Fast and Reliable Random Number Generators for Scientific Computing*

Richard P. Brent
Computing Laboratory
University of Oxford, UK
PARA04@rpbrent.co.uk

22 June 2004

*Invited talk at PARA04, Lyngby, Denmark.

 PARA04t

Abstract

Fast and reliable pseudo-random number generators are required for simulation and other applications in Scientific Computing. Because of Moore's law, random number generators that were satisfactory in the past may be inadequate today. We outline the current requirements for good uniform random number generators, and describe a class of generators having very fast vector/parallel implementations. We consider how to combine two generators to give a generator with better statistical and/or cryptographic properties, and also discuss the problem of initialization.

Outline

- Requirements for RNGs
- Linear Congruential RNGs
- Generalized Fibonacci and LFSR RNGs
- Ordering of triples
- Irreducible and primitive trinomials
- Almost primitive trinomials
- Application to RNGs
- Initialization of LFSR generators
- Combining generators
- Comments on some available RNGs
- Some suggestions for good RNGs

Introduction

Pseudo-random number generators (RNGs) are widely used in simulation.

A program running on a fast computer or cluster of PCs might use 10^9 random numbers per second for many hours (or weeks). Small correlations or other deficiencies could easily lead to spurious results.

In order to have confidence in the results of simulations, we need to have confidence in the statistical properties of the random numbers used. There are several examples of flaws in a random number generator being mistaken for significant physical phenomena.

Some Requirements

- **Uniformity.** This is the easiest requirement to achieve, at least when considered over the whole period.
- **Independence.** d-tuples should be uniformly and independently distributed in d-dimensional space (say for $d \le 6$). Subsequences (e.g. odd/even) of the main sequence should be independent.
- **Long Period.** The period (length of a cycle in the random numbers generated) should be large, certainly at least 10^{16} and preferably much larger (say at least 10^{32}).

  A few years ago, a generator in the library of at least one supercomputer manufacturer had period 2^{31} and ran through a complete cycle a few seconds! This and similar generators are obsolete and must be avoided.

Requirements continued

- **Statistical Tests.** The generator should appear random when subjected to any "natural" statistical test (i.e. one which does not depend on a detailed knowledge of the algorithm used to generate the "random" numbers) using an amount of computation comparable to that in applications (say 10^{16} operations).

  There are many statistical tests.
  Here are a few examples:

  - Marsaglia's "birthday spacings" test.
  - Shchur, Heringa and Blöte's 1D random-walk test.
  - A test on orderings of triples (u_n, u_{n-i}, u_{n-j}) for $0 < j < i \le B$, $B^2/2 \le 10^{16}$ say. This test is designed to detect generators based on 3-term recurrences with lags $\le B$.
  - Similarly for k-tuples with $B^{k-1}/(k-1)! \le 10^{16}$, $3 \le k \le 7$ say.

Requirements continued

- **Proper Initialization** The initialization of random number generators, especially those with a large amount of state information, is an important and often neglected topic. In some applications only a short sequence of random numbers is used after each initialization of the generator, and it is important that short sequences produced with different seeds are uncorrelated, even if the seeds are consecutive integers.
- **Disjoint Subsequences.** If a simulation is run on a parallel machine or on several independent machines, the sequences of random numbers generated on each machine must be independent (at least with very high probability).

Requirements continued

- **Unpredictability** In cryptographic applications, it is not sufficient for the sequence to pass standard statistical tests for randomness; it also needs to be *unpredictable* in the sense that there is no efficient deterministic algorithm for predicting u_n (with probability of success significantly greater than chance) from $(u_0, u_1, \ldots, u_{n-1})$, unless n is so large that the prediction is infeasible.

  Unpredictability is not required in scientific applications. However, if a random number generator is predictable then we can always devise a statistical test (albeit an artificial one) that the generator will fail. Thus, it is a wise precaution to use an unpredictable generator if the cost of doing so is not too high.

  Unpredictability implies uniformity, independence, and a (very) long period. However, it is worthwhile to state these simpler requirements separately.

