Thoughts on Pseudorandom Number Generators

B.D. Ripley

These notes are a modified version of Ripley (1990).

1. What is a good generator?

The ideal properties of a good general-purpose pseudorandom number generator are easy to agree but impossible to achieve simultaneously. They include

- a very good approximation to a uniform distribution,
- very close to independent output in a moderate number of dimensions,
- repeatability from a simply specified starting point,
- speed,
- a very long period (at least 2^{50}).

The first two of these are axiomatic (or a definition of *pseudorandom*) since pseudorandom numbers will be used as if they were a realization of an independent $U(0,1)$ stream. Repeatability is very important for debugging (in particular, debugging the work of others) and for understanding a stochastic process. For example, Ripley & Kirkland (1990) show some summaries of the simulation of a Markov random field which show an abrupt change at one point in the supposedly converging iterative process. Because a repeatable sequence was used, the process could be run up to just before that point and stopped, so the critical phase could be examined in detail.

The last two points are also closely linked. I believe the time taken for pseudorandom number generation should be negligible compared to the operations performed on the numbers produced. Some of our iterative simulations on a Sun 3 workstation spent 30% of the time generating numbers from a rather fast ($13\,\mu$secs/call) generator, and so use 1.5 million random numbers per minute. A single run can easily take 10 hours or 9×10^8 random numbers. The period of many commonly used generators is around 2^{31}, one quarter of this number. Simulations in statistical physics on supercomputers can use very much larger numbers. There need be no great disadvantage in repeating the sequence within a simulation provided that it can be guaranteed to be done in an asynchronous way (such as when the simulation is non-stationary). The real need for a long period is to satisfy the first two requirements simultaneously, since a relatively small number N of k-tuples cannot fill $[0,1]^k$ adequately. An argument based on spatial statistics (Ripley, 1987, p. 26) suggests that we need $N \gg 200n^2$ if there are n iid points in the problem. Thus for small problems with up to 1000 points, a period of about 2^{31} is adequate, but for problems with a million or more points, periods of 2^{50} or more are necessary.

Even in more mundane applications a fast generator can avoid having to construct complicated non-uniform variate generators, either to minimize the number of uniforms used or by 're-using' uniforms. On modern hardware with standard functions built into maths co-processor chips the elementary functions can be fast so simple routines can also be very fast. This is illustrated in table 1, which shows on the Sun 3 chipsets the calculation of a logarithm is faster than two calls to any pseudorandom number generator.

One solution to the problems of finding a good pseudorandom number generator is to use a machine-readable record of bytes produced by a physical device, as suggested by Luc Devroye. A laser disc could store of the order of 10^9 such bytes and can be addressed randomly, so a suitable 'seed' would be the starting address. Although this may provide fewer random real numbers than we need, the sequence could be reused in some 'random' way. The problems are at one extreme to deliver the numbers fast enough in certain sorts of parallel computers, and at the other, cost of the drive and intermediate storage. Physical devices are also used; the

Institute of Statistical Mathematics in Tokyo has a 200kbytes/sec physical generator, but this is of course not repeatable.

Park & Miller (1988) comment that examples of good generators are hard to find, and suggest *LGM* (below) as a 'minimal standard'. Their search was, however, in the computer science literature, and mainly in texts at that; random number generation seems to be one of the most misunderstood subjects in computer science!

2. Some Example Generators.

Congruential generators

Most commonly used generators are from the *linear congruential* family,

$$X_i = (aX_{i-1} + c) \bmod M, \qquad U_i = X_i/M \tag{1}$$

with widely used examples including

Name	a	c	M	period	reference
LGM	16807	0	$2^{31}-1$	$2^{31}-2$	Lewis, Goodman, Miller (1969)
PRB	630360016	0	$2^{31}-1$	$2^{31}-2$	Payne, Rabung, Bogyo (1969)
Marsg	69069	1	2^{32}	2^{32}	Marsaglia (1972), also VAX
Los Alamos	5^{19}	0	2^{48}	2^{46}	Beyer, at workshop
Atari ST	3141592621	1	2^{32}	2^{32}	from the OS ROM
rand	1103515245	12345	2^{31}	2^{31}	Unix
drand48	25214903917	11	2^{48}	2^{48}	Unix and copies

