TOWARD A UNIVERSAL RANDOM NUMBER GENERATOR

by

George Marsaglia
Supercomputer Computations Research Institute
and
Department of Statistics
The Florida State University
Tallahassee, FL 32306

and

Arif Zaman
Department of Statistics
The Florida State University
Tallahassee, FL 32306

FSU Statistics Report No. M 766

September 1987
Toward A Universal Random Number Generator

George Marsaglia and Arif Zaman

Supercomputer Computation Research Institute
and
Department of Statistics
The Florida State University
Tallahassee, Florida 32306

Abstract. This article describes an approach toward a random number generator that passes all of the stringent tests for randomness we have put to it, and that is able to produce exactly the same sequence of uniform random variables in a wide variety of computers, ranging from TRS80, Apple, Macintosh, Commodore, Kaypro, IBM PC, AT, PC and AT clones, VAX, Sun, IBM 380/370, 3000, Amdahl and CDC Cyber to 205 and ETA supercomputers.

Introduction. An essential property of a random number generator is that it produce a satisfactorily "random" sequence of numbers. Increasingly sophisticated uses have raised questions about the suitability of many of the commonly available generators—see, for example, reference [1]. Another shortcoming in many, indeed most, random number generators is they are not able to produce the same sequence of variables in a wide variety of computers. Such a requirement seems essential for an experimental science that lacks standardized equipment for verifying results.

We address these deficiencies here, suggesting a combination generator tailored particularly for reproducibility in all CPU's with at least 16 bit integer arithmetic. The random numbers themselves are reals with 24-bit fractions, uniform on [0,1). We provide a suggested Fortran implementation of this "universal" generator, together with suggested sample output with which one may verify that a particular computer produces exactly the same bit patterns as the computers enumerated above. The Fortran code is so straightforward that versions may be readily written for other languages; so far, students have written and confirmed results for Basic, Fortran, Pascal, Modula II and Ada versions.
A list of desirable properties for a random number generator might include:

1. **Randomness.** Provides a sequence of independent uniform random variables suitable for all reasonable applications. In particular, passes all the latest tests for randomness and independence.

2. **Long Period.** Able to produce, without repeating the initial sequence, all of the random variables for the huge samples that current computer speeds make possible.

3. **Efficiency.** Execution is rapid, with modest memory requirements.

4. **Repeatability.** Initial conditions (seed values) completely determine the resulting sequence of random variables.

5. **Portability.** Identical sequences of random variables may be produced in a wide variety of computers, for given starting values.

6. **Homogeneity.** All subsets of bits of the numbers must be random, from the most- to the least-significant bits.

**Choice of the Method.** Our choice of a generator that goes to meet these criteria is a combination generator, in which the principal, long period, component is based on the binary operation $x \cdot y$ on reals $x$ and $y$ defined by

$$x \cdot y = \begin{cases} x - y, & \text{if } x \geq y \\ x - y + 1, & \text{else} \end{cases}$$

We require a sequence of reals on $[0, 1)$: $U_1, U_2, U_3, \ldots$, each with a 24-bit fraction. We chose 24 bits because it is the most common fraction size for single-precision reals and because the operation $x \cdot y$ can be carried out exactly, with no loss of bits, in most computers—those with reals having fractions of 24 or more bits.

The basic component of our universal generator uses this operation to produce a lagged-Fibonacci sequence, designated by $F(r, s, e)$. This sequence of real numbers is defined by:

$$x_1, x_2, x_3, \ldots \quad \text{with} \quad x_n = x_{n-r} \cdot x_{n-s}$$

The lags $r$ and $s$ are chosen so that the sequence is satisfactorily random and has a very long period. If the initial, seed values, $x_1, x_2, \ldots, x_r$ are each 24-bit fractions, $x_i = I_i/2^{24}$, then the resulting sequence, generated by $x_n = x_{n-r} \cdot x_{n-s}$, will produce a sequence with period and structure identical to that of the corresponding sequence of integers

$$I_1, I_2, I_3, \ldots \quad \text{with} \quad I_n = I_{n-r} - I_{n-s} \mod 2^{24}.$$
For suitable choices of the lags \( r \) and \( s \) the period of the sequence is \((2^{24} - 1) \times 2^{r-1}\). The need to choose \( r \) large for long period and randomness must be balanced with the resulting memory costs: a table of the \( r \) most recent \( x \) values must be stored. We have chosen \( r = 27, s = 33 \). The resulting cost of \( 97 \) storage locations for the circular list needed to implement the generator seems reasonable. A few hundred memory locations more or less is no longer the problem it used to be. The period of the resulting generator is \((2^{24} - 1) \times 2^{66}\), about \(2^{120}\), which we boost to \(2^{144}\) by the other part of the combination generator, described below. Methods for establishing periods for \( F(r,s,\mod 2^k) \) generators are given in reference [2].

