Random Number Generation and Simulation on Vector and Parallel Computers

Richard P. Brent

Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
rpb@comlab.ox.ac.uk
www.comlab.ox.ac.uk/oucl/people/richard.brent.html

Abstract. Pseudo-random numbers are often required for simulations performed on parallel computers. The requirements for parallel random number generators are more stringent than those for sequential random number generators. As well as passing the usual sequential tests on each processor, a parallel random number generator must give different, independent sequences on each processor. We consider the requirements for a good parallel random number generator, and discuss generators for the uniform and normal distributions. We also describe a new class of generators for the normal distribution (based on a proposal by Wallace). These generators can give very fast vector or parallel implementations. Implementations of uniform and normal generators on vector and vector/parallel computers are discussed.

1 Introduction

Pseudo-random numbers have been used in Monte Carlo calculations since the earliest days of digital computers [32]. In this paper we are concerned here with random number generators (RNGs) on fast, modern computers – typically either vector processors or parallel computers using vector or pipelined RISC processors. What we say about vector processors often applies to pipelined RISC processors with a memory hierarchy (the vector registers of a vector processor corresponding to the first-level cache of a RISC processor).

With the increasing speed of vector processors and parallel computers, considerable attention must be paid to the quality of random number generators. A program running on a supercomputer might use 10^8 random numbers per second over a period of many hours or even months in the case of QCD calculations, so 10^{14} 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.

Applications require random numbers with various distributions (uniform, normal, exponential, binomial, Poisson, etc.) but the algorithms used to generate these random numbers usually require a good uniform random number

D. Pritchard, J. Reeve (Eds.): Euro-Par'98, LNCS 1470, pp. 1–20, 1998.

generator – see for example [2,5,14,24,34,39]. In this paper we consider the generation of uniformly and normally distributed numbers.

Pseudo-random numbers generated in a deterministic fashion on a digital computer can not be truly random. What is required is that finite segments of the sequence behave in a manner indistinguishable from a truly random sequence. In practice, this means that they pass all statistical tests which 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.

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

Following are some of the more important requirements for a good uniform pseudo-random number generator and its implementation in a subroutine library (the modifications for a normal generator are obvious) –

- *Uniformity.* The sequence of random numbers should pass statistical tests for uniformity of distribution. In one dimension this is easy to achieve. Most generators in common use are provably uniform (apart from discretisation due to the finite wordlength) when considered over their full period.
- *Independence.* Subsequences of the full sequence $u_0, u_1, \cdots$ should be independent. For example, members of the even subsequence $u_0, u_2, u_4, \cdots$ should be independent of their odd neighbours $u_1, u_3, \cdots$. Thus, the sequence of pairs (u_{2n}, u_{2n+1}) should be uniformly distributed in the unit square. More generally, 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 "reasonable" values of d (certainly for all $d \leq 6$).
- *Long Period.* As mentioned above, a simulation might use 10^{14} random numbers. In such a case the period p must exceed 10^{14}. For many generators there are strong correlations between $u_0, u_1, \cdots$ and $u_m, u_{m+1}, \cdots$, where $m = p/2$ (and similarly for other simple fractions of the period). Thus, in practice the period should be *much* larger than the number of random numbers which will ever be used.
- *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 [22]. 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.
- *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. In practice it will be expensive to simulate a long wordlength on a machine with a short wordlength, but the converse should be easy – a machine with a long wordlength (say

$w = 64$) should be able to simulate a machine with a smaller wordlength without loss of efficiency.

- *Disjoint Subsequences.* 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. This partitioning method is sometimes called "decimation" or "leapfrog" [11]. For efficiency each processor should be able to "skip over" the terms which 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. This requires some efficient method of generating u_m for large m without generating all the intermediate values $u_1, \ldots, u_{m-1}$.
- *Efficiency.* It should be possible to implement the method efficiently so that only a few arithmetic operations are required to generate each random number and all vector/parallel capabilities of the machine are used. To minimise subroutine call overheads, the random number routine should return an array of (optionally) several numbers at a time.

Several recent reviews [4,6,11,16,22,24,28,33] of uniform random number generators are available. The most important conclusion regarding uniform generators is that good ones may exist, but are hard to find [33]. Linear congruential generators with a "short" period (less than say 2^{48}) are certainly to be avoided. Generalised (or "lagged") Fibonacci generators using the "exclusive or" operation are also to be avoided; other generalised Fibonacci generators may be satisfactory if the lags are sufficiently large (if they use the operation of addition then the lags should probably be at least 1000). See, for example, [12, Table 2]. Our recommendation, implemented as `RANU4` on Fujitsu VP2200 and VPP300 vector/parallel processors, is a generalised Fibonacci generator with very large lags, e.g. $(79500, 132049)$ (see [21]), and careful initialisation which avoids any initial atypical behaviour and ensures disjoint sequences on parallel processors. For further details see [6].

In the interests of conserving space, we refer the reader to the reviews cited above for uniform generators, and concentrate our attention on the less often considered, but still important, case of normal random number generation on vector/parallel processors. "Classical" generators are considered in §2, and an interesting new class of "Wallace" generators [40] is considered in §3.

We do not attempt to cover the important topic of *testing* random number generators intended for use on vector/parallel computers. A good, recent survey of this topic is [12]. The user should always remember that a deterministic sequence of pseudo-random numbers can not truly be random; all that testing can do is inspire confidence that a generator is indistinguishable from random in a particular application [37,38]. In practice, testing is essential to cull bad generators, but can not provide any guarantees.

2 Normal RNGs based on Uniform RNGs

In this section we consider some "classical" methods for generating normally distributed pseudo-random numbers. The methods all assume a good source of uniform random numbers which is transformed in some manner to a sequence of normally distributed random numbers. The transformation is not necessarily one to one.

The most well-known and widely used methods for generating normally distributed random variables on sequential machines [2,5,14,20,24,26] involve the use of different approximations on different intervals, and/or the use of "rejection" methods [14,24], so they often do not vectorise well. Simple, "old-fashioned" methods may be preferable. In §2.1 we describe two such methods, and in §§2.2–2.3 we consider their efficient implementation on vector processors, and give the results of implementations on a Fujitsu VP2200/10. In §§2.4–2.5 we consider some other methods which are popular on serial machines, and show that they are unlikely to be competitive on vector processors.

