JOURNAL OF COMPUTATIONAL PHYSICS **40**, 517–526 (1981)

Note

A Very Fast Shift-Register Sequence Random Number Generator

Introduction

In the most widely used class of pseudo-random number generators [1, 2], each random integer, x_i, is obtained from its predecessor, x_{i-1}, by

$$x_i = ax_{i-1} \qquad (\text{mod}\, m). \tag{1}$$

It is convenient to discuss the integer sequence $\{x_i \mid 0 < x_i < m\}$, although in practice one would convert the x_i into floating point numbers distributed over some fixed finite interval, such as (0, 1). If m is prime and a is a positive primitive root of m, i.e., $a^m = 1$ (mod m) but $\forall_{n<m}\, a^n \neq 1$ (mod m), then a single periodic sequence of integers is generated by (1) with the maximum possible cycle length, $m-1$. For computers with 32 bit words, the Mersenne prime $m = 2^{31} - 1$ is a convenient modulus. A subroutine GGL [3], using (1) with $a = 7^5 = 16807$, which is primitive with respect to $2^{31} - 1$, has seen extensive use on IBM computers. It is found to have good statistical properties [4], and is quite fast. On an IBM 370/168 it requires roughly 2 μsec to generate each new random number.

Its drawback is its cycle length, $2^{31} - 1 \approx 2 \times 10^9$ steps, which can be exhausted in about one hour on a modern high speed computer. It is no longer unusual for a single simulation to consume more than 10^8 random numbers. The customary procedure for initializing a random number generator is to supply an arbitrary value for x_1. Unless the user is careful to let x_1 be the last random number generated in the previous simulation, there is a significant likelihood that overlapping portions of the basic sequence will be generated in successive simulations. This may lead to redundant results. By combining n distinct random number sequences of cycle length $(m-1)$ one can in principle [6] increase the effective cycle length to $(m-1)^n$, but this costs at least an n-fold increase in the time required to generate each random number.

Use of d successive pseudo-random numbers as sample coordinates in a d-dimensional space introduces a second need for longer cycle length. A single-step recurrence like (1) produces only $m-1$ distinct successor d-tuples. Since there are $(m-1)^d$ possible positions in the d-dimensional space, the set of positions generated is sparse. Worse, there are known to be unfortunate choices of a for which the d-tuple sample positions are very unevenly distributed for some d, even though the cycle length is maximal [7]. In order to assure uniformly distributed sampling in d-space, therefore, a cycle length exceeding $(m-1)^d$ is desirable.

0021-9991/81/040517–10$02.00/0

581/40/2-17

An algorithm for producing pseudo-random bit sequences of effectively unlimited period was introduced by Tausworthe [8, 9]. Let the kth bit, a_k, in the sequence be given by

$$a_k = c_1 a_{k-1} + c_2 a_{k-2} + \cdots c_{p-1} a_{k-p+1} + a_{k-p} \qquad (\text{mod } 2). \tag{2}$$

Then since each p-tuple of successive bits depends only upon the previous n-tuple, the maximum possible cycle length is $2^p - 1$. This is achieved iff the polynomial $1 + c_1 x + c_2 x^2 + \cdots + c_{p-1} x^{p-1} + x^p$ is primitive over GF(2) [2]. A sequence of m-bit random integers $\{x_i\}$ can be thought of as m columns of random bits, and bitwise addition without carry is simply the "exclusive or" operation, denoted $\oplus$, commonly available as a primitive machine instruction. Thus the algorithm [10]

$$x_k = c_1 x_{k-1} \oplus c_2 x_{k-2} \oplus \cdots \oplus c_{p-1} x_{k-p+1} \oplus x_{k-p}, \tag{3}$$

may generate very long pseudo-random sequences. Primitive trinomials, $1 + x^q + x^p$, $p > q$, have been identified up to quite large order [11]. Using a primitive trinomial reduces (3) to

$$x_k = x_{k-q} \oplus x_{k-p}, \tag{4}$$

which requires only one exclusive or and some address calculations for each new integer generated. A random number generator based on (4) may therefore be as fast as or faster than a multiplicative congruential generator of the type (1), while providing a period $2^p - 1$ and requiring only the extra space to store the previous p iterates.