The Second Part of the Combination. We now turn to choice of a generator to combine with the \( F(97,33,s) \) chosen above. We are not content with that generator alone, even though it has an extremely long period and appears to be suitably random from the stringent tests we have applied to it. But it fails one of the tests: the Birthday-Spacings Test. This test goes as follows: let each of the generated values \( x_1, x_2, \ldots \) represent a “birthday” in a “year” of \(2^{24}\) days. Choose, say, \( m = 512 \) birthdays, and let \( x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(m)} \) be the first \( m \) birthdays sorted in increasing order. The spacings are defined by \( y_1 = x_{(1)}, y_2 = x_{(2)} - x_{(1)}, y_3 = x_{(3)} - x_{(2)}, \ldots, y_m = x_{(m)} - x_{(m-1)}, \) with \( y_{(1)} \leq y_{(2)} \leq \cdots \leq y_{(m)} \) being the ordered spacings. The test statistic \( J \) is the number of duplicate values in the spacings, i.e., the number of times \( y_{(i)} = y_{(i-1)} \) for \( i = 2, \ldots, m \). The resulting \( J \) should have a Poisson distribution with mean \( \lambda = m^{m/(4n)} = m^{m/2^{26}} \). Lagged-Fibonacci generators \( F(r,s,e) \) fail this test, unless the lag \( r \) is more than 500 or the binary operation \( e \) is, say, multiplication for odd integers mod \( 2^k \).

In order to get a generator that passes all the stringent tests we have applied, we have resorted to combining the \( F(97,33,*) \) generator with a second generator. Combining different generators has strong theoretical support; see [1]. Our choice of the second generator is a simple arithmetic sequence for the prime modulus \(2^{24} - 3 = 16777213\). For an initial integer \( I \), subsequent integers are \( I-k, I-2k, I-3k, \ldots \mod 16777216\). This may be implemented in 24-bit reals, again with no bits lost, by letting the initial value be, say \( c = 362436/1766216 \), then forming successive 24-bit reals by the operation \( c \times d \), defined as \( c \times d = \{ (\text{if } c \geq d \text{ then } c - d, \text{ else } c - d + 16777213/16777216) \} \). Here \( d \) is some convenient 24-bit rational, say \( d = 7654321/16777216 \). The resulting sequence has period \(2^{24} - 3\), and while it is far too regular for use alone, it serves, when combined by means of the \( * \) operation with the \( F(97,33,*) \) sequence, to provide a composite sequence that meets all of the criteria mentioned in the introduction, except for simplicity. All of the operations in the combination generator are simple, and the generation part is quite simple, but the setup procedure, setting the initial \( 97 \times \) values, is more complicated than the generating procedure. We now turn to details of implementation.
Implementation. We have two binary operations, each able to produce exact arithmetic on reals with 24-bit fractions:

\[ x \ast y = \begin{cases} 
  x - y, & \text{if } x \geq y \\
  x - y + 1, & \text{else}
\end{cases} \]

\[ c \circ d = \begin{cases} 
  c - d, & \text{if } c \geq d \\
  c - d + 16777213/16777216, & \text{else}
\end{cases} \]

We require computer instructions that will generate two sequences:

\[ x_1, x_2, x_3, \ldots, x_{97}, x_{98}, \ldots \quad \text{with} \quad x_n = x_{n-97} \ast x_{n-33} \]

\[ c_1, c_2, c_3, \ldots \quad \text{with} \quad c_n = c_{n-1} \circ (7854321/16777216) \]

then produce the combined sequence

\[ U_1, U_2, U_3, \ldots \quad \text{with} \quad U_n = x_n \ast c_n \]

The \( c \) sequence requires only one initial value, which we arbitrarily set to \( c_1 = 362436/16777216 \). The \( x \) sequence requires 97 initial, seed, values, each a real of the form \( I/16777216 \), with \( 0 \leq I \leq 16777215 \). The main problem in implementing the universal generator is in finding a suitable way to set the 97 initial values, a way that is both random and consistent from one computer to another. The \( F(97,33, - \text{ mod } 1) \) generator is quite robust, in that it gives good results even for bad initial values. Nonetheless, we feel that the initial table should itself be filled by means of a good generator, one that need not be fast because it is used only for the setup. Of course, we might ask that the user provide 97 seed values, each with an exact 24-bit fraction, but that seems too great a burden.

After considerable experimentation, we recommend the following procedure: assign values bit-by-bit to the initial table \( U(1), U(2), \ldots, U(97) \) with a sequence of bits \( b_1, b_2, b_3, \ldots \). Thus \( U(1) = .b_1b_2b_3 \ldots b_{24}, U(2) = .b_{25}b_{26} \ldots b_{48} \) and so on. The sequence of bits is generated by combining two different generators, each suitable for exact implementation in any computer: one a 3-lag Fibonacci generator, the other an ordinary congruential generator for modulus 169.

