

Xorshift RNGs

George Marsaglia *
The Florida State University

Abstract

Description of a class of simple, extremely fast random number generators (RNGs) with periods $2^k - 1$ for $k = 32, 64, 96, 128, 160, 192$. These RNGs seem to pass tests of randomness very well.

1 Introduction

A **xorshift** random number generator (xorshift RNG) produces a sequence of $2^{32} - 1$ integers x , or a sequence of $2^{64} - 1$ pairs x, y , or a sequence of $2^{96} - 1$ triples x, y, z , etc., by means of repeated use of a simple computer construction: exclusive-or (xor) a computer word with a shifted version of itself. In C, the basic operation is $y \wedge (y \ll a)$ for shifts left, $y \wedge (y \gg a)$ for shifts right. (In Fortran, a single form, `ieor(y, ishft(y, a))` will produce the desired result, with negative a for shifts right, nonnegative for shifts left.)

Combining such xorshift operations for various shifts and arguments provides extremely fast and simple RNGs that seem to do very well on tests of randomness. To give an idea of the power and effectiveness of xorshift operations, here is the essential part of a C procedure that, with only three xorshift operations per call, will provide $2^{128} - 1$ random 32-bit integers, given four random seeds x, y, z, w :

```
tmp=(x^(x<<15)); x=y; y=z; z=w; return w=(w^(w>>21))^(tmp^(tmp>>4));
```

Such a procedure is very fast, typically over 200 million/second, and the resulting random integers pass all the tests of randomness that have been applied to them, particularly the "tough" tests in [3] and the new version of the Diehard Battery [2].

Background theory for establishing the periods and choices that provide a variety of xorshift RNGs with periods up to 2^{160} are given here. Longer periods are available, for example, from multiply-with-carry RNGs, but they use integer multiplication and require keeping a (sometimes large) table of the most recently generated values. More detailed comparisons are given in the final, summary section.

2 Theory

A mathematical model for most RNGs can be put in the following form: We have a **seed set** \mathcal{Z} made up of m -tuples (x_1, x_2, \dots, x_m) , and a one-to-one function $f()$ on \mathcal{Z} . Most commonly, \mathcal{Z} is just a set of integers, but for better RNGs, it may be a set of pairs, triples, etc. If z is chosen uniformly and randomly from \mathcal{Z} , then the output of the RNG is the sequence $f(z), f^2(z), f^3(z), \dots$, where $f^2(z)$ means $f(f(z))$, etc. Because f is 1-1 over \mathcal{Z} , the random variable $f(z)$ is itself uniform over \mathcal{Z} , as is $f^2(z)$; indeed, each element of the sequence $f(z), f^2(z), \dots$ is uniformly distributed over the seed set \mathcal{Z} , but they are not independent.

For xorshift RNGs, the seed set \mathcal{Z} is the set of $1 \times n$ binary vectors, $\beta = (b_1, b_2, \dots, b_n)$, excluding the zero vector. Usually, n will be 32, 64, 96, etc., so that its elements can be made up by adjoining 32-bit computer words. The elements of the vectors β are in the field $\{0, 1\}$, so that addition of binary vectors can be implemented by xor'ing the constituent 32-bit parts. For our xorshift RNG, we need an invertible function over \mathcal{Z} , and for that we use a linear transformation over the binary vector space, characterized by a nonsingular $n \times n$ binary matrix T . If β is a uniform random choice, (the **seed**), from \mathcal{Z} , then each member of the sequence $\beta T, \beta T^2, \beta T^3, \dots$ is also uniformly distributed

*While the author is now Professor Emeritus, portions of the research for this article were done under by grants from The National Science Foundation.

over \mathbb{Z} , so we have a sequence of ID, Identically Distributed, uniform elements from \mathbb{Z} , but they are not IID, that is, Independent Identically Distributed. But it turns out here, as for many RNGs, functions of the ID elements often have distributions very close to those of the same functions of elements of an IID sequence. That is the remarkable property of certain choices of functions $f()$ over seed sets \mathbb{Z} that justifies their usefulness in computers for the past fifty years.

