European Journal of Operational Research 63 (1992) 76–85
North-Holland

Theory and Methodology

Pseudorandom number generators for supercomputers and classical computers: A practical introduction *

Jack P.C. Kleijnen and Ben Annink
Tilburg University, Tilburg, Netherlands

Received October 1989; revised February 1991

Abstract: When the cycle of a multiplicative congruential generator with a modulus that is a power of two is split into two parts, then the pseudorandom numbers across parts turn out to lie on only two parallel lines. These 'long range' correlations have consequences for computers with a traditional or a new architecture. For vector computers, simple alternative computer implementations are presented. These implementations are faster than the standard subroutines available on a specific vector computer, namely the CYBER 205.

Keywords: Simulation; Monte Carlo; parallel algorithms; random numbers; software

1. Introduction

Is there really any need for more research on pseudorandom number generators; is it not like beating a dead horse? But no: new properties of old generators are still being discovered, and new generators must be developed to accommodate new architectures of computers, as this paper will show. Recently some interesting results on pseudorandom number generators have been published; unfortunately these results are scattered over various journals that are not easily accessible to readers of this journal. This paper gives a practice-oriented survey, and does not require mathematical sophistication.

We concentrate on one class of generators, namely linear congruential generators. So we do not discuss that other important class. Tausworthe generators. Moreover we focus on linear congruential generators with zero additive constant ($c = 0$ in (2.1)) and a modulus that is a power of two ($m = 2^w$ in (2.1)). This is a class of generators that are widely used, although they are not ideal. Recent publications on pseudorandom number generation are: Afflerbach and Grothe (1988), Bratley et al. (1987), Durst (1989),

* The first author was sponsored by the Supercomputer Visiting Scientist Program at Rutgers University, The State University of New Jersey, during July 1988. In 1989, computer time on the CYBER 205 in Amsterdam was made available by SURF/NFS. Useful comments on earlier versions of this paper were made by Bert Bettonvil (Katholieke Universiteit Brabant/Technische Universiteit Eindhoven) and three referees.

Correspondence to: J.P.C. Kleijnen, Department of Information Systems and Auditing, School of Business and Economics, Katholieke Universiteit Brabant (Tilburg University) P.O. Box 90153, 5000 LE, Tilburg, Netherlands.

Kalos and Whitlock (1986), L'Ecuyer (1990), Knuth (1981), Marsaglia et al. (1990), Morgan (1984), Park and Miller (1988), and Ripley (1987).

This paper is based on the number theoretic results of De Matteis and Pagnutti (1988), but presents their results in simpler terms, including a number of graphs. Their results are extended to antithetic pseudorandom numbers; moreover statistical tests are applied to corroborate their number-theoretic results. (Note that De Matteis and Pagnutti's results do not come out of the blue; see, for example, Ripley (1987, p. 42)). In the light of these results, alternative generators for supercomputers are examined.

There are several types of computers: vector computers should be distinguished from traditional scalar computers and truly parallel computers. Traditional computers such as the IBM 370 and the VAX series, execute one instruction after the other; so they operate sequentially. Truly parallel computers such as the HYPERCUBE, have many Central Processing Units (CPUs) that can operate independently of each other; this is called coarse grain parallelism. Vector computers such as the CRAY 1 and the CYBER 205, have a 'vector processing' capability: fine grain parallelism. This paper focusses on the CYBER 205, but generators for other supercomputers can be evaluated and improved along the same lines.

This paper is organized as follows. Section 2 summarizes basic results for linear congruential generators, needed in the sequel. In Section 3 the full cycle of the multiplicative generator with modules 2^w is split into equal parts, first into two parts (Section 3.1), then into 2^k parts (Section 3.2), which shows that the pseudorandom numbers lie on two and on no more than 2^{k-1} parallel lines if $k \leq 2$ and $k \geq 3$, respectively. Antithetic pseudorandom numbers are briefly considered in Section 3.3; the conditional variances and the correlation coefficient of the pseudorandom numbers paired across two parts (of the 2^k parts) are studied in Section 3.4. The disadvantages of splitting a pseudorandom number stream into parts are summarized in Section 3.5. Section 4 gives alternative computer implementations for vector computers. First the 'assembly line' architecture of vector computers, such as the CYBER 205, is explained. Next Section 4.1 gives one implementation that requires computation of J multipliers, and Section 4.2 gives a related parallel algorithm that requires computation of a single multiplier and initializing a vector with J successive numbers. Finally Section 4.3 compares these two implementations to the standard scalar routine RANF and the standard vector routine VRANF on the CYBER 205. Section 5 summarizes our conclusions.