Requirements continued

- **Efficiency.** Only a few arithmetic operations should be required to generate each random number. Procedure call overheads should be minimised (e.g. one call could fill an array with random numbers).
- **Repeatability.** For testing and development it is useful to be able to repeat *exactly* the same sequence as was used in another run, but not necessarily starting from the beginning of the sequence.

  Thus, it should be easy to save all the state information required to restart the generator.
- **Portability.** For testing and development it is useful to be able to generate *exactly* the same sequences on different machines.

Linear Congruential RNGs

Introduced by D. H. Lehmer in 1948.

$$U_{n+1} = (aU_n + c) \bmod m$$

where $m > 0$ is the modulus, a is the multiplier, and c is an additive constant.

Often $m = 2^w$ is chosen as a convenient power of 2. In this case it is possible to get period m. However, $w = 32$ is not nearly large enough ($10^{16} > 2^{53}$).

If m is prime and $c = 0$, we can get period $m - 1$ by choosing a to be a primitive root.

Linear congruential generators have difficulty passing the spectral test ("Random numbers fall mainly in the planes" – Marsaglia).

Marsaglia and Zaman have introduced "add with carry" and "subtract with borrow" generators, which are essentially linear congruential generators with large moduli of a special form.

Generalized Fibonacci Generators

$$U_n = U_{n-r} \; \theta \; U_{n-s}$$

where r and s are fixed "lags" and θ is some binary operator. We always assume $0 < s < r$.

For example, the choice of $\theta = + \pmod{2^w}$ that we usually assume below is convenient on a binary machine. In this case the period is at most

$$2^{w-1}(2^r - 1) \,,$$

and this is attained if $x^r + x^s + 1$ is a primitive polynomial over GF(2) and the initial values $U_0, \ldots, U_{r-1}$ are not all even.

The case of addition in GF(2), i.e. $w = 1$, $\theta = \oplus$, gives a *linear feedback shift register* (LFSR) generator which is easy to implement in hardware. These (and the case $w > 1$, $\theta = \oplus$) are also called Tausworthe generators.

Problem with Ordering of Triples

If $\theta = + \pmod{m}$, then the orderings of certain triples do not occur with the correct probability (1/6). This is unacceptable if $r = 2$ (Fibonacci), but it is not so serious if r is large. Neglecting probabilities $O(1/m)$, we have:

Ordering	**Probability**
$U_{n-r} < U_{n-s} < U_n$	1/4
$U_{n-r} < U_n < U_{n-s}$	0
$U_{n-s} < U_{n-r} < U_n$	1/4
$U_{n-s} < U_n < U_{n-r}$	0
$U_n < U_{n-r} < U_{n-s}$	1/4
$U_n < U_{n-s} < U_{n-r}$	1/4

We can implement a statistical test that gives a significant result for ordering of triples, knowing only that $0 < s < r \le B$ say, with $O(B)$ words of memory, $B + O(\log B)$ random number calls, and $O(B^2 \log B)$ overall operations.

Generating Functions

Suppose $x_n = x_{n-r} + x_{n-s}$ over any field F. We define the *generating function*

$$G(t) = \sum_{n \geq 0} x_n t^n \ .$$

It is easy to see that

$$G(t) = \frac{G_0(t)}{P(t)} \ ,$$

where $G_0(t)$ is a polynomial of degree (at most) $r-1$ defined by the initial values $x_0, \ldots, x_{r-1}$, and

$$P(t) = 1 - t^s - t^r$$

is defined by the recurrence.

The generating function can be used to obtain various theoretical results (expected values, correlations, etc).

Some Definitions

For the sake of simplicity, when considering generating functions we only consider polynomials over GF(2). *We won't repeat this statement every time!*

Recall that, for polynomials u, v over GF(2), $2u = 2v = 0$. This implies that $u - v = u + v$ and $(u+v)^2 = u^2 + v^2$.

