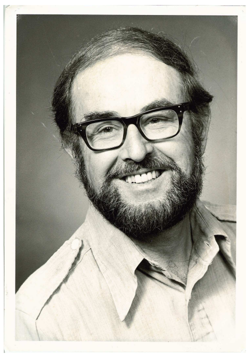
Journal of Modern Applied Statistical Methods
May,2003, Vol.2,No.1,2–13

Random Number Generators

George Marsaglia
Professor Emeritus of Statistics
Florida State University



The author discusses some promising new random number generators, as well as formulates the mathematical basis that makes them random variables in the same sense as more familiar ones in probability and statistics, emphasizing his view that randomness exists only in the sense of mathematics. He discusses the need for adequate seeds that provide the axioms for that mathematical basis, and gives examples from Law and Gaming, where inadequacies have led to difficulties. He also describes new versions of the widely used Diehard Battery of Tests of Randomness.

Key words: Random number generator, Diehard Test

Introduction

In 1985 I was invited to give the keynote address "A current view of random number generators" at Statistics and Computer Science: XVI Symposium on the Interface. An article based on that address was published in the Proceedings of that conference,[5]. Judging from newsgroups and citations, the article seems to have been widely read, although such proceedings are often difficult to access. Availability of the file keynote.ps in the CDROM [6], stat.fsu.edu/pub/diehard, may have made the article easier to get. Two other postscript files in that CDROM provide more detail on topics of the present article: mwc1.ps and monkey.ps.

In this article I will update that "current" view, dwelling at some length on what I see as more important kinds of RNGs, particularly Multiply-With-Carry (MWC) and Complimentary-Multiply-With-Carry (CMWC), because they have simple implementations, are very fast, can have incredibly long periods and pass tests randomness at least as well as, and often better than, other kinds of RNGs.

But first I will provide a summary discussion of *congruential* RNGs, because they remain the most common kind, and of *xorshift* RNGs, because they are as fast and simple as congruential but better behaved in tests of randomness. I will list all 648 of the full-period, 32-bit xorshift RNGs. There will also be a short description of *lagged Fibonacci* RNGs. These have diminished in importance, because MWC and CMWC RNGs provide far far longer periods for the same effort, and have better performance on tests.

George Marsaglia is Professor Emeritus of Statistics at Florida State University & Professor Emeritus of Pure andApplied Mathematics & Computer Science at Washington State University. His PhD was in Mathematics under H.B. Mann at Ohio State Univ., 1950, and he was a Fulbright Scholar under M. S. Bartlett and Alan Turing at Univ. of Manchester in 1949-50, then Associate under Harold Hotelling at Univ. N. Carolina. He was Professor and Director of the School of Computer Science at McGill Univ. 1970-78. He has published articles in over fifty math, computer science, statistics, physics, medicine and law journals, & is probably best known for work on random numbers, generating non-uniform variates and testing for randomness. His email address is geo@stat.fsu.edu

But one kind is still important because it can provide floating point uniforms directly, without the usual floating of integers.

I will also dwell on the problem of seeds and their relation to randomness, and on the need for an adequate number of seeds that has arisen in Law and Gaming. Finally, I will discuss the latest version of my *DIEHARD Battery of Tests of Randomness* [6], which includes some new, difficult-to-pass tests.

Random Number Generators (RNGs)

The mathematics of random number generators requires a set $\mathcal{Z}$, an invertible function f over $\mathcal{Z}$, and, for a random choice of a seed z from $\mathcal{Z}$, the sequence of random values in $\mathcal{Z}$ produced by iterating the function f:

$$f(z), f^2(z), f^3(z), \ldots,$$

where $f^2(z)$ means $f(f(z))$, $f^3(z)$ means $f(f^2(z))$, etc. Sometimes—in fact, most often—the set $\mathcal{Z}$ is just the set of integers represented by 32-bit computer words, but for RNGs that meet more stringent requirements, the set $\mathcal{Z}$ might be the set of all m-tuples $(x_1, x_2, \ldots, x_m)$ of 32-bit integers, and f a function that converts one such m-tuple into another.

If f is a one-to-one function over $\mathcal{Z}$, then for any seed z chosen uniformly from $\mathcal{Z}$, the random variable $f(z)$ is also uniformly distributed over $\mathcal{Z}$. (Just as if you randomly choose a digit d from $\mathcal{Z} = \{0, 1, 2, \ldots, 8, 9\}$ and I instead choose $3d + 5 \bmod 10$, my choice has the same uniform distribution as yours, since $f(x) = 3x+5 \bmod 10$ is one-to-one over $\mathcal{Z}$.) For the general case with seed set $\mathcal{Z}$, the choice of a random seed z from $\mathcal{Z}$ will provide, through $f(z), f^2(z), \ldots$ a long sequence of uniform random choices from $\mathcal{Z}$. They will not be independent random choices, but for many purposes they may behave as though they were, allowing us, with the minimal effort of choosing a random seed from $\mathcal{Z}$, to provide the huge samples that many simulation studies call for.

Note that when $\mathcal{Z}$ is $\{(x_1, x_2, \ldots, x_m)\}$, a set of m-tuples, and a random seed z from $\mathcal{Z}$ leads to a sequence of uniform but not-independent random choices from Z: $f(z), f^2(z), \ldots$, then the elements of each $(x_1, x_2, \ldots, x_m)$ may themselves be uniform over their range, and furthermore, substrings such as x_1, x_2, x_3 may be quite close to uniform and independent over their joint product set. This suggests, as experience shows, that RNGs with seed sets $\mathcal{Z}$ made up of m-tuples $(x_1, x_2, \ldots, x_m)$, may be more desirable, although they may require that the user provide many more than the usual single random integer seed. Perhaps the axiom: "You get what you pay for" applies.