2.1 Matrices T that generate all non-null binary vectors

First given in [1], here is the main result:

Theorem *In order that a nonsingular $n \times n$ binary matrix T produce all possible non-null $1 \times n$ binary vectors in the sequence $\beta, \beta T, \beta T^2, \dots$ for every non-null initial $1 \times n$ binary vector β , it is necessary and sufficient that, in the group of nonsingular $n \times n$ binary matrices, the order of T is $2^n - 1$*

Proof: First, the necessity: If the period of $\beta T, \beta T^2, \dots$ is $k = 2^n - 1$ then $\beta T^k = \beta$ for every $1 \times n$ binary vector β , so the null space of the matrix $T^k + I$ is the whole space, and thus $T^k + I$ must be the zero matrix, that is, $T^k = I$. If $T^j = I$ for some $j < k$, then the period of $\beta T, \beta T^2, \dots$ would be less than $2^n - 1$.

Then the sufficiency: If the order of T is $k = 2^n - 1$, then the matrices T, T^2, T^3, \dots, T^k are nonsingular and distinct, and through the characteristic polynomial of T and Euclid's algorithm, each of them can be represented as a polynomial in T of degree $< n$. Since there are $k = 2^n - 1$ non-null polynomials in T of degree $< n$, they must be, in some order, the distinct nonsingular matrices T, T^2, \dots, T^k . In particular, if a polynomial in T is a singular matrix, then it must reduce, through T 's characteristic polynomial, to the zero matrix. It follows that the period of $\beta T, \beta T^2, \dots$ must $k = 2^n - 1$, because $\beta T^j = \beta$ for some non-null β and $j < k$ would mean that $T^j + I$ is singular.

3 Application to Xorshift RNGs

For a binary vector y , the operation yT in a computer is likely to be expensive unless the matrix T has a special form. If L is the $n \times n$ binary matrix that effects a left shift of one position on a binary vector y , that is, L is all 0's except for 1's on the principal subdiagonal, then, with $T = I + L^a$, the xorshift operation in C, $y^{\wedge}(y << a)$ produces the linear transformation yT : add, mod 2, the binary vector y to a left-shift- a version of itself. Similarly, if R is the right-shift-1 matrix, (the transpose of L), then the xorshift operation $y^{\wedge}(y >> b)$ may be used to form yT , with $T = I + R^b$.

The $n \times n$ matrices $I + L^a$ and $I + R^b$ are nonsingular, since, for example, $L^n = 0$ and thus the finite series $I + L^a + L^{2a} + L^{3a} + \dots$ is the inverse of $(I + L^a)$. So an obvious candidate for a matrix that has order $2^n - 1$ is $T = (I + L^a)(I + R^b)$ or $T = (I + R^b)(I + L^a)$. Unfortunately, when n is 32 or 64, no choices for a and b will provide such a T with the required order. (A preliminary test on candidates is to square T n times. If the result is not T , then T cannot have order $2^n - 1$. By representing the rows of a binary matrix as a computer word, then xoring all of the rows for which a (left) multiplying binary vector has 1's, very fast binary matrix products can be evaluated with simple C or Fortran programs, and it only takes a few minutes to check every possible a, b in $T = (I + L^a)(I + R^b)$.)

However, there are many choices for a, b, c for which the matrices $T = (I + L^a)(I + R^b)(I + L^c)$ have order $2^n - 1$, when $n = 32$ or $n = 64$. There are 81 triples (a, b, c) , $a < c$, for which the 32×32 binary matrix $T = (I + L^a)(I + R^b)(I + R^c)$ has order $2^{32} - 1$:

1, 3,10	1, 5,16	1, 5,19	1, 9,29	1,11, 6	1,11,16	1,19, 3	1,21,20	1,27,27
2, 5,15	2, 5,21	2, 7, 7	2, 7, 9	2, 7,25	2, 9,15	2,15,17	2,15,25	2,21, 9
3, 1,14	3, 3,26	3, 3,28	3, 3,29	3, 5,20	3, 5,22	3, 5,25	3, 7,29	3,13, 7
3,23,25	3,25,24	3,27,11	4, 3,17	4, 3,27	4, 5,15	5, 3,21	5, 7,22	5, 9,7
5, 9,28	5, 9,31	5,13, 6	5,15,17	5,17,13	5,21,12	5,27, 8	5,27,21	5,27,25
5,27,28	6, 1,11	6, 3,17	6,17, 9	6,21, 7	6,21,13	7, 1, 9	7, 1,18	7, 1,25
7,13,25	7,17,21	7,25,12	7,25,20	8, 7,23	8,9,23	9, 5,1	9, 5,25	9,11,19
9,21,16	10, 9,21	10, 9,25	11, 7,12	11, 7,16	11,17,13	11,21,13	12, 9,23	13, 3,17
13, 3,27	13, 5,19	13,17,15	14, 1,15	14,13,15	15, 1,29	17,15,20	17,15,23	17,15,26