2.1 Some Normal Generators

Assume that a good uniform random number generator which returns uniformly distributed numbers in the interval $[0, 1)$ is available, and that we wish to sample the normal distribution with mean μ and variance σ^2. We can generate two independent, normally distributed numbers x, y by the following old algorithm due to Box and Muller [31] *(Algorithm B1):*

1. Generate independent uniform numbers $u, v \in [0, 1)$.
2. Set $r \leftarrow \sigma\sqrt{-2\ln(1-u)}$.
3. Set $x \leftarrow r\sin(2\pi v) + \mu$ and $y \leftarrow r\cos(2\pi v) + \mu$.

The proof that the algorithm is correct is similar to the proof of correctness of the Polar method given in Knuth [24].

Algorithm B1 is a reasonable choice on a vector processor if vectorised square root, logarithm and trigonometric function routines are available. Each normally distributed number requires 1 uniformly distributed number, 0.5 square roots, 0.5 logarithms, and 1 sin or cos evaluation. Vectorised implementations of the Box-Muller method are discussed in §2.2.

A variation of Algorithm B1 is the *Polar* method of Box, Muller and Marsaglia described in Knuth [24, Algorithm P]:

1. Generate independent uniform numbers $x, y \in [-1, 1)$.
2. Set $s \leftarrow x^2 + y^2$.
3. If $s \in (0, 1)$ then go to step 4 else go to step 1 (i.e. *reject* x and y).
4. Set $r \leftarrow \sigma\sqrt{-2\ln(s)/s}$, and return $rx + \mu$ and $ry + \mu$.

It is easy to see that, at step 4, (x, y) is uniformly distributed in the unit circle, so s is uniformly distributed in $[0, 1)$.

A proof that the values returned by Algorithm P are independent, normally distributed random numbers (with mean μ and variance σ^2) is given in Knuth [24]. On average, step 1 is executed $4/\pi$ times, so each normally distributed number requires $4/\pi \simeq 1.27$ uniform random numbers, 0.5 divisions, 0.5 square roots, and 0.5 logarithms. Compared to Algorithm B1, we have avoided the sin and cos computation at the expense of more uniform random numbers, 0.5 divisions, and the cost of implementing the acceptance/rejection process. This can be done using a vector gather. Vectorised implementations of the Polar method are discussed in §2.3.

2.2 Vectorised Implementation of the Box-Muller Method

We have implemented the Box-Muller method (Algorithm B1 above) and several refinements (B2, B3) on a Fujitsu VP2200/10 vector processor at the Australian National University. The implementations all return double-precision real results, and in cases where approximations to sin, cos, sqrt and/or ln have been made, the absolute error is considerably less than 10^{-10}. Thus, statistical tests using less than about 10^{20} random numbers should not be able to detect any bias due to the approximations. The calling sequences allow for an array of random numbers to be returned. This permits vectorisation and amortises the cost of a subroutine call over the cost of generating many random numbers.

Our method B2 is the same as B1, except that we replace calls to the library sin and cos by an inline computation, using a fast, but sufficiently accurate, approximation (for details see [7]).

Times, in machine cycles per normally distributed number, for methods B1, B2 (and other methods described below) are given in Table 1. In all cases the generalised Fibonacci random number generator RANU4 (described in [6]) was used to generate the required uniform random numbers, and a large number of random numbers were generated, so that vector lengths were long. RANU4 generates a uniformly distributed random number in 2.2 cycles on the VP2200/10. (The cycle time of the VP2200/10 at ANU is 3.2 nsec, and two multiplies and two adds can be performed per clock cycle, so the peak speed is 1.25 Gflop.)

The Table gives the total times and also the estimated times for the four main components:

1. ln computation (actually 0.5 times the cost of one ln computation since the times are per normal random number generated).
2. sqrt computation (actually 0.5 times).
3. sin or cos computation.
4. other, including uniform random number generation.

The results for method B1 show that the sin/cos and ln computations are the most expensive (65% of the total time). Method B2 is successful in reducing the sin/cos time from 33% of the total to 19%.

Table 1. Cycles per normal random number

component	B1	B2	B3	P1	P2	R1
ln	13.1	13.1	7.1	13.1	7.1	0.3
sqrt	8.8	8.8	1.0	8.8	1.0	0.0
sin/cos	13.8	6.6	6.6	0.0	0.0	0.0
other	5.9	5.6	11.6	11.9	13.8	35.1
total	41.6	34.1	26.3	33.8	21.9	35.4

In Method B2, the computation of $\sqrt{-\ln(1-u)}$ consumes 64% of the time. An obvious way to reduce this time is to use a fast approximation to the function

$$f(u) = \sqrt{-\ln(1-u)},$$

just as we used a fast approximation to sin and cos to speed up method B1. However, this is difficult to accomplish with sufficient accuracy, because the function $f(u)$ is badly behaved at both endpoints of the unit interval. Method B3 overcomes this difficulty in the following way.

1. We approximate the function

$$g(u) = u^{-1/2} f(u) = \sqrt{\frac{-\ln(1-u)}{u}},$$

rather than $f(u)$. Using the Taylor series for $\ln(1-u)$, we see that $g(u) = 1 + u/4 + \cdots$ is well-behaved near $u = 0$.
2. The approximation to $g(u)$ is only used in the interval $0 \le u \le \tau$, where $\tau < 1$ is suitably chosen. For $\tau < u < 1$ we use the slow but accurate library ln and sqrt routines.
3. We make a change of variable of the form $v = (\alpha u + \beta)/(\gamma u + \delta)$, where $\alpha, \ldots, \delta$ are chosen to map $[0, \tau]$ to $[-1, 1]$, and the remaining degrees of freedom are used to move the singularities of the function $h(v) = g(u)$ as far away as possible from the region of interest (which is $-1 \le v \le 1$). To be more precise, let ρ be a positive parameter. Then we can choose

$$\tau = 1 - \left(\frac{\rho}{\rho+2}\right)^2,$$

$$v = (\rho+1)\left(\frac{(\rho+2)u - 2}{2(\rho+1) - (\rho+2)u}\right),$$

and the singularities of $h(v)$ are at $\pm(\rho+1)$.