We say that a polynomial $P(x)$ is *reducible* if it has nontrivial factors; otherwise it is *irreducible.*

If $P(x)$ is irreducible of degree $r > 1$, then $\mathrm{GF}(2^r) \approx Z_2[x]/(P(x))$. If x is generator for the multiplicative group of $Z_2[x]/(P(x))$, then we say that $P(x)$ is *primitive.*

Since the multiplicative group has order $2^r - 1$, we need to know the complete factorization of $2^r - 1$ in order to test if an irreducible polynomial is primitive. However, if r is a *Mersenne exponent*, i.e. $2^r - 1$ is prime, then irreducibility implies primitivity.

Some Well-Known Results

The following results can be found in texts such as Lidl, Menezes et al. Here μ is the Möbius function, and ϕ is Euler's phi function.

1. $x^{2^n} + x$ is the product of all irreducible polynomials of degree $d|n$. For example,

   $$x^8 + x = x(1+x)(1+x+x^3)(1+x^2+x^3) \ .$$

2. Let J_n be the number of irreducible polynomials of degree n. Then

   $$\sum_{d|n} dJ_d = 2^n \ \text{ and } \ J_n = \frac{1}{n}\sum_{d|n} 2^d \mu(n/d) \ .$$

   In particular, if n is prime then $J_n = (2^n - 2)/n$.

3. The number of primitive polynomials of degree n is $P_n = \phi(2^n - 1)/n \leq J_n$.

   In particular, if n is a Mersenne exponent, then $P_n = J_n = (2^n - 2)/n$.

The Reciprocal Polynomial

If $P(x) = \sum_{j=0}^{r} a_j x^j$ is a polynomial of degree r, with $a_0 \neq 0$, then

$$P_R(x) = x^r P(1/x) = \sum_{j=0}^{r} a_j x^{r-j}$$

is the *reciprocal polynomial.* Clearly $P(x)$ is irreducible (or primitive) iff $P_R(x)$ is irreducible (or primitive).

In particular, if

$$P(x) = 1 + x^s + x^r \ , \ \ 0 < s < r$$

is a trinomial, then the reciprocal trinomial is

$$P_R(x) = 1 + x^{r-s} + x^r \ .$$

If it is convenient, we can assume that $s \leq r/2$ (else consider the reciprocal trinomial).

When applied to random number generation, the reciprocal polynomial generates the sequence in reverse order.

Irreducible Trinomials

For applications such as random number generation, we want irreducible (or better, primitive) polynomials of high degree r and with a small number of nonzero terms. Let's restrict attention to *trinomials* of the form

$$P(x) = P_{r,s}(x) = 1 + x^s + x^r \ , \quad 0 < s < r \ .$$

Swan's Theorem

Swan (1962) determines the parity of the number of irreducible factors by an argument involving the discriminant (actually, Swan's Theorem is a rediscovery of 19th century results).

If r is an odd prime, then Swan's theorem implies that $P_{r,s}(x)$ has an even number of irreducible factors (and hence is reducible) if $r = \pm 3 \bmod 8$ and $s \neq 2$ or $r - 2$.

The condition on s can not be omitted, e.g. $x^{29} + x^2 + 1$ is irreducible.

Some Primitive Trinomials

In Table 1 (next slide) we give a table of primitive trinomials $x^r + x^s + 1$ where r is a Mersenne exponent (i.e. $2^r - 1$ is prime). We assume that $0 < 2s \leq r$ (so $x^r + x^{r-s} + 1$ is not listed).

Results for smaller r can be found on my website `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/` or in the literature.

The entries for $r \leq 3021377$ have been checked by running at least two different programs on different machines.

During this checking process, the entry with

$$r = 859433, \quad s = 170340$$

was found. This was surprising, because Kumada *et al.* claimed to have searched the whole range for $r = 859433$. It turns out that Kumada *et al.* missed this entry because of a bug in their sieving routine!

Some Primitive Trinomials cont.

In Table 1, $x^r + x^s + 1$ is primitive over GF(2).

The entries for $r = 132049$ are by Heringa *et al.* The second entry for $r = 859433$ is from Kumada *et al.*. The other entries were found by Brent, Larvala and Zimmermann.

