Chaotic Random Number Generators with Random Cycle Lengths

by Agner Fog

This document is published at www.agner.org/random/theory, December 2000, revised November 25, 2001.

Abstract

A known cycle length has hitherto been considered an indispensable requirement for pseudo-random number generators. This requirement has restricted random number generators to suboptimal designs with known deficiencies. The present article shows that the requirement for a known cycle length can be avoided by including a self-test facility. The distribution of cycle lengths is analyzed theoretically and experimentally. As an example, a class of chaotic random number generators based on bit-rotation and addition are analyzed theoretically and experimentally. These generators, suitable for Monte Carlo applications, have good randomness, long cycle lengths, and higher speed than other generators of similar quality.

Introduction

The literature on random number generators has often emphasized that random number generators must be supported by theoretical analysis (Knuth 1998). In fact, most treatises on random number generators have focused mainly on very simple generators, such as linear congruential generators, in order to make theoretical analysis possible (e.g. Knuth 1998, Niederreiter 1992, Fishman 1996). Unfortunately, such simple generators are known to have serious defects (Entacher 1998). The sequence produced by a pseudo-random number generator is deterministic, and hence not absolutely random. It always has some kind of structure. The difference between good and bad generators is that the good ones have a better hidden structure, i.e. a structure that is more difficult to detect by statistical tests and less likely to interfere with specific applications (Couture and L'Ecuyer 1998). This criterion appears to be in direct conflict with the need for mathematical tractability. The best random number generators are likely to be the ones that are most difficult to analyze theoretically. The idea that mathematical intractability may in fact be a desired quality has so far only been explored in cryptographic applications (e.g. Blum et al 1986), not in Monte Carlo applications. Mathematicians have done admirable efforts to analyze complicated random number generators, but this doesn't solve the fundamental dilemma between mathematical tractability and randomness. Thus, we are left with the paradox that it may be impossible to know which type of generator is best.

Two characteristics of random number generators need to be analyzed: randomness and cycle length.

Randomness may be tested either by theoretical analysis or by statistical tests. Both methods are equally valid in the sense that a particular defect may be detected by either method. Some defects are most easily detected by theoretical analysis; other defects are easier to detect experimentally. Thus, it is recommended that a generator be subjected to both types of testing. The discipline of designing random number generators has reached a state where good generators pass all experimental tests. Attempts at improvement have therefore in recent years relied increasingly on theoretical testing.

It is often required that random number generators have very long cycle lengths (L'Ecuyer 1999). Theoretical analysis is the only way to find the exact cycle length in case the cycle is too long to measure experimentally. However, experimental tests can assure that the cycle is longer than the sequence of random numbers needed for a particular application. As is demonstrated below, such a test can be performed "on the fly" in a very efficient way if the state transition function is invertible.

Cycle lengths in random maps

A pseudo random number generator is based on the sequence

$$s_n = f(s_{n-1}), \quad s \in S \tag{1}$$

where the state transition function *f* maps the finite set *S* into itself. The number of possible states is the cardinality of *S*:

$$m = |S| \tag{2}$$

Let $\mathcal{F}$ denote the set of all possible state transition functions:

$$\mathcal{F} = \{f : S \to S\} \tag{3}$$

Assume that we have picked a state transition function *f* at random from the m^m members of $\mathcal{F}$. We now want to make sure that *f* can produce a sequence of c_{min} random numbers from a given seed s_0 without getting cyclic. Let ν be the length of the limiting cycle, and μ the length of any transient aperiodic sequence. In other words:

$$i < j < \mu + \nu \Rightarrow s_i \neq s_j \quad \text{and}$$

$$n \geq \mu \Rightarrow s_n = s_{n+\nu} \tag{4}$$

The mean values of ν and μ have been calculated by Harris (1960):

$$E(\nu) \approx \tfrac{1}{4}\sqrt{2\pi m} \qquad E(\mu) = E(\nu) + 1 \tag{5}$$

and the standard deviations

$$\sigma_\mu = \sigma_\nu \approx \sqrt{\left(\frac{2}{3} - \frac{\pi}{8}\right) m} \tag{6}$$

Thus, to make sure that the first c_{min} states are different, i.e. that $\mu+\nu > c_{min}$, we have to let $m >> c_{min}^2$. Unfortunately, setting $m >> c_{min}^2$ is no guarantee that the first c_{min} states are different, because we have no easy way of picking *f* out of $\mathcal{F}$ that is sufficiently random to guarantee that (5) and (6) hold; and experimental verification can be quite time-consuming.

The situation becomes easier when the state transition function *f* is invertible. Let $\mathcal{G}$ denote the subset of F that contains all invertible mappings of *S* onto itself:

$$\mathsf{G} = \left\{ f \in \mathsf{F} \;\middle|\; \forall s,t \in S : s \neq t \Rightarrow f(s) \neq f(t) \right\}. \tag{7}$$

If *f* is invertible then all trajectories must be cyclic, i.e. $\mu = 0$.