For simplicity, we choose $\rho = 1$, which experiment shows is close to optimal on the VP2200/10. Then $\tau = 8/9$, $v = (6u-4)/(4-3u)$, and $h(v)$ has singularities at $v = \pm 2$, corresponding to the singularities of $g(u)$ at $u = 1$ and $u = \infty$. A polynomial of the form $h_0 + h_1 v + \cdots + h_{15} v^{15}$ can be used to approximate

$h(v)$ with absolute error less than 2×10^{-11} on $[-1, 1]$. About 30 terms would be needed if we attempted to approximate $g(u)$ to the same accuracy by a polynomial on $[0, \tau]$. We use polynomial approximations which are close to minimax approximations. These may easily be obtained by truncating Chebyshev series, as described in [10].

It appears that this approach requires the computation of a square root, since we really want $f(u) = u^{1/2}g(u)$, not $g(u)$. However, a trick allows this square root computation to be avoided, at the expense of an additional uniform random number generation (which is cheap) and a few arithmetic operations. Recall that u is a uniformly distributed random variable on $[0, 1)$. We generate *two* independent uniform variables, say u_1 and u_2, and let $u \leftarrow \max(u_1, u_2)^2$. It is easy to see that u is in fact uniformly distributed on $[0, 1)$. However, $u^{1/2} = \max(u_1, u_2)$ can be computed without calling the library sqrt routine. To summarise, a non-vectorised version of method B3 is:

```
1. Generate independent uniform numbers $u_1, u_2, u_3 \in [0, 1)$.
2. Set $m \leftarrow \max(u_1, u_2)$ and $u \leftarrow m^2$.
3. If $u > 8/9$ then
   3.1. set $r \leftarrow \sigma\sqrt{-\ln(1 - u)}$ using library routines, else
   3.2. set $v \leftarrow (6u - 4)/(4 - 3u)$, evaluate $h(v)$ as described above, and
        set $r \leftarrow \sigma m h(v)$.
4. Evaluate $s \leftarrow \sin(2\pi u_3 - \pi)$ and $c \leftarrow \cos(2\pi u_3 - \pi)$ as in [7].
5. Return $\mu + cr\sqrt{2}$ and $\mu + sr\sqrt{2}$, which are independent, normal
   random numbers with mean $\mu$ and standard deviation $\sigma$.
```

Vectorisation of method B3 is straightforward, and can take advantage of the "list vector" technique on the VP2200. The idea is to gather those $u > 8/9$ into a contiguous array, call the vectorised library routines to compute an array of $\sqrt{-\ln(1 - u)}$ values, and scatter these back. The gather and scatter operations introduce some overhead, as can be seen from the row labelled "other" in the Table. Nevertheless, on the VP2200, method B3 is about 23% faster than method B2, and about 37% faster than the straightforward method B1. These ratios could be different on machines with more (or less) efficient implementations of scatter and gather.

Petersen [35] gives times for normal and uniform random number generators on a NEC SX-3. His implementation *normalen* of the Box-Muller method takes 55.5 nsec per normally distributed number, i.e. it is 2.4 times faster than our method B1, and 1.51 times faster than our method B3. The model of SX-3 used by Petersen has an effective peak speed of 2.75 Gflop, which is 2.2 times the peak speed of the VP2200/10. Considering the relative speeds of the two machines and the fact that the SX-3 has a hardware square root function, our results are encouraging.

2.3 Vectorised Implementation of the Polar Method

The times given in Table 1 for methods B1–B3 can be used to predict the best possible performance of the Polar method (§2.1). The Polar method avoids the computation of sin and cos, so could gain up to 6.6 cycles per normal random number over method B3. However, we would expect the gain to be less than this because of the overhead of a vector gather caused by use of a rejection method. A straightforward vectorised implementation of the Polar method, called method P1, was written to test this prediction. The results are shown in Table 1. 13.8 cycles are saved by avoiding the sin and cos function evaluations, but the overhead increases by 6.0 cycles, giving an overall saving of 7.8 cycles or 19%. Thus, method P1 is about the same speed as method B2, but not as fast as method B3.

Encouraged by our success in avoiding most ln and sqrt computations in the Box-Muller method (see method B3), we considered a similar idea to speed up the Polar method. Step 4 of the Polar method (§2.1) involves the computation of $\sqrt{-2\ln(s)/s}$, where $0 < s < 1$. The function has a singularity at $s = 0$, but we can approximate it quite well on an interval such as $[1/9, 1]$, using a method similar to that used to approximate the function $g(u)$ of §2.2.

Inspection of the proof in Knuth [24] shows that step 4 of the Polar method can be replaced by

```
4a. Set $r \leftarrow \sigma\sqrt{-2\ln(u)/s}$,
    and return $rx + \mu$ and $ry + \mu$
```

where u is any uniformly distributed variable in $(0, 1]$, provided u is independent of $\arctan(y/x)$. In particular, we can take $u = 1 - s$. Thus, omitting the constant factor $\sigma\sqrt{2}$, we need to evaluate $\sqrt{-\ln(1-s)/s}$, but this is just $g(s)$, and we can use exactly the same approximation as in §2.2. This gives us method P2. To summarise, a non-vectorised version of method P2 is:

```
1. Generate independent uniform numbers $x, y \in [-1, 1)$.
2. Compute $s \leftarrow x^2 + y^2$.
3. If $s \geq 1$ then go to step 1 (i.e. reject $x$ and $y$) else go to step 4.
4. If $s > 8/9$ then
   4.1. set $r \leftarrow \sigma\sqrt{-\ln(1-s)/s}$ using library routines, else
   4.2. set $v \leftarrow (6s-4)/(4-3s)$, evaluate $h(v)$ as described in §2.2, and
        set $r \leftarrow \sigma h(v)$.
5. Return $xr\sqrt{2} + \mu$ and $yr\sqrt{2} + \mu$, which are independent, normal
   random numbers with mean $\mu$ and standard deviation $\sigma$.
```

To vectorise steps 1-3, we simply generate vectors of x_j and y_j values, compute $s_j = x_j^2 + y_j^2$, and compress by omitting any triple (x_j, y_j, s_j) for which $s_j \geq 1$. This means that we can not predict in advance how many normal random numbers will be generated, but this problem is easily handled by introducing a level of buffering.