The last line, for $r = 24036583$, is for the Mersenne prime announced in May 2004. No serious attempt has yet been made to find a primitive trinomial of this degree.

r	s
132049	7000, 33912, 41469, 52549, 54454
756839	215747, 267428, 279695
859433	170340, 288477
3021377	361604, 1010202
6972593	3037958
24036583	?

Table 1: Some primitive trinomials

Almost Primitive Trinomials

There is a large gap between some of the Mersenne exponents r for which primitive trinomials exist. For example, there are none in the interval $859433 < r < 3021377$, even though there are three Mersenne exponents in this interval. This is because Swan's theorem rules out about half of the Mersenne exponents – it rules out most exponents of the form $r = \pm 3 \bmod 8$.

The usual solution is to consider pentanomials (five nonzero terms) instead of trinomials, but a faster alternative is to use *almost primitive* trinomials (introduced by Brent and Zimmermann).

Definition. We say that a polynomial $P(x)$ is *almost primitive* with *exponent* r and *increment* $\delta < r$ if $P(x)$ has degree $r + \delta$, $P(0) \neq 0$, and $P(x)$ has a primitive factor of degree r.

Almost Primitive Trinomials cont.

For example, the trinomial $x^{16} + x^3 + 1$ is almost primitive with exponent 13 and increment 3, because

$$x^{16} + x^3 + 1 = (x^3 + x^2 + 1)D(x),$$

where

$$D(x) = x^{13}+x^{12}+x^{11}+x^9+x^6+x^5+x^4+x^2+1$$

is primitive.

In Table 2 (next slide) we list some almost primitive trinomials. In fact, we give at least one for each Mersenne exponent $r < 10^7$ for which no primitive trinomial of degree r exists.

The search is complete for Mersenne exponent degrees $r < 10^7$.

r	δ	s	f
13	3	3	7
19	3	3	7
61	5	17	31
107	2	8, 14, 17	3
2203	3	355	7
4253	8	1806 1960	255 85
9941	3	1077	7
11213	6	227	63
21701	3	6999, 7587	7
86243	2	2288	3
216091	12	42930	3937
1257787	3	74343	7
1398269	5	417719	21
2976221	8	1193004	85

Table 2:
Some almost primitive trinomials over GF(2).
$x^{r+\delta} + x^s + 1$ has a primitive factor of degree r;
δ is minimal; $2s \le r + \delta$; the period $\rho = (2^r - 1)f$.

A Larger Example

Consider the entry $r = 216091$, $\delta = 12$, $s = 42930$ in Table 2. We have

$$x^{216103} + x^{42930} + 1 = S(x)D(x),$$

where

$$S(x) = x^{12} + x^{11} + x^5 + x^3 + 1,$$

and $D(x)$ is a (dense) primitive polynomial of degree 216091.

The factor $S(x)$ of degree 12 splits into a product of two primitive polynomials,

$$x^5 + x^4 + x^3 + x + 1 \text{ and}$$

$$x^7 + x^5 + x^4 + x^3 + x^2 + x + 1.$$

The contribution to the period from these factors is $f = \mathrm{LCM}(2^5 - 1, 2^7 - 1) = 3937$.

Application to RNGs

If $T(x) = x^{r+\delta} + x^s + 1$ is almost primitive with exponent r, we can use the corresponding linear recurrence

$$U_n = U_{n-r-\delta} + U_{n-s} \bmod 2^w$$

as a generalized Fibonacci (or LFSR) random number generator.

The period will be a multiple of $2^r - 1$ provided $U_0, \ldots, U_\delta$ are odd. This condition ensures that a recurrence with lags $\le \delta$ (corresponding to the degree-δ factor of $T(x)$) is not satisfied.

Initialization

When discussing requirements we mentioned the problem of initialization. Many random number generators fall down here. One example is the generator recommended in Volume 2 of Knuth's *The Art of Computer Programming*, third edition (fixed at the ninth printing, Jan. 2002).

Using the theory of generating functions (or, less efficiently, linear algebra), it is possible to "skip ahead" n terms in the sequence for a generalized Fibonacci or LFSR RNG in $O(\log n)$ arithmetic operations. The idea is similar to that of forming n-th powers by squaring and multiplication.