The chief weakness of (4) is that it requires careful initialization. If the ith and jth bits are identical in each of the first p integers of the sequence they will remain identical throughout. If the first p entries in the ith and jth bit positions are nearly identical it may take many iterations before they become independent. Lewis and Payne [10] propose initializing $\{x_1 \cdots x_p\}$ such that each column of bits contains a delayed replica of the same basic bit sequence, using a delay of order $100p$ steps between columns. Initialization to generate n-bit random numbers can be done with $100pn$ calls to the basic subroutine, and thus is moderately time-consuming. Extensive statistical tests [11, 12] have confirmed the safety of this initialization procedure.

We propose a quicker and less cumbersome initialization procedure. Simply use a good random number generator of type (1) to produce the first p integers, then continue with algorithm (4). The risk in doing this is that the columns of bits one starts with may not be linearly independent. If that occurs, the sequence will not generate all possible floating point numbers, and thus will not have the maximum period [9, 12]. For $p > 50$ this is extremely unlikely, but a simple construction can be incorporated into the initialization to guarantee linear independence of the columns if desired. (For this suggestion we are indebted to J. Arthur Greenwood.) Let s be the number of bits in each mantissa. Choose s distinct numbers from among the q initial random numbers and think of the bits in their mantissas as forming an s by s element square array. Replace the diagonal elements of this array by '1's, and the lower triangle of the array by '0's, and restore the modified numbers to their original positions. Finally, initialization by giving a single seed to the first random number generator makes it simple to reproduce a simulation while the extreme cycle length of (4) minimizes the danger that two different seeds will create overlapping sequences.

We have used GGL [3] to initialize a shift-register sequence based on the primitive trinomial with $q=103$, $p=250$. Comparisons of the resulting generator, denoted R250, with GGL are presented in the next section. The subroutine used is exhibited (in IBM 370 assembly language) in the Appendix. If a smaller storage requirement, or a still longer cycle is desired, other values of p and q may be substituted in the program. The sets $(p, q) = (98, 27)$ and $(521, 32)$ both give primitive trinomials [12, 13].

Comparisons

Figures 1a–c each display 10,000 random numbers, grouped in pairs and plotted in the unit square. The numbers in Fig. 1a were created with GGL, those in Figs. 1b and c with R250. The quicker initialization procedure was followed in constructing Fig. 1b, while the full delayed replica process was employed in Fig. 1c. To cursory inspection, all three distributions look uniform and correlation-free. There is no evidence of the striated density variations identified by Marsaglia [5]. However, on closer inspection, one's eyes invariably detect local patterns, whether they have any statistical validity or not. A more quantitative statistical analysis is needed to determine whether any given process generates numbers which are neither too evenly distributed or too clumped to be independent.

Triples of random numbers are often used in simulations. To test the uniformity of successive triples of random numbers generated by the two algorithms, 10^6 such triples were constructed for each and assigned to cells in the unit cube, with a resolution of $32 \times 32 \times 32$ cells. This tests properties of the leading five bits of each random number in the sequence. Since all columns of bits generated by R250 have the same statistical characteristics, results of this test apply to any subset of bits in the sequence of numbers generated by R250. A useful measure of the cell-to-cell variation is the χ^2-like quantity

$$\varphi = L^{-3} \sum_{\text{cells } i,j,k=1}^{L} [n(i,j,k) - n_0]^2/n_0, \tag{5}$$

where $n(i, j, k)$ denotes the number of triples falling into cell (i, j, k) and n_0 is the mean number of triples per cell. If the x_i are independently distributed, $\varphi \sim 1$. For both R250 and GGL we obtained $\varphi = 1.00 \pm 0.005$, taking several samples of 10^6 numbers each.