The second-last column of Table 1 gives results for method P2. There is a saving of 11.9 cycles or 35% compared to method P1, and the method is 17% faster than the fastest version of the Box-Muller method (method B3). The cost of logarithm and square root computations is only 37% of the total, the remainder being the cost of generating uniform random numbers (about 13%)

and the cost of the rejection step and other overheads (about 50%). On the VP2200/10 we can generate more than 14 million normally distributed random numbers per second.

2.4 The Ratio Method

The Polar method is one of the simplest of a class of rejection methods for generating random samples from the normal (and other) distributions. Other examples are given in [2,5,14,24]. It is possible to implement some of these methods in a manner similar to our implementation of method P2. For example, a popular method is the Ratio Method of Kinderman and Monahan [23] (also described in [24], and improved in [26]). In its simplest form, the Ratio Method is given by *Algorithm R:*

1. Generate independent uniform numbers $u, v \in [0, 1)$.
2. Set $x \leftarrow \sqrt{8/e}(v - \frac{1}{2})/(1 - u)$.
3. If $-x^2 \ln(1 - u) > 4$ then go to step 1 (i.e. reject x) else go to step 4.
4. Return $\sigma x + \mu$.

Algorithm R returns a normally distributed random number using on average $8/\sqrt{\pi e} \simeq 2.74$ uniform random numbers and 1.37 logarithm evaluations. For the proof of correctness, and various refinements which reduce the number of logarithm evaluations, see [23,24,26]. The idea of the proof is that x is normally distributed if the point (u, v) lies inside a certain closed curve C which in turn is inside the rectangle $[0, 1] \times [-\sqrt{2/e}, +\sqrt{2/e}]$. Step 3 rejects (u, v) if it is outside C.

The function $\ln(1 - u)$ occurring at step 3 has a singularity at $u = 1$, but it can be evaluated using a polynomial or rational approximation on some interval $[0, \tau]$, where $\tau < 1$, in much the same way as the function $g(u)$ of §2.2.

The refinements added by Kinderman and Monahan [23] and Leva [26] avoid most of the logarithm evaluations. The following step is added:

2.5. If $P_1(u, v)$ then go to step 4
else if $P_2(u, v)$ then go to step 1
else go to step 3.

Here $P_1(u, v)$ and $P_2(u, v)$ are easily-computed conditions. Geometrically, P_1 corresponds to a region R_1 which lies inside C, and P_2 corresponds to a region R_2 which encloses C, but R_1 and R_2 have almost the same area. Step 3 is only executed if (u, v) lies in the borderline region $R_2 \backslash R_1$.

Step 2.5 can be vectorised, but at the expense of several vector scatter/gather operations. Thus, the saving in logarithm evaluations is partly cancelled out by an increase in overheads. The last column (R1) of Table 1 gives the times for our implementation on the VP2200. As expected, the time for the logarithm computation is now negligible, and the overheads dominate. In percentage terms the times are:

1%	logarithm computation (using the library routine),
17%	uniform random number computation,
23%	scatter and gather to handle borderline region,
59%	step 2.5 and other overheads.

Although disappointing, the result for the Ratio method is not surprising, because the computations and overheads are similar to those for method P2 (though with less logarithm computations), but only half as many normal random numbers are produced. Thus, we would expect the Ratio method to be slightly better than half as fast as method P2, and this is what Table 1 shows.

2.5 Other Methods

On serial machines our old algorithm GRAND [5] is competitive with the Ratio method. In fact, GRAND is the fastest of the methods compared by Leva [26]. GRAND is based on an idea of Von Neumann and Forsythe for generating samples from a distribution with density function $c\exp(-h(x))$, where $0 \le h(x) \le 1$:

1. Generate a uniform random number $x \in [0, 1)$, and set $u_0 \leftarrow h(x)$.
2. Generate independent uniform random numbers $u_1, u_2, \ldots \in [0, 1)$ until the first $k > 0$ such that $u_{k-1} < u_k$.
3. If k is odd then return x,
   else reject x and go to step 1.

A proof of correctness is given in Knuth [24].

It is hard to see how to implement GRAND efficiently on a vector processor. There are two problems –

1. k is not bounded, even though its expected value is small. Thus, a sequence of gather operations seems to be required. The result would be similar to Petersen's implementation [35] of a generator for the Poisson distribution (much slower than his implementation for the normal distribution).
2. Because of the restriction $0 \le h(x) \le 1$, the area under the normal curve $\exp(-x^2/2)/\sqrt{2\pi}$ has to be split into different regions from which samples are drawn with probabilities proportional to their areas. This complicates the implementation of the rejection step.

For these reasons we would expect a vectorised implementation of GRAND to be even slower than our implementation of the Ratio method. Similar comments apply to other rejection methods which use an iterative rejection process and/or several different regions.

3 Vectorisation of Wallace's Normal RNG

Recently Wallace [40] proposed a new class of pseudo-random generators for normal variates. These generators do not require a stream of uniform pseudo-random numbers (except for initialisation) or the evaluation of elementary functions such as log, sqrt, sin or cos (needed by the Box-Muller and Polar methods).

The crucial observation is that, if x is an n-vector of normally distributed random numbers, and A is an $n \times n$ orthogonal matrix, then $y = Ax$ is another n-vector of normally distributed numbers. Thus, given a pool of nN normally distributed numbers, we can generate another pool of nN normally distributed numbers by performing N matrix-vector multiplications. The inner loops are very suitable for implementation on vector processors. The vector lengths are proportional to N, and the number of arithmetic operations per normally distributed number is proportional to n. Typically we choose n to be small, say $2 \le n \le 4$, and N to be large.

Wallace implemented variants of his new method on a scalar RISC workstation, and found that its speed was comparable to that of a fast uniform generator, and much faster than the "classical" methods considered in §2. The same performance relative to a fast uniform generator is achievable on a vector processor, although some care has to be taken with the implementation (see §3.6).

In §3.1 we describe Wallace's new methods in more detail. Some statistical questions are considered in §§3.2–3.5. Aspects of implementation on a vector processor are discussed in §3.6, and details of an implementation on the VP2200 and VPP300 are given in §3.7.