It is by now well known that well-designed congruential generators have a k-dimensional output which fills a lattice in $[0,1)^k$. All of these examples have adequate lattice structure in up to 8 dimensions (Ripley, 1987, §2.4) and so satisfy the first two requirements, at least weakly. It hardly seems worth searching for 'optimal' multipliers since most choices are reasonably good. (I find the criteria of Fishman & Moore (1986) unreasonably stringent. Generators whose criteria differ by a factor of two are for practical purposes indistinguishable.) The worst performance here in terms of the ratio r of lattice sides is the Los Alamos generator with $r \approx 16.8$ in $[0,1]^3$, but in absolute terms this is *much* better than all the shorter-period generators. It is not worth going beyond 8 dimensions, since 2^{32} points in $[0,1]^8$ will be at least 1/16 apart, and even 2^{48} will be 1/64 apart.

Some of these generators (notably *Marsg*) have a poor two-dimensional discrepancy. This arises because one axis of the lattice of points from this generator is closely aligned with the axes in $[0,1]^2$, the basis vectors being $2^{-32}(1, 69069)$ and $2^{-32}(-62184, 19400)$. The discrepancy is about that of independent 18-bit reals! The position of *LGM* is even worse; Fishman & Moore (1986) quote its disrepancy as being in the interval $10^{-5}[1.488, 5.952]$, about that of independent 16-bit reals, whereas other generators with the same modulus and period can achieve 10^{-8}. I believe that discrepancy is not a good measure of a *pseudo*random number generator, and may not be a good measure of a *quasi*random sequence, since Koksma's inequality is only an upper bound for the accuracy of a quasi-Monte Carlo integral.

The difficulty in implementing (1) is to perform the modular arithmetic rapidly. It is usually possible to implement *LGM* and *Marsg* in double-precision reals, and with suitable hardware this need not be outrageously slow (see table 1 below).

```
REAL FUNCTION UNIF(SEED)
DOUBLE PRECISION SEED
SEED = MOD(69069.0D0*SEED+1.0D0, 4294967296.0D0)
UNIF = SEED*2.3283064D-10
END
```

For this to be valid the double precision needs at least 49 bits of precision, which even the most frugal implementations provide. Schrage (1979) gave a portable integer implementation of *LGM*, improved in Bratley, Fox & Schrage (1983, p. 202):

```
REAL FUNCTION UNIF(ISEED)
INTEGER*4 ISEED,K
K = ISEED/127773
ISEED = 16807*(ISEED-127773*K)-2836*K
IF (ISEED .LT. 0) ISEED = ISEED+2147483647
UNIF = 4.6566128E-10*ISEED
END
```

With floating-point hardware the double-precision method is often as fast.

In an attempt to construct a very portable pseudorandom number generator, Wichmann & Hill (1982, 1987) combined three short-period congruential generators by addition modulo 1. On a machine with 32-bit integers their code is

```
REAL FUNCTION RND()
INTEGER X,Y,Z
DATA X,Y,Z/...,...,.../
X = MOD(171*X, 30269)
Y = MOD(172*Y, 30307)
Z = MOD(170*Z, 30323)
RND = MOD(X/30269.0 + Y/30307.0 + Z/30323.0, 1.0)
END
```

and the papers contain code for machines with 16-bit integers. This generator has period $\mathrm{lcm}\,(30268,\ 30306,\ 30322) = 6.95 \times 10^{12}$, but the three divisions and the mod make it rather slow on all the computers I use (Table 1). Thus I dispute that it is 'efficient' (their 1982 title).

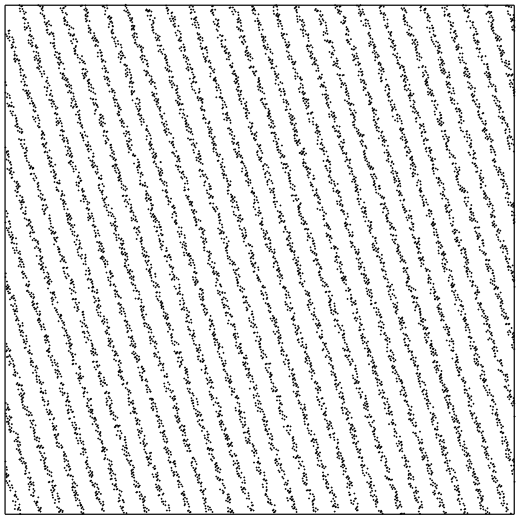

Figure 1. Plot of all successive pairs (U_{2i}, U_{2i+1}) from one IBM PC BASIC generator.