2. Linear congruential generators

Linear congruential generators have the form

$$x_{j+1} = (ax_j + c) \bmod m, \quad j = 0, 1, 2, \ldots \quad (2.1)$$

where a, c, m, and x_0 are integers; the seed x_0 and the multiplier a are positive, but smaller than the modulus m; the additive constant c is a non-negative integer smaller than m. When c is zero, the generator is called *multiplicative* congruential. The generator has a specific cycle length or period h, which means that if the generator starts with seed x_0, then $x_h = x_0$, so $x_{h+1} = x_1$ and so on. Obviously the pseudorandom numbers $r_j = x_j/m$ satisfy $0 \leq r_j < 1$. An efficient algorithm results when $m = 2^w$ where w depends on the computer's word size; for example, CDC's vector computer CYBER 205 uses $m = 2^{47}$ (see CDC, 1986), but IMSL uses $m = 2^{31} - 1$ for traditional computers; NAG uses $m = 2^{59}$ (double word on traditional computers). However, there are other considerations besides efficiency.

Generators should yield pseudorandom numbers that are statistically independent; that is, the observed sequence $r_0, r_1, \ldots, r_n$ should not provide any information about the next sequence $r_{n+1}, r_{n+2}, \ldots$ It is extremely difficult to meet this requirement; see the earlier references.

It is possible to derive mathematical conditions that are necessary but not sufficient. For example, the following lemma is well known, and will be used later on.

Lemma 1. *If in* (2.1), $m = 2^w$ *(with* $w \geq 3$*) and* $c = 0$, *then the maximum cycle length is* $h = 2^{w-2}$; *this maximum is reached if* $a = 4g + 1$ *with odd integer* g.

Note that if m is a prime number (so $m \neq 2^w$), then longer cycles are possible; see the references

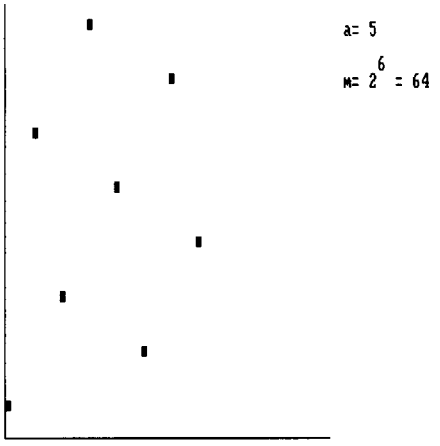

Figure 1. Plot of all successive pairs (x_{2j}, x_{1+2j}) with $j = 0, 1, \ldots, \frac{1}{2}h - 1$ for a multiplicative generator with $m = 2^6$ and $a = 5$

cited above. Because these mathematical conditions are not sufficient, statistical tests should be applied to the generator's output $(r_0, r_1, \ldots)$ to check if several types of statistical dependence are absent indeed. For example, two-tuples (r_0, r_1), (r_2, r_3), (r_4, r_5), etc. should be uniformly distributed over the unit square. Figure 1 shows results for a pedagogical example that can be easily checked by the reader; to improve the readability 'dots' are shown as 'big black squares'. We shall return to this figure.

3. Partitioning the cycle

Kleijnen (1989) surveys several types of linear congruential generators for vector computers. Section 4 will discuss vector computers; here it is only mentioned that Kleijnen (1989) discusses splitting the cycle of pseudorandom numbers into 65535 $(= 2^{16} - 1)$ non-overlapping parts. The present paper shows that this approach is wrong! The proof reveals properties of generators that also concern traditional computers. Section 3 is restricted to multiplicative generators with modulus $m = 2^w$, a multiplier resulting in a cycle length $h = 2^{w-2}$, and a seed $x_0 = 1$; see again Lemma 1.

3.1. Partitioning into two parts

Suppose the cycle of length $h = 2^{w-2}$ is split into two equal parts (of length $\frac{1}{2}h$). Kleijnen (1989) assumes that the pseudorandom numbers r_j, $j = 0, \ldots, h$, are statistically independent. Unfortunately, the numbers r_j, or equivalently the integers x_j, are statistically dependent. More specifically, De Matteis and Pagnutti (1988) give number-theoretic results that guide our present research.

Let us return to the pedagogical example of Figure 1 with $m = 2^6$ and $h = 2^{6-2} = 16$. Splitting the cycle into two parts yields a first part consisting of $x_0, x_1, \ldots, x_7$, and a second part comprising $x_8, x_9, \ldots, x_{15}$. Now plot the pairs corresponding *across* the two parts: (x_0, x_8), $(x_1, x_9), \ldots, (x_7, x_{15})$. So in this paper we are interested, not in first-order autocorrelation (Figure 1), but in long-range correlation. This yields Figure 2.