3.1 Wallace's Normal Generators

The idea of Wallace's new generators is to keep a pool of nN normally distributed pseudo-random variates. As numbers in the pool are used, new normally distributed variates are generated by forming appropriate combinations of the numbers which have been used. On a vector processor N can be large and the whole pool can be regenerated with only a small number of vector operations[1].

The idea just outlined is the same as that of the generalised Fibonacci generators for uniformly distributed numbers – a pool of random numbers is transformed in an appropriate way to generate a new pool. As Wallace [40] observes, we can regard the uniform, normal and exponential distributions as maximum-entropy distributions subject to the constraints:

$0 \le x \le 1$ (uniform)
$E(x^2) = 1$ (normal)
$E(x) = 1,\ x \ge 0$ (exponential).

We want to combine $n \ge 2$ numbers in the pool so as to satisfy the relevant constraint, but to conserve no other statistically relevant information. To simplify notation, suppose that $n = 2$ (there is no problem in generalising to $n > 2$). Given two numbers x, y in the pool, we could satisfy the "uniform" constraint by forming

$$x' \leftarrow (x + y) \bmod 1,$$

and this gives the family of generalised Fibonacci generators [6].

[1] The process of regenerating the pool will be called a "pass".

We could satisfy the "normal" constraint by forming

$$\begin{pmatrix} x' \\ y' \end{pmatrix} \leftarrow A \begin{pmatrix} x \\ y \end{pmatrix},$$

where A is an orthogonal matrix, for example

$$A = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

or

$$A = \frac{1}{5} \begin{pmatrix} 4 & 3 \\ -3 & 4 \end{pmatrix}.$$

Note that this generates two new pseudo-random normal variates x' and y' from x and y, and the constraint

$${x'}^2 + {y'}^2 = x^2 + y^2$$

is satisfied because A is orthogonal.

Suppose the pool of previously generated pseudo-random numbers contains $x_0, \ldots, x_{N-1}$ and $y_0, \ldots, y_{N-1}$. Let $\alpha, \ldots, \delta$ be integer constants. These constants might be fixed throughout, or they might be varied (using a subsidiary uniform random number generator) each time the pool is regenerated.

One variant of Wallace's method generates $2N$ new pseudo-random numbers $x'_0, \ldots, x'_{N-1}$ and $y'_0, \ldots, y'_{N-1}$ using the recurrence

$$\begin{pmatrix} x'_j \\ y'_j \end{pmatrix} = A \begin{pmatrix} x_{\alpha j + \gamma \bmod N} \\ y_{\beta j + \delta \bmod N} \end{pmatrix} \tag{1}$$

for $j = 0, 1, \ldots, N-1$. The vectors x' and y' can then overwrite x and y, and be used as the next pool of $2N$ pseudo-random numbers. To avoid the copying overhead, a double-buffering scheme can be used.

3.2 Desirable Constraints

In order that all numbers in the old pool (x, y) are used to generate the new pool (x', y'), it is essential that the indices

$$\alpha j + \gamma \bmod N$$

and

$$\beta j + \delta \bmod N$$

give permutations of $\{0, 1, \ldots, N-1\}$ as j runs through $\{0, 1, \ldots, N-1\}$. A necessary and sufficient condition for this is that

$$\mathrm{GCD}(\alpha, N) = \mathrm{GCD}(\beta, N) = 1\ . \tag{2}$$

For example, if N is a power of 2, then any odd α and β may be chosen.

The orthogonal matrix A must be chosen so each of its rows has at least two nonzero elements, to avoid repetition of the same pseudo-random numbers. Also, these nonzeros should not be too small.

For implementation on a vector processor it would be efficient to take $\alpha = \beta = 1$ so vector operations have unit strides. However, statistical considerations indicate that unit strides should be avoided. To see why, suppose $\alpha = 1$. Thus, from (1),

$$x'_j = a_{0,0} x_{j+\gamma \bmod N} + a_{0,1} y_{\beta j + \delta \bmod N} ,$$

where $|a_{0,0}|$ is not very small. The sequence (z_j) of random numbers returned to the user is

$$\begin{array}{l} x_0, \ldots, x_{N-1}, y_0, \ldots, y_{N-1}, \\ x'_0, \ldots, x'_{N-1}, y'_0, \ldots, y'_{N-1}, \ldots \end{array}$$

so we see that z_n is strongly correlated with $z_{n+\lambda}$ for $\lambda = 2N - \gamma$.

Wallace [40] suggests a "vector" scheme where $\alpha = \beta = 1$ but γ and δ vary at each pass. This is certainly an improvement over keeping γ and δ fixed. However, there will still be correlations over segments of length $O(N)$ in the output, and these correlations can be detected by suitable statistical tests. Thus, we do not recommend the scheme for a library routine, although it would be satisfactory in many applications.

We recommend that α and β should be different, greater than 1, and that γ and δ should be selected randomly at each pass to reduce any residual correlations.

For similar reasons, it is desirable to use a different orthogonal matrix A at each pass. Wallace suggests randomly selecting from two predefined 4×4 matrices, but there is no reason to limit the choice to two[2]. We prefer to choose "random" 2×2 orthogonal matrices with rotation angles not too close to a multiple of $\pi/2$.

3.3 The Sum of Squares

As Wallace points out, an obvious defect of the schemes described in §§3.1–3.2 is that the sum of squares of the numbers in the pool is fixed (apart from the effect of rounding errors). For independent random normal variates the sum of squares should have the chi-squared distribution χ^2_ν, where $\nu = nN$ is the pool size.

To overcome this defect, Wallace suggests that one pseudo-random number from each pool should not be returned to the user, but should be used to approximate a random sample S from the χ^2_ν distribution. A scaling factor can be introduced to ensure that the sum of squares of the ν values in the pool (of which $\nu - 1$ are returned to the user) is S. This only involves scaling the matrix A, so the inner loops are essentially unchanged.

[2] Caution: if a finite set of predefined matrices is used, the matrices should be multiplicatively independent over $GL(n, R)$. (If $n = 2$, this means that the rotation angles (mod 2π) should be independent over the integers.) In particular, no matrix should be the inverse of any other matrix in the set.

