Fast and Reliable Random Number Generators for Scientific Computing

Richard P. Brent[1]

Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK

random@rpbrent.co.uk

Abstract. Fast and reliable pseudo-random number generators are required for simulation and other applications in Scientific Computing. We outline the requirements for good uniform random number generators, and describe a class of generators having very fast vector/parallel implementations with excellent statistical properties. We also discuss the problem of initialising random number generators, and consider how to combine two or more generators to give a better (though usually slower) generator.

1 Introduction

Monte Carlo methods are of great importance in simulation [36], computational finance, numerical integration, computational physics [13, 24], etc. Due to Moore's Law and increases in parallelism, the statistical quality of random number generators is becoming even more important than in the past. A program running on a supercomputer might use 10^9 random numbers per second over a period of many hours (or even months in some cases), so 10^{16} or more random numbers might contribute to the result. Small correlations or other deficiencies in the random number generator could easily lead to spurious effects and invalidate the results of the computation, see e.g. [13, 34].

Applications require random numbers with various distributions (e.g. normal, exponential, Poisson, ...) but the algorithms used to generate these random numbers almost invariably require a good uniform random number generator. A notable exception is Wallace's method [7, 39] for normally distributed numbers. In this paper we consider only the generation of uniformly distributed numbers. Usually we are concerned with *real* numbers u_n that are intended to be uniformly distributed on the interval $[0, 1)$. Sometimes it is convenient to consider *integers* U_n in some range $0 \le U_n < m$. In this case we require $u_n = U_n/m$ to be (approximately) uniformly distributed.

Pseudo-random numbers generated in a deterministic fashion on a digital computer can not be truly random. What is required is that finite segments

 rpb217

of the sequence $(u_0, u_1, \cdots)$ behave in a manner indistinguishable from a truly random sequence. In practice, this means that they pass all statistical tests that are relevant to the problem at hand. Since the problems to which a library routine will be applied are not known in advance, random number generators in subroutine libraries should pass a number of stringent statistical tests (and not fail any) before being released for general use.

Random numbers generated by physical sources are available [1, 2, 15, 23, 37, 38]. However, there are problems in generating such numbers sufficiently fast, and experience with them is insufficient to be confident of their statistical properties. Thus, for the present, we recommend treating such physical sources of random numbers with caution. They can be used to initialise (and perhaps periodically reinitialise) deterministic generators, and can be combined with deterministic generators by the algorithms considered in §6. In the following we restrict our attention to deterministic pseudo-random number generators.

A sequence $(u_0, u_1, \cdots)$ depending on a finite state must eventually be periodic, i.e. there is a positive integer ρ such that $u_{n+\rho} = u_n$ for all sufficiently large n. The minimal such ρ is called the *period*.

In §§2–3 we consider desiderata for random number generators. In §§4–5, we describe one popular class of random number generators. In §6 we discuss how to combine two or more generators to give a (hopefully) better generator. Finally, in §7 we briefly mention some implementations.

2 Requirements for good random number generators

Requirements for a good pseudo-random number generator have been discussed in many surveys, e.g. [5, 9, 11, 17, 20, 22, 25]. Due to space limitations we can not cover all aspects of random number generation here, but we shall attempt to summarize and comment briefly on the most important requirements. Of course, some of the requirements listed below may be irrelevant in certain applications. For example, there is often no need to skip ahead (§2.4). In some applications, such as Monte Carlo integration, it may be preferable to use numbers that definitely do not behave like random numbers: they are "quasi-random" rather than random [32].

2.1 Uniformity

The sequence of random numbers should pass statistical tests for uniformity of distribution. This is usually easy for deterministic generators implemented in software. For physical/hardware generators, the well-known technique of Von Neumann, or similar but more efficient techniques [16], can be used to extract uniform bits from a sequence of independent but possibly biased bits.