Assume now that we have picked a state transition function *f* at random from the *m*! members of G. Starting with the random seed s_0, we apply the transition *f* repeatedly in order to find the cycle length ν. If the first *i* states are different, then the probability that the next state will close the cycle is $\frac{1}{m-i}$ because only the initial state can close the circle.

The probability that the cycle length ν=c is then

$$p_c = \left(1 - \sum_{i=1}^{c-1} p_i\right) \frac{1}{m-c+1} = \frac{1}{m} \tag{8}$$

Thus, starting with a random seed, the length of the first cycle found will be uniformly distributed:

$$c_1 \in U(1, m) \tag{9}$$

An estimate of the expected number of cycles $n_c(m)$ can be derived from the fact that each subsystem of orbits has the same qualities as the whole system. If we have found a cycle of length c_1 then the number of remaining cycles should be $n_c(m-c_1)$. The average of $n_c(m-c_1)$ over all possible values of c_1 should then be equal to $n_c(m)$-1.

$$\tfrac{1}{m} \sum_{c_1=1}^{m} n_c(m - c_1) = n_c(m) - 1 \tag{10}$$

For big *m* we can ignore quantization and replace the sum with an integral:

$$\tfrac{1}{m} \int_1^m n_c(m - c_1)\, dc_1 \approx n_c(m) - 1 \tag{11}$$

This integral equation can be solved, using the approximation $dn_c(m)/dm \approx n_c(m) - n_c(m-1)$, and we get the approximate solution:

$$n_c(m) \approx \log_e(m+1) \tag{12}$$

A different path, described by Harris (1960), gives a slightly different approximation:

$$n_c(m) \approx \log_e(m) + \gamma \tag{13}$$

where γ = 0.577 is Euler's constant.

If we can find all cycles in the system, then we will see that their lengths are evenly distributed on a logarithmic scale:

$$\log_e c_i \in U(0, \log_e m) \tag{14}$$

The apparent discrepancy between (9) and (14) is explained by the fact that the former formula expresses the length of the cycle you find from a given seed. Hitting a long cycle is more probable than hitting a short cycle. (14), on the other hand, expresses the distribution of all cycles in the system.

If we need C_{min} random numbers for a particular application, then we can calculate the probability of getting into a cycle of insufficient length from (9):

$$P(c_1 \le c_{\min}) = \frac{c_{\min}}{m}. \qquad (15)$$

The advantage of choosing an invertible state transition function is not only that the mean cycle length gets longer, but also that it is easier to verify experimentally that a sequence contains no repetitions, because we only have to compare each new state s_i with the initial state s_0. All states between s_0 and s_i are not possible successors of s_i.

It is recommended that this verification method be built into the code. Such a self-test can be very fast (see below), and it provides the same certainty that a sequence is non-cyclic as the use of a generator with known cycle length.

RANROT generators

The principle of random cycle lengths is exemplified by a new class of random number generators similar to the additive or lagged Fibonacci generators, but with extra rotation or swapping of bits. Several types are exemplified below. In a RANROT generator type A, the bits are rotated after the addition, in type B they are rotated before the addition. You may have more than two terms as in type B3 below, and you may rotate parts of the bitstrings separately as in type W.

RANROT type A:

$$X_n = ((X_{n-j} + X_{n-k}) \bmod 2^b) \text{ rotr } r \qquad (16)$$

RANROT type B:

$$X_n = ((X_{n-j} \text{ rotr } r_1) + (X_{n-k} \text{ rotr } r_2)) \bmod 2^b \qquad (17)$$

RANROT type B3:

$$X_n = ((X_{n-i} \text{ rotr } r_1) + (X_{n-j} \text{ rotr } r_2) + (X_{n-k} \text{ rotr } r_3)) \bmod 2^b \qquad (18)$$

RANROT type W:

$$Z_n = ((Y_{n-j} \text{ rotr } r_3) + (Y_{n-k} \text{ rotr } r_1)) \bmod 2^{b/2}$$

$$Y_n = ((Z_{n-j} \text{ rotr } r_4) + (Z_{n-k} \text{ rotr } r_2)) \bmod 2^{b/2}$$

$$X_n = Y_n + Z_n \cdot 2^{b/2} \qquad (19)$$

Where X_n is an unsigned binary number of b bits, Y_n and Z_n are b/2 bits.
Y rotr r means the bits of Y rotated r places to the right (00001111_2 rotr 3 = 11100001_2).

i, *j* and *k* are different integers. For simplicity, it is assumed that $0 < i < j < k$, and that each $r \in [0,b)$, except for type W where $r \in [0,b/2)$.

The performance of these generators and the selection of good parameters is discussed below.