A more realistic generator has a bigger modulus m and hence a longer cycle h. We present plots only for $m = 2^{12}$ and $a = 5$ (these plots are easily obtained on a Personal Computer); Lemma 2 implies that the pattern shown by these plots holds for all generators considered in this section (Lemma 2 is presented in the next subsection). Figure 3 shows the plot for partitioning into two parts: $(x_0, x_{h/2}), (x_1, x_{(h/2)+1}), \ldots, (x_{(h/2)-1}, x_h)$. In both Figures 2 and 3 all $\frac{1}{2}h$ pairs lie on only two parallel lines, with slope one; these lines have no overlapping domains; a small number in the first part $(0 < r_j < 0.5)$ goes together with a high number in the second part $(0.5 < r_{(h/2)+j} < 1)$; so the pseudorandom numbers are negatively correlated (see Table 2 later on.) Figure 3 displays r

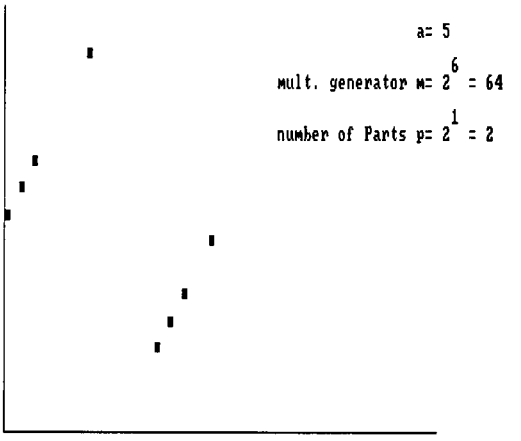

Figure 2. Pairs across two parts $(x_j, x_{j+h/2})$ with $j = 0, \ldots, \frac{1}{2}h - 1$ for a multiplicative generator with $m = 2^6$ and $a = 5$

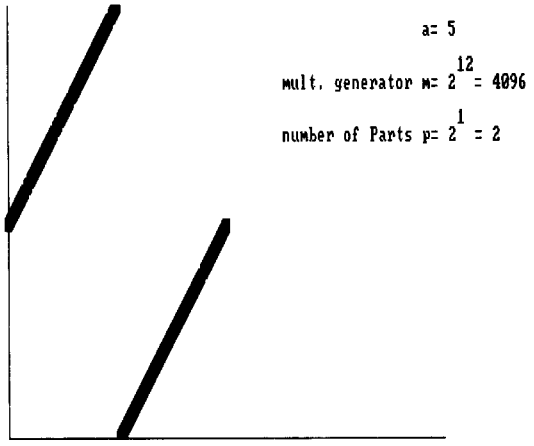

Figure 3. Pairs across two parts for $m = 2^{12}$ and $a = 5$

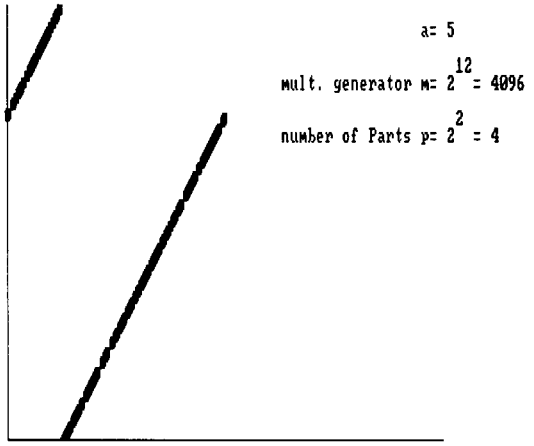

Figure 4. Pairs across first two parts when splitting into four equal parts ($m = 2^{12}$ and $a = 5$)

$(0 < r < 1)$, not the integers x $(0 < x < m)$, in order to make the plots independent of the modulus m; the remaining Figures 4 through 7 also refer to the unit square (Figures 1 and 2 serve only pedagogical purposes).

3.2. *Partitioning into 2^k parts*

What happens if the number of parts is doubled? First, notice the relationship between partitioning into two and four equal parts, respectively. Consider the didactic example with $m = 2^6$ in Figure 2. When the cycle is split into two parts, the following pairs are plotted (x_0, x_8), $(x_1, x_9), \ldots, (x_7, x_{15})$. When the cycle is now partitioned into four parts, each part has length $h = 2^{w-2}/4 = 2^{6-2}/2^2 = 4$; part No. 1 is (x_0, x_1, x_2, x_3), part No. 2 is (x_4, x_5, x_6, x_7), part No. 3 is $(x_8, x_9, x_{10}, x_{11})$, and part No. 4 is $(x_{12}, x_{13}, x_{14}, x_{15})$. Then the pairs across parts No. 1 and No. 3 are: (x_0, x_8), (x_1, x_9), (x_2, x_{10}), (x_3, x_{11}). *But these four pairs also occurred in the plot for two parts only!* So if splitting into two parts gives unacceptable results, then splitting into four parts and using all parts does not help! The cycle must be split into more parts and *only the first two parts can be used.* Figure 4 displays the plot for parts No. 1 and No. 2: $(x_0, x_{h/4})$, $(x_1, x_{(h/4)+1})$, $\ldots, (x_{(h/4)-1}, x_{(h/2)-1})$. Again all $\frac{1}{4}h$ pairs lie on only two parallel lines, with slope one; these lines still have no overlapping domains; compared with splitting into only two parts (Figure 3) these lines are shifted to the left (the correlation is still negative but smaller in absolute magnitude; see Table 2).