2.2 Independence

Subsequences of the full sequence $(u_0, u_1, \cdots)$ should be independent. Random numbers are often used to sample a d-dimensional space, so the sequence of

d-tuples $(u_{dn}, u_{dn+1}, \ldots, u_{dn+d-1})$ should be uniformly distributed in the d-dimensional cube $[0, 1]^d$ for all "small" values of d (certainly for all $d \le 6$). For random number generators on parallel machines, the sequences generated on each processor should be independent.

2.3 Long period

As mentioned above, a simulation might use 10^{16} random numbers. In such a case the period ρ must exceed 10^{16}. For many generators there are strong correlations between $u_0, u_1, \cdots$ and $u_m, u_{m+1}, \cdots$, where $m \approx \rho/2$ (and similarly for other simple fractions of the period). Thus, in practice the period should be *much* larger than the number of random numbers that will ever be used. A good rule of thumb is to use at most $\rho^{1/2}$ numbers. In fact, there are reasons, related to the *birthday spacings test* [28], for using at most $\rho^{1/3}$ numbers: see [22, §6].

2.4 Ability to skip ahead

If a simulation is to be run on a machine with several processors, or if a large simulation is to be performed on several independent machines, it is essential to ensure that the sequences of random numbers used by each processor are disjoint. Two methods of subdivision are commonly used. Suppose, for example, that we require 4 disjoint subsequences for a machine with 4 processors. One processor could use the subsequence $(u_0, u_4, u_8, \cdots)$, another the subsequence $(u_1, u_5, u_9, \cdots)$, etc. For efficiency each processor should be able to "skip over" the terms that it does not require.

Alternatively, processor j could use the subsequence $(u_{m_j}, u_{m_j+1}, \cdots)$, where the indices m_0, m_1, m_2, m_3 are sufficiently widely separated that the (finite) subsequences do not overlap, but this requires some efficient method of generating u_m for large m without generating all the intermediate values $u_1, \ldots, u_{m-1}$.

For generators satisfying a linear recurrence, it is possible to skip ahead by forming high powers of the appropriate matrix (see [22, §3.5] for details). However, it is not so well known that more efficient methods exist using generating functions. Essentially, we can replace matrix multiplications by polynomial multiplications. Multiplying two $r \times r$ matrices is much more expensive than multiplying two polynomials modulo a polynomial of degree r. Details are given in [4] and an implementation that is practical for r of the order of 10^6 is available [3].

2.5 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, so it is important that short sequences produced with different seeds are uncorrelated.

For example, suppose that a random number generator with seed s produces a sequence $(u_1^{(s)}, u_2^{(s)}, u_3^{(s)}, \ldots)$. If we use m different seeds $s_1, s_2, \ldots, s_m$ and generate n numbers from each seed, we get an $m \times n$ array U with elements $U_{i,j} = u_j^{(s_i)}$. We do not insist that the seeds are random – they could for example be consecutive integers.

Packages such as Marsaglia's *Diehard* [26] typically test a 1-D array of random numbers. We can generate a 1-D array by concatenating the rows (or columns) of U. Irrespective of how this is done, we would hope that the random numbers would pass the standard statistical tests. However, many current generators fail because they were intended for the case $m = 1$ (or small) and n large [14, 18]. The other extreme is m large and $n = 1$. In this case we expect $u_1^{(s)}$ to behave like a pseudo-random *function* of s.

2.6 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 expected by chance) from $(u_0, u_1, \ldots, u_{n-1})$, unless n is so large that the prediction is infeasible.

At first sight it appears that 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 seems a wise precaution to use an unpredictable generator if the cost of doing so is not too high. We discuss techniques for this in §6.

Strictly speaking, unpredictability implies uniformity, independence, and a (very) long period. However, it seems worthwhile to state these simpler requirements separately.

2.7 Efficiency

It should be possible to implement the method efficiently so that only a few arithmetic operations are required to generate each random number, all vector/parallel capabilities of the machine are used, and overheads such as those for subroutine calls are minimal. Of course, efficiency tends to conflict with other requirements such as unpredictability, so a tradeoff is often involved.