Some examples of congruential generators in common use have *both* too small a period and an incorrect implementation. Sawitzki (1985) quotes one version of the built-in generator in IBM PC BASIC which was chosen as a = 214013, c = 13523655, $M = 2^{24}$ and hence would have a rather short period. What was actually implemented was

$$X_i = (214013(X_{i-1} \bmod 2^{16}) + 13523655) \bmod 2^{24}$$

which has period 2^{16} but with numbers not uniformly spread in $\{0, \ldots, 2^{24}-1\}$ (although the unevenness of the top 8 bits is small). The main problem is its performance in higher dimensions; for instance figure 1 shows a plot of successive pairs! Thus this generator will fail in many problems small enough to be programmed in BASIC.

One common objection to congruential generators with modulus $M = 2^\beta$ is that their "lower order bits have a small period". This applies to $\{X_i\}$ not $\{U_i\}$, and is only relevant if the generator is used to produce many bits per U_i. This seems a futile exercise, and usually takes longer than `INT(2.0*RND())` for each new bit!

Shift-register generators

The alternative family is that of *shift-register* generators. These generate L-bit integers by a recurrence of the form

$$Y_i = Y_{i-p} \oplus Y_{i-(p-q)} \tag{2}$$

where $\oplus$ denote bitwise exclusive or (the *C* operation ^, and available in most Fortran dialects but few Pascals). Recurrence (2) can be implemented by a simple circular buffer:

```
REAL FUNCTION RND()
PARAMETER(IP='p', IQ='q', IPQ=IP-IQ, L='L')
COMMON /RNDBLK/ Y(IP)
DATA I,J/IPQ,IP/
IY = XOR(Y(I), Y(J))
Y(J) = IY
I = I-1
IF (I .EQ. 0) I = IP
J = J-1
IF (J .EQ. 0) J = IP
RND = IY*(2.0**-L)
END
```

(The `XOR` function is valid in Sun Fortran; other systems have other conventions given in the Appendix). Before use `Y(1),...,Y(P)` are initialized with $Y_{-1}, \ldots, Y_{-p}$. Each of the L bits of Y_i then follows part of the sequence

$$b_j = b_{j-p} \oplus b_{j-(p-q)} \tag{3}$$

with $j = i - t_k$ for bit k. Often this general form is called a *GFSR* but some authors consider only the case $t_k = (k-1)\tau$. For suitable p and q the sequence $\{Y_i\}$ has period $2^p - 1$, and if in addition the starting values are chosen suitably the sequence is k-distributed for $k \le \lfloor p/L \rfloor$ (Fushimi & Tezuka, 1983; Ripley, 1987, §2.3). In practical terms this means that k-tuples $(U_i, \ldots, U_{i+k-1})$ have an almost independent distribution. One popular choice is p = 521, q = 32, L = 32, for which 16-tuples are practically independent, and single-precision reals U_i will be to a very good approximation uniformly distributed. As Table 1 shows, this generator can be implemented to run nearly as fast as a 32-bit congruential generator, so it comes close to meeting our aims in §1.

The snag with GFSRs is their initialization. The procedure given in Fushimi & Tezuka (1983) is trial-and-error, given in a modified form by Marsaglia & Tsay (1985). One very simple way to initialize our example GFSR is to use the generator

$$b_j = b_{j-p} \oplus b_{j-q} \tag{4}$$

(which is just (3) run backwards). The first p values are arbitrary; then (4) is used to find p further values. Take the jth bit of `Y(i)` as b_{i+16j}, corresponding to $t_j = 16j$. Then the sequence $\{Y_i\}$ is 16-distributed, and as Table 1 shows the initialization is acceptably rapid. Fortran code is given in the Appendix. To reduce the p = 521 bits needed to initialize to a reasonable quantity we use a congruential generator (in fact *LGM* in Schrage's form).

Fushimi (1989) considers a variant on this idea, with a time-consuming initialization but the same theoretical properties. Table 1 shows that initialization cannot be performed routinely since it takes as long as 250,000 calls. Nevertheless, initialization is acceptable on mainframes and C code can be more rapid; on the Atari it took 13 seconds for the second and subsequent initializations.

Initialization for GFSRs is still a subject of debate, and other ideas may yet be proved better. However, 16-distribution suffices for just about any conceivable practical purpose.