Congruential RNGs

Given a suitable modulus m, multiplier a, additive constant k and initial random seed x_0, use of the sequence $x_n = ax_{n-1} + k \bmod m$ is probably the oldest and most common method of producing random integers. If a is a primitive root of the prime p, and x_0 is a random seed from

$$\mathcal{Z} = \{1, 2, \ldots, p-1\},$$

then the sequence generated by $x_n = ax_{n-1} \bmod p$ will be strictly periodic, with period $p - 1$, and each element of that sequence will be a uniform random variable on the set Z, but of course they will not be independent.

Getting $ax \bmod p$ for a prime p is usually much more difficult than getting $ax \bmod 2^{32}$, as the latter is virtually automatic in most CPUs. Thus sequences such as $x_n = ax_{n-1} + k \bmod 2^{32}$, with k odd and $a = \pm 3 \bmod 8$ have dominated, since, given a random seed

$$x_0 \in \mathcal{Z} = \{0, 1, \ldots, 2^{32}-1\},$$

each element in the sequence will be uniformly distributed over $\mathcal{Z}$, and the sequence will have period 2^{32}.

Congruential RNGs have the flaw of "falling mainly in the planes",[2]. For example, if x, y, z are any three successive integers produced by a congruential RNG with multiplier a, then the point (x, y, z) falls on the lattice of points generated by all linear combinations, with integer coefficients, of the three points $(1,a,a^2),(0,m,0),(0,0,m)$; any point (x,y) lies on the lattice generated by $(1,a),(0,m)$; any four consecutive outputs provide a point (x,y,z,w) in 4-space that must fall on the lattice of integer combinations of the four points $(1,a,a^2,a^3)$,$(0,m,0,0)$,$(0,0,m,0)$,$(0,0,0,m)$, etc.. See [2,3,4]. The last reference describes a simple way to characterize the lattice of a congruential RNG, in terms of ratios of edges of a unit cell—nearly cubic is better than long and thin.

Partly because it is one of the few exact measures for congruential RNGs, a widely used assessment is that of Knuth's 'Spectral test', [1] which dwells on the lattice stucture only tangentially. The spectral test amounts to characterizing a lattice by the minimum distance between its hyperplanes. Although I discovered the lattice structure of congruential RNGs, I have never found it a very useful measure of their goodness or badness, but it remains a measure that is frequently taken, because of its exact, interesting mathematical underpinnings.

Cracking a Congruential RNG

Because congruential RNGs are so common, and are often a system RNG, it may be worth pointing out a simple method for determining whether an unknown RNG is congruential, and if so, how to determine its modulus and multiplier. I have used this method for over thirty years, and it is implicit in references [2,3], but has not been stated explicitly in a journal before this. Suppose the rule for the RNG is $x_n = ax_{n-1} + k \bmod m$. Suppose α,β,γ are any three points in the plane with coordinates successive integers produced by that congruential RNG. Then the determinant of the 2×2 matrix with rows $\beta-\alpha$ and $\gamma-\alpha$ is the volume of the parallelepiped determined by the three points, and must be an integer multiple of m, the unit-cell volume of the lattice. Thus the gcd of five or six such determinants will usually provide m, from which a and k may be found.

For example, a certain simple RNG produces integers
308,785,930,695,864,237,1006,819,204,777,378, 495,376,357,70,747,356,...,
leading to points $\alpha_1=(308,785)$,$\alpha_2=(785,930)$, $\alpha_3=(930,695)$,...,

Then the parallelepiped determined by $\alpha_1,\alpha_2,\alpha_3$ has volume 133120, that determined by $\alpha_2,\alpha_3,\alpha_4$ has volume 30720, etc. The sequence of volumes determined by $\alpha_i,\alpha_{i+1},\alpha_{i+2}$ is 133120,30720,118784,263168,474112,..., and the gcd of the first two, then the first three, then the first four,..., leads to the sequence 10240,2048,1024,1024,1024,1024,..., and thus to the inference that $m = 1024$. Solving
$\{308a+k = 785,\ 785a+k = 930\} \bmod 1024$
yields $a = 69, k = 13$. Thus, with $x_0 = 308$, the sequence $x_n = 69x_{n-1} + 13 \bmod 1024$ produces the above output of the RNG.
Query: Which congruential RNG produced 768,54,747,221,321,48,225,669,414,163,260, 723,127,119,420,685,809,630?

Xorshift RNGs

Theory behind these RNGs is based on viewing a 32- (or 64-) bit integer as an element of a vector space with components in the field mod 2. For such, addition of two vectors can be implemented with the exclusive-or (xor) operation. That, combined with the shift operation, can be used to create certain linear transformations over that vector space. Here the seed set $\mathcal{Z}$ is the set of all non-zero 1×32 binary vectors and f is a linear transformation on $\mathcal{Z}$, represented by a 32×32 binary matrix T, nonsingular. Then for a random seed $y \in \mathcal{Z}$, the sequence is $yT, yT^2, yT^3 \ldots$. If, and only if, the order of T is $2^{32}-1$ in the group of 32×32 nonsingular binary matrices, then sequence $yT, yT^2, yT^3, \ldots$ will have period $2^{32}-1$.

Applications require a simple and fast way to form the matrix product yT, and that can be done if, say, $T = (I + L^a)(I + R^b)(I + L^c)$, where L is the matrix that effects a left shift of one (in C, `y^=(y<<1)`), so that yL^a in C is `y^=(y<<a)`. The matrix R, the transpose of L, effects a right shift of one. Thus, for $T = (I + L^a)(I + R^b)(I + L^c)$, for a random 32-bit seed y from $\mathcal{Z}$, each new y in the sequence $yT, yT^2, yT^3, \ldots$ can be produced in C by successive application of the three instructions

```
  y^=y<<13; y^=y>>17; y^=y<<5;
```

Such xorshift sequences are among the most desirable of simple RNGs: quick and easy, with seemingly better performance than congruential on tests of randomness.

