General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

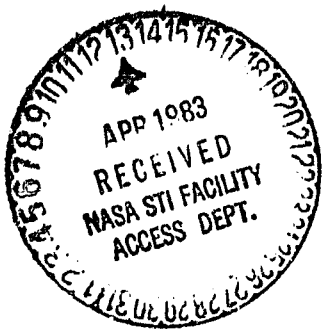
- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

NASA TECHNICAL MEMORANDUM



NASA TM- 82516

PSEUDO-RANDOM NUMBER GENERATOR FOR THE SIGMA V COMPUTER

By Stanley N. Carroll
Systems Dynamics Laboratory

February 1983

NASA

George C. Marshall Space Flight Center
Marshall Space Flight Center, Alabama

TECHNICAL REPORT STANDARD TITLE PAGE

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3. RECIPIENT'S CATALOG NO.
NASA TM-82516		

4. TITLE AND SUBTITLE	5. REPORT DATE
Pseudo-Random Number Generator for the Sigma V Computer	February 1983
	6. PERFORMING ORGANIZATION CODE
7. AUTHOR(S)	8. PERFORMING ORGANIZATION REPORT #
Stanley N. Carroll	
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. WORK UNIT NO.
George C. Marshall Space Flight Center Marshall Space Flight Center, Alabama 35812	11. CONTRACT OR GRANT NO.
	13. TYPE OF REPORT & PERIOD COVERED
12. SPONSORING AGENCY NAME AND ADDRESS	Technical Memorandum
National Aeronautics and Space Administration Washington, D.C. 20546	14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

Prepared by Systems Dynamics Laboratory, Science and Engineering.

16. ABSTRACT

A technique is presented for developing a pseudo-random number generator based on the linear congruential form. The two numbers used for the generator are a prime number and a corresponding primitive root, where the prime is the largest prime number that can be accurately represented on a particular computer. The primitive root is selected by applying Marsaglia's lattice test. The technique presented has been applied to write a new random number program for the Sigma V computer. The new program, named S:RANDOM1, is judged to be superior to the older program named S:RANDOM. For applications requiring several independent random number generators, a table is included showing several acceptable primitive roots. The technique and programs described in the report can be applied to any computer having word length different from that of the Sigma V.

17. KEY WORDS	18. DISTRIBUTION STATEMENT
Pseudo-random Random-number generator Prime number Primitive root	Unclassified – Unlimited

19. SECURITY CLASSIF. (of this report)	20. SECURITY CLASSIF. (of this page)	21. NO. OF PAGES	22. PRICE
Unclassified	Unclassified	17	NTIS

ACKNOWLEDGMENT

The author gratefully acknowledges the advice and assistance of Mr. Warren Adams of the Systems Dynamics Laboratory, Marshall Space Flight Center.

TABLE OF CONTENTS

LIST OF TABLES

TECHNICAL MEMORANDUM

PSEUDO-RANDOM NUMBER GENERATOR FOR THE SIGMA V COMPUTER

INTRODUCTION

This report describes a basic approach for developing a random number generator. Although the approach is not restricted to a specific computer, the concept is illustrated with application to the Sigma V computer at the Marshall Space Flight Center. As will be seen later, a few of the specific computer characteristics must be accounted for when preparing code for some of the routines.

Many of the more popular pseudo-random number generators use the linear congruential form

$$Z(i+1) = Z(i) * C \quad \text{Mod } (M) \tag{1}$$

Here M and C are integers and Z(i) is the previous number. The expression Mod (M) implies the product Z(i)*C be expressed as the remainder for modulus M. Many choices are available for selecting the integers M and C to build a random number generator; however this report is intentionally restricted to the special case where M is a prime number and C is a primitive root of M. Other choices for selecting M and C may be found in Reference 3. The definition and pertinent properties of a primitive root are covered in the Appendix. Although by strict definition this report deals only with pseudo-random numbers, the terms pseudo-random and random are used interchangeably unless a specific distinction is needed for clarity.

A highly desirable characteristic of any random number generator is a long cycle length, or period, where cycle length is the count of distinct numbers generated before any number is repeated. The type of generator discussed herein has a cycle length of exactly M-1. This maximum length period only applies when C is a primitive root; if C is not a primitive root the maximum period is in the range of 2 to (M-1)/2. An example of this property is illustrated in the Appendix which contains a sample table dealing with the prime number 19.