The two sequences that are combined to produce bits \( b_1, b_2, b_3, \ldots \) are:

\[ y_1, y_2, y_3, y_4, \ldots \quad \text{with} \quad y_n = y_{n-3} \times y_{n-2} \times y_{n-1} \text{ mod } 179. \]

\[ z_1, z_2, z_3, z_4, \ldots \quad \text{with} \quad z_n = 53z_{n-1} + 1 \text{ mod } 169. \]

Then \( b_i \) in the sequence of bits is formed as the sixth bit of the product \( y_i z_i \), using operations which may be carried out in most programming languages: \( b_i = \{ \text{if } y_i z_i \mod 64 < 32 \text{ then } 0, \text{ else } 1 \} \).

Choosing the small moduli 179 and 169 ensures that arithmetic will be exact in all computers, after which combining the two generators by multiplication and bit extraction stays within the range of 16-bit integer arithmetic. The result is a sequence of bits that passes extensive tests for randomness, and thus seems well suited for initialising a universal generator.
The user's burden is reduced to providing three seed values for the 3-lag Fibonacci sequence \( y_n \), and one seed value for the congruential sequence \( z_n \).

For Fortran implementations of the universal generator, we recommend that a table \( U(1), \ldots, U(97) \) be shared, in (labelled) COMMON, with a setup routine, say RSTART(I,J,K,L), and the function subprogram, UNI(), that returns the required uniform variate. An alternative approach is to have a single subprogram that includes an entry for the setup procedure, but not all Fortran compilers allow multiple entries to a subprogram. The initial, seed values for the setup are \( I, J, K, \) and \( L \). Here \( I, J, K \) must be in the range 1 to 178, and not all 1, while \( L \) may be any integer from 0 to 168. More simply, one could simply pick all arguments between 1 and 168. If (positive) integer values are assigned to \( I, J, K, L \) outside the specified ranges, the generator will still be satisfactory, but may not produce exactly the same bit patterns in different computers, because of uncertainties when integer operations involve more than 15 bits.

To use the generator, one must first CALL RSTART(I,J,K,L) to set up the table in labelled common. Subsequent uniform random variables are obtained by using UNI() in any arithmetic expression—e.g., for example, in \( X=\text{UNI()} \) or \( Y=2.\text{UNI()}-\text{ALOG(UNI())} \), etc.

FORTRAN subprograms for initialising and calling UNI

```
FUNCTION UNI()
C *** FIRST CALL RSTART(I,J,K,L)
C *** WITH I,J,K,L INTEGERS
C *** FROM 1...168 NOT ALL 1
REAL U(97)
COMMON /SET1/ U,C,CD,CM
DATA I,J/97,33/
UNI=U(I)-U(J)
IF(U(I).LT.0.) UNI=UNI+1.
I=I+1
IF(I.EQ.0) I=97
J=J-1
IF(J.EQ.0) J=97
C=G-CD
IF(C.LT.0.) C=G+CM
UNI=UNI-C
IF(U(I).LT.0.) UNI=UNI+1.
RETURN
END

SUBROUTINE RSTART(I,J,K,L)
REAL U(97)
COMMON /SET1/ U,C,CD,CM
DO 2 II=1,97
S=S+1
2 DO 3 JJ=1,4
M=NOD(NOD(I+J,179)*K,179)
I=J
J=K
K=M
L=NOD(53*L+1,169)
IF(NOD(L*M,64).GE.32) S=S+T
DO 4 JJ=1,3
3 T=T+S
4...
```

The required initialisation, CALL RSTART(I,J,K,L), can be circumvented by using a BLOCK DATA subprogram to set the the values in the COMMON /SET1/. The values returned by CALL RSTART(12,34,56,78), used in the verification
program listed below, seem a good choice for standardising initial values. A
listing for such a subprogram is not provided because there is no standard way
to enter exact floating point data on different machines. In other languages,
analogous devices for initialisation can be used. In this way, RSTART can be
used to specify seed values, but need not be called for those who wish to use
the standard sequence of UNI's.

Verifying the Universality. We now suggest a short Fortran program for
verifying that the universal generator will produce exactly the same 24-bit reals
that other computers produce. Conversion to an equivalent Basic, Pascal or
other program should be transparent. Assume then that you have implemented
the UNI routine with its RSTART setup procedure in your computer. Running
this short program or an equivalent:

```
CALL RSTART(12,34,56,78)
DO 2 I=1,20000
  2 X=UNI()
  PRINT 3,(4096.*UNI(),I=1,6)
FORMAT(3F12.1)
END
```

should give the following output:

```
6533892.0 14220222.0 7275067.0
6172232.0 6354498.0 10633180.0
```

If it does, you will almost certainly have a universal random number gen-
erator that passes all the standard tests, and all the latest——more stringent——
tests for randomness, has an incredibly long period, about $2^{144}$, and, for given
RSTART values I,J,K,L, produces the same sequence of 24-bit reals as do almost
all other commonly-used computers.

Good Luck.

References

Address, Computer Science and Statistics: Sixteenth Symposium

Random Number Sequences”, Linear Algebra and its Applications, 67,