For 32- (or 64-) bit binary vectors, there are no two-shift matrices $T = (I + L^a)(I + R^b)$ that have full period, and certainly no one-shift, so 3-shift T's are needed. 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 $2^{32} - 1$, listed in four columns:

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			

If $T = (I + L^a)(I + R^b)(I + L^c)$ has full period, then so does $(I + L^c)(I + R^b)(I + L^a)$, and so does $(I + L^a)(I + L^c)(I + R^b)$, leading to 4×81 T's with order $2^{32} - 1$. But then the transpose of each also has full period. That provides $8 \times 81 = 648$ matrices. Any $[a, b, c]$ in the above table of 81 yields eight lines of C code:

```
     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;
```

In summary: for each of the above 81 triples $[a, b, c]$ with $a < c$, any one of those eight lines of C can provide the instructions for a 32-bit RNG with period $2^{32} - 1$.

For 64-bit integers, there are 8×275 or 2200 such xorshift T's with periods of $2^{64} - 1$. A list is available from the author, as well as a fast C program for finding all full period xorshift T's

Lagged Fibonacci RNGs

The basic recurrence for a lagged Fibonacci RNG is $x_n = x_{n-r} \bullet x_{n-s}$, for 'lags' r and s, with $r > s$. Here $\bullet$ is a binary relation for pairs of elements in some set $\mathcal{X}$ and the seed set $\mathcal{Z}$ is the set of r-tuples $(x_1, x_2, \ldots, x_r)$ with the x's in $\mathcal{X}$. Usually, $\mathcal{X}$ is the set of 32-bit integers and $\bullet$ is addition or subtraction mod 2^{32} or addition of binary vectors (exclusive-or: $\oplus$). A promising choice has $\mathcal{X}$ the set of odd integers and $\bullet$ multiplication mod 2^{32}. Theory for the latter may be based on expressing elements x, y from $\mathcal{X}$ in the form $x = \pm 3^a, y = \pm 3^b \bmod 2^{32}$ so that $x \bullet y = \pm 3^{(a+b \bmod 2^{30})}$ and the recurrence rules for addition mod 2^{30} apply.

The notation $F(r, s, \bullet)$ is used for a lagged Fibonacci RNG. For proper choice of the lags r, s, the period of $F(r, s, \pm \bmod 2^{32})$ can be 2^{32+r}, while that of $F(r, s, \oplus)$ will at best be 2^r, whatever the word size. For proper choice $r > s$, the period of $F(r, s, *$ on odds $\bmod 2^{32})$ is 2^{30}.

As with other RNGs, formal definition of lagged Fibonacci RNGs requires a seed set $\mathcal{Z}$ and a function f on $\mathcal{Z}$. Here, $\mathcal{Z}$ is the set $\{[x_1, x_2, \ldots, x_r]\}$, with x's in the set $\mathcal{X}$ on which we have the binary relation, and the function f:

$f([x_1,x_2,\dots,x_r]) = [x_2,\dots,x_r,x_1 \bullet x_{r-s+1}]$.

Implementing lagged Fibonacci sequences with lags $r > s$ requires keeping a table of the r most recent values. Their periods, around 2^{32+r}, are far short of the possible 2^{32r} that is attainable with certain RNGs that also keep a table of the r most recent values, discussed in the next two sections. One of the most useful applications of lagged Fibonacci RNGs is in the generation of floating point uniform [0,1) variates directly, without the usual floating of random integers. For example, suppose we want to generate 64-bit (C's `double` or Fortran's `double precision`) uniform [0,1) random variables using the IEEE 754 standard: 1 sign bit, 11 exponent bits, 52 fraction bits, with the implied 1 leading the fraction part. For our binary relation $x \bullet y$ we use the rule:

`If` $x \geq y$ `then` $x-y$, `else` $x-y+1$.

If x and y are floating point representations of rationals $a/2^{53}$ and $b/2^{53}$, then $x \bullet y$ will produce the (exact) floating point version of $c/2^{53}$, with $c = x - y \bmod 2^{53}$.

A single precision version of this is in the widely used 'Universal' generator [9], while a double precision version is the RNG in Matlab and described in the new DVD version of [6]. Both combine an $F(99,33,-)$ sequence with a simple Weyl sequence $y_n = y_{n-1} + d$, with d a constant and the $y's$ double representations of rationals of the form $j/2^k$, with $k = 23$ or 53. Then the double precision operation

`if` $x<y$ `then` $x-y$ `else` $x-y+p/2^k$

produces rationals with denominators 2^k and numerators the difference modulo the largest prime $p < 2^k$.

Multiply-With-Carry (MWC) RNGs

An early description of MWC RNGs is in the file mwc1.ps of [6]. For another, suppose we extend the example of the second paragraph: You randomly choose a number from 1 to 58 as a pair ${}^cx$—that is, 23 is represented as ^{23}, 49 as ^{49}, etc. Your seed set $\mathcal{Z}$ is the 58 pairs ${}^cx$, $0 \leq c < 6$, $0 \leq x < 10$, excluding ^{00} and ^{59}. I convert your choice ${}^cx$ into a new pair by means of the function $f({}^cx) = {}^{c'}x'$, with $c' = \lfloor (6x+c)/10 \rfloor$ and $x' = 6x + c \bmod 10$. Thus $f({}^25) = {}^32$, $f({}^51) = {}^11$, etc. For each uniform choice of $z \in \mathcal{Z}$, my result $f(z)$ will be uniform in $\mathcal{Z}$, and the sequence $f(z), f^2(z), \dots$ will be a sequence of uniform choices from $\mathcal{Z}$. If you randomly choose, say, $z = {}^35$, the result of $f(z), f(z^{(2)}) \dots$ is a period-58 sequence that will contain every element of $\mathcal{Z}$:

${}^33,\ {}^21,\ {}^08,\ {}^48,\ {}^52, \dots, {}^19,\ {}^55,\ {}^35,\ {}^33,\ {}^21, \dots,$

each of them the realization of a true random variable in the mathematics sense, uniformly distributed over the finite set $\mathcal{Z}$.