The above definition of cycle length infers that any sequence of M-1 consecutive numbers must contain all integers between 1 and M-1. This feature provides the generator with the property of being uniformly distributed. The random property is based on the order, or position, of the M-1 integers within a given string. Thus, as can be deduced from the above equation, the random property is controlled by the primitive root. Also, each primitive root will "shuffle" the M-1 integers into a different order. Dividing the integer sequence by the number M produces a new sequence with elements on the open interval (0,1). Elements of this new sequence are said to be uniformly distributed over the range 0 to 1, exclusive. A major reason for developing a high quality uniformly distributed random number generator is that it in turn is the basic element for constructing different number generators with other type distributions, e.g., Poisson, normal, etc.

The user of a random number generator should be aware that most generators have some degree of limitation. Although a particular generator may receive an excellent rating based on previous application and utilization, this rating does not guarantee that the generator will be suitable for a future

application significantly different from what has been done in the past. An example of this is cited later where a particular generator is ideal for 2 dimensional problems, but the same generator can cause erroneous results when used in 3 dimensional problems. To avoid misusing a generator, the user should become familiar with the generator's characteristics and limitations.

Every primitive root yields the same set of numbers but in a different sequence, thus one strongly suspects that some of these sequences will yield better results than other sequences. The primitive roots which give the better sequences should be used for the constant C in equation (1). Likewise, some of these sequences show very poor traits and the corresponding primitive roots should not be used for this application. The tool for determining which primitive root to use and which to discard is provided by the lattice test.

LATTICE TEST

The lattice test developed by Marsaglia [4] is a theoretical method for evaluating pseudo-random number generators based on the linear congruential form. This test is based on the way the numbers are generated and does not require a sample for statistical analysis. The only two numbers needed to execute the lattice test are M and C.

A problem of many generators is the lack of independence between pairs, triplets, etc. Results from the lattice test give a qualitative measure of this independence and only a minimum amount of computational effort is required. The test itself constructs an n-lattice cell in n-space. Measure of goodness is determined by the ratio of the largest cell dimension to the smallest cell dimension. In 3-space the perfect generator would have a cubic lattice; whereas, a bad generator would take the form of a very long tube having a small cross sectional area. In the section on APL Generators an example is given of a generator having a very good 2-lattice structure but a very poor 3-lattice structure.

For n dimensions the lattice test uses the n rows of the following matrix form:

$$\begin{bmatrix} 1 & C & C^2 & \cdots & C^{n-1} \\ 0 & M & 0 & \cdots & 0 \\ 0 & 0 & M & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & M \end{bmatrix}$$

Next Marsaglia's BEST2 algorithm is applied to the above matrix. Defining the rows as Pi and Pj ($i < j$) the steps of BEST2 are:

1) If $PjPj^T < PiPi^T$, interchange Pi and Pj

2) Replace Pj by (Pj - L*Pi); L is the integer closest to $PiPj^T/PiPi^T$.

3) If for new Pj, $PjPj^T > PiPi^T$, increment j and go to 1, otherwise go to 1 without incrementing j.

Repeat BEST2 between all pairs of rows until no alternations occur. Measure of goodness is the ratio

$$PnPn^T/P1P1^T$$

Guidelines on judging "good" from "bad" are given by Marsaglia. A ratio less than 2 is good while a ratio larger than 3 is bad.

To construct a good random number generator one is faced with the problem of finding a favorable primitive root which gives a small cell size ratio. The criteria to have a large cycle length requires M to be very large (around 2.1 billion on the Sigma). A large M however means many primitive roots (around 500 million) must be analyzed; thus, because of the large quantity of cases involved, performing a search on the entire spectrum to find the absolute best is deemed impractical. Instead, the approach used was to search a reasonably sized interval and find all primitive roots with acceptable cell size ratios. The next section discusses the prime number selection for building the random number generator S:RANDOM1 and the criteria used to pick candidate primitive roots.