There are several good approximations to the χ^2_ν distribution for large ν. For example,

$$2\chi^2_\nu \simeq \left(x + \sqrt{2\nu - 1}\right)^2 \ , \tag{3}$$

where x is $N(0,1)$. More accurate approximations are known [1], but (3) should be adequate if ν is large.

3.4 Restarting

Unlike the case of generalised Fibonacci uniform random number generators [8], there is no well-developed theory to tell us what the period of the output sequence of pseudo-random normal numbers is. Since the size of the state-space is at least 2^{2wN}, where w is the number of bits in a floating-point fraction and $2N$ is the pool size (assuming the worst case $n = 2$), we would expect the period to be at least of order 2^{wN} (see Knuth [24]), but it is difficult to guarantee this. One solution is to restart after say 1000N numbers have been generated, using a good uniform random number generator with guaranteed long period combined with the Box-Muller method to refill the pool.

3.5 Discarding Some Numbers

Because each pool of pseudo-random numbers is, strictly speaking, determined by the previous pool, it is desirable not to return all the generated numbers to the user[3]. If $f \geq 1$ is a constant parameter[4], we can return a fraction $1/f$ of the generated numbers to the user and "discard" the remaining fraction $(1 - 1/f)$. The discarded numbers are retained internally and used to generate the next pool. There is a tradeoff between independence of the numbers generated and the time required to generate each number which is returned to the user. Our tests (described in §3.7) indicate that $f \geq 3$ is satisfactory.

3.6 Vectorised Implementation

If the recurrence (1) is implemented in the obvious way, the inner loop will involve index computations modulo N. It is possible to avoid these computations. Thus $2N$ pseudo-random numbers can be generated by $\alpha + \beta - 1$ iterations of a loop of the form

```
do j = low, high
xp(j) = A00*x(alpha*j + jx) + A01*y(beta*j + jy)
yp(j) = A10*x(alpha*j + jx) + A11*y(beta*j + jy)
enddo
```

[3] Similar remarks apply to some uniform pseudo-random number generators [24,27].

[4] We shall call f the "throw-away" factor.

where `low, high, jx,` and `jy` are integers which are constant within the loop but vary between iterations of the loop. Thus, the loop vectorises. To generate each pseudo-random number requires one load (non-unit stride), one floating-point add, two floating-point multiplies, one store, and of order

$$\frac{\alpha+\beta}{N}$$

startup costs. The average cost should is only a few machine cycles per random number if N is large and $\alpha+\beta$ is small.

On a vector processor with interleaved memory banks, it is desirable for the strides α and β to be odd so that the maximum possible memory bandwidth can be achieved. For statistical reasons we want α and β to be distinct and greater than 1 (see §3.2). For example, we could choose

$$\alpha = 3, \;\; \beta = 5,$$

provided $\mathrm{GCD}(\alpha\beta, N) = 1$ (true if N is a power of 2). Since $\alpha+\beta-1 = 7$, the average vector length in vector operations is about $N/7$.

Counting operations in the inner loop above, we see that generation of each pseudo-random $N(0,1)$ number requires about two floating-point multiplications and one floating-point addition, plus one (non-unit stride) load and one (unit-stride) store. To transform the $N(0,1)$ numbers to $N(\mu,\sigma^2)$ numbers with given mean and variance requires an additional multiply and add (plus a unit-stride load and store) [5]. Thus, if f is the throw-away factor (see §3.5), each pseudo-random $N(\mu,\sigma^2)$ number returned to the user requires about $2f+1$ multiplies and $f+1$ additions, plus $f+1$ loads and $f+1$ stores.

If performance is limited by the multiply pipelines, it might be desirable to reduce the number of multiplications in the inner loop by using fast Givens transformations (i.e. diagonal scaling). The scaling could be undone when the results were copied to the caller's buffer. To avoid problems of over/underflow, explicit scaling could be performed occasionally (e.g. once every 50-th pass through the pool should be sufficient).

The implementation described in §3.7 does not include fast Givens transformations or any particular optimisations for the case $\mu = 0$, $\sigma = 1$.

3.7 RANN4

We have implemented the method described in §§3.5–3.6 in Fortran on the VP2200 and VPP300. The current implementation is called `RANN4`. The implementation uses `RANU4` [6] to generate uniform pseudo-random numbers for initialisation and generation of the parameters $\alpha, \ldots, \delta$ (see (1)) and pseudo-random orthogonal matrices (see below). Some desirable properties of the uniform random number generator are inherited by `RANN4`. For example, the processor id is appended to the seed, so it is certain that different pseudo-random sequences

[5] Obviously some optimisations are possible if it is known that $\mu = 0$ and $\sigma = 1$.

will be generated on different processors, even if the user calls the generator with the same seed on several processors of the VPP300.

The user provides RANN4 with a work area which must be preserved between calls. RANN4 chooses a pool size of $2N$, where $N \ge 256$ is the largest power of 2 possible so that the pool fits within part (about half) of the work area. The remainder of the work area is used for the uniform generator and to preserve essential information between calls. RANN4 returns an array of normally distributed pseudo-random numbers on each call. The size of this array, and the mean and variance of the normal distribution, can vary from call to call.

The parameters $\alpha, \ldots, \delta$ (see (1)) are chosen in a pseudo-random manner, once for each pool, with $\alpha \in \{3, 5\}$ and $\beta \in \{7, 11\}$. The parameters γ and δ are chosen uniformly from $\{0, 1, \ldots, N-1\}$. The orthogonal matrix A is chosen in a pseudo-random manner as

$$A = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix},$$

where $\pi/6 \le |\theta| \le \pi/3$ or $2\pi/3 \le \theta \le 5\pi/6$. The constraints on θ ensure that $\min(|\sin\theta|, |\cos\theta|) \ge 1/2$. We do not need to compute trigonometric functions: a uniform generator is used to select $t = \tan(\theta/2)$ in the appropriate range, and then $\sin\theta$ and $\cos\theta$ are obtained using a few arithmetic operations. The matrix A is fixed in each inner loop (though not in each complete pass) so multiplications by $\cos\theta$ and $\sin\theta$ are fast.

For safety we adopt the conservative choice of throw-away factor $f = 3$ (see §3.5), although in most applications the choice $f = 2$ (or even $f = 1$) is satisfactory and significantly faster.