2.8 Repeatability

For testing and development it is useful to be able to repeat a run with *exactly* the same sequence of random numbers as was used in an earlier run. This is usually easy if the sequence is restarted from the beginning (u_0). It may not be so easy if the sequence is to be restarted from some other value, say u_m for a large integer m, because this requires saving the state information associated with the random number generator.

2.9 Portability

Again, for testing and development purposes, it is useful to be able to generate *exactly* the same sequence of random numbers on two different machines, possibly with different wordlengths. This was more difficult to achieve in the past than it is nowadays, when nearly all computers have wordlengths of 32 or 64 bits, and their floating-point arithmetic satisfies the IEEE 754 standard.

3 Equidistribution

We should comment on the concept of *equidistribution*, which we have not listed as one of our requirements. Definitions and examples can be found in [22, §4.2] and in [30, §1.2].

Consider concatenating the leading v bits from k consecutive random numbers. According to [30], a random number generator is said to be k-distributed to v-bit accuracy if each of the 2^{kv} possible combinations of bits occurs the same number of times over a full period of the random number generator, except that the sequence of all zero bits is allowed to occur once less often. Some generators with period $\rho = 2^r$ or $2^r - 1$ can be proved to satisfy this condition for $kv \le r$. This is fine in applications such as Monte Carlo integration. However, the probability that a periodic but otherwise random sequence will satisfy the condition is vanishingly small. If we perform a "chi-squared" test on the output of a k-distributed generator, the test will be failed because the value of χ^2 is too *small* !

To give a simply analogy: if I toss a fair coin 100 times, I expect to get about 50 heads and 50 tails, but I would be mildly surprised to get exactly the same number of heads as tails (the probability of this occurring is about 0.08). If (with the aid of a computer) I toss a fair coin 10^{12} times, I should be *very* surprised to get exactly the same number of heads as tails. (For $2n$ tosses, the probability of an equal number of heads and tails occurring is about $1/\sqrt{n\pi}$.) This is another reason for using at most $\sqrt{\rho}$ numbers from the full period of length ρ (compare §2.3).

4 Generalized Fibonacci generators

In this section we describe a popular class of random number generators. For various generalizations, see [22].

Given a circular buffer of length r words (or bits), we can generate pseudo-random numbers from a linear or nonlinear recurrence

$$u_n = f(u_{n-1}, u_{n-2}, \ldots, u_{n-r}) \ .$$

For speed it is desirable that $f(u_{n-1}, u_{n-2}, \ldots, u_{n-r})$ depends explicitly on only a small number of its r arguments. An important case is the class of "generalized Fibonacci" or "lagged Fibonacci" random number generators [17].

Marsaglia [25] considers generators $F(r, s, \theta)$ that satisfy

$$U_n = U_{n-r} \,\theta\, U_{n-s} \mod m$$

for fixed "lags" r and s ($r > s > 0$) and $n \geq r$. Here m is a modulus (typically 2^w if w is the wordlength in bits), and θ is some binary operator, e.g. addition, subtraction, multiplication or "exclusive or". We abbreviate these operators by $+, -, *$ and $\oplus$ respectively. Generators using $\oplus$ are also called "linear feedback shift register" (LFSR) generators or "Tausworthe" generators. Usually U_n is normalised to give a floating-point number $u_n = U_n/m \in [0, 1)$.

It is possible to choose lags r, s so that the period ρ of the generalized Fibonacci generators $F(r, s, +)$ is a large prime p or a small multiple of such a prime. Typically, the period of the least-significant bit is p; because carries propagate from the least-significant bit into higher-order bits, the overall period is usually $2^{w-1}\rho$ for wordlength w. For example, [9, Table 1] gives several pairs (r, s) with $r > 10^6$. (The notation in [9] is different: $r + \delta$ corresponds to our r.)