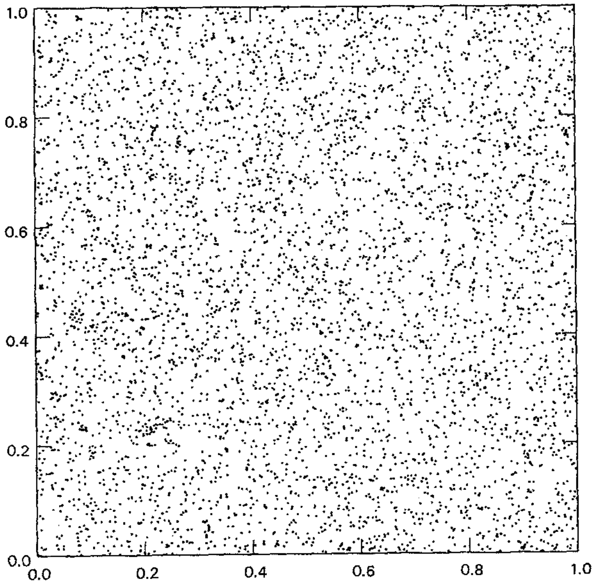

FIG. 1a. 10,000 random numbers uniformly distributed in the unit interval, generated by GGL and plotted as 5000 sequential (x, y) pairs.

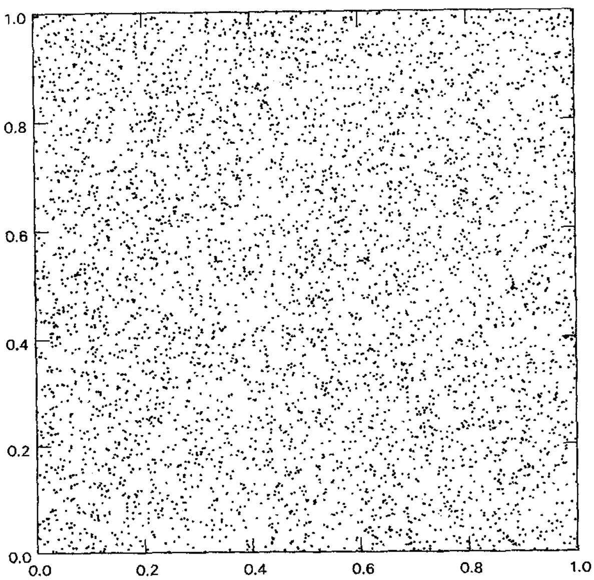

FIG. 1b. 10,000 random numbers, plotted as in Fig. 1a, but generated by R250 with the simplified initialization.

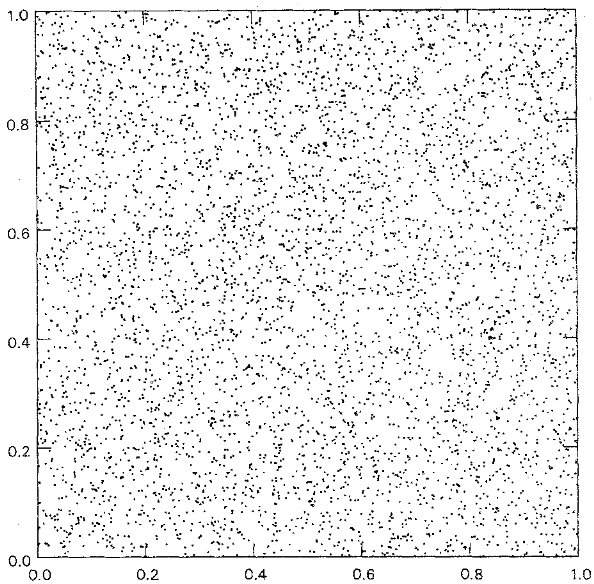

FIG. 1c. As in Figs. 1a and b, but using a sequence generated by R250 following the full delayed columns initialization procedure suggested in Ref. [9].