If I use the x-component of each pair in the first cycle, I get a small sample of 58 random digits:

$$3,1,8,8,2,7,3,\dots,2,2,3,9,5,5.$$

and if I take, in reverse order, the x's from the full cycle, then attach a decimal point, I get the decimal expansion $\frac{33}{59} =$ .55932203389830508474576271186440677966101694915254237288135593···.

Now take an eminently practical example, (used as one of the components in the KISS RNG below): Let $a = 698769069, b = 2^{32}$. You randomly choose one of the $ab-2$ seeds from the set $\mathcal{Z}$ of pairs $[c,x]$, $0 \leq c < a$, $0 \leq x < b$, excluding $[0,0]$ and $[a{-}1, b{-}1]$. For each choice of seed z, form the sequence $f(z), f^2(z), f^3(z), \dots$, where

$$f([a,c]) = [\lfloor (ax+c)/b \rfloor, (ax+c) \bmod b].$$

The resulting sequence will have period $ab-2$, about $2^{60.4}$ or $10^{18.2}$. The x components of each element of that sequence of pairs will pass tests of randomness at-least-as-well-as, and usually better-than, most commonly used RNGs that produce 32-bit integers, and with a period far greater than the $\approx 2^{32}$ of most RNGs. But you must pay a little more for that longer period: two random seeds, the c in $0 \leq c < a$ and the x in $0 \leq x < b$. (The forbidden seeds in the examples: $[0,0]$ and $[a-1, b-1]$, have the property that $f(z) = z$ and thus produce sequences with period 1. This nuisance restriction is overcome in the next section on CMWC.)

Another feature of this example, and true in general: the generated x's will form, in reverse order, the base $b = 2^{32}$ 'digits' of the expansion of $j/(ab-1)$ for some integer $0 < j < ab - 1$, while the forbidden seeds $[0, 0]$ and $[a-1, b-1]$ will provide base-b expansions of $0/(ab-1)$ and $(ab-1)/(ab-1)$.

And still another feature of the MWC sequence generated on pairs $[c, x]$ by means of $f([a, c]) = [\lfloor(ax + c)\rfloor, (ax + c) \bmod b]$ is that **the resulting x's are just the elements of the congruential sequence $y_n = ay_{n-1} \bmod (ab-1)$, reduced mod b.** For example, with seed $z = [123, 456789]$ in that last example, the sequence of x's becomes

939722732,3858638025,3534982343,
2658951225,1839178858,1673917006...,

while with seed $y_0 = 123b+45678$, the congruential sequence $y_n = ay_{n-1} \bmod (ab-1)$ produces 319190024259564, 656649178557850825, 269 6296900490136775, 2470136321377329209... and that sequence, taken mod 2^{32}, yields the x's of the MWC sequence.

The above MWC sequences may be described by $x_n = ax_{n-1} + \text{carry} \bmod b$, with the 'carry' c being the number of b's dropped in the modular reduction that produced the new x: $c = \lfloor(ax_{n-1} + c)/b\rfloor$. These are lag-1 MWCs. For lag-r MWCs, as with any RNG, we need a collection $\mathcal{Z}$ of seeds and an invertible function f. In this case, $\mathcal{Z}$ is the set of $(r+1)$-tuples

$$\mathcal{Z} = \{[c; x_0, x_1, \dots, x_{r-1}]\},$$

with $0 \le c < a, 0 \le x < b$, except for $[0; 0, \dots, 0]$ and $[a-1; b-1, \dots, b-1]$.
Then the function f is

$$f([c; x_0, x_1, \dots, x_{r-1}]) = \lfloor(ax_0 + c)/b\rfloor; \\ x_1, x_2; \dots, x_{r-1}, ax_0 + c \bmod b].$$

For example, with $a = 5, b = 10$ and $r = 6$, the lag-6 MWC generator $x_n = 5x_{n-6} + c \bmod b$, starting with seed $z = [4; 2, 3, 5, 3, 9, 4]$, will produce this sequence of z's:
$[1; 3, 5, 3, 9, 4], [1; 5, 3, 9, 4, 4], [2; 3, 9, 4, 4, 6]$, $[1; 9, 4, 4, 6, 6], [4; 4, 4, 6, 6, 7], \dots,$ with output the sequence of x's: 4,4,6,6,7,....

The period of the sequence is the order of 10 for the prime $p = ab^6 - 1 = 5999999$, which is $(p-1)/2 = 2,499,999$.

Here is an example of a C program to compute the sequence through a little more than a full period, and to provide basis for comments on programming the general lag-r MWC RNG:

```
int main(void){
unsigned long i,t,x0=2,x1=3,
x2=5,x3=3,x4=9,x5=4,c=4;
for(i=1;i<2500006;i++){
{t=5*x0+c;c=t/10;x0=x1;x1=x2;
x2=x3;x3=x4;x4=x5;x5=t%10;
if(i<7 || i>2499993)printf{
"%7d,%d;%d,%d,%d,%d,%d\n",
i,c,x0,x1,x2,x3,x4,x5);   }}
```

The output of that C progam will give the first six, then the last six z's in the cycle of length 2,499,999, as well as confirming that the first six of the second cycle match those of the first cycle.