This technique allows us to guarantee that different seeds give different sequences for all practical purposes (e.g. use segments of the full sequence separated by more than 10^{18} numbers). With care we can ensure that the first random number in each sequence behaves like a random function of the seed.

Lazy Initialization

Consider a generalized Fibonacci RNG based on a primitive trinomial of degree r, using addition mod 2^w. There are $W = 2^{r(w-1)}(2^r - 1)$ ways to initialize r words of w bits so that not all words are even. Each cycle has length $L = 2^{w-1}(2^r - 1)$. Thus there are

$$C = W/L = 2^{(r-1)(w-1)}$$

distinct cycles. Provided $w \geq 2$ and r is not too small, C is very large.

If we initialize the RNG twice using an independent RNG, it is *extremely* unlikely that the two sequences will be in the same cycle. In fact, the probability is $1/C = 2^{-(r-1)(w-1)}$. Thus, in practice this "lazy" method of initialization is adequate whenever we want to generate different random sequences (e.g. on different processors of a parallel computer).

Improving Generators

We have shown how three-term generators with large periods can be obtained. Now we show how their statistical properties can be improved. This is necessary because the three-term property necessarily implies a deterministic relation between certain triples in the sequence.

Improving a RNG by "Decimation"

If $(x_0, x_1, \ldots)$ is generated by a 3-term recurrence, we can obtain a (hopefully better) sequence $(y_0, y_1, \ldots)$ by defining $y_j = x_{jp}$, where $p > 1$ is a suitable constant. In other words, use every p-th number and discard the others. ("Decimation" refers to $p = 10$, though the term is inaccurate because with $p = 10$ we discard 90% and not 10% of the numbers.)

Consider the case $w = 1$ (i.e. LFSR generators). If $p = 2$, the y_j satisfy the same 3-term recurrence! However, if $p = 3$ we do get something new. Using generating functions, it is easy to show that the y_j satisfy a 5-term recurrence.

Ziff's "Fast Decimation"

For LFSR generators, Ziff has given a method for obtaining 5-term recurrences in some cases of decimation with $p = 3$, 5, and 7. Note that the case $p = 3$ and some of the cases $p = 5$ are not recommended because of certain 4-point correlations. The remaining cases should be much better for RNG than the original 3-term recurrences, and not much slower.

For example, consider a 3-term generator G with lags r and s, where $7|(2r - s)$. Ziff's "rule (5b)" says that the 7-decimation of G is a 5-term generator with lags $r, (5r + s)/7, (r + 3s)/7, s$.

For example, the 7-decimation of the 3-term generator with lags 3021377 and 1010202 is a 5-term generator with lags 3021377, 2302441, 1010202 and 864569. The 5-term generator is slightly slower than the 3-term generator, but has much better statistical properties in 3 and 4 dimensions.

Lüscher's Decimation by Blocks

An improvement (suggested by Lüscher) over simple decimation is decimation by blocks. Choose a blocksize, say B. (For 3-term generators choose $B \le r$.) Use one block of B numbers, then discard the next $p - 1$ blocks. Roughly speaking, the justification is that any correlations between blocks should be "washed out" by a mixing process if p is sufficiently large.

Decimation by blocks is slower than Ziff's method, e.g. for $p = 7$ it is 3 to 4 times slower. However, it is more general, as it can be applied to any random number generator, and it should give better statistical results.

Combining Generators by Addition

We can combine some number K of generalized Fibonacci generators by addition (mod 2^w) (similarly for $\oplus$). If each component generator is defined by a primitive trinomial

$$T_k(x) = x^{r_k} + x^{s_k} + 1 \ ,$$

with distinct prime degrees r_k, then the combined generator has period

$$2^{w-1} \prod_{k=1}^{K} (2^{r_k} - 1)$$

and satisfies a 3^K-term linear recurrence.

Because the speed of the combined generator decreases like $1/K$, we would probably take $K \le 3$ in practice. The case $K = 2$ seems to be better (and more efficient) than "decimation" with $p = 3$. It should also be better, and of comparable efficiency, to decimation with $p \le 7$ by Ziff's method.