Of those 81 triples with $a < c$, the triple (c, b, a) also provides a full period T , making 162 in all. In addition, another 162 full-period T 's arise from taking transposes: use triples (a, b, c) to form $T = (I + R^c)(I + L^b)(I + R^a)$, for

a total of 324. Then, finally, for each of the 324 T 's of the form $(I + L^a)(I + R^b)(I + L^c)$ or $(I + R^a)(I + L^b)(I + R^c)$, the corresponding matrices $(I + L^a)(I + L^c)(I + R^b)$, and $(I + R^a)(I + R^c)(I + L^b)$, also turn out to have period $2^{32} - 1$, making a total of 648 choices.

In summary, for each of the above 81 triples a, b, c with $a < c$, any one of these eight lines of C can serve as the basis for a 32-bit xorshift RNG with period $2^{32} - 1$:

```

y^=y<<a; y^=y>>b; y^=y<<c;
y^=y<<c; y^=y>>b; y^=y<<a;
y^=y>>a; y^=y<<b; y^=y>>c;
y^=y>>c; y^=y<<b; y^=y>>a;
y^=y<<a; y^=y<<c; y^=y>>b;
y^=y<<c; y^=y<<a; y^=y>>b;
y^=y>>a; y^=y>>c; y^=y<<b;
y^=y>>c; y^=y>>a; y^=y<<b;

```

For 64-bit integers, the following 275 triples provide 64×64 matrices $T = (I + L^a)(I + R^b)(I + L^c)$ whose order is $2^{64} - 1$:

1, 1,54	1, 1,55	1, 3,45	1, 7, 9	1, 7,44	1, 7,46	1, 9,50	1,11,35	1,11,50
1,13,45	1,15, 4	1,15,63	1,19, 6	1,19,16	1,23,14	1,23,29	1,29,34	1,35, 5
1,35,11	1,35,34	1,45,37	1,51,13	1,53, 3	1,59,14	2,13,23	2,31,51	2,31,53
2,43,27	2,47,49	3, 1,11	3, 5,21	3,13,59	3,21,31	3,25,20	3,25,31	3,25,56
3,29,40	3,29,47	3,29,49	3,35,14	3,37,17	3,43, 4	3,43, 6	3,43,11	3,51,16
3,53, 7	3,61,17	3,61,26	4, 7,19	4, 9,13	4,15,51	4,15,53	4,29,45	4,29,49
4,31,33	4,35,15	4,35,21	4,37,11	4,37,21	4,41,19	4,41,45	4,43,21	4,43,31
4,53, 7	5, 9,23	5,11,54	5,15,27	5,17,11	5,23,36	5,33,29	5,41,20	5,45,16
5,47,23	5,53,20	5,59,33	5,59,35	5,59,63	6, 1,17	6, 3,49	6,17,47	6,23,27
6,27, 7	6,43,21	6,49,29	6,55,17	7, 5,41	7, 5,47	7, 5,55	7, 7,20	7, 9,38
7,11,10	7,11,35	7,13,58	7,19,17	7,19,54	7,23, 8	7,25,58	7,27,59	7,33, 8
7,41,40	7,43,28	7,51,24	7,57,12	8, 5,59	8, 9,25	8,13,25	8,13,61	8,15,21
8,25,59	8,29,19	8,31,17	8,37,21	8,51,21	9, 1,27	9, 5,36	9, 5,43	9, 7,18
9,19,18	9,21,11	9,21,20	9,21,40	9,23,57	9,27,10	9,29,12	9,29,37	9,37,31
9,41,45	10, 7,33	10,27,59	10,53,13	11, 5,32	11, 5,34	11, 5,43	11, 5,45	11, 9,14
11, 9,34	11,13,40	11,15,37	11,23,42	11,23,56	11,25,48	11,27,26	11,29,14	11,31,18
11,53,23	12, 1,31	12, 3,13	12, 3,49	12, 7,13	12,11,47	12,25,27	12,39,49	12,43,19
13, 3,40	13, 3,53	13, 7,17	13, 9,15	13, 9,50	13,13,19	13,17,43	13,19,28	13,19,47
13,21,18	13,21,49	13,29,35	13,35,30	13,35,38	13,47,23	13,51,21	14,13,17	14,15,19
14,23,33	14,31,45	14,47,15	15, 1,19	15, 5,37	15,13,28	15,13,52	15,17,27	15,19,63
15,21,46	15,23,23	15,45,17	15,47,16	15,49,26	16, 5,17	16, 7,39	16,11,19	16,11,27
16,13,55	16,21,35	16,25,43	16,27,53	16,47,17	17,15,58	17,23,29	17,23,51	17,23,52
17,27,22	17,45,22	17,47,28	17,47,29	17,47,54	18, 1,25	18, 3,43	18,19,19	18,25,21
18,41,23	19, 7,36	19, 7,55	19,13,37	19,15,46	19,21,52	19,25,20	19,41,21	19,43,27
20, 1,31	20, 5,29	21, 1,27	21, 9,29	21,13,52	21,15,28	21,15,29	21,17,24	21,17,30
21,17,48	21,21,32	21,21,34	21,21,37	21,21,38	21,21,40	21,21,41	21,21,43	21,41,23
22, 3,39	23, 9,38	23, 9,48	23, 9,57	23,13,38	23,13,58	23,13,61	23,17,25	23,17,54
23,17,56	23,17,62	23,41,34	23,41,51	24, 9,35	24,11,29	24,25,25	24,31,35	25, 7,46
25, 7,49	25, 9,39	25,11,57	25,13,29	25,13,39	25,13,62	25,15,47	25,21,44	25,27,27
25,27,53	25,33,36	25,39,54	28, 9,55	28,11,53	29,27,37	31, 1,51	31,25,37	31,27,35
33,31,43	33,31,55	43,21,46	49,15,61	55, 9,56				