Autocorrelation statistics for both generators agree with the results expected for uniformly and independently distributed random variables. The nth autocorrelation coefficient, $g(n)$,

$$g(k) = \langle x_i x_{i-k} \rangle - \langle x_i \rangle^2, \tag{6}$$

was obtained for sequences of 10^4 to 10^6 random numbers and $0 \leqslant k \leqslant 20$, using both GGL and the simplified form of R250. In Table I we show averages of $g(k)$ over 100 samples of 10^4 random numbers each. The variance expected when $g(k)$ is measured for a sequence of N numbers is

$$\begin{aligned}\{\langle g(k)^2 \rangle - \langle g(k) \rangle^2\}^{1/2} \\ = N^{-1/2}\{\langle x_i^2 \rangle^2 + 2\langle x_i^2 \rangle \langle x_i \rangle^2 - 3\langle x_i \rangle^4\}^{1/2} \\ \approx 0.3N^{-1/2}.\end{aligned} \tag{7}$$

The results in Table I are in good agreement with this, as were experiments with other values if N. We also note that there is no obvious dependence of either $\langle g(k) \rangle$ or its variance on the separation, k. Although the variability observed with GGL is slightly greater than that of R250, the difference seems too small to have any observable impact on a simulation.

Testing for the probability of runs of increasing or decreasing x_i gives a measure of any higher order correlations not detected in the two point correlation $g(k)$. We consider that a sequence such that $x_{-1} > x_0 < x_1 < \cdots < x_{s-1} < x_s > x_{s+1}$ contains a run of length s between x_0 and x_s. The expected number of such runs (of all lengths) in a sequence of N independent random numbers is [4] $\sim 2N/3$. The expected number

TABLE I

Autocorrelation Statistics[a]

10000 RANDOM NOS WITH R250 100 TIMES				
AVERAGED AUTOCORRELATION COEFFICIENTS= 0.333374				
0.249928	0.250060	0.249970	0.249855	0.249933
0.249965	0.250088	0.249890	0.250044	0.250154
0.250034	0.250073	0.250095	0.250110	0.249976
0.250092	0.249940	0.250081	0.249979	0.250107
VARIANCE OF AUTOCORRELATION COEFFICIENTS= 0.003109				
0.003138	0.003105	0.003088	0.003071	0.003014
0.002923	0.003113	0.002913	0.003025	0.002994
0.003081	0.002926	0.003011	0.003008	0.003017
0.003063	0.003160	0.002998	0.003025	0.002971
10000 RANDOM NOS WITH GGL2 100 TIMES				
AVERAGED AUTOCORRELATION COEFFICIENTS= 0.334018				
0.250610	0.250521	0.250523	0.250467	0.250709
0.250513	0.250510	0.250666	0.250519	0.250600
0.250533	0.250520	0.250480	0.250565	0.250532
0.250675	0.250518	0.250539	0.250650	0.250638
VARIANCE OF AUTOCORRELATION COEFFICIENTS= 0.003215				
0.003160	0.003127	0.002979	0.003169	0.003153
0.003133	0.003226	0.003245	0.003121	0.003157
0.003108	0.003095	0.003311	0.003229	0.003108
0.003100	0.003283	0.003154	0.003278	0.003303

[a] Autocorrelation coefficient, $g(k)$ (5), and its variance (6), for the multiplicative congruential random number generator GGL (lower table) and the shift register sequence R250 (upper table). The results for $k = 0$ are given in the title lines, followed by each quantity for $k = 1$ to 20.

TABLE II

Run Length Statistics[a]