The pattern of the plots changes as we go on doubling the number of equal parts! Figure 5 gives the plot for the first two parts in case of 2^3 parts: $(x_0, x_{h/8})$, $(x_1, x_{(h/8)+1}), \ldots, (x_{(h/8)-1}, x_{(h/4)-1})$. Again all $\frac{1}{4}h$ pairs lie on parallel lines with slope one, but there are now four lines and some of these lines have partially overlapping domains; a small number in the first part 'goes together' with two different values in the second part (strictly speaking, one particular value of x_j corresponds to a unique value for $x_{(h/8)+j}$ since all numbers x are different in a multiplicative generator; we shall return to this issue).

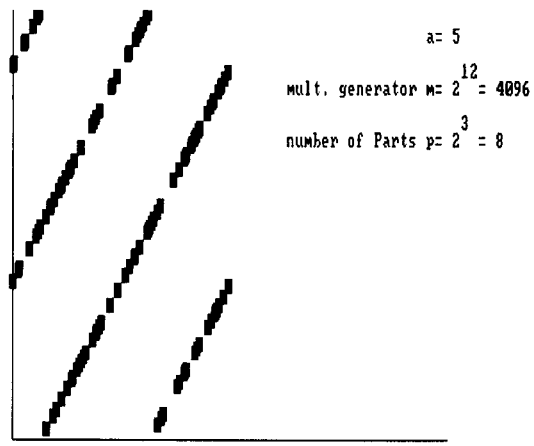

Figure 5. Pairs across first two parts for 2^3 parts ($m = 2^{12}$ and $a = 5$)

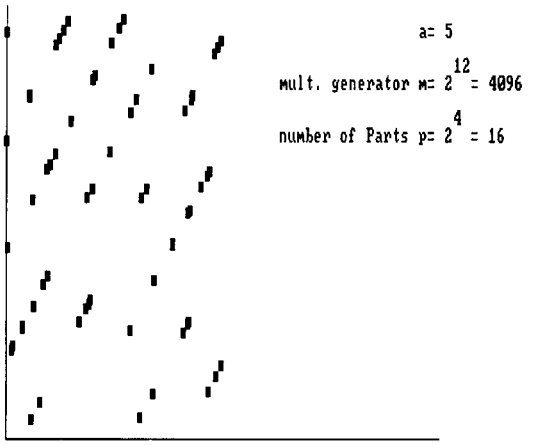

Figure 6. Pairs across two parts for 2^4 parts ($m = 2^{12}$ and $a = 5$)

Figure 6 plots the pairs when the cycle is split into 2^4 parts. Again all $\frac{1}{16}h$ pairs lie on parallel lines with slope one, but there are now eight such lines with more overlap of domains. Finally Figure 7 gives results for 2^5 parts. All $\frac{1}{32}h$ pairs still lie on parallel lines with slope one, but there are now so many lines that these lines are hard to distinguish (there are few points per line). And so we could continue. Actually De Matteis and Pagnutti (1988, p.604) prove the following lemma.

Lemma 2. *Suppose the modulus of the multiplicative generator is $m = 2^w$ with $w > 4$, the multiplier a is chosen such that the cycle length is $h = 2^{w-2}$, and the seed is $x_0 = 1$. Divide the resulting sequence into 2^k parts with $k < w - 2$. If $k \leq 2$, then the points $(x_j, x_{(h/2^k)+j})$ lie on two parallel lines with slope one $(0 \leq j < h/2^k)$. If $k > 2$, then there are no more than 2^{k-1} parallel lines with slope one.*

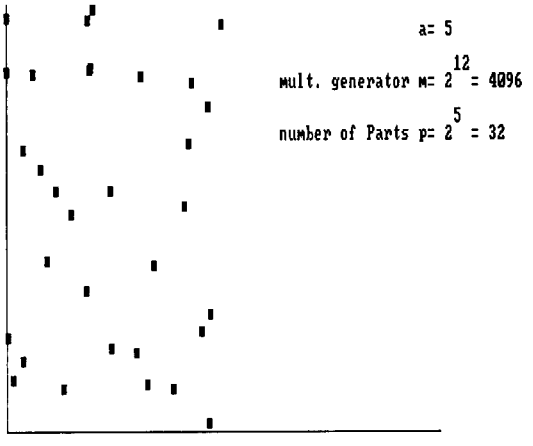

Figure 7. Pairs across two parts for 2^5 parts ($m = 2^{12}$ and $a = 5$)

3.3. Antithetic pseudorandom numbers

Kleijnen (1974, p.254) proves that the antithetic pseudorandom numbers $1 - r_j$ can be generated by starting with the seed $m - x_0$. Hence the antithetic numbers (say) y_j satisfy $y_j = m - x_j$ for $j = 1, 2, \ldots, h-1, h$; that is, the points (x_j, y_j) lie on a single line with slope minus one. We combine this result with Lemma 2 (which implies slope plus one) to conclude that *the cycle of the antithetic numbers y_j has no element in common with the cycle of the 'original' numbers x_j*: $\{y_1, \ldots, y_h\} \cap \{x_1, \ldots, x_h\} = \emptyset$.