As with the lag-1 MWCs, the more general lag-r MWC:

$$x_n = ax_{n-r} + \text{carry} \bmod b$$

will produce a sequence of x's that are, in reverse order, the digits in the base-b expansion of $j/(ab^r - 1)$, with $0 < j < ab^r - 1$. For example, from the above C program, the sequence of x's in reverse order are $935328987 \cdots 8322467664$, and, sure enough,

$$\tfrac{4676644}{5999999} = .93532898706579741315594 \cdots \\ 16916123383224676644\ 935328 \cdots,$$

the trailing digits of which can be determined by expanding ($10^{499979} \times 4676644 \bmod 5999999$) to 30 places. To find which j provides the expansion of j/p, just put a decimal point in front of the reversed x's that end a cycle—for the above case, .9353289..., and find that 4676644 is the integer closest to $.935289p$.

A possibly simpler way is to use the inverse function of f, say

$$f^{-1}(z) = g([c; x_0, x_1, \dots, x_{r-1}]) =$$

$$[bc + x_{r-1} \bmod a;$$
$$\lfloor (bc + x_{r-1})/a) \rfloor, x_0, x_1, \ldots, x_{r-2}].$$

Then, with $z = [4; 2, 3, 5, 3, 9, 4]$, the rightmost x's in the sequence z,$g(z)$,$g^2(z)$,... will generate, in order, the digits of the base-b expansion of $4676644/p$, just as the rightmost x's in $f(z), f^2(z), f^3(z), \ldots$ will generate those digits in reverse order.

For computer implementation, we often choose $b = 2^{32}$, and then it is clear that the f sequence is more practical than the g sequence, as the integer operations $t = ax + c; c = \lfloor t/b \rfloor; x = t \bmod b$ are built into most CPUs. One merely forms $t = ax + c$ in 64 bits, then c is the top-32 and x the bottom-32 bits of t.

In the C program above, for lag-6, it is just barely feasible to keep the last six x's by means of promotions: `x0=x1; x1=x2;` and so on. Keeping a (circular) table of the r most recent x's in an array `Q[ ]`, and a pointer that rotates through the elements provides simple and very fast MWC RNG's. An example is this C procedure that provides 32-bit random integers with period greater than $2^{33245} \approx 10^{10007}$:

```
static unsigned long Q[1038],c=123;
unsigned long MWC1038(void){
static unsigned long i=1037;
unsigned long long t,a=611373678LL;
t=a*Q[i]+c; c=(t>>32);
if(--i) return(Q[i]=t);
i=1037; return(Q[0]=t);     }
```

You need to assign random 32-bit seeds to the static array Q[1038].

Note: Unlike simple MWC RNGs

$$x_n = ax_{n-1} + \text{carry} \bmod m,$$

which can be expressed as the reduction, mod b, of the congruential sequence

$$y_n = ay_{n-1} \bmod ab-1,$$

there seems to be no such simple relation between lag-r MWCs

$$x_n = ax_{n-r} + \text{carry} \bmod b$$

and the congruential sequence

$$y_n = ay_{n-r} \bmod ab^r-1.$$

Complimentary-Multiply-With-Carry (CMWC) RNGs

A few nagging problems come with MWC RNGs $x_n = ax_{n-r} + c \bmod b$ when $b = 2^{32}$ is chosen for computer implementation: the period is the order of b for the modulus $m = ab^r - 1$, but even when $p = ab^r - 1$ is a prime, the period cannot be $p-1$ because $b = 2^{32}$ is a square. Thus, as in the above example, MWC1038(), even though $p = ab^{1038} - 1$ is prime, (as is $(p-1)/2$), the generated 32-bit integers will have period $(p-1)/2$, and they will, in reverse order, form either the base-2^{32} digits of the expansion of j/p for some j in the subgroup $\{b, b^2, b^3, \ldots, b^{(p-1)/2} \bmod p\}$, or else for some j in the coset $\{hb, hb^2, hb^3, \ldots, hb^{(p-1)/2} \bmod p\}$, where h is some group element not in the cyclic subgroup generated by b.

Thus, strictly speaking, we do not have a seed set $\mathcal{Z}$ until we choose the seed $[c; x_0, x_1, \ldots, x_{1037}]$. Half of the choices will lead to the digits in the expansion of j/p for j in the group, half for j in the coset. (An interesting sidelight: if $[c, x_0, x_1, \ldots, x_{r-1}]$ is a seed whose subsequent x's form the reversed digits in j/p, with j in the subgroup, then the seed $[a-1-c; b-1-x_0, b-1-x_1, \ldots, b-1-x_{r-1}]$ will form the reversed digits in the expansion of k/p, with k in the coset—indeed, $k = p - j$.)

Another nuisance feature of MWC RNGs is that the two seeds $[a-1; b-1, \ldots, b-1]$ and $[0; 0, \ldots, 0]$ must be avoided, as they have the property that $f(z) = z$, so that their periods are 1 (with reversed digits corresponding to the base-b expansions of $0/p$ and p/p, as, in base 10, $23/23 = .9999999\cdots$).

Complimentary-multiply-with-carry RNGs (CMWC) permit us to avoid both of those difficulties. By making $b = 2^{32} - 1$, we can still exploit the way that integer arithmetic is carried out in modern CPUs (with a little fiddling for reductions mod $2^{32} - 1$ rather than mod 2^{32}). For this, we seek primes of the form $p = ab^r + 1$ with $b = 2^{32} - 1$ a primitive root of p. Then the CMWC recursion is $x_n = (b-1) - [ax_{n-r} + c \bmod b]$, where rather than the x of MWC, we return the $(b-1)$-complement of that x. The period will be $p - 1 = ab^r$.