Bifurcation

Good random number generators appear to be chaotic, and this is in fact a desired quality. It is therefore tempting to apply chaos theory to these systems. Unfortunately, chaos theory traditionally applies to continuous systems only. A theory of deterministic chaos for discrete systems has been proposed by Waelbroeck & Zertuche (1999), but their theory and definitions apply only to infinite systems. A chaotic system with a finite number of discrete states has been studied experimentally, but with little theoretical analysis (Cernák 1996).

The most distinctive characteristic of a chaotic system is bifurcation, i.e. divergence of trajectories from adjacent starting points. A common measure of bifurcation is the Liapunov exponent, λ, defined for continuous systems by

$$\left| f^t(S_0+\varepsilon) - f^t(S_0) \right| = \varepsilon \cdot e^{t\lambda} \tag{20}$$

where S is the state descriptor, f is the function that transforms a state into the next state, t is the number of iterations, and ε is a small perturbation in the initial state. The formal definition of the Liapunov exponent is the limit of λ for $\varepsilon \to 0$ and $t \to \infty$:

$$\lambda(S_0) = \lim_{t\to\infty}\lim_{\varepsilon\to 0}\frac{1}{t}\log_e\left|\frac{f^t(S_0+\varepsilon) - f^t(S_0)}{\varepsilon}\right| \tag{21}$$

(Schuster 1995). This definition applies to idealized analytical models only. In physical systems you cannot go to the limits. Since physical variables are limited, the distance between the two trajectories cannot go towards infinity, and consequently λ will be zero in the limit $t \to \infty$. The distance can be expected to follow what resembles a logistic curve. In the beginning it grows exponentially, but at some point it levels off when the limit of the physical variables is approached. Similarly, the value of ε is limited downwards by precision, noise, and quantum effects. A reasonable value of λ would be measured at a point where the distance grows exponentially.

The same should apply to finite discrete systems. Assume that the state descriptor S consists of n binary digits:

$$S = (s_{n-1}s_{n-2}...s_0)_2 = \sum_{i=0}^{n-1} 2^i s_i \tag{22}$$

A difference between two states is defined by the Hamming distance, which is the number of bits that differ (Waelbroeck & Zertuche, 1999):

$$\mathrm{d_H(S,S')} = \sum_{\mathrm{i=0}}^{\mathrm{n-1}} \left|\mathrm{s_i - s'_i}\right| \tag{23}$$

Since the lowest possible value of ε is 1, we can define the Liapunov exponent for a finite discrete system as:

$$\lambda(S) = \frac{1}{t} \log_e d_H(f^t(S), f^t(S')) \qquad (24)$$

where $d_H(S,S') = 1$ and t is chosen within the area where the Hamming distance between the two trajectories grows exponentially. The highest possible Hamming distance is n, and the average distance between random states is n/2. We can therefore expect the average distance between two trajectories from adjacent starting points to grow exponentially in the beginning, and finally level off towards n/2. Experiments show that after a short period of exponential growth, the distance grows approximately linearly until it finally levels off at the value n/2 (fig. 1).

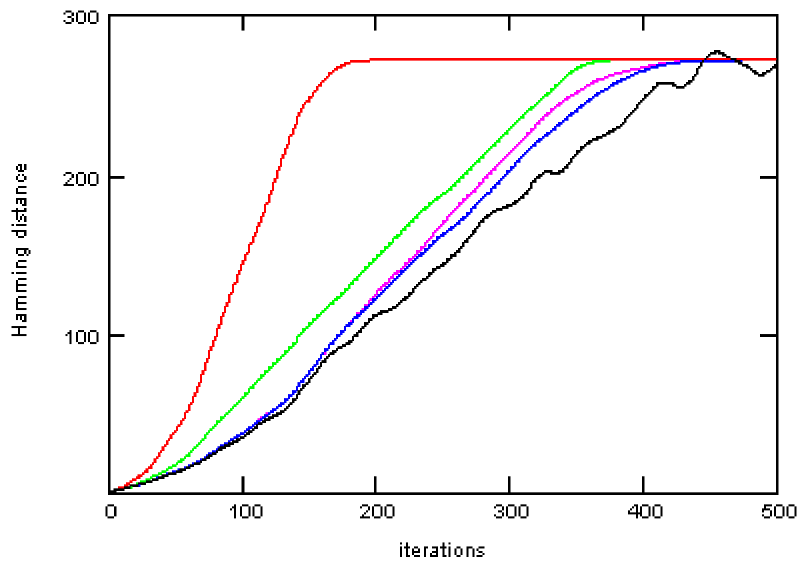

fig. 1. Average Hamming distance between trajectories from adjacent starting points in RNG's with k=17, b=32. Each curve is averaged over 100,000 experiments.
Legend (left to right):

red	= RANROT B3
green	= RANROT W
purple	= RANROT-A
blue	= RANROT-B
black	= lagged Fibonacci

The tiny kinks on the curves are systematic and reproducible.