Lemma 1 stated that a multiplicative generator with $m = 2^w$ has a maximum cycle of length $h = 2^{w-2}$. Now we can explain this cycle length as follows. The modulus $m = 2^w$ results in odd values only: half the cycle running from 0 through $m - 1$ is lost that way. Another half lies in the antithetic cycle!

3.4. Statistical analysis

The preceding plots illustrated *number-theoretic* results. What are the *statistical* consequences? First note that, within a cycle, no number x_j occurs more than once, whereas the statistical analysis of simulation output assumes that random numbers are sampled independently and hence specific values can occur more than once. In the statistical analysis this phenomenon is always ignored. In the same way the analysis of the preceding plots assumed *continuous* lines, parallel and equidistant in the unity quadrant.

We assume that the generator does yield a uniform marginal distribution; hence $\text{var}(r) = \frac{1}{12}$. It is easy to derive the variance of $r_{(h/2^k)+j}$ given r_j and a partitioning of the cycle into 2^k parts $(j = 0, \ldots, (h/2^k) - 1)$. For example, for $k = 3$, Figure 5 gives four lines such that two values $r_{(h/8)+j}$ correspond with each r_j. For simplicity's sake we assume that these two values are equally probable. Obviously the distance between two neighboring lines is $\frac{1}{2}$. Hence

$$\text{var}\left(r_{(h/8)+j} \mid r_j\right) = \left\{\left(\tfrac{1}{4}\right)^2 + \left(\tfrac{1}{4}\right)^2\right\}\tfrac{1}{2} = \tfrac{1}{16}.$$

This yields Table 1. This Table shows that the *conditional* variance increases monotonically to $\frac{1}{12}$, which is the variance if the second part would be *independent* of the first part (so the assumptions used to derive this table seem realistic).

We also test the *correlation coefficient* between the pairs $(r_j, r_{(h/2^k)+j})$. If the r's were multivariate *normally* distributed, then zero correlation would imply independence. In case of non-normality this is not true; for example, when

$$r_{(h/2^k)+j} = \begin{cases} r_j & \text{for } 0 < r_j < 0.5, \\ 1 - r_j & \text{for } 0.5 < r_j < 1, \end{cases} \tag{3.2}$$

then their correlation is zero; yet they are not independent (as (3.2) shows). To test for *zero* correlation of the uniformly distributed r's we use the 'Spearman rank correlation test'; see Churchill (1983, pp.596–598). Because this test assumes independent pairs, we assume that short-range correlations can be ignored, and we test long-range correlations. So if the rank of r_j is v_j and that of $r_{(h/2^k)+j}$ is $w_{(h/2^k)+j}$, then we compute

$$R = 1 - \frac{6 \sum_{j=1}^{n} (v_j - w_{(h/2^k)+j})^2}{n(n^2-1)}. \tag{3.3}$$

Obviously $\max(R) = 1$. The following statistic has an approximate t-distribution with $n-2$ degrees of freedom:

$$T = \frac{R(n-2)^{1/2}}{(1-R^2)^{1/2}}. \tag{3.4}$$

Table 2 shows T for $n = 1000$ and a popular generator, namely $m = 2^{32}$ and $a = 69069$. This table gives non-significant correlation for $k = 3$, since $t^{\alpha}_{n-1} = 1.65$ for $\alpha = 0.05$ and $n = 1000$. Nevertheless Figure 5 and Table 1 suggest a strong dependence; also see the example in (3.2).

In summary, this subsection shows that splitting the cycle into a few parts (small k) does not give independent pseudorandom numbers, even though the estimated correlation coefficient may be non-significant. But, if pseudorandom numbers are dependent, then the simulation fed by these numbers does not give independent results. Yet the statistical analysis of the simulation output assumes independence when estimating variances and confidence intervals; so this analysis may then give misleading results.

Table 2
Spearman rank test for zero correlation of $(r_{(h/2^k)+j}, r_j)$, when partitioning the cycle into 2^k parts; $m = 2^{32}$ and $a = 69069$; $n = 1000$

$k = 1$	2	3	4	5
$T = -17.94$	-4.56	-1.05	0.68	-0.19

3.5. *Summary of splitting approach*

Kleijnen (1989) assumes that the pseudorandom number r_j are truly independent. Then it makes sense to generate (say) J numbers in parallel by selecting J seeds such that the full cycle is split into J equal parts. However, number-theoretical results derived by De Matteis and Pagnutti (1988) imply that these parts may be correlated, especially if J is small. Acceptable statistical behavior requires that the cycle be split into at least 2^5 parts and that only the first two parts be used. So only $2 \times 2^{w-2-5}$ numbers of the full cycle of length $h = 2^{w-2}$ can be used! That useful part can be split into J subparts for parallel generation of pseudorandom numbers; see Kleijnen (1989). Long-range correlation also causes problems on traditional computers, if relatively many pseudorandom numbers are needed.