Formally, if $p = ab^r + 1$ is a prime for which b is a primitive root, then the seed set

$$\mathcal{Z} = \{[c; x_0, x_1, \ldots, x_{r-1}]\},$$
$$0 \le c < a, 0 \le x < b,$$

has ab^r elements, and for any $z \in \mathcal{Z}$, (including all 0's and $c = a-1$ with all $x_i = b-1$), the sequence $f(z), f^2(z), \ldots$ will have period ab^r. Here

$$f(z) = f([c; x_0, x_1, \ldots, x_{r-1}]) =$$
$$[\lfloor (ax_0 + c)/b \rfloor; x_1, x_2; \ldots, x_{r-1},$$
$$(b-1) - (ax_0 + c \bmod b)].$$

Furthermore, the sequence of trailing x's in the sequence $f(z), f^2(z), \ldots$ will, in reverse order, form the base-b digits in the expansion of j/p for some $0 < j < p$.

Example: $b = 10$ is a primitive root of the prime $p = 7b^2 + 1 = 701$. The seed set is the 700 elements $\mathcal{Z} = \{[c; x, y]\}$ with $0 \le c < 7$, $0 <= x < 10$, $0 \le y < 10$. The iterating function is $f([c, x, y]) = [\lfloor (7x + c)/10 \rfloor; y, 9 - (7x + c \bmod 10)]$. Starting with seed $z = [2; 3, 4]$, the sequence $f(z), f^2(z), \ldots$ produces the 700 elements of $\mathcal{Z}$, then repeats:

$$[2;4,6], [3;6,9], [4;9,4], [6;4,2], [3;2,5], \ldots,$$
$$[4;6,6], [4;6,3], [4;3,3], [2;3,4], [2;4,6], [3;6,9], \ldots.$$

The trailing x's, taken in reverse order from the end of a cycle, are $4336\ldots$, and $.4336p = 303.9536$, so we expect $j = 304$, and so it is:

$$\tfrac{304}{701} = .4336661911554921 54\cdots$$
$$93295292439372325249643366 61912\cdots$$

provides the output from the CMWC $x_n = 9 - [7x_{n-2} + c \bmod 10]$ starting with $c = 2, x_0 = 3, x_1 = 6$ and put in reverse order.

Those digits could be produced in direct order with the sequence $z, g(z), g^2(z), \ldots$ where g is the inverse of f:

$$g([c; x, y]) = [10c + y \bmod 7; \lfloor (10c + y)/7 \rfloor, x].$$

Then the sequence $z, g(z), g^2(z), \ldots$ becomes

$$[2;3,4], [4;3,3], [4;6,3], [4;6,6], [1;6,6], \ldots,$$

and the trailing components, $43366\cdots$ form the digits in the expansion of 304/701.

The digits in the base-b expansion of j/p for a large prime p are likely to serve quite well as random integers from 0 to $b-1$, whether in direct or reverse order. But for computer implementation, with $b = 2^{32}$ or $b = 2^{32} - 1$, the arithmetic in $g(z)$ is much less well suited to computer operations than that in $f(z)$.

With period exceeding $2^{131086} \approx 10^{39461}$, here is a C procedure that produces the CMWC sequence

$$x_n = (b-1) - [ax_{n-r} + \text{carry} \bmod b],$$

with $b = 2^{32} - 1$, $a = 18782$:

```
static unsigned long Q[4096],c=123;
unsigned long CMWC(void){
unsigned long long t, a=18782LL;
static unsigned long i=4095;
unsigned long x,m=0xfffffffe;
i=(i+1)&4095;   t=a*Q[i]+c;
c=(t>>32); x=t+c; if(x<c){x++;c++;}
return(Q[i]=m-x);         }
```

The static array `Q[ ]` must be filled with random 32-bit integers for different runs. Rather than keeping the most recent 4096 x's in an array Q, smaller sizes $2048, 1024, 512, \ldots$ can be used. (Choice of array size 2^k simplifies incrementing the array index). Different choices of **r,a** require slight changes to the above C code for CMWC: Make **Q[ ]** have size **r**, change multiplier **a** and change the two 4095's to (decimal) **r-1**.

Here are a few good choices for **r** and **a**:

r	a	r	a
2048	1030770	64	987651206
2048	1047570	64	987657110
1024	5555698	32	987655670
1024	987769338	32	987655878
512	123462658	16	987651178
512	123484214	16	987651182
256	987662290	8	987651386
256	987665442	8	987651670
128	987688302	4	987654366
128	987689614	4	987654978

The results will be CMWC RNGs that seem to pass tests of randomness as well as any I know of, are simple and extremely fast, and have periods ab^r, with $b = 2^{32} - 1$, roughly 2^{32r+30}.

Choice of **r** and **a** have little effect on speed—about 18 nanoseconds on a 1.2MHz PC, or better than fifty million random numbers per second. Those wanting even more pairs **r,a** will need to find primes of the form $p = ab^{r} + 1$ for which $b = 2^{32} - 1$ is a primitive root.

Randomness and Choice of Seeds

Just as in geometry, where existence of and conclusions about complicated objects are premised on the existence of fundamentals such as points, lines, planes, etc., most distribution theory in probability is premised on the existence of more fundamental random variables. In particular, the distribution of elements in the RNG sequence $f(z), f^2(z), f^3(z), \ldots$ is premised on the existence of a random selection z from the seed set $\mathcal{Z}$, and it has as firm a basis in mathematics as do results in geometry, number theory and the like.

In my view, use of the name pseudo random number generators (PRNGs) is not appropriate. The qualifier *pseudo* can have several implications, most commonly: unreal, false, pretended, spurious, sham.

Use of *pseudo*, in the sense of *unreal* implies that there is **real** randomness, when the only kind we are sure of is in the sense of mathematics—exactly the sense that applies in our use of $f(z), f^2(z), \ldots$.

Use of *pseudo* in the sense of *false* is not appropriate either. If x and y are independent standard normal variates then we say that $x^2 + y^2$ is a chi-square variate; we do not call it a pseudo chi-square variate. Its properties may be deduced from that of its defining variates, just as are those of elements of the sequence $f(z), f^2(z), f^3(z), \ldots$—both considerations a real consequence of assumptions in the mathematical model.