NRUNS	CHISQ(6)	N(1)	N(2)	N(3)	N(4)	N(5)	N(6)	N(7)	N(8)	N(9)
666666	6.00	416667	183333	52778	11508	2034	303	39	4	0
666411	0.87	416315	183335	52839	11521	2040	315	38	6	0
666631	5.77	416918	182958	52702	11634	2037	335	40	6	0
666944	2.71	417009	183286	52883	11425	2017	282	36	4	1
666632	13.88	416879	182632	53420	11315	2029	306	45	4	1
667412	4.71	417762	183189	52696	11426	2007	289	38	3	1
666354	6.69	416259	183409	52645	11596	2114	278	52	0	0
666747	1.00	416804	183320	52706	11518	2066	293	35	3	1
666906	6.13	417192	183023	52848	11400	2103	282	47	8	2
667585	9.82	418125	183247	52301	11494	2060	305	44	8	0
666278	5.68	416195	183200	52887	11593	2087	274	35	6	0
666881	2.34	416861	183461	52801	11374	2024	316	42	1	0
666256	2.52	415859	183682	52854	11486	2023	308	36	7	0
667423	10.52	417774	183306	52608	11280	2105	305	38	6	0
667477	9.54	417708	183584	52463	11294	2063	313	44	6	1
666225	6.51	416381	182885	52918	11552	2102	330	51	5	0
666549	2.18	416625	183223	52710	11572	2047	325	43	3	0
667250	11.00	417599	183305	52301	11713	1990	302	38	1	0
666245	12.82	416231	183124	52727	11848	1985	287	35	7	0
667034	6.30	416891	183693	52763	11394	1941	309	39	3	0
667147	3.74	417593	182924	52751	11470	2061	312	32	2	1

[a] Numbers of runs of either strictly increasing or strictly decreasing random numbers in 20 sequences of 10^6 numbers each, generated by the shift register algorithm using the simplified initiation procedure. The top row gives the expected values for this sequence length.

of runs of length s will be $n(s) \sim 2N[(s+1)^2 + s]/(s+3)!$ Results for 20 cases using R250, each with $N = 10^6$, are compared in Table II with the expected values. The observed run lengths agree with the expected values. To analyze the variation observed we use

$$\chi^2(k) = \sum_{s=1}^{k} [n_{\rm obs}(s) - n_{\rm exp}(s)]^2/n_{\rm exp}(s). \tag{8}$$

Fluctuations in χ^2 are large. For 30 cases of $N = 10^6$ using R250 we obtained $\chi^2(6) = 6.2 \pm 3.5$. Published data [4] on GGL shows even higher variability for this test, with $\chi^2(k) > k$.

As a final test, we have employed R250 and GGL in Monte Carlo calculations of the energy and order parameter (magnetization) in the 2D Ising ferromagnet on a square lattice, for which exact results are known. The exact magnetization is given by [14]

$$m(T) = [1 - \operatorname{csch}^4(2J/k_B T)]^{1/8}. \tag{9}$$

We considered a temperature just below the critical temperature, $T_c \approx 2.264 J/k_B$. For the temperature chosen, (9) gives $m(0.96T_c) = 0.8146$ for an infinite system, while the internal energy is $U(0.96T_c) = -1.4547\,J$ per spin. A sample of 440×440 spins was used, with all spins initially set $= +1$. The random numbers were used both to determine which spin should attempt to flip and to compare with the Boltzmann factor in determining whether the attempt was successful. Results for $m(0.96T_c)$ and $U(0.96T_c)$ were averaged over periods for 10 successive groups of 20 MCS each, using both R250 and GGL. In each case, the answers are converging to the exact values. The congruential generator RANDU, known to have poor triplet distribution properties [7], fails this test. It leads to a magnetization roughly 10% too large.

TABLE III

2D SQ Ising Monte Carlo Test[a]

R250:											
$m(T)$	0.8759	0.8380	0.8297	0.8259	0.8245	0.8221	0.8256	0.8180	0.8193	0.8153	0.8146
$-U(T)$	1.5348	1.4708	1.4638	1.4606	1.4614	1.4588	1.4626	1.4554	1.4558	1.4556	1.4547
GGL:											
$m(T)$	0.8754	0.8328	0.8253	0.8224	0.8197	0.8218	0.8200	0.8202	0.8177	0.8116	0.8146
$-U(T)$	1.5334	1.4640	1.4600	1.4566	1.4564	1.4604	1.4572	1.4590	1.4584	1.4504	1.4547

[a] Monte Carlo calculations of magnetization and internal energy for 440×440 spins ina 2D Ising ferromagnet, each point averaged over 20 MCS.