The Liapunov exponent can be estimated for RANROT systems by calculating the probability that a differing bit will generate a carry in the addition operation. If a differing bit generates a carry in one of the trajectories, but not in the other, then the Hamming distance is increased by one. The average number of carries generated by one differing bit when adding b-bit integers is

$$C(b) = \frac{1}{b} \sum_{i=0}^{b-1} \sum_{j=1}^{b-1-i} 2^{-j} \qquad (25)$$

which is close to 1 when b is big. The fraction of bits involved in addition is 2a/k, where a is the number of addition operations in the state transition function. The expected value of the Liapunov exponent will then be

$$\lambda = \log_e\left(1 + \frac{2a}{k}C(b)\right) \tag{26}$$

A comparison between theoretical and experimental Liapunov exponents for small t is given in table 1.

RNG type	b	k	a	λ theoretical	λ experimental
A, B, W	32	17	1	0.10	0.12
B3	32	17	2	0.20	0.18

Table 1. Theoretical and experimental Liapunov exponents for RANROT generators

The shape of the curve quickly diverges from exponential growth. The reason for this is that carries from adjacent differing bits tend to cancel out so that the Hamming distance grows more slowly.

The Liapunov exponent does not appear to be a good measure of randomness because increasing k improves the randomness, but decreases λ. A RANROT generator can be converted to a lagged Fibonacci generator by setting r = 0. This degrades randomness but hardly affects λ. Finally, A random number generator that involves only exclusive-or and shift operations may have λ=0, yet generators of acceptable randomness have been constructed in this way (James 1990).

Nevertheless, bifurcation may in itself be a desired quality of random number generators, and designs with high values of λ may be desired. Very high Liapunov exponents can be obtained for generators that involve multiplication, such as multiple recursive generators (L'Ecuyer 1999) and multiply-with-carry generators (Couture & L'Ecuyer 1997).

Test of RANROT generators

Cycle lengths

Small RANROT systems were analyzed experimentally, identifying all cycles in each system in order to verify the distribution of cycle lengths. The biggest systems that were analyzed exhaustively for cycle lengths had m = 2^{32} states. Bigger systems were not analyzed in this way because of the high amount of computer resources needed for such an analysis.

For example, a RANROT type A system with j=1, k=4, b=7, r=4 has 24 cycles of the following lengths: 1, 5, 9, 11, 14, 21, 129, 6576, 8854, 16124, 17689, 135756, 310417, 392239, 432099, 488483, 1126126, 1355840, 1965955, 4576377, 7402465, 8393724, 57549556, 184256986.

A prime factorization of the cycle lengths showed that they share no more prime factors than you would expect from completely random numbers. This, however, holds only when the design rules in table 4 below are obeyed.

The number of cycles is always even. No explanation for this has been found. There appears to be no simple rule for predicting which states belong to the same cycle.

Plotting the binary logarithms of the cycle lengths showed that these were uniformly distributed in the interval between 0 and $k \cdot b$, in accordance with equation (14).

The average number of cycles was close to $\log_e(m+1) \approx \log_e m = k \cdot b \cdot \log_e 2$. Comparisons of the measured numbers of cycles to the theoretical values are listed in table 2 below for different types of RANROT generators.

The experimental measurement of the average number of cycles was somewhat problematic. All RANROT generators of the types defined above have one cycle of length 1, which is the trivial case where all bits in the system are 0. It is doubtful whether this cycle should be included in the count. However, the measured values fits the theoretical values best if this trivial cycle is included. Another problem is that the number of possible parameter sets that fit the design rules in table 4 is rather limited when the size of the system cannot exceed $m = 2^{32}$. Minor rule violations were unavoidable, with the result that some systems showed a few small extra cycles of the same length. It was difficult to decide whether these extra cycles should be regarded as caused by some undesired symmetry in the system or by mere coincidence. Such systems were manually excluded from the statistics, which inevitably causes a systematic error. A special version of the RANROT generator was designed in order to measure the average number of cycles without these problems. This is called type BX:

RANROT type BX:

$$X_n = (((X_{n-j} \oplus H) \text{ rotr } r_1) + (X_{n-k} \text{ rotr } r_2)) \bmod 2^b \qquad (27)$$

The $\oplus$ operator is a bitwise exclusive-or, inverting one or more bits when $H \neq 0$. Inverting bits prevents the trivial cycle of length one for the state of all-zeroes in the state descriptor. The H parameter can be varied arbitrarily which gives more possible parameter sets for testing.

RANROT type	number of tests	m	cycles/$\log_e m$	std.dev.
A	114	2^{20}-2^{32}	0.999	0.25
B	60	2^{20}-2^{32}	0.977	0.22
B3	62	2^{30}	1.035	0.21
BX	2033	2^{25}-2^{28}	1.0052	0.22
W	96	2^{30}	1.049*	0.23

Table 2**.** experimental values of mean number of cycles.