Combining by Shuffling

Suppose we have two pseudo-random sequences $X = (x_0, x_1, \ldots)$ and $Y = (y_0, y_1, \ldots)$. We can use a buffer V of size B say, fill the buffer using the sequence X, then use the sequence Y to generate indices into the buffer. If the index is j then the random number generator returns $V[j]$ and replaces $V[j]$ by the next number in the X sequence [Knuth, Algorithm M].

In other words, we use one generator to shuffle the output of another generator. This seems to be as good (and about as fast) as combining two generators by addition. B should not be too small.

Combining by Shrinking

Coppersmith *et al* suggested using one sequence to "shrink" another sequence.

Suppose we have two pseudo-random sequences $(x_0, x_1, \ldots)$ and $(y_0, y_1, \ldots)$, $\;x_i, y_i \in \mathrm{GF}(2)$. Suppose $y_i = 1$ for $i = s_0, s_1, \ldots$ Define a sequence $(z_0, z_1, \ldots)$ to be the subsequence $(x_{s_0}, x_{s_1}, \ldots)$ of $(x_0, x_1, \ldots)$. In other words, one sequence of bits (y_i) is used to decide whether to "accept" or "reject" elements of another sequence (x_i). This is sometimes called "irregular decimation".

Combining two sequences by shrinking is slower than combining the sequences by $\oplus$, but is less amenable to analysis based on linear algebra or generating functions, so is preferable in applications where the sequence needs to be unpredictable (e.g. in cryptography – see Menezes *et al*, §6.3).

Faster Shrinking

Methods for generating cryptographically strong sequences, such as Coppersmith's "shrinking" generator, give only one random bit at a time. This is slow. To speed up such generators, we could try to generate w bits at a time, but this could introduce a cryptographic weakness!

To see why this might be so, suppose we can find distinct w-bit outputs $z_\alpha, z_\beta, z_\gamma$ such that

$$z_\alpha = z_\beta \oplus z_\gamma \; .$$

This will occur by chance with probability 2^{-w}. Thus, if $w = 1$ the relation is probably just a coincidence, but if $w = 64$ it is extremely unlikely to be a coincidence, so it gives us some information about the "hidden" sequences (x_i) and (y_i).

Comments on Some Available RNGs

Many implementations of linear congruential generators are available. They usually have a period which is too short and do not give good d-dimensional uniformity for $d > 3$ (Marsaglia).

Marsaglia dislikes Tausworthe RNGs because they fail the "birthday spacings" test. He recommends add/subtract with carry/borrow ("Very Long Period") generators, but these may also fail the birthday spacings test or the gap test.

Shchur, Heringa and Blöte showed that generalized Fibonacci generators based on primitive trinomials of small degree fail a 1D random-walk test. To avoid this, we recommend using large degree and/or combining at least two generators.

The idea of combining generators is not original – it has been suggested by several people, although apparently it has seldom been used in practice.

Blocking of Output

It is easy to vectorise both linear congruential and generalized Fibonacci RNGs. This is only useful if batches of random numbers are generated together. Thus, the interface to a library routine should allow an array of random numbers to be returned.

This comment applies even on a scalar workstation, because returning an array of random numbers reduces subroutine-call overheads.

ranut

Many random number generators based on primitive trinomials have been documented in the literature, but the implementations are usually for a fixed trinomial. The choice of trinomial involves a tradeoff. Larger values of the degree r give generators with better statistical properties, but the space requirements and the time required for initialization increase with r. Thus, the optimal choice of a trinomial depends on the particular application and computing resources available.

We have implemented an open-source uniform pseudo-random number generator `ranut` that automatically selects a primitive or almost primitive trinomial whose degree depends on the size of the working space allocated by the user, and then implements a generalized Fibonacci generator based on that trinomial.

`ranut` has been tested with Marsaglia's *Diehard* package and no anomalous results have been observed.

ranut continued

`ranut` does not attempt to combine more than one generator (there are too many possibilities and tradeoffs). The user can easily combine two or more generators (some or all of these can be implemented by `ranut`) if this is desired.