4. Vector computers and generators

This section gives generators for vector computers that produce pseudorandom numbers *not* spread over the full cycle (because of long-range correlation). Moreover, these generators produce

Table 1
Conditional variance of $r_{(h/2^k)+j}$ given r_j for 2^k parts as a percentage of $\text{var}(r_{(h/2^k)+j}) = \frac{1}{12}$

k	1	2	3	4	5	6	7
$\text{var}(r_{(h/2^k)+j} \mid r_j)$	0	0	75%	93.75%	98.44%	99.61%	99.90%

numbers in exactly the same order as generators on traditional computers do; this characteristic facilitates debugging.

First consider the *pipeline* architecture of vector computers such as the CYBER 205. A simple example is provided by the inner product of two vectors, $\boldsymbol{v}_1'\boldsymbol{v}_2 = \sum_{j=1}^{J} v_{1j}v_{2j}$. This computation requires J scalar multiplications $v_{1j}v_{2j}$; these J operations can be done in parallel because the product $v_{1j}v_{2j}$ does not need the product $v_{1(j-1)}v_{2(j-1)}$. The pipeline architecture means that the computer works as an assembly line; hence, efficiency improves drastically if a large number of identical operations can be executed, independently of each other; see Levine (1982), Miller and Walker (1989), Oed (1982), and Zenios and Mulvey (1986). Vector computers are efficient only if these operations can be executed independently or in parallel, which excludes *recursive* statements. Unfortunately, the linear congruential generator is recursive: (2.1) shows that the computation of x_{j+1} needs the predecessor x_j. Moreover, because of fixed set-up costs, the 'assembly line' is efficient only if the number of basic operations is large; the literature suggests $J > 50$. Because the CYBER 205 uses 16 bits for addressing there is a technical upper limit on J, namely $J \leq 2^{16} - 1 = 65535$; see SARA (1984, p.26). So the computer should generate J pseudorandom numbers in parallel with $50 < J \leq 65535$. Hence a simulation experiment that requires N pseudorandom numbers calls this parallel routine $\lceil N/J \rceil$ times where $\lceil\ \rceil$ denotes rounding upwards to the next integer; for example, if $N = 1000000$ and $J = 65535$, then 16 calls are necessary. So image an $(I \times J)$-matrix of pseudorandom numbers, where J numbers are generated in parallel and I calls are made to that vector routine. Kleijnen (1989) surveys different solutions to this problem (namely, J different multipliers m_j and J additive constants c_j; sampling J seeds; selecting J seeds I apart; also see Section 3). He rejects the following idea because of overflow on the computer; we shall show, however, how to solve this problem.

4.1. Vector of multipliers

Fishman (1978) proves that, given a seed x_0 and J calls to the traditional multiplicative generator (see (2.1) with $c = 0$), the resulting number x_J can be derived without knowing the intermediate numbers $(x_1, x_2, \ldots, x_{J-1})$:

$$x_J = (a^J x_0) \bmod m. \tag{4.1}$$

So J pseudorandom numbers can be generated in parallel, provided we first generate, once and for all, the vector of J multipliers $\boldsymbol{a} = (a_1, a_2, \ldots, a_{J-1}, a_J)'$ with elements

$$a_j = (a^j) \bmod m, \quad j = 1, \ldots, J. \tag{4.2}$$

The vector $\boldsymbol{a}$ is multiplied by the scalar x_0 to give the vector $(x_1, x_2, \ldots, x_{J-1}, x_J)'$. Replacing the scalar x_0 by the last element of the latter vector, namely x_J, yields the next vector $(x_{J+1}, x_{J+2}, \ldots, x_{2J-1}, x_{2J})'$, and so on. In this way *the pseudorandom numbers are generated in exactly the same order as they would have been produced in scalar mode!*

At the end of the simulation run the last pseudorandom number should be stored, so that the simulation experiment can be continued later on or a new (unrelated) simulation experiment can start at a seed different from the default x_0; also see Celmaster and Moriarty (1986) and De Matteis and Pagnutti (1988, p.602). We shall return to this generator after we have discussed a closely related generator.

4.2. Vector of J successive numbers

Suppose there is available a vector of J successive pseudorandom numbers, which can be generated in the traditional way through (2.1):

$$\boldsymbol{x} = (x_0, x_1, x_2, \ldots, x_{J-2}, x_{J-1})'. \tag{4.3}$$

Multiplying this vector by the scalar multiplier (a^J) mod m gives a new vector that is identical to the new vector obtained by the technique of Section 4.1. Now, however, the vector of the last J numbers should be stored at the end of a simulation.

There is a computational problem in both approaches: *overflow* occurs when computing high powers of the multiplier a, such as a^J. (Overflow in traditional generators is discussed in Park and Miller, 1988, p.1195.) That problem, however, can be solved through 'controlled integer overflow' (Law and Kelton, 1982, pp.219–232), combined with the CYBER 205 'binary complement' representation of negative integers: the Appendix gives

a computer program based on (4.3), which will be the most efficient implementation in the next subsection.