PRIMITIVE ROOT SEARCH

The Sigma V computer is a 32 bit word machine and the largest positive integer which can be represented is

$$M = 2^{31} - 1 = 2,147,483,647$$

This integer happens to be a prime number; hence it was selected as the value to use in constructing the random number program S:RANDOM1. To use this value of M requires a special integer multiplication routine to avoid overflow problems that would occur with standard FORTRAN multiplication. The Intrinsic FORTRAN multiplication of 2 integers return an answer that is the right most 32 bits of a 64 bit word; thus, anytime a product requires more than 31 bits for the numerical representation, the returned answer will be in error. To avoid this problem the added multiplication routine utilizes the full 64 bits. Also, additional steps were incorporated into the multiplication routine to reduce the 64 bit result to a remainder for modulus M. The modulus operation insures the final output cannot be larger than M-1 and therefore can be accurately represented with 32 bits. Thus, no computer overflow problems can occur using this multiplication routine.

Following the technique outlined in the Appendix, all primitive roots can easily be found once any primitive root is known. A computer program for finding the least positive primitive root is currently on the Sigma V in the author's account. With the aid of this program the least positive primitive root of M was determined to be 7. The technique outlined in the Appendix shows that since 5 is relative prime to M-1, then another primitive root must be

$$7^5 \text{ Mod } (M) = 16,807$$

Although the above number is frequently cited in the literature [1,4], this number does not exhibit an outstanding lattice structure; in fact it does not rate high among the first few primitive roots generated. Using 7 as a base for the calculations, Table 1 summarizes the lattice test results for the first 15 primitive roots. The output number listed under each Li is the ratio of the largest cell size dimension to the smallest cell size dimension. The root sum squared of the 4 numbers, L2 through L5, is shown under the heading, RSS. The computer program used to generate the data provided only the RSS value for the primitive root 7. In the lower part of the table the data has been sorted according to increasing RSS values. Although one of these entries, exponent of 47, has ratios less than 2 for all dimensions tested, the entry has a RSS value which is 65 percent larger than the theoretical minimum of 2. Because the objective was to construct a generator which would be suitable for a wide range of applications, more emphasis was put on finding the primitive root(s) with an associated small RSS value(s).

To find a selection of primitive roots having reasonably small RSS values, a program was written to search a specified interval and output to the line printer the status on all new constants found which were better than the best found up to that point. Also, if a number was found with a RSS value within a few percent of the best RSS value then this number was also recorded. At the start of the search the percentage used was 10 percent, later it was dropped to 5, then 2.5, and finally to 1. The interval search was done in two phases. For the first part the search (i.e., exponents of 7) extended up to 80,000. No limitations were placed on the magnitude of acceptable primitive roots, and the minimum tolerance was down to 2.5 percent at the end of the search interval. For the second phase the interval was extended to 1 million but the acceptable primitive roots were constrained to be bigger than 500 million. One reason for adding the constraint was to speed up the search procedure; a second reason was the preference to have primitive roots in the range between mid-size to large. A constant 1 percent on the tolerance band was used for the second phase.

Table 2 contains a summary of the relevant output data which was generated by this search. Since only 0.046 percent of the total interval was searched no claim is made that these primitive roots are the best. They do however provide a basis for a very good random number generator for dimensions not exceeding 5. Should the need exist the information contained in Table 2 offers the user many excellent options for constructing independent generators. The last entry, C = 660,601,212, is the value used in the FORTRAN version of S:RANDOM1.

RANDOM1 PROGRAM

A binary file named B:RANDOM1 is on the Sigma V in the account name SCARROLL. This file has two separate pseudo-random number generators plus the necessary 64 bit multiplication routines. The distributions for the two generators are the uniform and the normal. Names and calling arguments for these two subroutines are:

Uniform: RNDU (X,N)

and

Normal: RNDN (X,N) .

Each subroutine must be initialized to establish the starting value for the random number calculations [i.e., Z(0)]. Initialization is done by calling the subroutine with a negative or zero value for N. Each subroutine has the intrinsic starting value, J0, of

$$J0 = 123,456,789$$

Initialization is done by the operation:

$$Z(0) = (J0)^{-N} \text{ Mod } (M) \quad .$$

After the call to initialize the subroutine all subsequent calls should be done with N positive. Specifics of each subroutine follows:

Uniform:

a) Range of output variable X is between 0 and 1.

b) N specifies the number of different random numbers to generate. The program can generate N numbers in one call in lieu of using N calls to get N numbers.

c) If N is larger than 1, main program must have a dimension statement for variable X.