Programming Considerations

Programming the algorithm (3) for a given p and q is straightforward. If the random numbers are to be used in floating point arithmetic operations, on most machines they need not be normalized, i.e., high order zeroes in the mantissa can be tolerated. On the other hand, comparing two unnormalized floating point numbers can give wrong results. For unnormalized random numbers the exclusive or operations can be applied to the floating point numbers directly, followed by an operation to reset the correct exponent. After calculating a vector of random numbers, such a subroutine must copy the last p iterates into the first p locations to be ready for subsequent calls.

If normalized random numbers are required, the subroutine must keep an array of the q most recent unnormalized mantissas in addition to the normalized results, some of which may have been left-shifted. The effort required is therefore two stores and three address calculations per cycle instead of one store and two address calculations. In the Appendix we give a example of algorithm (3) encoded to produce unnormalized numbers. Timing comparisons between normalized and unnormalized variants of the algorithm, and a fast version of GGL [3, 5] are given in Table IV. Because of the time required to shift the last 250 iterates, the full efficiency of the unnormalized algorithm is not realized in calculating short sequences. Tests with an unnormalized version of R250 which used an auxiliary rotating array suggested that the breakeven point lies around 1000 iterates.

In conclusion, the Tausworthe-type shift register algorithm with only simple precautions taken to insure a safe initialization provides statistical characteristics which are indistinguishable in the short run from those of the best known multiplicative congruential generators, and in the long run should be superior. There is no performance penalty associated with the longer cycle length. In fact, if unnormalized random floating point numbers are acceptable (or if one uses a computer for which the ratio of multiply to add time is greater than it is on the 370/168), the shift register algorithm is the faster.

TABLE IV

Performance Comparison[a]

GGL2	(normalized)	2.0 μsec
GGL2	(unnormalized)	1.5 μsec
R250	(normalized)	2.1 μsec
R250	(unnormalized)	1.0 μsec

[a] Time required to generate each random number, using assembly coded implementations of algorithms (1) and (4) on an IBM 370/168. 10^7 numbers were generated by each method to obtain timing.