4.3. Comparison of four implementations

Table 3 compares the computer execution times of different computer implementations of the same generator on the CYBER 205. This computer can use FORTRAN 200 (a superset of FORTRAN 77) that allows vector and scalar programming; see CDC (1986). Implementation No. 1 is RANF, a standard scalar subroutine that uses a multiplicative generator with $m = 2^{47}$ and $a = 84000335758957$ (or in hexadecimal notation, a = 00004C65DA2C866D); see CDC (1986, pp.10–29). The CYBER 205 uses words of 64 bits; 48 bits are used to represent integers, including one sign bit; hence $m = 2^{47}$. Implementation No. 2 is VRANF, a standard vectorized subroutine that uses the same modulus m and multiplier a as RANF does; see CDC (1986, pp.11–1). Implementation No. 3 uses the vector of multipliers of (4.2). Implementation No. 4 uses the vector of J preceding numbers x_j plus the multiplier a^J; see (4.3); Implementations Nos. 3 and 4 also have the same modulus and multiplier as RANF has. The last two generators can be implemented not only in vector mode but also in scalar mode; of course RANF is in scalar mode, and VRANF is in vector mode. The measurements in Table 3 do not include storing the last vector or scalar to continue simulation at the last pseudorandom number.

Our results for RANF and VRANF deviate substantially from An Mey (1983): he finds that VRANF is always slower than RANF, and his CPU times are a factor 1000 higher! (We double-checked our results, so we are convinced of the correctness of our data; we cannot explain An Mey's results.) Implementation No. 4 is slightly faster than No. 3 is. The latter implementation must store and fetch the last element of the vector of numbers x_j. Moreover, No. 3 needs two vectors, namely one vector for the multipliers a_j and one vector for the numbers x_j. So we recommend implementation No. 4. Of course it remains to be investigated, whether the generator implemented this way has acceptable statistical behavior. For example, the generator should have small short-range correlations; see the references in Section 1.

Table 3
Computer time in microseconds of different implementations on a CYBER 205

Type of implementation	Vector length J			
	5	500	50000	65535
No. 1. RANF:				
scalar mode	0.014	0.520	51.553	67.465
No. 2. VRANF:				
vector mode	0.021	0.208	19.507	25.652
No. 3. J multipliers:				
vector mode	0.013	0.079	7.713	9.923
scalar mode	0.026	1.572	157.763	206.843
No. 4. J numbers & a^J:				
vector mode	0.013	0.079	7.425	9.631
scalar mode	0.024	1.561	157.098	206.083

5. Conclusions

Kalos and Whitlock (1986, p.180) state: "The question of independence of separate sequences to be used in parallel remains a major research issue. Not enough is known about the long-term correlations within linear congruential generators to use equal subsequences with confidence". Matteis and Pagnutti (1988) prove that each multiplicative generator shows very strong 'long range' correlations: splitting its cycle into 2^k parts gives pseudorandom numbers that lie on no more than 2^{k-1} parallel lines if $k \geq 3$; if $k < 3$, then they lie on only two parallel lines. Consequently, on vector computers, pseudorandom numbers could be generated by partitioning the cycle into 2^5 parts and using only the first two parts. There are two better techniques, however, that require the computation, once and for all, of either J multipliers ($a_j = a^j \bmod m$) or the computation of one multiplier ($a^J \bmod m$) and the initialization of one vector with J successive numbers. These two techniques are faster than the standard subroutines (RANF and VRANF) on a well-known vector computer, the CYBER 205.

Appendix: The FORTRAN 200 program for implementation No. 4.

```
PROGRAM VARIANT4
IMPLICIT REAL (U-Z),
  INTEGER (A-T)
PARAMETER (N1 = 5,N4 = 65535,K = 1)
```

```
      PARAMETER (A1 = 37772072706109)
      INTEGER MVAST
      BIT BVAST
      DESCRIPTOR MVAST, BVAST
      DIMENSION T(N4), S1(N1)
      DIMENSION X1(N1)
      DATA MINT / X'0000800000000000' /
      CALL RANSET(K)
      DO 5 I = 1,N4
      U = RANF( )
      CALL RANGET(T(I))
    5 CONTINUE
C     ! N = 5
C     ! SCALAR
      S1(1;N1) = T(1;N1)
      ZPU1 = SECOND( )
      DO 10 I = 1,N1
      S1(I0 = A1 * S1(I)
      IF (S1(I).LT.0) S1(I) = S1(I)-MINT
      X1(I0 = S1(I)/MINT
   10 CONTINUE
      ZPU2 = SECOND( )
      U1 = ZPU2-ZPU1
C     ! VECTOR
      ASSIGN MVAST,.DYN.N1
      ASSIGN BVAST,.DYN.N1
      S1(1;N1) = T(1;N1)
      ZPU1 = SECOND( )
      S1(1;N1) = A1 * S1(1;N1)
      BVAST = S1(1;N1).LT.O
      MVAST = S1(1;N1)-MINT
      S1(1;N1) = Q8VCTRL(MVAST,BVAST;
        S1(1;N1))
      X1(1;N1) = S1(1;N1)/MINT
      ZPU2 = SECOND( )
      Z1 = ZPU2-ZPU1
      FREE
      PRINT *, 'BEGIN: VECTORISE
        SCALAR'
      PRINT *, 'N =   5',Z1,'  ',U1
      END
```

To enable the reader to check this program, we give three of the $J = 5$ seed values (x_0, x_1, x_2) and the outcomes of the first ten random numbers for those seeds. So we display x_5, x_6, x_7 on the first row, x_{10}, x_{11}, x_{12} on the second row, and so on (to save space we do not display x_8, x_9 and x_{13}, x_{14}, etc.) in Table 4.