*) statistically significant at 5%

The experimental values agree well with the theory. The differences from the theoretical values predicted by equation (12) are small, and not statistically significant (t-test, 2-tailed), except for type W. The differences from the alternative approximation (13) is highly significant for type BX. The discrepancy for type W is ascribed to violations of the design rules, which were more serious in this case.

Randomness

All types of RANROT generators have been tested with several experimental tests including the fifteen tests in the DIEHARD battery of tests published by George Marsaglia (1997). They have passed all tests when the design rules in table 4 were obeyed, and even passed most tests when these rules were grossly violated.

The most important theoretical test is the spectral test (Knuth 1998), which is a test of a possible lattice structure in t-dimensional space of points

$$P_n = (u_{n-t+1}, u_{n-t+2}, \dots , u_n). \tag{28}$$

Where the normalized output

$$u_i = 2^{-b}X_i \in [0,1). \tag{29}$$

The RANROT generators have no lattice structure for P_n with dimensionality $t \leq k$, because all sequences of k consecutive numbers are possible except for the sequence of k zeroes. A lattice structure can only be found for dimensionality $t > k$, or for points of non-consecutive numbers. The worst case for RANROT types A, B and W is the structure of points formed from non-consecutive numbers $Q_n = (u_{n-k}, u_{n-j}, u_n)$ in 3-dimensional space.

I will now analyze the lattice structure of Q_n for RANROT type A.

In order to translate the rotr function into manageable functions, we first define

$$u_{n-j} + u_{n-k} = 2^{-b}y_a + 2^{r-b}y_b + y_c \tag{30}$$

where y_a is the least significant r bits of the sum, y_b is the next b-r bits, and y_c is the carry bit from the addition:

$$y_a = (2^b(u_{n-j} + u_{n-k})) \bmod 2^r \tag{31}$$

$$y_b = \lfloor 2^{b-r}(u_{n-j} + u_{n-k}) \rfloor \bmod 2^{b-r} \tag{32}$$

$$y_c = \lfloor u_{n-j} + u_{n-k} \rfloor \tag{33}$$

Now the formula (16) for RANROT type A can be expressed as

$$u_n = 2^{-r}y_a + 2^{-b}y_b \tag{34}$$

When y_a and y_c are kept constant, we can isolate y_b from (30) and insert in (34):

$$u_n = 2^{-r}u_{n-j} + 2^{-r}u_{n-k} + (2^{-r}-2^{-b-r})y_a - 2^{-r}y_c \tag{35}$$

which defines a plane perpendicular to the vector $(2^{-r},2^{-r},-1)$.

For y_c=0, the 2^r possible values of y_a give 2^r parallel planes with the distance

$$\delta_1 = \frac{2^{-r}(1-2^{-b})}{\sqrt{2^{1-2r}+1}} \tag{36}$$

These planes are cut off to the half of the unit cube that is defined by $u_{n-j} + u_{n-k} < 1$. In the other half of the unit cube, where $y_c = 1$, we have a similar set of 2^r parallel planes with the same distance. The second plane of the second set is almost coincident with the extension of the first plane of the first set, except for the tiny offset of

$$\varepsilon = \frac{2^{-r-b}}{\sqrt{2^{1-2r}+1}} . \tag{37}$$

Ignoring this offset, we can describe the lattice structure as 2^r+1 parallel planes with the distance approximately 2^{-r}.

When y_b and y_c are kept constant, in stead of y_a and y_c, we get a different lattice structure:

$$u_n = 2^{b-r}u_{n-j} + 2^{b-r}u_{n-k} + (2^{-b}-1)y_b - 2^{b-r}y_c \tag{38}$$

which defines a plane perpendicular to the vector $(2^{b-r},2^{b-r},-1)$.

For y_c=0, the 2^{b-r} possible values of y_b give 2^{b-r} parallel planes with the distance

$$\delta_2 = \frac{1-2^{-b}}{\sqrt{2^{1+2(b-r)}+1}} \tag{39}$$

For y_c=1, we get a similar set of parallel planes. The distance between the last plane in the first set and the first plane in the second set is slightly more than δ_2:

$$\delta_3 = \frac{1-2^{-b}+2^{-r}}{\sqrt{2^{1+2(b-r)}+1}} \tag{40}$$

Combining these two sets of planes, we have 2^{b-r+1} parallel planes with distance approximately $2^{r-b-½}$.

The lattice structure defined by (35) will be dominating when r is small, while the lattice structure defined by (38) will dominate when r is big, i.e. close to b. The best resolution is obtained for intermediate values of r, where the points Q_n will lie on the lines defined by the intersections of the planes of the two lattice structures.

The special case r = 0 defines the well-known lagged Fibonacci or ADDGEN generator (Knuth 1998). According to (36) and (37) it has two parallel planes with the distance $1/\sqrt{3}$, in accordance with the findings of L'Ecuyer (1997). Interestingly, the same result is obtained by setting $r = b$ in equation (40).