As with the 32-bit case, a selection of any one of the 275 a, b, c choices for 64-bit sequences, and any one of the above eight lines of C code, will provide, for 64-bit words, a xorshift RNG with period $2^{64} - 1$, for a total of $8 \times 275 = 2200$ choices.

Here is a basic 32-bit xorshift C procedure that takes a 32-bit seed value y :

```

unsigned long xor(){
static unsigned long y=2463534242;
y^=(y<<13); y=(y>>17); return (y^=(y<<5)); }

```

It uses one of my favorite choices, $[a, b, c] = [13, 17, 5]$, and will pass almost all tests of randomness, except the binary rank test in Diehard [2]. (A long period xorshift RNG necessarily uses a nonsingular matrix transformation, so every successive n vectors must be linearly independent, while truly random binary vectors will be linearly independent only some 30% of the time.) Although I have only tested a few of them, any one of the 648 choices above is likely to provide a very fast, simple, high quality RNG.

For C compilers that have 64-bit integers, the following will provide an excellent period $2^{64}-1$ RNG, given a 64-bit seed x :

```

unsigned long long xor64(){
static unsigned long long x=88172645463325252LL;
x^=(x<<13); x^=(x>>7); return (x^=(x<<17)); }

```

but any of the above 2200 choices is likely to do as well.

3.1 Binary vector spaces of dimension $n = 96, 128, 160 \dots$

While it is convenient to use a 32-bit computer word to represent an element of a vector space of dimension 32, or dimension 64 for compilers that allow 64-bit integers, to get longer xorshift periods we need methods for representing elements of vector spaces of higher dimensions. A good way to do this is to allow, say, 1×96 vectors made up of 32-bit components (x, y, z) or 1×128 vectors with 32-bit components (x, y, z, w) , etc. We are then faced with the problem of choosing matrices T that define linear transformations over such vector spaces.

A natural choice is to make T a companion matrix in block form—that is, for $n = 64, 96, 128$,

$$T = \begin{pmatrix} 0 & A \\ I & B \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & A \\ I & 0 & C \\ 0 & I & B \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 & A \\ I & 0 & 0 & C \\ 0 & I & 0 & D \\ 0 & 0 & I & B \end{pmatrix}.$$