Because of our use of RANU4 to generate the parameters $\alpha, \ldots, \delta$ etc, it is most unlikely that the period of the sequence returned by RANN4 will be shorter than the period of the uniformly distributed sequence generated by RANU4. Thus, it was not considered necessary to restart the generator as described in §3.4. However, our implementation monitors the sum of squares and corrects for any "drift" caused by accumulation of rounding errors.

On the VP2200/10, the time per normally distributed number is approximately $(6.8f + 3.2)$ nsec, i.e. $(1.8f + 1.0)$ cycles. With our choice of $f = 3$ this is 23.6 nsec or 6.4 cycles. The fastest version, with $f = 1$, takes 10 nsec or 2.8 cycles. For comparison, the fastest method of those considered in [7] (the Polar method) takes 21.9 cycles. Thus, we have obtained a speedup by a factor of about 3.2 in the case $f = 3$.

Times on a single processor of the VPP300 are typically faster by a factor of about two, which is to be expected since the peak speed of a processor on the VPP300 is 2.285 GFlop (versus 1.25 Gflop on the VP2200/10). On the VPP300 with P processors, the time per normally distributed number is $11.4/P$ nsec if $f = 3$ and $5.4/P$ nsec if $f = 1$.

Various statistical tests were performed on RANN4 with several values of the throw-away factor f. For example:

- If (x, y) is a pair of pseudo-random numbers with (supposed) normal $N(0, 1)$ distributions, then $u = \exp(-(x^2 + y^2)/2)$ should be uniform in $[0, 1]$, and $v = \mathrm{artan}(x/y)$ should be uniform in $[-\pi/2, +\pi/2]$. Thus, standard tests for uniform pseudo-random numbers can be applied. For example, we generated batches of (up to) 10^7 pairs of numbers, transformed them to (u, v) pairs, and tested uniformity of u (and similarly for v) by counting the number of values occurring in $1,000$ equal size bins and computing the χ^2_{999} statistic. This test was repeated several times with different initial seeds etc. The χ^2 values were not significantly large or small for any $f \geq 1$.
- We generated a batch of up to 10^7 pseudo-random numbers, computed the sample mean, second and fourth moments, repeated a number of times, and compare the observed and expected distributions of sample moments. The observed moments were not significantly large or small for any $f \geq 3$. The fourth moment was sometimes significantly small (at the 5% confidence level) for $f = 1$.

A possible explanation for the behaviour of the fourth moment when $f = 1$ is as follows. Let the maximum absolute value of numbers in the pool at one pass be M, and at the following pass be M'. By considering the effect of the orthogonal transformations applied to pairs of numbers in the pool, we see that (assuming $n = 2$),

$$M/\sqrt{2} \leq M' \leq \sqrt{2}M \ .$$

Thus, there is a correlation in the size of outliers at successive passes. The correlation for the subset of values returned to the user is reduced (although not completely eliminated) by choosing $f > 1$.

4 Summary and Conclusions for Normal RNG

We showed that both the Box-Muller and Polar methods for normally distributed random numbers vectorise well, and that it is possible to avoid and/or speed up the evaluation of the functions (sin, cos, ln, sqrt) which appear necessary. On the VP2200/10 our best implementation of the Polar method takes 21.9 machine cycles per normal random number, slightly faster than our best implementation of the Box-Muller method (26.3 cycles).

We considered the vectorisation of some other popular methods for generating normally distributed random numbers, and showed why such methods are unlikely to be faster than the Polar method on a vector processor.

We showed that normal pseudo-random number generators based on Wallace's ideas vectorise well, and that their speed on a vector processor is close to that of the generalised Fibonacci uniform generators, i.e. only a small number of machine cycles per random number.

Because Wallace's methods are new, there is little knowledge of their statistical properties. However, a careful implementation should have satisfactory statistical properties provided distinct non-unit strides α, β satisfying (2) are used, the sums of squares are varied as described in §3.3, and the throw-away factor f is chosen appropriately. The pool size should be fairly large (subject to storage constraints), both for statistical reasons and to improve performance of the inner loops. Wallace uses 4×4 orthogonal transformations, but a satisfactory generator is possible with 2×2 orthogonal transformations.

It may appear that we have concentrated on vector rather than parallel implementations. If this is true, it is because vectorisation is the more interesting and challenging topic. Parallelisation of random number generators is in a technical sense "easy" since no communication is required after the initialisation on different processors. However, care has to be taken with this initialisation to ensure independence (see §1), and testing of parallel RNGs should not ignore this important requirement.

Acknowledgements

Thanks are due to:

– Don Knuth for discussions regarding the properties of generalised Fibonacci methods and for bringing some references to my attention.
– Wes Petersen for his comments and helpful information on implementations of random number generators on Cray and NEC computers [34,35].
– Chris Wallace for sending me a preprint of his paper [40] and commenting on my attempts to vectorise his method.
– Andy Cleary, Bob Gingold, Markus Hegland and Peter Price for their assistance on the Vector/Parallel Scientific Subroutine Library ("area 4") project.

This work was supported in part by a Fujitsu-ANU research agreement. The ANU Supercomputer Facility provided computer time for development and testing on Fujitsu VP2200 and VPP300 computers at the Australian National University.

References

1. M. Abramowitz and I. A. Stegun: Handbook of Mathematical Functions. Dover, New York, 1965, Ch. 26. 14
2. J. H. Ahrens and U. Dieter: Computer Methods for Sampling from the Exponential and Normal Distributions. Comm. ACM **15** (1972), 873–882. 2, 4, 9
3. S. Aluru, G. M. Prabhu and J. Gustafson: A Random Number Generator for Parallel Computers. Parallel Computing **18** (1992), 839.
4. S. L. Anderson: Random Number Generators on Vector Supercomputers and Other Advanced Architectures, SIAM Review **32** (1990), 221–251. 3
5. R. P. Brent: Algorithm 488: A Gaussian Pseudo-Random Number Generator (G5). Comm. ACM **17** (1974), 704–706. 2, 4, 9, 10
6. R. P. Brent: Uniform Random Number Generators for Supercomputers. Proc. Fifth Australian Supercomputer Conference, Melbourne, December 1992, 95–104. `ftp://nimbus.anu.edu.au/pub/Brent/rpb132.dvi.gz` 3, 5, 11, 15
7. R. P. Brent: Fast Normal Random Number Generators for Vector Processors. Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University, March 1993. `ftp://nimbus.anu.edu.au/pub/Brent/rpb141tr.dvi.gz` 5, 7, 16