The RANROT generators of the other types have been analyzed in a similar way. The results are summarized in table 3. All of the lattices have slight deviations from the nice pattern of exactly equidistant planes. The distance between a corner and the nearest plane is always less than or equal to the distance between planes.

RANROT type	conditions	number of planes or hyperplanes[1)]	perpendicular to	max distance between planes or hyperplanes
A	r_1 small	2^r+1	$(1, 1, -2^r)$	$\leq 2^{-r}$
	r_1 big	2^{1+b-r}	$(1, 1, -2^{r-b})$	$\leq 2^{r-b-½}(1+2^{-r})$
B	r_1, r_2 small	$2^{r\text{max}}+2^{\lvert r1-r2 \rvert}$	$(2^{-r2}, 2^{-r1}, -1)$	$\leq 2^{-r\text{max}}$
	r_1 small r_2 big	$2^{b+r1-r2} + 2^{r1}$	$(2^{b-r2}, 2^{-r1}, -1)$	$\leq 2^{r2-r1-b}$
	r_1 big r_2 small	$2^{b+r2-r1} + 2^{r2}$	$(2^{-r2}, 2^{b-r1}, -1)$	$\leq 2^{r1-r2-b}$
	r_1, r_2 big	$2^{b-r1} + 2^{b-r2}+1$	$(2^{b-r2}, 2^{b-r1}, -1)$	$\leq 2^{r\text{min}-b}$
B3	r_1, r_2, r_3 small	$2^{r\text{max}} \cdot (1+2^{-r1}+2^{-r2}+2^{-r3})-1$	$(2^{-r3},2^{-r2},2^{-r1},-1)$	$\leq 2^{-r\text{max}}$
	r_1, r_2, r_3 big	$2^{b-r1} + 2^{b-r2} + 2^{b-r3} +1$	$(2^{b-r3},2^{b-r2},2^{b-r1},-1)$	$\leq 2^{r\text{min}-b}$
W	r_1, r_3 small	$\approx 2^{b/2+\lvert r1-r3 \rvert} + 2^{b/2} +2^{\max(r1,r3)}$	$(2^{b/2-r1}, 2^{b/2-r3}, -1)$	$\approx 2^{-b/2-\lvert r1-r3 \rvert}$
	r_1 small r_3 big	$\approx 2^{b+r1-r3} + 2^{b/2} + 2^{r1}$	$(2^{b/2-r1}, 2^{b-r3}, -1)$	$\approx 2^{r3-r1-b}$
	r_1 big r_3 small	$\approx 2^{b+r3-r1} + 2^{b/2} + 2^{r3}$	$(2^{b-r1}, 2^{b/2-r3}, -1)$	$\approx 2^{r1-r3-b}$
	r_1, r_3 big	$\approx 2^{b-r1} + 2^{b-r3} + 1$	$(2^{b-r1}, 2^{b-r3}, -1)$	$\approx 2^{-b+\max(r1,r3)}$
	r_2, r_4 small	$\approx 2^{b/2+\max(r2,r4)} + 2^{\lvert r2-r4 \rvert} + 1$	$(2^{-b/2-r2}, 2^{-b/2-r4}, -1)$	$\approx 2^{-b/2-\max(r2,r4)}$
	r_2 small r_4 big	$\approx 2^{b/2+r2} +2^{b/2+r2-r4} +1$	$(2^{-b/2-r2}, 2^{-r4}, -1)$	$\approx 2^{-b/2-r2}$
	r_2 big r_4 small	$\approx 2^{b/2+r4} +2^{b/2+r4-r2} +1$	$(2^{-r2}, 2^{-b/2-r4}, -1)$	$\approx 2^{-b/2-r4}$
	(r_2, r_4 big)	$\approx 2^{b/2} + 2^{b/2-r2}+2^{b/2-r4}$	$(2^{-r2}, 2^{-r4}, -1)$	$\approx 2^{-b/2}$

table 3. Worst case lattice structure of points defined by non-consecutive random numbers.

*r*min and *r*max are the smallest and the biggest of the *r*'s respectively.

1) Almost coincident planes are counted as one.

Choice of parameters

While the choice of parameters for the RANROT generators is not very critical, certain rules should be observed for best performance. Most importantly, all bits in the state buffer should be interdependent. For this reason, *j* and *k* must be relatively prime. If *j* and *k* (and *i* for type B3) share a factor *p*, then the system can be split into *p* independent systems. For the same reason, *k* – *j* must be odd in type W.