Appendix

```
********* R250U *************S KIRKPATRICK 1-21-80 *******************
*                                                                    *
*        THIS SUBROUTINE GENERATES PSEUDO-RANDOM NUMBERS BY THE      *
*        ALGORITHM:  M(I)=M(I-147) XOR M(I-250)                      *
*        (WHERE M IS THE MANTISSA OF FLOATING POINT NO X(I)          *
*        EXTREMELY LONG CYCLE LENGTH IS A CONSEQUENCE OF FACT THAT   *
*        X**250 + X**103 + 1 IS A PRIMITIVE POLYNOMIAL               *
*                                                                    *
*        IN THIS VERSION, FIRST 250 LOCATIONS OF X MUST CONTAIN      *
*        RANDOM NUMBERS ON ENTRY.  THE NEXT N RECEIVE UNNORMALIZED   *
*        RANDOM NUMBERS, THE LAST 250 OF WHICH ARE ALSO SAVED IN     *
*        LOCATIONS 1-250.                                            *
*                                                                    *
*        IT MAY BE INVOKED FROM A FORTRAN PROGRAM BY THE STATEMENT:  *
*        CALL  R250 (X,N)                                            *
*        TO INVOKE FROM PL/I, FORCE CALL BY NAME LINKAGE:            *
*        CALL R250 (X(1),N);                                         *
*                                                                    *
*        WHERE X = ANY REAL*4 ARRAY OF DIMENSION >= N + 250          *
*              N = ANY INTEGER*4 VARIABLE (N>=1), THE NUMBER OF REAL *
*                  RANDOM NUMBERS TO BE GENERATED AND PLACED IN X    *
*                                                                    *
*        INTERNAL INDICES: K = POSITION IN ARRAY X                   *
*              I = POSITION OF FIRST PREVIOUS NO IN ROTATING LIST M  *
*              J = POSITION OF SECOND PREVIOUS NO IN M               *
*                                                                    *
*        REFERENCE:  LEWIS AND PAYNE, J ACM 20, 456 (1973)           *
*                                                                    *
**********************************************************************
*
R250U    CSECT
         USING *,15
SAVE     STM   14,12,12(13)    SAVE REGISTER CONTENTS
         ST    13,SAVEAREA+4
         LR    2,13
         LA    13,SAVEAREA
         ST    13,8(2)
INC      EQU   2
N        EQU   3
XADR     EQU   6
I        EQU   7
J        EQU   8
K        EQU   9
CHAR     EQU   10
L        EQU   11
         LM    J,L,INITJ       INITIALIZE J,K,CHAR IN R8-R10
         LA    INC,4           INCREMENT=4 BYTES (1 WORD)
         LM    6,7,0(1)        R6-R7 CONTAIN ADDRESSES OF X(1),N
         L     N,0(0,7)        R3=N
         SLA   N,2             R3=4*N
         SR    XADR,INC        R6=ADDRESS OF WORD BEFORE X
         LR    I,INC           I=4 INITIALLY
         CNOP  2,8             POSITION START OF LOOP FOR BEST TIMING
LOOP     AR    J,INC           INCREMENT J
         L     4,0(I,XADR)     LOAD I-TH RANDOM INTEGER
         AR    K,INC           INCREMENT K
         X     4,0(J,XADR)     XOR WITH INTEGER AT J
         OR    4,CHAR          SLAP THE APPROPRIATE EXPONENT ON IT
         ST    4,0(K,XADR)     SAVE THF RESULT
ENDLOOP  BXLE  I,INC,LOOP      TEST FOR DONE
*        MOVE THE LAST 250 RESULTING NOS INTO FIRST 250 POSITIONS
         LR    J,I
         AR    J,XADR          COMPUTE ADDR OF FIRST NO TO SAVE
         AR    XADR,INC        COMPUTE ADDR OF X(1)
         LR    I,L
         LR    K,L             LENGTH OF AREA TO SAVE
         MVCL  XADR,J          MOVE EM OUT
*
         L     13,SAVEAREA+4   TERMINAL LINKAGE
         RETURN (14,12)
*
*                              CONSTANTS
INITJ    DC    F'412'
INITK    DC    F'1000'
DCCHAR   DC    X'40000001'     THE 1 ASSURES THAT X>0.
LEN      DC    F'1000'
SAVEAREA DS    18F
         END
```

References

1. D. H. Lehmer, 2nd Symposium on Large-Scale Digital Calculating Machinery," Cambridge, Mass, 1951, p. 141.
2. D. Knuth, "The Art of Computer Programming," Vol. 2, Addison–Wesley, Reading, Mass., 1969.
3. IBM Subroutine Library—Mathematics, User's Guide (1971), program number 5736-XM7.
4. P. A. W. Lewis, A. S. Goodman, and J. M. Miller, *IBM Systems J.* **8** (1969), 136.
5. F. Gustavson and W. Liniger, *Computing* **6** (1971), 221.
6. D. Knuth, "The Art of Computer Programming," Vol. 2, p. 30, Addison–Wesley, Reading, Mass., 1969.

7. G. MARSAGLIA, *Proc. Natl. Acad. Sci.* **61** (1968), 25.
8. R. C. TAUSWORTHE, *Math. Comput.* **19** (1965), 201.
9. S. W. GOLOMB, "Shift Register Sequences," Holden–Day, San Francisco, 1967.
10. T. G. LEWIS AND W. H. PAYNE, *J. Assoc. Comput. Mach.* **20** (1973), 456.
11. N. ZIERLER AND J. BRILLHART, *Inform. Contr.* **14** (1969), 566.
12. H. S. BRIGHT AND R. L. ENISON, *Computing Surveys* **11** (1979), 357.
13. T. G. LEWIS, "Distribution Sampling for Computer Simulation," Heath, Lexington, Mass., 1975.
14. C. N. YANG, *Phys. Rev.* **85** (1952), 809.

RECEIVED: April 1, 1980

SCOTT KIRKPATRICK

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ERICH P. STOLL

IBM Research Laboratory
Säumerstrasse 4
8803 Ruschlikon, Switzerland