Table 4

Starting values x_0, x_1, x_2:		
84000335758957	42546483841641	118602654327989
Random numbers (x_5, x_6, x_7), (x_{10}, x_{11}, x_{12}), etc.:		
51635577448441	112073726270213	28809031491361
113554934179413	42036299976753	24524090886877
110015530009153	81298600819629	42705761318569
110447784126845	115384045819961	106866938963525
46264685920969	121717687575957	117131050270321
80793675172325	56567339750529	119127659069677
69425314839441	129916739502781	128201070008441
82909967323533	92291160590089	49025954510037
32167420825241	120236138515749	85010458949313
55571152067189	39458910421457	94340002081789
Random numbers $r(= x/2^{47})$:		
0.3668928446276	0.7963317207086	0.2047004805047
0.8068563359089	0.2986858758671	0.1742541463079
0.7817073566880	0.5776613023984	0.3034426848001
0.7847787069213	0.8198529557998	0.7593352717345
0.3287303650336	0.8648561872061	0.8322661690152
0.5740735898905	0.4019351234101	0.8464529278006
0.4932965313702	0.9231139550733	0.9109233901117
0.5891107500384	0.6557681373215	0.3483503584081
0.2285632719551	0.8543291479821	0.6040356407006
0.3948567841916	0.2803724216097	0.6703260317080

References

Afflerbach, L., and Grothe, H. (1988), "The lattice structure of pseudo-random vectors generated by matrix generators", *Journal of Computational and Applied Mathematics* 23, 127–131.

An Mey, D. (1983), "Erste Erfahrungen bei der Vektorisierung numerischer Verfahren", (First experiences when vectorizing numerical procedures), Computer Center, Technical University, Aachen Germany.

Bratley, P., Fox, B.L., and Schrage, L.E. (1987), *A Guide to Simulation*, Springer-Verlag, New York.

CDC (1986), "Fortran 200 Version 1 reference manual", Publication No. 60480200, Control Data Corporation, Sunnyvale, CA.

Celmaster, W., and Moriarty, K.J.M. (1986), "A method for vectorized random number generators", *Journal of Computational Physics* 64, 271–275.

Churchill, G. (1983), *Marketing Research*, 3rd ed., Dryden Press, Chicago, IL.

De Matteis, A., and Pagnutti, S. (1988), "Parallelization of random number generators and long-range correlations", *Numerische Mathematik* 53, 595–608.

Durst, M.J. (1989), "Using linear congruential generators for parallel random number generation", in: E.A. MacNair, K.J. Musselman and P. Heidelberger (eds.), *Proceedings of the 1989 Winter Simulation Conference*, 462–466.

Fishman, G.S. (1978), *Principles of Discrete Event Simulation*, Wiley/Interscience, New York.

Kalos, M.H., and Whitlock, P.A. (1986), *Monte Carlo Methods, Volume I: Basics*, Wiley, New York.

Kleijnen, J.P.C. (1974), *Statistical Techniques in Simulation, Volume 1*, Marcel Dekker, New York.

Kleijnen, J.P.C. (1989). "Pseudorandom number generation on supercomputers", *Supercomputer* 6, 34–40.

Knuth, D.E. (1981), *The Art of Computer Programming, Volume 2: Semi-numerical Algorithms*, Addison-Wesley, Reading, MA.

Law, A., and Kelton, W. (1982), *Simulation Modelling and Analysis*, McGraw-Hill, New York.

L'Ecuyer, P. (1990), "Random numbers for simulation", *Communications of the ACM* 33, 85–97.

Levine, R.D. (1982), "Supercomputers", *Scientific American*, 112–125.

Marsaglia, G., Zaman, A., and Tsang, W.W. (1990), "Towards a universal random number generator", *Statistics and Probability Letters* 8, 35–39.

Miller, R.K., and Walker, T.C. (1989), *Parallel Processing*, The Fairmont Press, Lilburn, CA.

Morgan, B.J.I. (1984), *Elements of Simulation*, Chapman & Hall, London.

Oed, W. (1982), "Monte Carlo simulation on vector machines", *Angewandte Informatik* 7, 358–364.

Park, S.K., and Miller, K.W. (1988), "Random number generators: Good ones are hard to find", *Communications of the ACM* 31, 1192–1201.

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

SARA (1984), "Cyber 205 user's guide, Part 3. Optimization of Fortran programs", SARA (Stichting Academisch Rekencentrum Amsterdam, Foundation Academic Computer Center Amsterdam), Amsterdam.

Zenios, S.A., and Mulvey, J.M. (1986), "Nonlinear network programming on vector supercomputers: A study on the Cray X-MP", *Operations Research* 34, 667–682.