Toward A Universal Random Number Generator

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

Wai Wan Tsang
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong

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, including TRS80, Apple, Macintosh, Commodore, Kaypro, IBM PC, AT, PC and AT clones, Sun, Vax, IBM 360/370, 3090, Amdahl, CDC Cyber and even 205 and ETA supercomputers.

1 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 in the abstract. The Fortran code is so straightforward that versions may be readily written for other languages; so far, correspondents have written or confirmed results for Basic, Fortran, Pascal, C, 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.

2 Choice of the Method

We seek a generator that has all of these desirable properties. (All? Well, almost all; the generator we propose falls short on *efficiency*, for it is slower than some of the standard, simple, machine-dependent generators. But all of the standard generators fail one or more of the stringent tests for randomness. See [1].)

Our choice is a combination generator. It combines two different generators. The principal component of the two has a very long period, about 10^{36}. It is a lagged-Fibonacci generator based on the binary operation $x \bullet y$ on reals x and y defined by

$$x \bullet y = \{\text{if } x \geq y \text{ then } x - y, \text{ else } x - y + 1\}.$$

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

This choice allows us to use a lagged-Fibonacci generator, designated F(r,s,•), as the basic component of our universal generator. It provides a sequence of reals by means of the operation $x \bullet y$:

$$x_1, x_2, x_3, \ldots \qquad \text{with } x_n = x_{n-r} \bullet x_{n-s}.$$

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 $x_n = x_{n-r} \bullet x_{n-s}$, will produce a sequence with period and structure identical to that of the corresponding sequence of integers

$$I_1, I_2, I_3, \ldots \qquad \text{with } I_n = I_{n-r} - I_{n-s} \bmod 2^{24}.$$

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 $r = 97, s = 33$. 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 $(2^{24} - 1) \times 2^{96}$, about 2^{120}, which we boost to 2^{144} by the other part of the combination generator, described below. Methods for establishing periods for lagged-Fibonacci generators are given in reference [2].

3 The Second Part of the Combination

We now turn to choice of a generator to combine with the F(97,33,•) 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. A typical version of this test goes as follows: let each of the generated values $x_1, x_2, \ldots$ represent a "birthday" in a "year" of, say, 2^{24} days. Choose, say, $m = 512$ birthdays, $x_1, x_2, \ldots, x_m$. Sort these to get $x_{(1)} \le x_{(2)} \le \cdots \le x_{(m)}$. Form spacings $y_1 = x_{(1)}, y_2 = x_{(2)} - x_{(1)}, y_3 = x_{(3)} - x_{(2)}, \ldots, y_m = x_{(m)} - x_{(m-1)}$. Sort the spacings, getting $y_{(1)} \le y_{(2)} \le \cdots \le y_{(m)}$. The test statistic is J, the number of duplicate values in the sorted spacings. i.e., initialize J← 0 then for $i = 2$ to m, put $J \leftarrow J + 1$ if $y_{(i)} = y_{(i-1)}$. The resulting J should have a Poisson distribution with mean $\lambda = m^3/(4n) = m^3/2^{26}$.

Lagged-Fibonacci generators F(r,s,•) fail this test, unless the lag r is more than 500 or the binary operation • is multiplication for odd integers mod 2^k. The count J, the number of duplicate spacings, is only asymptotically Poison distributed, requiring that n, the length of the year, be large. Applications of the Birthday Spacings Test typically choose n to be 100,000 or more—for example, using the leftmost 18 or more bits of the random number to provide a "birthday".

Detailed discussion of the test and test results will appear elsewhere, but here are results of a typical test on four different generators: two lagged-Fibonacci generators using subtraction, a lagged-Fibonacci generator using multiplication on odd integers, and a popular congruential generator, $x_n = 69069x_{n-1}$, all for modulus 2^{32}. The leftmost 25 bits are used to

choose one of 512 birthdays. Thus $n = 2^{25}$ and $m = 2^9$, so J should be Poisson distributed with $\lambda = m^3/(4n) = 1$. Of the four, only the F(97,33,•) and the congruential generator pass. The two lagged-Fibonacci generators using subtraction fail the test. Their duplicate-spacing counts are far from Poisson distributed, and remain so, whatever the choice of seed values, (and for a wide variety of choices of n, m and lags r, s as well).

A Birthday-Spacings Test for Four Generators

duplicate spacings	number expected	$F(97,33,-)$ observed	$F(55,24,-)$ observed	$F(97,33,*)$ observed	Congruential observed
0	36.79	41	29	41	36
1	36.79	16	14	33	37
2	18.39	18	34	20	20
≥ 3	8.03	25	23	6	7
Chi-square for	3 d.f.	48.1	56.91	1.53	.29
Probability of	better fit	1.0000	1.0000	.432	.33

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 $c = 362436/17666216$, then forming successive 24-bit reals by the operation $c \circ d$, defined as

$$c \circ d = \{\text{if } c \geq d \text{ then } c - d, \text{ else } c - d + 16777213/16777216\}.$$

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 efficiency. All of the operations in the combination generator are simple and efficient, and the generation part is quite simple, but the setup procedure, setting the initial 97 x values, is more complicated than the generating procedure. We now turn to details of implementation.