Combining generators by addition is extremely straightforward. Since `ranut` gives floating-point numbers in $[0, 1)$, it is only necessary to add these and discard the integer part of the result.

Some Suggestions

Although it is easy to suggest bad random number generators, there is no "best" generator. Here are a few possibilities for good generators based on primitive trinomials $1 + x^s + x^r$.

1. Combine two generalized Fibonacci generators based on primitive trinomials with $(r, s) = (1279, 418)$ and $(2281, 1029)$.

The period is $> 10^{1000}$.

The generator is fast, and uses only 28KB memory for 64-bit random numbers. The numbers satisfy a 9-term recurrence, so we expect good statistical properties.

2. Combine $(r, s) = (109, 2)$, $(127, 30)$, and $(521, 158)$ by addition (and perhaps $\oplus$).

The period is $> 10^{227}$. Uses only 6KB memory, so should run in cache (important for speed).

The output satisfies a 27-term recurrence, so we expect excellent statistical properties.

Some RNGs continued

3. Combine two generators by shrinking, where the "X" generator uses $(r, s) = (44497, 21034)$, the "Y" generator uses $(r, s) = (3217, 576)$. and the Y generator "shrinks" the output of the X generator. For maximum cryptographic strength, this should be done one bit at a time. (For speed the X and Y generators can be implemented to generate more than one bit at a time, and buffer their output one bit at a time.)

To trade off speed against cryptographic strength, the X generator could produce bytes or words instead of single bits.

The period is $> 10^{14000}$. The memory requirement ranges from 6KB (for 1-bit output) to 350KB (for 64-bit output).

Some RNGs continued

4.
The generator

$$x_n = x_{n-3021377} \oplus x_{n-2302441} \oplus x_{n-1010202} \oplus x_{n-864569}$$

obtained by 7-decimation from a primitive trinomial of degree $r = 3021377$. This should be very fast. See Ziff's paper for a "one line" implementation if an array of size 2^{22} is used.

The period is

$$2^r - 1 > 10^{909525} ,$$

and

$$r^4/4! \approx 3.47 \times 10^{24} ,$$

so the statistical properties should be adequate so long as no more than 10^{24} numbers are used.

References

[1] S. L. Anderson, Random number generators on vector supercomputers and other advanced architectures, *SIAM Rev.* **32** (1990), 221–251.

[2] R. P. Brent, *Some uniform and normal random number generators,* version 1.03 (January 2002). Available from `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/random.html`

[3] R. P. Brent, On the periods of generalized Fibonacci recurrences, *Math. Comp.* **63** (1994), 389–401. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub133.html`

[4] R. P. Brent, Random number generation and simulation on vector and parallel computers (extended abstract), *Proc. Fourth Euro-Par Conference*, *LNCS* **1470**, Springer-Verlag, Berlin, 1998, 1–20. ···`/pub185.html`

[5] R. P. Brent, S. Larvala and P. Zimmermann, A fast algorithm for testing irreducibility of trinomials mod 2 ..., *Math. Comp.* 72 (2003), 1443–1452. ···`/pub199.html`

[6] R. P. Brent, S. Larvala and P. Zimmermann, A primitive trinomial of degree 6972593, *Math. Comp.*, to appear. ···`/pub214.html`

[7] R. P. Brent and P. Zimmermann, Random number generators with period divisible by a Mersenne prime, *LNCS* **2667**, Springer-Verlag, Berlin, 2003, 1–10. ···`/pub211.html` .

[8] R. P. Brent and P. Zimmermann, Algorithms for finding almost irreducible and almost primitive trinomials, in *Primes and Misdemeanours* ..., Fields Institute, Toronto, to appear. ···`/pub212.html` .

[9] A. Compagner and A. Hoogland, Maximum-length sequences, cellular automata, and random numbers, *J. Computational Physics* **71** (1987), 391–428.