GFSRs have had a bad press. Knuth (1981, p. 30) warns against them:

> "*Caution:* Several people have been trapped into believing that this random bit-generation technique can be used to generate random whole-word fractions $(.X_0, X_1 \ldots X_{k-1})_2$, $(.X_k, X_{k+1} \ldots X_{2k-1})_2, \ldots$; but this is actually a poor source of random fractions, even though the bits are individually quite random."

In a similar vein, Marsaglia & Tsay (1985) give several objections, including

> "The exclusive-or operation $\oplus$, is no faster than + or $-$ in most computers so, taken with the poor statistical performance and relatively short periods, one wonders why $F(r, s, \oplus)$ generators have ever been given serious consideration.
>
> But they have. . . ."

Here $F(r, s, op)$ refers to the generator $Y_i = Y_{i-r}\ op\ Y_{i-s}$. Many of the objections to GFSRs are based on extrapolation from inadequate examples, and some are of the nature of Marsaglia's original objection to congruential generators on the basis of their lattice behaviour (to abandon any method with a known potential flaw). I do not believe any of these objections stand serious scrutiny.

Shuffling

One often advocated way to improve a generator is to use shuffling. The best-regarded method is that of Bays and Durham (1976) given in the following code:

```
      FUNCTION RAN0(ISEED)
      REAL V(98),Y
      LOGICAL FIRST
      DATA FIRST/.TRUE./
      IF (FIRST) THEN
        FIRST = .FALSE.
        DO 10 I = 1,98
10        V(I) = RND(ISEED)
        Y = RND(ISEED)
      ENDIF
      J = 1+97*Y
      Y = V(J)
      V(J) = RND(ISEED)
      RAN0 = Y
      END
```

The idea is to keep an array of previously generated pseudorandom numbers, and to use the *previous* number to pick an element to return and replace. As table 1 shows, the use of `RAN0` roughly doubles the timings. In §4 we will see an advantage of shuffling when running the same generator on different processors; if each is given a different seed they will be extremely unlikely ever to get into step.

Inversive generators

These were introduced by Lehn and co-workers. Suppose p is prime. Then all non-zero integers in Z_p have an inverse in that

$$\breve{x} \cdot x = 1 \pmod{p}$$

Assign zero the inverse 0. The same holds for odd integers modulo 2^α. The generator is then

$$X_i = (a\breve{X}_{i-1} + c) \bmod M$$

which gives a purely periodic sequence, and can for some values of a and b have period p if M is prime, and can have period $M/2$ if $M = 2^\alpha$ if and only is $a \bmod 4 = 1$ and $c \bmod 4 = 2$.

Results are also known for the case $M = p^\alpha$. These generators distribute k-tuples less regularly in $[0, 1]^k$ than the previous examples. There are still at most M k-tuples, though.

Similar remarks apply to quadratic congruential generators

$$X_i = aX_{i-1}^2 + bX_{i-1} + c \bmod M$$

whose k-dimensional output is a union of grids.

Matrix generators

A way to achieve more k-tuples is to use more history. Matrix generators are of the form

$$x_n = Ax_{n-1} \bmod p$$

for a r-dimensional column vector x, which corresponds to

$$x_n = a_{r-1}x_{n-1} + \cdots + a_1x_{n-r} \bmod p$$

The period is at most $p^r - 1$, and this is attained if and only if the minimal polynomial is primitive of degree r over $GF(p)$. There are lots of these, so they can be found by trial and error.

These generators also have a lattice structure in k dimensions.

Add-with-Carry Generators

This is a recent idea of Marsaglia & Ziman (Florida). Iterate based on (x_1, x_2, c) where x_1 and x_2 are points in the history of the generator, and c is the 'carry' from the previous operation (addition or subtraction). Thus

$$X_n = X_{n-r} + X_{n-s} + c \bmod M$$

or

$$X_n = X_{n-r} - X_{n-s} - c \bmod M$$

or

$$X_n = X_{n-s} - X_{n-r} - c \bmod M$$

An example is

$$X_n = X_{n-22} - X_{n-43} - c \bmod 2^{32} - 5$$

which has period $M^{43} - M^{22} \approx 2^{1376} \approx 10^{414}$ Other versions can be done exactly in floating point with a 24-bit mantissa:

$$U_n = U_{n-25} - U_{n-39} - c \bmod 1$$

with carry $c = 0, 2^{-24}$ and period approximately 10^{279}.