There are several ways to improve the performance of generalized Fibonacci generators on statistical tests such as the Birthday Spacings and Generalized Triple tests [25, 28]. The simplest is to include small odd integer multipliers α and β in the generalized Fibonacci recurrence, e.g.

$$U_n = \alpha U_{n-r} + \beta U_{n-s} \mod m \;.$$

Other ways to improve statistical properties (at the expense of speed) are to include more terms in the linear recurrence [19], to discard some members of the sequence [24], or to combine two or three generators in various ways (see §6).

With suitable choice of lags (r, s), the generalised Fibonacci generators satisfy the requirements of uniformity, long period, efficiency, and ability to skip ahead. Because there are only three terms in the recurrence, each number depends on only two previous numbers, so there may be difficulty satisfying the requirement for independence, at least if r is small. Because they are based on a linear recurrence, they do *not* satisfy the requirement for unpredictability. In §6 we show how to overcome these difficulties.

5 Short-term and long-term properties

When considering pseudo-random number generators, it is useful to consider their short-term and long-term properties separately.

short-term means properties that can be tested by inspection of relatively short segments of the full cycle. For example, suppose that a uniform random number generator is used to simulate throws of a dice. If consective sixes never occur in the output (or occur with probability much lower than expected), then the generator is faulty, and this can be tested by inspection of the results of a few hundred simulated throws. For a more subtle example, consider a single-bit LFSR generator of the form discussed in §4, with largest lag r, and period $2^r - 1$. We can form an $r \times r$ matrix from r^2 consecutive bits of output. This matrix is nonsingular (considered as a matrix over GF(2)). However, the probability that a random $r \times r$ matrix over GF(2) is nonsingular is strictly less than 1 (about 0.289 for large r, see [8]). Thus, by inspecting $O(r^2)$ consecutive bits of the output, we can detect a short-term nonrandomness.

long-term means properties that can be tested by inspection of a full cycle (or a significant fraction of a full cycle). For example, a uniform random number generator might have a small bias, so the expected output is $1/2 + \varepsilon$ instead of $1/2$. This could be detected by taking a sample of size slightly larger than $1/\varepsilon^2$.

Generalized Fibonacci generators based on primitive trinomials generally have good long-term properties, but bad short-term properties. To improve the short-term properties we can use *tempering* (transforming the output vectors by a carefully-chosen linear transformation), as suggested by Matsumoto and Kurita (see [30] and [22, §4.5]), or the other devices mentioned in §4.

6 Improving generators

In this section we consider how generators that suffer some defects can be improved.

6.1 Improving a generator 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.

Consider the case $F(r, s, \oplus)$ with $w = 1$ (LFSR) and $p = 3$. (If $p = 2$, the y_j satisfy the same 3-term recurrence as the x_j.)

Using generating functions, it is easy to show that the y_j satisfy a 5-term recurrence. For example, if $x_n = x_{n-1} \oplus x_{n-127}$, then $y_n = y_{n-1} \oplus y_{n-43} \oplus y_{n-85} \oplus y_{n-127}$. A more elementary approach for $p \le 7$ is given in [40].

A possible improvement over simple decimation is decimation by blocks [24].

6.2 Combining generators by addition or xor

We can combine some number K of generalized Fibonacci generators by addition (mod 2^w). 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 at least $2^{w-1} \prod_{k=1}^{K} (2^{r_k} - 1)$ 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 $p = 3$.

Alternatively, we can combine K generalized Fibonacci generators by bitwise "exclusive or" operating on w-bit words. This has the advantage of mixing different algebraic operations (assuming that addition mod 2^w is used in the generalized Fibonacci recurrence). Note that the least-significant bits will be the same for both methods.

6.3 Combining by shuffling

Suppose that 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 [17, 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.

6.4 Combining by shrinking

Coppersmith *et al* [12] suggested using one sequence to "shrink" another sequence.