Use of *pseudo* in the sense of *pretended* might be considered the least objectionable, for it might seem that we are pretending that our sequence $f(z), f^2(z), f^3(z) \ldots$ produces truly random numbers. But to many, (including me), true randomness exists only in the sense of mathematics—whether or not we understand it, the Universe is unfolding as it must. So the *pretending* is that there is such a thing as true randomness.

And finally, use of *pseudo* in the sense of *spurious* or *sham* is worst of all. Few would argue over the usefulness of RNGs for the past fifty years, and considerable effort has gone into studying their mathematics. Unfortunately, joint distribution theory for elements in the sequence $f(z), f^2(z), f^3(z) \ldots$ is not readily determined other than through simulation. Thus, except in cases such as the lattice structure of congruential RNGs, use of number or matrix theory to establish the periods, presence or absence of various m-tuples, relation to base-b decimal expansions, most of what we know about RNGs has been determined from extensive use, and those are far from *spurious* or *sham* endeavors.

If you must use 'pseudo', it would be more appropriate to say that our random-in-the-sense-of-mathematics numbers are *pseudo* independent, for we **are** pretending that they are independent—our random variables are identically distributed (id) uniform, but not independent identically distributed (iid) uniform.

Choice of Seeds

It is often convenient to choose just one or two random integers as seeds, even when several hundred may be needed to specify an element z of the seed set $\mathcal{Z}$. The other parts of the seed z may already have been assigned by default or previous use. Use of a RNG sequence can be likened to randomly choosing a starting position on a huge wheel of numbers (the RNG's cycle), then using them sequentially from that selected starting point. Even though that wheel might contain over 10^{10000} numbers, a single 32-bit integer can provide over four billion potential starting points, and the features you may want to study are likely to be consistent with the underlying probability theory at all but a few of those starting points.

But there are some applications where the seed selection procedure must be able to provide every element in the seed set $\mathcal{Z}$. Such requirements arise in Law. Many states have laws that permit use of computers (and hence a RNG) to select a jury venire—a panel of citizens selected to serve on juries. For example, Flor.Stat.ch.40.225(2000) authorizes use of computers to select jury venires, if such drawing *is by lot and at random by a method approved by The Florida Supreme Court.*

I was retained by that court to see if methods used in the various court districts were meeting this statutory requirement. It turned out that they were not,[8]. In most counties, a seed of perhaps ten digits is chosen by the staff or by the computer clock for the RNG of some proprietory administrative system. Suppose the task were to chooses 80 potential jurors from a list of 200 eligibles. There are $\binom{200}{80} > 10^{57}$ ways to choose such a panel, and the $\approx 10^{10}$ ways of selecting a single seed, (or worse, the 65,536 possible 16-bit integers from a computer clock) can not come close to providing the necessary number of choices.

The requirement that selection be *by lot and at random* means that a litigant should be entitled to any one of the possible venires; in this case, a RNG requiring at least eight 32-bit random integers to determine the element z of the seed set $\mathcal{Z}$ would be required. The Florida Supreme Court has implemented recommended procedures for choosing the necessary number of random seeds, from publicly available data—for example, from a coming week's stock market—which is unpredictable yet verifiable after the fact, see [7,8].

Another place where the need for adequate seeds arises is in the gaming industry. For example, The Michigan Game Control Board received an application to license a computer poker game that would permit the player to play as many as fifty games of poker at a time. The application was initially rejected because the Board ruled that a player was entitled to the chance that his fifty hands would all be straight flushes, and the RNG used by the machine had far too meager a seed set $\mathcal{Z}$. The company was presumably able to get a license after I advised them on overcoming the problem by extending the seed set $\mathcal{Z}$ for the version of my KISS RNG that they were using (without permission, as published mathematics or algorithms are not protected by patent or copyright law).

Combination RNGs

A RNG produces a random sequence of uniform selections from the seed set $\mathcal{Z}$. They are used to provide a sequence of integers $x_1, x_2, \ldots$, each x coming from one of the random elements in $\mathcal{Z}$, either, for simple RNGs , as the element $x_n = f^n(z)$ itself, or as one of the x-components of an r-tuple of x's that make up each $z \in \mathcal{Z}$. When, as is the most common case, the RNG merely produces a sequence of integers with period 2^{32}, then it can produce only $1/2^{32}$ of the possible pairs (x, y), only $1/2^{64}$ of the possible triples (x, y, z), etc. If your simulation concerns certain properties of triples (x, y, z) that are adequately represented in the limited supply that the RNG provides, then fine. But with such a limited supply, it is likely that there will be many simulations for which such a RNG is not suitable.

There are ways to overcome this difficulty. One of them is to use lag-r MWC or CMWC RNGs, from which every possible r-tuple of x's can appear. Another method is to combine simple, short period generators. One of the first examples of this was `Super Duper`, which combined a congruential and a 2-shift xorshift RNG. One of the most widely used is the KISS RNG, which I named during the early days of the Clinton administration when "Keep It Simple Stupid" was a buzz phrase relating to economic policy. The KISS RNG combines three simple RNGs: congruential, xorshift and the lag-1 MWC described above. It has had wide use, because it seems to pass all tests of randomness and yet uses simple computer instructions and needs no tables.

Here is a C version:

```
unsigned long KISS(){
static unsigned long
x=123456789,y=362436000,
z=521288629,c=7654321;
unsigned long long t;
 x=69069*x+12345;
 y^=y<<13; y^=y>>17; y^=y<<5;
 t=698769069LL*z+c; c=t>>32;
 return x+y+(z=t);  }
```

Seeds x,y,z,c may be changed from the default values. With a period $> 2^{124}$, KISS() may be more suitable than single-seed RNGs for applications. It is used in the gaming industry in North America and Australia. The speed may be only 20+ million per second compared to the 50+ million or so for MWC or CMWC, but a RNGs speed is usually not important in the overall time of a simulation or in game processing. For Fortran, which has no easy way to access the 64-bit product of two 32-bit integers, versions of KISS adjoin two lag-1 MWC's on 16 bits.