Normal:

a) Output variable is X. Distribution has a mean of zero and standard deviation one.

b) Same as b under Uniform.

c) Same as c under Uniform.

d) Implementation technique uses Marsaglia's rectangle-wedge-tail method as outlined in Reference 3. This method has essentially perfect accuracy and a very fast execution time. It requires only one uniform number calculation approximately 88 percent of the time.

This binary file is available to any interested user. To conserve computer memory the user should use the LYNX command to link B:RANDOM1.SCARROLL with their binary file name in lieu of copying the file to the individual's account. The file B:RANDOM1 contains four separate subroutines; in addition to the two random number programs named above, there are two multiplication routines named MULA and MULMOD.

APL GENERATOR

The APL software package on the Sigma V computer uses the same form generator as equation (1) but with

$$C = 65{,}539 = 3 + 2^{16}$$

and

$$M = 2^{31} = 2{,}147{,}483{,}648 \quad .$$

While this generator may be satisfactory for some applications, hidden trouble can result for applications where groups of 3 random numbers are used. A warning message about this generator can be found in Reference 2, where a derivation is given to show the correlation between three consecutive integers within the sequence.

The following results were obtained by applying the lattice test to the APL generator. The parameter N is the dimension and Li, i=2,3,4,5, has the same meaning as used previously.

N	L2	L3	L4	L5
2	1.0			
3	2.0	1819		
4	3.6	928	936	
5	1.1	173	179	179

For each value of N, the N-1 numbers represent the ratio of the N-1 cell dimensions to the smallest cell dimension. Except for two discrepancies these numbers agree with those found in Reference 4; the reference gives a 528 for L3 instead of a 928, and a 173 instead of a 179 for L4. The reference material is suspected to be in error since the data was extracted from another reference, and exact agreement occurs in 3 out of the 10 numbers.

The above mentioned correlation problem is also inherent to the generator using

$$C = 3 + 2^{18}$$

and

$$M = 2^{35} \quad .$$

This generator is discussed in Reference 4, and it also has appeared in some computer software packages.

To provide Sigma V APL users a better generator, a request was made to add an APL option which would be basically the same as that developed in FORTRAN. The only difference between the two generators is the choice for the multiplicative constant. The request to improve the APL version is documented in Reference 6, together with an explanation on the deficiency of the existing generator. The value of C selected for FORTRAN will not work in APL because the mantissa in APL should be limited to about 56 bits. This is approximately the level where round off errors start to occur. Since 31 bits are required for Z(i), the number C must be limited to no more than 25 bits (33,554,432) to

satisfy the 56 bit constraint. Three numbers were found which were considered satisfactory. These numbers are tabulated in Table 3. Actually, two new APL options were added to the Sigma V, these being denoted by APL1 and APL2. For APL1 the multiplicative constant is 16,807; for APL2 the constant is 29,903,947. In both cases the value for M is the same as used in the FORTRAN version. These APL options are obtained from logging on by requesting APL1 or APL2 instead of the normal APL.

Both primitive roots selected were tested in APL language to insure there were no overflow or round off problems with multiplication. Test results obtained in APL were verified by doing the same operation in FORTRAN using the 64 bit integer multiplication routine.

CONCLUSIONS AND RECOMMENDATIONS

This effort has produced an improved version for a random number generator and is available to all Sigma V users in both FORTRAN and APL. The recommendation is made that use of the older FORTRAN program named S:RANDOM be terminated and replaced with the new version S:RANDOM1.