4 Implementation

We have two binary operations, each able to produce exact arithmetic on reals with 24-bit fractions:

$$\begin{aligned} x \bullet y &= \{\text{if } x \geq y \text{ then } x - y, \text{ else } x - y + 1\} \\ c \circ d &= \{\text{if } c \geq d \text{ then } c - d, \text{ else } c - d + 16777213/16777216\}. \end{aligned}$$

We require computer instructions that will generate two sequences:

$$\begin{aligned} &x_1, x_2, x_3, \ldots, x_{97}, x_{98}, \ldots && \text{with} && x_n = x_{n-97} \bullet x_{n-33}, \\ &c_1, c_2, c_3, \ldots && \text{with} && c_n = c_{n-1} \circ (7654321/16777216), \end{aligned}$$

then produce the combined sequence

$$U_1, U_2, U_3, \ldots \quad \text{with} \quad U_n = x_n \bullet c_n.$$

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,-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 random sequence of bits $b_1, b_2, b_3, \ldots$. Thus $U(1) = .b_1 b_2 \ldots b_{24}$, $U(2) = .b_{25} b_{26} \ldots b_{48}$ 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 using multiplication, 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:

$$\begin{aligned} &y_1, y_2, y_3, y_4, \ldots && \text{with} && y_n = y_{n-3} \times y_{n-2} \times y_{n-1} \bmod 179. \\ &z_1, z_2, z_3, z_4, \ldots && \text{with} && z_n = 53 z_{n-1} - 1 \bmod 169. \end{aligned}$$

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: $b_i = \{\text{if } y_i z_i \bmod 64 < 32 \text{ then } 0, \text{ else } 1\}$.

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 initializing a universal generator.

The user's burden is reduced to providing three seed values for the 3-lag Fibonacci sequence, and one seed value for the congruential sequence $z_n = 53z_{n-1} + 1 \bmod 169$. For Fortran implementations of the universal generator, we recommend that a table U(1),...,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 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. 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, then get subsequent uniform random variables by using UNI() in an expression—as, for example, in X=UNI() or Y=2.*UNI()-ALOG(UNI()), etc.

FORTRAN SUBPROGRAMS FOR INITIALIZING AND CALLING UNI

```
      SUBROUTINE RSTART(I,J,K,L)
      REAL U(97)
      COMMON /SET1/ U,C,CD,CM,IP,JP
      DO 2 II=1,97
      S=0.
      T=.5
         DO 3 JJ=1,24
         M=MOD(MOD(I*J,179)*K,179)
         I=J
         J=K
         K=M
         L=MOD(53*L+1,169)
         IF(MOD(L*M,64).GE.32) S=S+T
    3    T=.5*T
    2 U(II)=S
      C=362436./16777216.
      CD=7654321./16777216.
      CM=16777213./16777216.
      IP=97
      JP=33
      RETURN
      END
```

```
      FUNCTION UNI()
C***  FIRST CALL RSTART(I,J,K,L)
C***  WITH I,J,K,L INTEGERS
C***  FROM 1 TO 168, NOT ALL 1
C***  NOTE: RSTART CHANGES I,J,K,L
C***  SO BE CAREFUL IF YOU REUSE
C***  THEM IN THE CALLING PROGRAM.
      REAL U(97)
      COMMON /SET1/ U,C,CD,CM,IP,JP
      UNI=U(IP)-U(JP)
      IF(UNI.LT.0.) UNI=UNI+1.
      U(IP)=UNI
      IP=IP-1
          IF(IP.EQ.0) IP=97
        JP=JP-1
      IF(JP.EQ.0) JP=97
      C=C-CD
      IF(C.LT.0.) C=C+CM
      UNI=UNI-C
      IF(UNI.LT.0.) UNI=UNI+1.
      RETURN
      END
```

5 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, but those who wish to may get the setup, generating and verification programs for various languages by writing to the authors.

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 6 II=1,20005
      X=UNI()
    6 IF(II.gt.20000) print 21,(MOD(INT(X*16.**I),16),I=1,7)
   21 FORMAT(6X,7I3)
      END
```

should produce this output:

```
  6  3 11  3  0  4  0
 13  8 15 11 11 14  0
  6 15  0  2  3 11  0
  5 14  2 14  4  8  0
  7 15  7 10 12  2  0
```

If it does, you will almost certainly have a universal random number generator 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

[1] George Marsaglia, "A current view of random number generators", Keynote Address, Computer Science and Statistics: Sixteenth Symposium on the Interface, Atlanta, March 1984. In *Proceedings of the Symposium*, Elsevier, 1986.

[2] George Marsaglia and Liang-Huei Tsay, "Matrices and the Structure of Random Number Sequences", *Linear Algebra and its Applications*, 67, 147-156, 1985.