8. R. P. Brent: On the Periods of Generalized Fibonacci Recurrences, Math. Comp. **63** (1994), 389–401. 14
9. R. P. Brent: A Fast Vectorised Implementation of Wallace's Normal Random Number Generator. Report TR-CS-97-07, Computer Sciences Laboratory, Australian National University, Canberra, April 1997. `ftp://nimbus.anu.edu.au/pub/Brent/rpb170tr.dvi.gz`
10. C. W. Clenshaw, L. Fox, E. T. Goodwin, D. W. Martin, J. G. L. Michel, G. F. Miller, F. W. J. Olver and J. H. Wilkinson: Modern Computing Methods. 2nd edition, HMSO, London, 1961, Ch. 8. 7
11. P. D. Coddington: Random Number Generators for Parallel Computers. The NHSE Review **2** (1996). `http://nhse.cs.rice.edu/NHSEreview/RNG/PRNGreview.ps` 3
12. P. D. Coddington and S-H. Ko: Techniques for Empirical Testing of Parallel Random Number Generators. Proc. International Conference on Supercomputing (ICS'98), Melbourne, Australia, July 1998, to appear. 3
13. S. A. Cuccaro, M. Mascagni and D. V. Pryor: Techniques for Testing the Quality of Parallel Pseudo-Random Number Generators. Proc. 7th SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1995, 279–284.
14. L. Devroye: Non-Uniform Random Variate Generation. Springer-Verlag, New York, 1986. 2, 4, 9
15. P. L'Ecuyer: Efficient and Portable Combined Random Number Generators. Comm. ACM **31** (1988), 742–749, 774.
16. P. L'Ecuyer: Random Numbers for Simulation. Comm. ACM **33**, 10 (1990), 85–97. 3
17. P. L'Ecuyer and S. Côté: Implementing a Random Number Package with Splitting Facilities. ACM Trans. Math. Software **17** (1991), 98–111.
18. W. Evans and B. Sugla: Parallel Random Number Generation. Proc. 4th Conference on Hypercube Concurrent Computers and Applications (ed. J. Gustafson), Golden Gate Enterprises, Los Altos, CA, 1989, 415.
19. 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.
20. P. Griffiths and I. D. Hill (editors): Applied Statistics Algorithms. Ellis Horwood, Chichester, 1985. 4
21. J. R. Heringa, H. W. J. Blöte and A. Compagner: New Primitive Trinomials of Mersenne-Exponent Degrees for Random-Number Generation. Internat. J. of Modern Physics C **3** (1992), 561–564. 3
22. F. James: A Review of Pseudo-Random Number Generators. Computer Physics Communications **60** (1990), 329–344. 2, 3
23. A. J. Kinderman and J. F. Monahan: Computer Generation of Random Variables Using the Ratio of Uniform Deviates. ACM Trans. Math. Software **3** (1977), 257–260. 9
24. D. E. Knuth: The Art of Computer Programming. Volume 2: Seminumerical Algorithms. 3rd edn. Addison-Wesley, Menlo Park, 1997. 2, 3, 4, 5, 8, 9, 10, 14
25. D. H. Lehmer: Mathematical Methods in Large-Scale Computing Units. Ann. Comput. Lab. Harvard Univ. **26** (1951), 141–146.
26. J. L. Leva: A Fast Normal Random Number Generator. ACM Trans. Math. Software **18** (1992), 449–453. 4, 9, 10
27. M. Lüscher:, A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations. Computer Physics Communications **79** (1994), 100–110. 14

28. G. Marsaglia: A Current View of Random Number Generators. Computer Science and Statistics: Proc. 16th Symposium on the Interface, Elsevier Science Publishers B. V. (North-Holland), 1985, 3–10. 3
29. M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson: A Fast, High-Quality, and Reproducible Lagged-Fibonacci Pseudorandom Number Generator. J. of Computational Physics **15** (1995), 211–219.
30. M. Mascagni, M. L. Robinson, D. V. Pryor and S. A. Cuccaro: Parallel Pseudorandom Number Generation Using Additive Lagged-Fibonacci Recursions. Springer-Verlag Lecture Notes in Statistics **106** (1995), 263–277.
31. M. E. Muller: A Comparison of Methods for Generating Normal Variates on Digital Computers. J. ACM **6** (1959), 376–383. 4
32. J. von Neumann: Various Techniques Used in Connection With Random Digits. The Monte Carlo Method, National Bureau of Standards (USA) Applied Mathematics Series **12** (1951), 36. 1
33. S. K. Park and K. W. Miller: Random Number Generators: Good Ones are Hard to Find. Comm. ACM **31** (1988) 1192–1201. 3
34. W. P. Petersen: Some Vectorized Random Number Generators for Uniform, Normal, and Poisson Distributions for CRAY X-MP. J. Supercomputing **1** (1988), 327–335. 2, 18
35. W. P. Petersen: Lagged Fibonacci Series Random Number Generators for the NEC SX-3. Internat. J. High Speed Computing **6** (1994), 387–398. 7, 10, 18
36. D. V. Pryor, S. A. Cuccaro, M. Mascagni and M. L. Robinson: Implementation and Usage of a Portable and Reproducible Parallel Pseudorandom Number Generator. Proc. Supercomputing ’94, IEEE, New York, 1994, 311–319.
37. I. Vattulainen, T. Ala-Nissila and K. Kankaala: Physical Tests for Random Numbers in Simulations. Phys. Rev. Lett. **73** (1994), 2513. 4
38. I. Vattulainen, T. Ala-Nissila and K. Kankaala: Physical Models as Tests of Randomness. Phys. Rev. E **52** (1995), 3205. 4
39. C. S. Wallace: Transformed Rejection Generators for Gamma and Normal Pseudorandom Variables. Australian Computer Journal **8** (1976), 103–105. 2
40. C. S. Wallace: Fast Pseudo-Random Generators for Normal and Exponential Variates. ACM Trans. Math. Software **22** (1996), 119–127. 3, 10, 11, 13, 18