Suppose we have two pseudo-random sequences $(x_0, x_1, \ldots)$ and $(y_0, y_1, \ldots)$, where $y_i \in \mathrm{GF}(2)$. Suppose $y_i = 1$ for $i = i_0, i_1, \ldots$ Define a sequence $(z_0, z_1, \ldots)$ to be the subsequence $(x_{i_0}, x_{i_1}, \ldots)$ 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" (compare §6.1).

Combining two sequences by shrinking is slower than combining the sequences by $+$ or $\oplus$, 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. Note that it is dangerous to take the x_i to be larger than a single bit. For example, if we tried to speed up the combination process by taking x_i to be a whole word, then the cryptographic security could be compromised.

7 Implementations

Several good random number generators are available. Following is a small sample, not intended to be exhaustive: Matsumoto and Nishimura's *Mersenne twister*, based on a primitive trinomial of degree 19937 with tempering to improve short-term properties [30]; L'Ecuyer's *maximally equidistributed combined LFSR generators* [21]; and the author's *ranut* (Fortran) and *xorgens* (C) generators [3]. The *xorgens* generators are simple, fast, have passed all statistical tests applied so far, and are based on a generalization of a recent idea of Marsaglia [27]. They are related to LFSR generators [6], but do not use trinomials, and can be implemented faster than most other LFSR generators because the degree r can be chosen to be a multiple of 64.

To close with a word of warning: all pseudo-random number generators fail some statistical tests – this is inevitable, since they are generated deterministically. It is ultimately the user's responsibility to ensure that the pseudo-random numbers appear sufficiently random for the application at hand.

References

1. Anonymous, *Random number generation and testing*, NIST, December 2000. `http://csrc.nist.gov/rng/` .
2. Anonymous, *True randomness on request*, University of Geneva and id Quantique, May 2004, `http://www.randomnumber.info` .
3. R. P. Brent, *Some uniform and normal random number generators,* ranut version 1.03 (January 2002) and xorgens version 2.01 (August 2004). Available from `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/random.html` .
4. R. P. Brent, On the periods of generalized Fibonacci recurrences, *Math. Comp.* **63** (1994), 389–401.
5. R. P. Brent, Random number generation and simulation on vector and parallel computers, *LNCS* **1470**, Springer-Verlag, Berlin, 1998, 1–20.
6. R. P. Brent, Note on Marsaglia's xorshift random number generators, *J. of Statistical Software* **11**, 5 (2004), 1–4. `http://www.jstatsoft.org` .
7. R. P. Brent, Some comments on C. S. Wallace's random number generators, *Computer J.*, to appear. Preprint at `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub213.html` .
8. R. P. Brent and B. D. McKay, Determinants and ranks of random matrices over Z_m , *Discrete Mathematics* **66** (1987), 35–49. $\cdots$`/pub094.html` .
9. R. P. Brent and P. Zimmermann, Random number generators with period divisible by a Mersenne prime, *LNCS* **2667**, Springer-Verlag, Berlin, 2003, 1–10. Preprint at $\cdots$`/pub211.html` .
10. R. P. Brent and P. Zimmermann, Algorithms for finding almost irreducible and almost primitive trinomials, in *Primes and Misdemeanours: Lectures in Honour of the Sixtieth Birthday of Hugh Cowie Williams*, Fields Institute, Toronto, 2004, 91–102. Preprint at $\cdots$`/pub212.html` .
11. P. D. Coddington, Random number generators for parallel computers, *The NHSE Review* **2** (1996). `http://nhse.cs.rice.edu/NHSEreview/RNG/PRNGreview.ps` .
12. D. Coppersmith, H. Krawczyk and Y. Mansour, The shrinking generator, *Advances in Crpyptology – CRYPTO'93, LNCS* **773**, Springer-Verlag, Berlin, 1994, 22–39.
13. A. M. Ferrenberg, D. P. Landau and Y. J. Wong, Monte Carlo simulations: hidden errors from "good" random number generators, *Phys. Review Letters* **69** (1992), 3382–3384.
14. P. Gimeno, *Problem with* `ran_array`, personal communication, 10 Sept. 2001.
15. M. Jakobsson, E. Shriver, B. K. Hillyer and A. Juels, A practical secure physical random bit generator, *Proc. Fifth ACM Conference on Computer and Communications Security*, November 1998. `http://www.bell-labs.com/user/shriver/random.html` .
16. A. Juels, M. Jakobsson, E. Shriver and B. K. Hillyer, How to turn loaded dice into fair coins, *IEEE Trans. on Information Theory* **46**, 2000, 911–921.
17. D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third edition), Addison-Wesley, Menlo Park, CA, 1998.
18. D. E. Knuth, *A better random number generator*, January 2002, `http://www-cs-faculty.stanford.edu/~knuth/news02.html` .
19. 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.
20. P. L'Ecuyer, Random numbers for simulation, *Comm. ACM* **33**, 10 (1990), 85–97.