Table 1 Times for various generators on a range of computers. All times are in μsecs except initialization which is in seconds.

	Atari ST	**Amstrad** 1640	**PS/2 60** + 80287	**Sun 3/60**	**3/80**	**4/60**	**4/370**
log(U)	1670	225	180	38	38	10	6.5
built-in*	210	200	120	141	156	11	10
Marsg (assembler)	60	—	—	—	—	6.1	4.5
BFSchrage	440	700	290	32	30	16	11
Marsg (Fortran)	2000	620	390	364	341	21	16
WichHill	1490	1220	550	98	91	47	34
GFSR521 init	1.25s	3.50s	1.25s	0.08s	0.10s	0.06s	0.05s
per call	210	175	90	19	18	3.1	2.3
Fushimi (1989) init	110s	220s	60s	3.5s	3.6s	1.3s	1.0s
per call	360	345	145	28	25	6.8	6.2
Fushimi† init	50s	150s	37s	2.9s	2.9s	1.1s	0.9s
per call	210	230	105	21	19	3.1	2.3
Shuffling‡	250	190	—	36	38	2.8	2.7
ranmar$^\diamond$	—	—	—	84	78	8.3	7.6
rcarry$^\bullet$	—	—	—	54	54	5.7	5.0

* varies from compiler to compiler.
† after some optimization of the code
‡ the *additional* time for shuffling.
$^\circ$ Eichenauer & Lehn (1986) with $m = 2^{31} - 1$.
$^\diamond$ James (1990), Marsaglia *et al.* (1990), period $\approx 10^{30}$
$^\bullet$ James (1990), period $\approx 10^{171}$

3. Some timings

To illustrate why speed might be important, the examples of §2 were tested on a range of machines. Some attempt was made to code them as efficiently as possible in Fortran, although where the authors gave Fortran code this was used. The Atari ST is a 16-bit personal computer based on a 8MHz 68000; the Prospero Fortran compiler was used. As this has no floating-point hardware the times are dominated by the $U_i = X_i/M$ step which contributes around 120 μseconds. Probably all the generators except the Fortran version of *Marsg* and *WichHill* are acceptable.

The Amstrad 1640 is a 8086-based XT-class machine with an 8087 coprocessor. The IBM PS/2 model 60 had a 10MHz 80286 and 8MHz 80287 coprocessor. The times given were in similar proportion across a range of 8088/8087 to 80386/80387 machines. The Prospero compiler was used (so the extra mathematics functions of the 80387 are not used). Here a fast routine is desirable, and for serious use one should code a routine in assembler (but assembler for this chip family is notoriously unfriendly).

The Sun 3/60 is a 20MHz 68020/68881 machine, the 3/80 a 20MHz 68030/68882 machine; the Sun 4/60 (also known as a SparcStation 1) and 4/370 are RISC machines. Timings are under SunOS4.0.3. Note that there can be big differences for nominally identical machines; another (very early) 4/60 gve similar times *except* 85 for BFSchrage, 23 for *rand* and 64 for *drand48*! Note also how relatively efficient the GFSRs and shuffling are on the RISC machines.

Timings in C are more difficult to report, partly due to the effects of its insistence on using double precision arguments for functions, and partly from the inefficiency of the compilers used (Lattice C on the Atari ST, OS 4.0.3 on the Suns).

Table 2 Times for various generators in C. All times are in μsecs except initialization which is in seconds.

	Atari ST	**Sun 3/60**	**3/80**	**4/60**	**4/370**	*comments*
log(U)	3600	220	190	16.0	10.4	
rand (integer)	54	7.7	8.4	5.1	3.9	returns $2^{31} - 1$
rand (real)	300	6.9	6.9	5.1	3.9	multiplied by $1/(2^{31} - 1)$
drand48	1700	370	420	27	21	
random	—	18	18	4.3	3.5	Unix additive feedback generator
ran0	1100	75	75	12.9	6.5	Press *et al* (1988)
Fushimi init	24s	1.3s	1.3s	0.48s	0.37s	
per call	330	29	25	4.7	3.5	
ranmar	—	250	290	12.9	10.7	

The Unix generator *rand* has been replaced by *drand48* (which is far too slow) and by a feedback generator *random* of the type $F(r, s, +)$. The shuffling generator *ran0* is an implementation of the Bays-Durham method applied to *rand*.