Then, for example, $(x, y, z)T = (y, z, xA + yC + zB)$, and we can seek 32×32 matrices A, B, C so the 32-bit operations xA, yC, zB are easy and T has order $2^{96}-1$ in the group of 96×96 nonsingular binary matrices. (I put the last column of blocks as A, B or A, C, B or A, C, D, B because it turns out that there are full period choices, $2^{64}-1, 2^{96}-1, 2^{128}-1$, by just choosing $A = (I + L^a)(I + R^b)$ and $B = (I + R^c)$, the other blocks all the zero matrix.) Thus, for suitable choices of the triple $[a, b, c]$, these matrices all have the required orders, respectively, $2^{64}-1, 2^{96}-1, 2^{128}-1$:

$$\begin{pmatrix} 0 & (I + L^a)(I + R^b) \\ I & (I + R^c) \end{pmatrix}, \begin{pmatrix} 0 & 0 & (I + L^a)(I + R^b) \\ I & 0 & 0 \\ 0 & I & (I + R^c) \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & (I + L^a)(I + R^b) \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & (I + R^c) \end{pmatrix}.$$

Some (four) choices for the triple $[a, b, c]$ are: For $n = 64$, $[10, 13, 10], [8, 9, 22], [2, 7, 3], [23, 3, 24]$; for $n = 96$, $[10, 5, 26], [13, 19, 3], [1, 17, 2], [10, 1, 26]$; for $n = 128$, $[5, 14, 1], [15, 4, 21], [23, 24, 3], [5, 12, 29]$. In each case, the order of the block matrix T is $2^n - 1$, and the essences of C procedures for generating sequences of the required length are, with x, y, z, w static unsigned longs and a temporary t :

```

t=(x^(x<<a)); x=y; return y=(y^(y>>c))^(t^(t>>b));    (period 264-1).
t=(x^(x<<a)); x=y; y=z; return z=(z^(z>>c))^(t^(t>>b));    (period 296-1).
t=(x^(x<<a)); x=y; y=z; z=w; return w=(w^(w>>c))^(t^(t>>b));    (period 2128-1).

```

These examples provide xorshift RNGs with period $2^{160}-1$, using the same promotion scheme, but requiring (static unsigned long) seeds x, y, z, w, v :

```

t=(x^(x>>a)); x=y; y=z; z=w; w=v; return v=(v^(v>>c))^(t^(t>>b));
with choice of parameters [a,b,c]=[2,1,4],[7,13,6],[1,1,20].

```

Long periods may also be found by choosing the last column of the block companion matrix to be $I + L^a$, $I + R^b$, $I + L^c$, $I + R^d$ as needed. Examples of the C generating code:

```
t=(x^(x<<3))^(y^(y>>19))^(z^(z<<6));x=y;y=z;return(z=t); period 296-1.
t=(x^(x<<20))^(y^(y>>11))^(z^(z<<27))^(w^(w>>6));x=y;y=z;z=w;return(w=t);
period 2128-1.
```

All of the above procedures return a 32-bit integer, although the procedures create sequences of pairs (x, y) , or triples (x, y, z) or quadruples (z, y, z, w) of the maximal period, $2^n - 1$, over binary vector spaces of dimension $n = 64, 96, 128$. For all of the above choices of parameters, the resulting xorshift RNGs pass all the tests in Diehard, and are exceptionally fast, all taking 4-6 nanoseconds, or about 200 million random numbers per second. They are so fast that a dominant part is the linkage: saving before—and restoring after—the registers used in the subroutine call.

Note that in the above, the last value: y in (x, y) , z in (x, y, z) , w in (x, y, z, w) , is the returned value, but it need not be. If it is merely assigned, the full sequence of pairs, triples, quadruples will still be generated, but any function of those values could be returned. This provides a variety of possibilities for interesting RNGs. For example, merely returning $69069 * x$ would provide a sort of congruential RNG with period $2^n - 1$, while a period of $2^{32}(2^n - 1)$ will result from combining (+ or xor) the regular xorshift with what I call a Weyl sequence: $d+=362437$; period 2^{32} (with any odd constant replacing 362437). Here is an example:

```
unsigned long xorwow(){
static unsigned long x=123456789,y=362436069,z=521288629,
w=88675123,v=5783321,d=6615241;

unsigned long t;
t=(x^(x>>2)); x=y; y=z; z=w; w=v; v=(v^(v<<4))^(t^(t<<1)); return (d+=362437)+v;
}
```

Simple and very fast (125 million/sec), the elements in its cycle of $2^{192} - 2^{32}$ easily pass all the tests in Diehard.