TABLE I SAMPLE PRIMITIVE ROOTS OF $2^{31} - 1$

EXPONENT OF 7	MULTIPLIER C	RSS	L2	L3	L4	L5
1	7	43403600.0				
5	16807	8.7	7.60	3.39	2.07	1.67
13	252246292	38.4	1.25	38.04	5.15	1.31
17	52959638	3.6	1.03	1.28	2.88	1.29
19	447489615	6.1	2.18	4.73	2.59	1.94
23	680742115	6.8	3.40	5.28	2.17	1.29
25	1144108930	4.3	1.38	2.11	3.17	1.49
29	373956417	3.6	1.59	1.21	2.52	1.70
37	655382362	4.1	1.44	1.96	2.52	2.15
41	1615021558	37.0	36.51	1.61	4.59	3.45
43	1826645050	261.1	3.15	261.00	6.97	2.01
47	613157876	3.3	1.74	1.13	1.93	1.67
53	1287767147	3.9	1.13	2.46	2.46	1.38
59	1693265200	6.5	2.04	2.41	3.35	4.54
61	1365616214	4.5	3.39	1.41	2.18	1.41
ARRANGED BY INCREASING RSS VALUE						
47	613157876	3.3	1.74	1.13	1.93	1.67
17	52958638	3.6	1.03	1.28	2.88	1.29
29	373956417	3.6	1.59	1.21	2.52	1.70
53	1287767147	3.9	1.13	2.46	2.46	1.38
37	655382362	4.1	1.44	1.96	2.52	2.15
25	1144108930	4.3	1.38	2.11	3.17	1.49
61	1365616214	4.5	3.39	1.41	2.18	1.41
19	447489615	6.1	2.18	4.73	2.59	1.94
59	1693265200	6.5	2.04	2.41	3.35	4.54
23	680742115	6.8	3.40	5.28	2.17	1.29
5	16807	8.7	7.60	3.39	2.07	1.67
41	1615021558	37.0	36.51	1.61	4.59	3.45
13	252246292	38.4	1.25	38.04	5.15	1.31
43	1826645050	261.1	3.15	261.00	6.97	2.01
1	7	43403600.0				

TABLE 2. SEARCH SUMMARY FOR FORTRAN GENERATOR

Exponent	C value	RSS	L2	L3	L4	L5
1	7	Not Known, but very large				
5	16,807	8.7	7.60	3.39	2.07	1.67
17	52,[illegible]58,638	3.6	1.03	1.28	2.88	1.29
47	613,157,876	3.3	1.74	1.13	1.93	1.67
127	992,518,913	2.5	1.04	1.26	1.16	1.58
2065	628,070,245	2.5	1.03	1.07	1.45	1.44
6403	821,299,934	2.5	1.24	1.17	1.19	1.38
7879	1,640,094,722	2.5	1.14	1.16	1.25	1.36
9347	701,309,109	2.4	1.04	1.19	1.25	1.3
46,315	1,805,241,783	2.37	1.05	1.36	1.11	1.21
[illegible],863	1,083,678,114	2.37	1.02	1.24	1.14	1.32
50,641	980,585,909	2.34	1.04	1.16	1.23	1.23
76,567	1,536,846,600	2.33	1.03	1.11	1.36	1.13
241,807	1,109,775,543	2.32	1.04	1.26	1.10	1.22
247,073	1,809,235,139	2.30	1.02	1.19	1.08	1.30
252,823	1,873,419,453	2.30	1.15	1.08	1.11	1.24
273,613	1,288,490,716	2.31	1.03	1.11	1.36	1.13
291,653	1,147,915,962	2.28	1.31	1.02	1.14	1.07
300,295	2,112,383,910	2.20	1.12	1.08	1.15	1.23
364,477	2,060,627,732	2.30	1.02	1.12	1.10	1.26
395,803	1,315,115,343	2.24	1.11	1.02	1.19	1.16
442,625	1,050,334,082	2.24	1.10	1.14	1.10	1.15
560,089	660,601,212	2.23	1.09	1.04	1.17	1.16

TABLE 3. SEARCH SUMMARY FOR APL GENERATOR

Search range	Exponent	C value	RSS	L2	L3	L4	L5
*	76567	1,536,846,600	2.33	1.03	1.11	1.36	1.13
80K-120K	NOTHING						
120K-1M	438,461	2,171,411	2.38	1.02	1.05	1.25	1.39
	602,479	20,903,947	2.34	1.04	1.3	1.22	1.09
1M-2M	NOTHING						
2M-3M	2,089,639	10,428,306	2.38	1.02	1.15	1.34	1.22

* - Value used for early Fortran study

APPENDIX

PRIMITIVE ROOTS OF PRIME NUMBERS

This Appendix discusses the definition and pertinent properties of a primitive root. The discussion is supplemented by an example using the prime number 19.

