Available online at www.sciencedirect.com
ScienceDirect
Journal of Computational and Applied Mathematics 216 (2008) 371–387
JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS
www.elsevier.com/locate/cam

The additive congruential random number generator—A special case of a multiple recursive generator

Roy S. Wikramaratna[a,*,1]

[a] *RPS Group plc, A31 Winfrith Technology Centre, Dorchester, Dorset DT2 8DH, UK*

Received 7 June 2006; received in revised form 16 May 2007

Abstract

This paper considers an approach to generating uniformly distributed pseudo-random numbers which works well in serial applications but which also appears particularly well-suited for application on parallel processing systems. Additive Congruential Random Number (ACORN) generators are straightforward to implement for arbitrarily large order and modulus; if implemented using integer arithmetic, it becomes possible to generate identical sequences on any machine.

Previously published theoretical analysis has demonstrated that a kth order ACORN sequence approximates to being uniformly distributed in up to k dimensions, for any given k. ACORN generators can be constructed to give period lengths exceeding any given number (for example, with period length in excess of 2^{30p}, for any given p). Results of empirical tests have demonstrated that, if p is greater than or equal to 2, then the ACORN generator can be used successfully for generating double precision uniform random variates.

This paper demonstrates that an ACORN generator is a particular case of a multiple recursive generator (and, therefore, also a special case of a matrix generator). Both these latter approaches have been widely studied, and it is to be hoped that the results given in the present paper will lead to greater confidence in using the ACORN generators.
© 2007 Elsevier B.V. All rights reserved.

MSC: 65C10; 68U20

Keywords: Pseudo-random number generator; Algorithm; Implementation; Theoretical analysis

1. The ACORN generator

The kth order additive congruential random number (ACORN) generator is defined by Wikramaratna [11,12] from an integer modulus M, an integer seed Y_0^0 satisfying $0 < Y_0^0 < M$ and an arbitrary set of k integer initial

* Tel.: +44 1305 217440.
E-mail addresses: roy@wikramaratna.fsnet.co.uk, WikramaratnaR@rpsgroup.com (R.S. Wikramaratna).

[1] This paper is dedicated to the memory of my uncle, P.H. Diananda (born April 1919, Weligama, Sri Lanka; died 3rd March 2007, Singapore), who was Professor of Mathematics at the University of Singapore from 1962 until 1979 and who was the first of many mathematicians in the family.

0377-0427/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2007.05.018

values Y_0^m, $m = 1, \ldots, k$, each satisfying $0 \leqslant Y_0^m < M$ by the equations

$$Y_n^0 = Y_{n-1}^0, \quad n \geqslant 1, \tag{1}$$

$$Y_n^m = (Y_n^{m-1} + Y_{n-1}^m)_{\text{mod } M}, \quad n \geqslant 1, \quad m = 1, \ldots, k, \tag{2}$$

where by $(Y)_{\text{mod } M}$ we mean the remainder on dividing Y by M. The numbers Y_n^k can be normalised to the unit interval by dividing by M

$$X_n^k = Y_n^k / M, \quad n \geqslant 1. \tag{3}$$

It turns out that the numbers X_n^k defined by Eqs. (1)–(3) approximate to being uniformly distributed on the unit interval in up to k dimensions, provided a few simple constraints on the initial parameter values are satisfied. In short, the modulus M needs to be a large integer (typically a prime power), while the seed Y_0^0 and the modulus should be chosen to be relatively prime. This is the approach that we have adopted in most of our experiments with the ACORN generator, and it appears to work very successfully.

The original implementation proposed in [11] used real arithmetic modulo one, calculating the X_n^k directly. This implementation suffered from a number of conceptual limitations (although in practice these did not appear to detract from the usefulness of the algorithm as a source of uniformly distributed pseudo-random numbers): owing to the effects of rounding errors in real arithmetic the sequences were not necessarily reproducible on different machines or with different compilers (although the different sequences still exhibited similar statistical behaviour); consequently, although period lengths were large they could not be predicted or determined with any certainty; finally, it was not possible to make a clear statement of how best to initialise the generator. Use of an integer implementation based on Eqs. (1)–(3) overcomes these limitations (see Ref. [12]) and the original implementation is therefore considered by the author to be superseded by this newer approach, which is illustrated in Section 4.1.

Extensive empirical testing, including that documented in [11], has demonstrated that the numbers X_n^k approximate to being uniformly distributed in the unit interval [0,1) and satisfy a range of statistical tests of randomness. These tests suggest that increasing the order of the generator improves the randomness; it has also been observed that increasing the modulus improves the randomness of the generator. Further empirical tests have been carried out more recently by the author, making use of the Diehard statistical test suite, Marsaglia [8]. The outcomes of these additional tests are reviewed briefly later in the paper.

Theoretical analysis given by Wikramaratna [12] has shown that the numbers Y_n^m are of the form

$$Y_n^m = \left(\sum_{i=0}^{m} Y_0^i Z_n^{m-i} \right)_{\text{mod } M}, \tag{4}$$

where

$$Z_n^{m-i} = \frac{(n+m-i-1)!}{(n-1)!(m-i)!}. \tag{5}$$

The analysis goes on to show that in fact the kth order ACORN generator approximates to being uniformly distributed in all dimensions up to k (in the sense that the j-tuples $(X_n^k, X_{n+1}^k, \ldots, X_{n+j-1}^k)$ approximate to being uniformly distributed in j dimensions for each $j \leqslant k$).