4 Summary

A variety of simple and extremely fast RNGs can be developed by combining xorshift operations in different ways, yet in spite of their simplicity, the random numbers they produce do extremely well in tests of randomness. For the hundreds of choices of $[a, b, c]$ given above, the simplest ones use the basic C operation $x^=(x<<a); x^=(x>>b); (x^=(x<<c));$ to produce periods $2^{32} - 1$ for 32-bit words, or period $2^{64} - 1$ for 64-bit words. They are suitable by themselves or for use in combination with other methods.

A period of 2^{32} is considered low by modern standards. With very little additional computer time, xorshift operations can be used to provide sequences of pairs (x, y) of period $2^{64} - 1$, triples (x, y, z) of period $2^{96} - 1$, or quadruples (x, y, z, w) of period $2^{128} - 1$ or quintuples (x, y, z, w, v) , period $2^{192} - 1$. Extensions to higher k -tuples are feasible, but the general procedure: compute the new value as a function of the previous k values, then promote: $x=y; y=z; z=w;$ etc. becomes less efficient than methods that keep a circular table of the k previously generated values. With such tables, the multiply-with-carry and complimentary-multiply-with-carry methods can provide periods as large as 2^{131102} . I have described various versions through newsgroup postings, and a general description is in [4].

Suppose we compare a xorshift RNG, period $2^{128} - 1$, with a multiply-with-carry RNG of comparable period. First, the xorshift:

```
unsigned long xor128(){
static unsigned long x=123456789,y=362436069,z=521288629,w=88675123;
unsigned long t;
t=(x^(x<<11));x=y;y=z;z=w; return( w=(w^(w>>19))^(t^(t>>8)) );
}
```

then the multiply-with-carry (MWC):

```
unsigned long mwc(){
static unsigned long x=123456789,y=362436069,z=77465321,c=13579;
```

```

unsigned long long t;
t=916905990LL*x+c; x=y; y=z; c=(t>>32); return z=(t&0xffffffff);
}

```

The MWC RNG generates a sequence $x_n = ax_{n-3} + \text{carry} \bmod b$, with $a = 916905990$, $b = 2^{32}$. It keeps the three previous values, x, y, z and the current carry c , forms $t=ax+c$ in 64 bits, then promotes, $x=y, y=z$. Then the new c (the carry) is the top 32 bits of t and the new z is the bottom 32. The period is $(ab^3 - 1)/2 \approx 2^{125}$ (the order of b for the prime $ab^3 - 1$).

Both routines pass all in the Diehard battery of tests [2].

Both use just a few C instructions; the xorshift RNG has to keep the four most recent values; the MWC has to keep the three most recent as well as the latest carry c .

The seed set for xor128 is four 32-bit integers x, y, z, w not all 0, while the seed set for MWC is three 32-bit integers x, y, z and an initial $c < a$, excluding the two cases $x=y=z=c=0$, and $x=y=z=b-1, c=a-1$.

But xor128() is much faster than mwc(). On an 1800MHz PC, xor128() takes 4.4 nanoseconds (> 220 million numbers/sec), while mwc() takes 21 nanosecs (48 million/sec). But of course 48 million numbers/second is not likely to be considered a bottleneck in simulation problems that call for random numbers.

If you are interested in RNGs with extremely long periods, then you might consider MWC or CMWC methods that attain periods up to 2^{131102} , but require keeping a table of the k most recent values, with k 's of hundreds or thousands. But if not, and you are content with periods of $2^{160}, 2^{128}, 2^{96}$ or less, then one of the xorshift RNGs that merely keeps the last x or x, y or x, y, z in a straightforward matter, yet still provides periods that seem large enough for most applications, is worth considering as a workhorse RNG. Particularly when such xorshift RNGs are so simple, so fast and do so well on tests of randomness.

References

- [1] Marsaglia, George and Tsay, L. H., 1985, Matrices and the structure of random number sequences, *Linear Algebra and its Applications*, **67**, 147–156.
- [2] Marsaglia, George, 1995, The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness, produced at Florida State University under a grant from The National Science Foundation. Access available at www.stat.fsu.edu/pub/diehard, and a revised version of the Diehard tests at www.csis.hku.hk/~diehard.
- [3] Marsaglia, George and Tsang, Wai Wan, 2002, Some difficult-to-pass tests of randomness, *Journal Statistical Software*, **7**, Issue 3.
- [4] Marsaglia, George, 2003, Random number generators, *Journal of Modern Applied Statistical Methods*, **2** No. 2.