It is clear from the way a binary addition is implemented, that there is a flow of information from the low bits to the high bits through the carries, but no flow of information the other way. This problem is seen in the lagged Fibonacci generator, where the least significant bit of all words in the buffer form an independent system. It has been proposed to solve this problem by adding the carry from the most significant bit position to the least significant bit in the next addition in the so-called ACARRY generator (Marsaglia, et. al. 1990). This mechanism improves the cycle length but hardly the randomness. The rotation of bits in the RANROT generators serves the same purpose of providing a flow of information from the high bits to the low bits, but at the same time improves the lattice structure. At least one of the *r*'s must be non-zero in order to make all bit positions interdependent. The lattice structure described above is rather coarse if the *r*'s are too small (i.e. close to zero) or too big (i.e. close to b, or for type W: b/2). The finest lattice structure is obtained for values or *r* near *b*/2 for type A, near *b*/3 and 2*b*/3 respectively for type B, and near *b*/4, *b*/2, 3*b*/4 for type B3.

The biggest theoretical problem relating to the choice of parameters is to make sure that the distribution of cycle lengths is random, in accordance with (14). If all *r*'s are zero, then we have the situation of the lagged Fibonacci or ADDGEN generator, which has many relatively short cycles with commensurable lengths. The maximum cycle length in this case is $(2^k-1)2^{b-1}$ for optimal choices of *j* and *k* (Knuth 1998, Lidl & Niederreiter 1986). A random distribution of cycle lengths is usually obtained if at least one of the *r*'s is non-zero. But for certain unlucky choices of *r*'s you may see certain regularities or symmetries that cause the system to have a number of small cycles with the same or commensurable lengths, while the remaining cycles have the desired distribution. For example, a RANROT type A with $r=1$ has 2^{b-1} cycles of length 1 because any state with all *X*'es equal and the most significant bit=0 will be transformed into itself. Several rules of thumb have been developed to avoid such symmetries. These rules are summarized in table 4. Unfortunately, there is no known design principle which can provide an absolute guarantee that the cycle lengths have the distribution (14). Therefore, a self test as described above is needed for detecting the (very unlikely) situation of getting into a cycle of insufficient length. (This has never happened during several years of extensive use).

For optimal performance, the parameters should be chosen according to the following rules:

rule #	rule	RANROT type			
		A	B	B3	W
1	j, k (and i) have no common factor	+++	+++	+++	+++
2	$1 < j < k$-1	++	+	+	+
3	k-j odd	-	-	-	+++
4	one $r \neq 0$	+++	+++	+++	++
5	all r's $\neq 0$	+++	++	+	-
6	r's different	n. a.	++	++	++
7	$r > 1$	+++	++	+	+
8	r relatively prime to b	+	+	+	+
9	k relatively prime to b	+	+	+	+
Explanation of symbols:					
+++	important rule				
++	some small cycles may occur if not obeyed				
+	minor importance				
-	no significance				

Table 4**.** Design rules for RANROT generators.
All rules applied to an r also applies to the corresponding b-r (for type W: $b/2 – r$).

If the desired resolution is higher than the microprocessor word size, then you may use an implementation like type W, where parts of the bitstring are rotated separately, because it is faster to rotate two words than to rotate one double-word. You may let r_3 and r_4 be zero for the sake of speed.

The value of k determines the size of the state buffer. While a high value of k improves randomness and cycle length, a moderate value between 10 and 20 will generally suffice for RANROT generators. An excessive value of k will take up unnecessary space in the memory cache, which will slow down execution in applications that exhaust the cache.

History and speed considerations

Treatises on random number generators traditionally pay little or no attention to speed, although speed can be a quite important factor in Monte Carlo applications.

The RANROT generator is designed to be a fast random number generator. I invented this generator several years ago when computers were not as fast as today and when most random number generators were either quite bad or quite slow. I needed a random number generator with good randomness and high speed. Since multiplication and especially division are quite time-consuming instructions, I was left with additive generators. I searched for other microprocessor instructions that provided a good shuffling of bits and at the same time were fast. The bit rotate instruction turned out to be the best candidate for this purpose. The result was the fast RANROT generator, which turned out to have a quite chaotic behavior.

The problem that the cycle length is unknown and random was solved by means of the self-test. This relieved some serious design constraints and allowed me to optimize for speed rather than for mathematical tractability.

The self-test is implemented by saving a copy of the initial contents of the state buffer. After each execution of the algorithm, the first word of the state buffer is compared to the copy. (Note that the first word of the circular buffer is the one pointed to by the *n-k* pointer, not the one that physically comes first). The rest of the buffer only has to be compared in the rare case that the first word is matching. The self-test therefore takes only one CPU clock cycle extra.

Most applications require a floating point output u_n in the interval [0,1). The conversion of the integer X_n to a floating-point value u_n has traditionally been done simply by multiplication with 2^{-b}. This involves the slow intermediate steps of converting the unsigned *b*-bit integer to a signed integer with more than *b* bits, and converting this signed integer to a normalized floating-point number. A much faster method can be implemented by manipulating the bits of a floating-point representation as follows: Set the binary exponent to 0 (+ bias) and the fraction part of the significand to random bits. This will generate a floating-point number with uniform distribution in the interval [1,2). A normalized floating-point number in the interval [0,1) is then obtained by subtracting 1. The binary representation of floating-point numbers usually follows the IEEE-754 standard (IEEE Computer Society 1985). If portability is important then you have to choose a floating-point precision that is available on all computers.