For a really powerful combination, I would recommend CMWC()+KISS(), for the rare chance that the 4096-tuples provided by CMWC might benefit from tweaking with KISS. (The "+" can be ordinary addition mod 2^{32}).

Testing RNGs

For some 25 years, in graduate math/cpt.sci/stat courses, I had discussed using a battery of tests of randomness for RNGs, and in one of the classes a Chinese student, who knew the word 'battery' only from pervasive TV ads for Sears car batteries, used the term Diehard in referring to the tests we were discussing—much to the amusement of the class, but the name stuck. Subsequently, under a grant from NSF, I developed

The Marsaglia Random Number CDROM
with
The Diehard Battery of Tests of Randomness

The CDROM contained 600 megabytes of random bits produced by combining the output of good RNGs with the output of physical devices purporting to provide random bits. Some 1000 free copies of that CDROM were distributed to researchers worldwide. The CDROM also contained Fortran and C code for what I called The Diehard Battery of Tests of Randomness. Although the free CDROMs were soon gone, presence of two of them at sites on the web made it readily available, and it seems to be in wide use.

A new version of those tests is now available at www.csis.hku.hk/~diehard and will be included in a DVD version of [6], with more extensive files of random bits. The new Diehard tests contain several new 'tough' tests that relate to use of random integers in computational number theory and cryptography.

Some of the most effective tests for randomness are based on what I call Monkey Tests. The idea is that some part of each random number can be used to specify a 'keystroke' that produces a letter from an alphabet. Few images invoke the mysteries and ultimate certainties of a random sequence as well as that of the proverbial monkey at a typewriter. A discussion of such 'monkey tests' is in the file monkey.ps of [6].

The most effective monkey tests are those for which counts are maintained for the number of appearances of each k-letter word in a long string of random letters, and it turns out that if

$$Q_k = \sum(\text{observed-expected})^2/\text{expected}$$

is the naive Pearson statistic for k-letter words, then $Q_k - Q_{k-1}$ is the quadratic form in the weak inverse of the covariance matrix of the k-letter word counts, and is asymptotically chisquare distributed with $L^k - L^{k-1}$ degrees of freedom, when the alphabet has L letters.

It usually happens that the number of possible k-letter words is too great to maintain a count for each word. In that case, the number of k-letter words missing from a long string of N random letters is used. That number will be close to normally distributed with mean $L^k e^{-N/L^k}$. Except for smaller cases $k = 2, 3, 4$, the variance must be estimated by simulation.

One of the tests in the new version of

Diehard is called the 'Gorilla test', in the sense of a strong monkey test. Many RNGs fail it. The Gorilla test counts the number of bit strings of length 26 that are missing from a sequence of $2^{26} + 25$ bits. Such a string is formed by specifying the bit position for each 32-bit word, then taking that bit from each of $2^{26} + 25$ calls to the RNG. The Gorilla test reports a p-value for the number of missing 26-bit strings for each of the 32 bits.

Another new test is the gcd test. That test uses two successive 32-bit integers u, v produced by the RNG, then finds k, the number of steps needed to find the gcd of u, v by Euclid's algorithm, and x, the resulting gcd. It tests to see if a sample of ten million such k's and x's have distributions consistent with underlying theory. All congruential RNGs—even those with prime modulus—fail the gcd test for distribution of k's, and many as well for distribution of x's.

The new tests include a stronger version of my 'birthday spacings' test, and others. These, with the gorilla and gcd tests, all relate to the suitability of numbers as random integers, an area of increasing importance in cryptography and computational number theory.

What might be called more conventional tests are mainly concerned with the performance of UNIs, that is, the uniform [0,1) variables that result from floating the RNG's integers. In a way, it may be reassuring that most RNGs pass such tests, because most real-life applications of RNGs seem concerned with the UNIs that result, not with observed non-uniformity in the bits of the integers that produce the UNIs (although poor performance of leading bits often portends bad sets of UNIs).

The availability of transfers through the Internet makes it easy to get the new version of the Diehard Battery of Tests at `www.csis.hku.hk/~diehard/`, which interested readers are invited to try for themselves.

References

[1] Knuth, D. E.(1998). *The Art of Computer Programming, Volume II*, 3rd Ed., Addison Wesley, Reading, Mass.

[2] Marsaglia, G. (1968). Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, **61**, 25–28.

[3] Marsaglia, G. (1970). Regularities in congruential random number generators. *Numerische Mathematik* **16,** 8–10.

[4] Marsaglia, G. (1972). The structure of linear congruential sequences. In *Applications of Number Theory to Numerical Analysis*, Z. K. Zaremba, ed., Academic Press, 249–285.

[5] Marsaglia, G. (1985). A current view of random number generators. Keynote Address, Statistics and Computer Science: XVI Symposium on the Interface, Atlanta, *Proceedings*, Elsevier.

[6] Marsaglia G. (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. Available at sites:
`www.stat.fsu.edu/pub/diehard`
`www.csis.hku.hk/~diehard/cdrom`.

[7] Marsaglia, G. (2001). Problems with the use of computers for selecting jury panels. *Jurimetrics* **41** No. 4, 425–427.

[8] Marsaglia, G. (2003). Seeds for random number generators. *Communications ACM* May 2003.

[9] Marsaglia, G., Tsang, W. W. and Zaman, A. (1989). Toward a universal random number generator. *Statistics and Probability Letters* **8,** No. 5.