21. P. L'Ecuyer, Tables of maximally equidistributed combined LFSR generators, *Mathematics of Computation* **68** (1999), 261–269.
22. P. L'Ecuyer, Random number generation, Chapter 2 of *Handbook of Computational Statistics*, J. E. Gentle, W. Haerdle, and Y. Mori, eds., Springer-Verlag, 2004, 35–70.
23. W. Knight, Prize draw uses heat for random numbers, *New Scientist*, 17 August 2004. `ttp://www.newscientist.com/news/news.jsp?id=ns99996289` .
24. M. Lüscher, A portable high-quality random number generator for lattice field theory simulations, *Computer Physics Communications* **79** (1994), 100–110.
25. G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface*, Elsevier Science Publishers B. V.,1985, 3–10.
26. G. Marsaglia, *Diehard*, 1995. Available from `http://stat.fsu.edu/~geo/` .
27. G. Marsaglia, Xorshift RNGs, *J. of Statistical Software* **8**, 14 (2003), 1–9. `http://www.jstatsoft.org` .
28. G. Marsaglia and W. W. Tsang, Some difficult-to-pass tests of randomness *J. of Statistical Software* **7**, 3 (2002), 1–9. `http://www.jstatsoft.org` .
29. M. Mascagni, M. L. Robinson, D. V. Pryor and S. A. Cuccaro, Parallel pseudorandom number generation using additive lagged-Fibonacci recursions, *Lecture Notes in Statistics* **106**, Springer-Verlag, Berlin, 1995, 263–277.
30. M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Transactions on Modeling and Computer Simulations* **8**, 1998, 3–30. Also `http://www.math.keio.ac.jp/~matumoto/emt.html` .
31. 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/` .
32. H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, CBMS-NSF Regional Conference Series in Applied Mathematics **63**, SIAM, Philadelphia, 1992.
33. W. P. Petersen, Lagged Fibonacci series random number generators for the NEC SX-3, *Internat. J. High Speed Computing* **6** (1994), 387–398.
34. 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.
35. S. Tezuka, P. L'Ecuyer, and R. Couture, On the add-with-carry and subtract-with-borrow random number generators, *ACM Trans. on Modeling and Computer Simulation* **3** (1993), 315–331.
36. I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical tests for random numbers in simulations, *Phys. Review Letters* **73** (1994), 2513–2516.
37. J. Walker, *HotBits: Genuine random numbers, generated by radioactive decay*, Fourmilab Switzerland, April 2004. `http://www.fourmilab.ch/hotbits/`.
38. C. S. Wallace, Physically random generator, *Computer Systems Science and Engineering* **5** (1990), 82–88.
39. C. S. Wallace, Fast pseudo-random generators for normal and exponential variates, *ACM Trans. on Mathematical Software* **22** (1996), 119–127.
40. R. M. Ziff, Four-tap shift-register-sequence random-number generators, *Computers in Physics* **12** (1998), 385–392.