The optimized code for a RANROT type W executes in just 18 clock cycles on an Intel Pentium® II or III microprocessor, including the time required for the self-test and conversion to floating point. This means that it can produce more than 50 million floating point random numbers per second with 63 bits resolution on a 1GHz microprocessor (Fog 2001).

New generations of microprocessors can do multiplications in a pipelined manner. This means that it can start a new multiplication before the previous one has finished (Fog 2000). This makes multiplicative random number generators more attractive, although they still take more time than the RANROT.

A few years ago, the so-called Mersenne Twister was proposed as a very good and fast random number generator (Matsumoto & Nishimura 1998). Unfortunately, the Mersenne Twister uses quite a lot of RAM memory. The effect of memory use on speed does not show in a simple speed test that just calls the generator repeatedly, but an excessive memory use may slow down execution significantly in larger applications that exhaust the memory cache in the microprocessor.

Conclusion

It is possible to make a good pseudo-random number generator with unknown cycle length when a self-test provides the desired guarantee against repeated states. This self-test can be very fast when the state transition function is invertible.

The traditional requirement that cycle lengths can be calculated theoretically has been avoided by introducing the self-test. This gives a freedom of design that makes it possible to optimize for speed and randomness and to take hardware-specific considerations.

The RANROT generators tested here are faster than other generators of similar quality, and they have performed very well in both experimental and theoretical tests for randomness.

Under ideal conditions, the distribution of cycle lengths is expected to be random, according to equation (14), and the average number of cycles is close to $\log_e(m)$. This ideal behavior is approximated quite well by all variants of the RANROT generators tested here, except for the most unfavorable choices of parameters. The general principles described in this article may be applied to other designs as well.

For the most demanding applications, you may combine the output of two different generators, one traditional and one with random cycle length, in order to get the best of both worlds.

Examples of implementation in the C++ and assembly languages are given by Fog (2001).

References

Blum, L, Blum, M, and Schub M. 1986: A simple unpredictable pseudo-random number generator. *SIAM Journal of Computing.* vol. 15, no. 2, pp. 364-383.

Cernák, J. 1996. Digital generators of chaos. *Physics Letters A.* vol. 214, pp. 151-160.

Couture, R and L'Ecuyer, P. 1997: Distribution Properties of Multiply-with-Carry Random Number Generators. *Mathematics of Computation*, vol. 66, p. 591.

Couture, R and L'Ecuyer, P. 1998. Guest Editors' Introduction. *ACM transactions on Modeling and Computer Simulation.* vol. 8, no. 1, pp. 1-2.

Entacher, Karl. 1998. Bad Subsequences of Well-Known Linear Congruential Pseudorandom Number Generators. *ACM transactions on Modeling and Computer Simulation.* vol. 8, no. 1, pp. 61-70.

Fishman, George S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications.* New York: Springer.

Fog, A. 2000. *How to optimize for the Pentium family of microprocessors.* http://www.agner.org/assem. [A copy is archived at the Royal Library Copenhagen]

Fog, A. 2001. *Pseudo random number generators.* http://www.agner.org/random.

Harris, B. 1960. Probability distributions related to random mappings. *Annals of Mathematical Statistics.* vol. 31, pp. 1045-1062.

IEEE Computer Society 1985: *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985).*

James, F. 1990. A review of pseudorandom number generators*. Computer Physics Communications.* vol. 60, pp. 329-344.

Knuth, D. E. 1998. *The art of computer programming.* vol. 2, 3rd ed. Addison-Wesley. Reading, Mass.

L'Ecuyer, P. 1997. Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal of Computing* vol. 9, no. 1, pp. 57-60.

L'Ecuyer, P. 1999. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research*, vol 47, no. 1. pp. 159-164.

Lidl, R. and Niederreiter, H. 1986. *Introduction to finite fields and their applications*. Cambridge University Press.

Marsaglia, G. 1997. *DIEHARD*. http://stat.fsu.edu/~geo/diehard.html or http://www.cs.hku.hk/internet/randomCD.html.

Marsaglia, G., Narasimhan, B., and Zaman, A. 1990. A random number generator for PC's. *Computer Physics Communications.* vol. 60, p. 345.

Matsumoto, M. and Nishimura, T. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* vol. 8, no. 1, pp. 31-42.

Niederreiter, H. 1992. *Random Number Generation and Quasi-Monte Carlo Methods.* Philadelphia: Society for Industrial and Applied Mathematics.

Schuster, H. G. 1995. *Deterministic Chaos: An Introduction.* 3'rd ed. VCH. Weinheim, Germany.

Waelbroeck, H. & Zertuche, F. (1999). Discrete Chaos*. J. Phys. A.* vol 32, no. 1, pp. 175-189.