[10] D. Coppersmith, H. Krawczyk and Y. Mansour, The shrinking generator, *Proc. CRYPTO'93, LNCS* **773** (1994), 22–39.

[11] A. M. Ferrenberg, D. P. Landau and Y. J. Wong, Monte Carlo simulations: Hidden errors from "good" random number generators, *Phys. Rev. Lett.* **69** (1992), 3382–3384.

[12] GIMPS, The Great Internet Prime Search, `http://www.mersenne.org/`

[13] S. W. Golomb, *Shift register sequences,* Holden-Day, San Francisco, 1967.

[14] J. R. Heringa, H. W. J. Blöte and A. Compagner. New primitive trinomials of Mersenne-exponent degrees for random-number generation, *International J. of Modern Physics C* **3** (1992), 561–564.

[15] F. James, A review of pseudorandom number generators, *Computer Physics Communications* **60** (1990), 329–344.

[16] D. E. Knuth, *The art of computer programming, Volume 2: Seminumerical algorithms* (third ed.), Addison-Wesley, Menlo Park, CA, 1998.

[17] T. Kumada, H. Leeb, Y. Kurita and M. Matsumoto, New primitive t-nomials (t = 3, 5) over GF(2) whose degree is a Mersenne exponent, *Math. Comp.* **69** (2000), 811–814. Corrigenda: *ibid* **71** (2002), 1337–1338.

[18] Y. Kurita and M. Matsumoto, Primitive t-nomials ($t = 3, 5$) over GF(2) whose degree is a Mersenne exponent ≤ 44497, *Math. Comp.* **56** (1991), 817–821.

[19] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge Univ. Press, Cambridge, second edition, 1994.

[20] M. Lüscher, A portable high-quality random number generator for lattice field simulations, *Computer Physics Communications* **79** (1994), 100-110.

[21] G. Marsaglia, Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sci. USA* **61** (1968), 25-28.

[22] G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface* (edited by L. Billard), Elsevier Science Publishers B. V. (North-Holland), 1985, 3–10.

[23] G. Marsaglia and L. H. Tsay, Matrices and the structure of random number sequences, *Linear Algebra Appl.* **67** (1985), 147–156.

[24] G. Marsaglia and A. Zaman, A new class of random number generators, *Ann. of Appl. Prob.* **1** (1991), 462-480.

[25] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997. `http://cacr.math.uwaterloo.ca/hac/`

[26] J. F. Reiser, *Analysis of additive random number generators,* Ph. D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1977. Also Technical Report STAN-CS-77-601.

[27] E. R. Rodemich and H. Rumsey, Jr., Primitive trinomials of high degree, *Math. Comp.* **22** (1968), 863–865.

[28] L. N. Shchur and P. Butera, *The RANLUX generator: resonances in a random walk test*, May 1998, preprint `hep-lat/9805017` available from `http://xxx.anl.gov`.

[29] L. N. Shchur, J. R. Heringa and H. W. J. Blöte, Simulation of a directed random-walk model: the effect of pseudo-random-number correlations, *Physica A* **241** (1997), 579.

[30] W. Stahnke, Primitive binary polynomials, *Math. Comp.* **27** (1973), 977–980.

[31] R. G. Swan, Factorization of polynomials over finite fields, *Pacific J. Math.* **12** (1962), 1099–1106.

[32] R. C. Tausworthe, Random numbers generated by linear recurrence modulo two, *Math. Comp.* **19** (1965), 201–209.

[33] S. Tezuka, Efficient and portable combined Tausworthe random number generators, *ACM Trans. on Modeling and Computer Simulation* **1** (1991), 99–112.

[34] I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical tests for random numbers in simulations, *Phys. Rev. Lett.* **73** (1994), 2513–2516.

[35] E. J. Watson, Primitive polynomials (mod 2), *Math. Comp.* **16** (1962), 368–369.

[36] N. Zierler, Primitive trinomials whose degree is a Mersenne exponent, *Inform. and Control* **15** (1969), 67–69.

[37] R. M. Ziff, Four-tap shift-register-sequence random-number generators, *Computers in Physics* **12** (1998), 385–392.