4. Conclusions

The reader may already have guessed which generators I actually use. For all but the most demanding problems *Marsg* suffices. It is simple to program in assembler (and I have done so on 6502, 68000 and 68020 chips as well as on a VAX) and easy to check against a double precision implementation. If any doubt arises from the results of the simulation experiment I use GFSR521 as a cross-check, but have *never* had cause to doubt *Marsg*. However, the period of *Marsg* is beginning to seem too small, and GFSRs have the edge for the future, since they provide the fastest way to achieve an effectively infinite period *and* have demonstrably good theoretical properties.

The whole history of pseudorandom number generation is riddled with myths and extrapolations from inadequate examples. A healthy dose of scepticism is needed in reading the literature.

References

Bays, C & Durham, S.D. (1976) Improving a poor random number generator. *ACM Trans. Math. Soft.* **2**, 59-64.

Bratley, P., Fox, B.L. & Schrage, L.E. (1983) *A Guide to Simulation*. Springer, New York.

l'Ecuyer, P. (1990) Random numbers for simulation. *Comm. Assoc. Comput. Mach.* **33**, 85-97.

Eichenauer, J. and Lehn, L. (1986) A non-linear congruential pseudo random number generator. *Statistiche Hefte* **27**, 315–326.

Fishman, G.S. & Moore, L.R. (1986) An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM J. Sci. Stat. Comput.* **7**, 24-45.

Fushimi, M. (1989) Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$. *This volume*.

Fushimi, M. & Tezuka, S. (1983) The k-distribution of the generalized feedback shift register pseudorandom numbers. *Commun. Assoc. Comput. Mach.* **26**, 516-523.

James, F. (1990) A review of pseudorandom numbers. *Comp. Phys. Comm.* **60**, 329–344

Knuth, D.E. (1981) *The Art of Computer Programming, Vol* **2***, Seminumerical Algorithms.* Addison-Wesley, Reading, Mass.

Lewis, P.A.W., Goodman, A.S. & Miller, J.M. (1969) A pseudo-random number generator for the System/360. *IBM Sys. J.* **8**, 136-145.

Marsaglia, G. (1972) The structure of linear congruential sequences. In *Applications of Number Theory to Numerical Analysis* ed. S.K. Zaremba, Academic Press, London, 249-285.

Marsaglia, G. & Tsay, L.-H. (1985) Matrices and the structure of random sequences. *Linear Algebra Appl.* **67**, 147-156.

Marsaglia, G., Zaman, A. and Tsang, W. (1990) Towards a universal random number generator. *Statistics and Probability Letters* **9**, 35– .

Park, S.K. & Miller, K.W. (1988) Random number generators: good ones are hard to find. *Comm. Assoc. Comput. Mach.* **31**, 1192-1201.

Payne, W.H., Rabung, J.H. & Bogyo, T.P. (1969) Coding the Lehmer pseudorandom number generator. *Comm. Assoc. Comput. Mach.* **12**, 85-86.

Press, W.H., Flannery, B.P., Teubolsky, S.A. & Vetterling, W.T. (1988) *Numerical Recipes in C.* Cambridge University Press, Cambridge.

Ripley, B.D. (1987) *Stochastic Simulation*. Wiley, New York.

Ripley, B.D. & Kirkland, M.D. (1990) Iterative simulation methods. *J. Comput. Appl.Math.* **165-172.**,

Ripley, B.D. (1990) Thoughts on pseudorandom numbers. *J. Comput. Appl.Math.* **31**, 153–163.

Sawitzki, G. (1985) Another random number generator which should be avoided. *Stat. Soft. Newsl.* **11**, 81-82.

Schrage, L. (1979) A more portable Fortran random number generator. *ACM Trans. Math. Soft.* **5**, 132-138.

Sylwestrowicz, J.D. (1982) Parallel processing in statistics. *Compstat 1982 Proc.* **I**, 131-136, Physica-Verlag, Berlin.

Wichmann, B.A. & Hill, I.D. (1982) Algorithm AS183. An efficient and portable pseudo-random number generator. *Applied Statistics* **31**, 188-190. Correction **33**, 123.

Wichmann, B.A. & Hill, I.D. (1987) Building a random-number generator. *Byte* **12,3**, 127-128.