Definitions:

M: prime number.

M1: M-1.

S: set of integers between 1 and M1, inclusive.

p: member of set S.

Primitive Root:

Consider the sequence

$$p\ ,\ p^2\ ,\ p^3\ ,\ \ldots\ p^n \text{ Mod } (M)\ ,\ n \leqslant M1\quad . \tag{A-1}$$

Since each element is calculated by Mod (M), no element can be larger than M1, and at least one element will be unity provided n is extended to M1. Consider the case when the remainder is unity, i.e.,

$$1 = p^n \text{ Mod } (M)\quad . \tag{A-2}$$

The definition of a primitive root is associated with what values of n satisfy (A-2). The number p is said to be a primitive root of M if the smallest value for n satisfying (A-2) is M1. For this case the sequence in (A-1) will contain all elements of the set S, i.e., all the integers between 1 and M1 will appear in some pseudo-random order and no number will appear twice.

Cycle length is defined as the number of integers in the sequence before any repetition occurs; hence, the cycle length for a primitive root is always M1. Table A-1 shows all possible sequences for the prime number 19. All candidate primitive roots, p, are listed in the first column; for each row, the 18 columns correspond to the powers of p per (A-1). Looking across the row for p = 2, observe that the first time the integer 1 appears is for n = 18; therefore, by the above definition, a primitive root of 19 is the number 2.

Knowing any primitive root, all other primitive roots can be found in an easy manner. The steps for this are:

1) Factor M1

2) Form p^n Mod (M), n=1,2,...M1.

3) The result in (2) is a primitive root if n is relative prime to M1.

In the sample table those numbers which are relative prime to 18 are indicated by an asterisk above the number. This data shows that, in addition to the integer 2, other primitive roots are 13, 14, 15, 3, and 10. Note that all primitive roots are obtained by this procedure regardless of the starting primitive root.

Most techniques for finding primitive roots do so by first finding the least positive root. In principle the algorithm for accomplishing this procedure is to start with the test number 2 and form the powers of 2. Looking at 2^n, if this number (after applying modulus M) is 1 but n is not M1, then increment the test number by 1 and start the process over. Continue this process until the first primitive root is found. In practice there are several short cuts which speed up the process and save considerable computer time. First, this test is not required beyond the midpoint, n = M1/2. If at the midpoint a unity remainder has not been calculated and the value for the midpoint calculation is M1, then the test number must be a primitive root. A further reduction in required work is that not all powers of p must be tested. The factors of M1 determine which powers must be checked [5]. This reference shows that all possible cycle lengths are some combination of the factors of M1. In the example for M = 19,

$$M1 = M - 1 = 2 * 3 * 3$$

and the only possible cycle lengths are 2, 3, 6, 9, and 18. Also, the table shows that only the exponents 6 (=18/3) and 9 (=18/2) must be checked to test if the number is a primitive root. This short cut saves a lot of time since the minimum number of powers to check is equal to the number of distinct factors of M1, which typically fall between 2 and 7.

TABLE A-1. PRIMITIVE ROOTS OF PRIME NUMBER 19

	1	2*	3	4	5*	6	7*	8	9	10	11*	12	13*	14	15	16	17*	18
2	2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

REFERENCES

1. Atkinson, A. C.: Tests of Pseudo-Random Numbers. Applied Statistics, Vol. 29, pp. 164-171.

2. Forsythe, G. E., et al.: Computer Methods for Mathematical Computations. Prentice Hall, 1977.

3. Knuth, D. E.: The Art of Computer Programming, Volume 2, Seminumerical Algorithms. Addison Welsey, 1969.

4. Marsaglia, G.: The Structure of Linear Congruential Sequences. In Applications of Number Theory to Numerical Analysis, S. K. Zaremba (ed.), Academic Press, 1972.

5. Ore, O.: Number Theory and Its History. McGraw-Hill, 1948.

6. Rheinfurth, M.: Marshall Space Flight Center Memorandum, ED01-34-82, April 1982.