The ACORN generator has found practical application as a source of pseudo-random numbers, for example in the GSLIB geostatistical library (see [2,3]).

Wikramaratna [13] has considered issues related to the parallelisation of computations that make use of the ACORN algorithm as a source of uniformly-distributed pseudo-random numbers. That paper demonstrates the simplicity of an efficient implementation using a splitting approach to allow parallel Monte-Carlo simulations to be carried out efficiently with arbitrary numbers of processors.

2. Multiple recursive generators

2.1. Definition

In a multiple recursive generator of order k each variate is defined as a linear combination of the previous k variates, calculated using integer arithmetic modulo M. Thus (see for example [6,10]) we can write

$$y_j = (a_1 y_{j-1} + a_2 y_{j-2} + \cdots + a_k y_{j-k})_{\text{mod } M}. \tag{6}$$

As before the y_j can be normalised to the unit interval by dividing by M

$$x_j = \frac{y_j}{M} = \frac{(a_1 y_{j-1} + a_2 y_{j-2} + \cdots + a_k y_{j-k})_{\text{mod } M}}{M}. \tag{7}$$

When $k = 1$ this is simply a multiplicative congruential generator (or equivalently a linear congruential generator with zero additive constant). When $k > 1$, it is called a multiple recursive multiplicative congruential generator, or simply a multiple recursive generator. Knuth [7] has shown that for M prime and under appropriate conditions the maximum period length is equal to $M^k - 1$. For large values of M computational efficiency becomes an issue and various authors have sought particular choices of the a_i for which especially efficient implementations are possible (for example, Deng and Lin [1] restrict the values taken by the a_i to 0, +1 and −1, and consider combinations which satisfy the conditions for maximal period length given by Knuth for the case of $M = 2^{31} - 1$). L'Ecuyer et al. [5] have studied the statistical properties of multiple recursive generators and make recommendations on the choice of multipliers for certain specific moduli, including in particular $M = 2^{31} - 1$.

It will be useful to consider also a slightly more general form of Eq. (6) which we will call a generalised multiple recursive generator of order k

$$y_j = (a_1 y_{j-1} + a_2 y_{j-2} + \cdots + a_k y_{j-k} + c)_{\text{mod } M}, \tag{8}$$

where c is a constant. As before, the y_j can be normalised to the unit interval by dividing by the modulus M. We observe in passing that a linear congruential generator with non-zero additive constant is a generalised multiple recursive generator of order one.

2.2. The ACORN generator as a special case of a multiple recursive generator

Theorem 1. *Every kth order ACORN generator, defined by Eqs.* (1) *and* (2), *is equivalent to a particular case of a generalised multiple recursive generator of order k, Eq.* (8), *for which the constant c is equal to the seed* Y_0^0; *furthermore, the coefficients* a_i *are given by*

$$a_i = \left((-1)^{i+1} \frac{k!}{(k-i)!i!} \right)_{\text{mod } M}, \quad i = 1, \ldots, k. \tag{9}$$

Proof. Consider an arbitrary kth order ACORN generator. We assert that the following equation holds for each $p = 1, \ldots, k$ (and indeed also for $p = k + 1$ provided we define Y_{n+p}^{-1} in an appropriate manner) and for every $n \geqslant 1$

$$Y_{n+p}^{k-p} = (Y_{n+p}^{k-p+1} - Y_{n+p-1}^{k-p+1})_{\text{mod } M} = \left(\sum_{i=0}^{p} (-1)^i \frac{p!}{(p-i)!i!} Y_{n+p-i}^{k} \right)_{\text{mod } M}. \tag{10}$$

In each case, the first part of each equation (10) is a straightforward rearrangement of Eq. (2) where we have set $m = k - p$ and we have made use of the fact that Eq. (2) hold for all $n \geqslant 1$ and therefore in particular for $n + p$.

As an aside, we observe that the first part of Eq. (10) also holds trivially when $p = k + 1$ if we define Y_{n+p}^{-1} in the obvious way, extending Eq. (2) to the case $m = 0$ and rearranging as before.

We proceed to show that the second part of Eq. (10) also holds, by induction on p, with the case $p = 1$ being trivial (observe that in this case there are only two terms in the summation, and that each of the factorial terms

is equal to one). If the second part of Eq. (10) is shown to hold for some $p \leqslant k$, then

$$\begin{aligned} Y_{n+(p+1)}^{k-(p+1)} &= \left(Y_{n+(p+1)}^{k-(p+1)+1} - Y_{n+(p+1)-1}^{k-(p+1)+1}\right)_{\text{mod } M} = \left(Y_{n+p+1}^{k-p} - Y_{n+p}^{k-p}\right)_{\text{mod } M} \\ &= \left(\sum_{i=0}^{p}(-1)^i \frac{p!}{(p-i)!i!} Y_{n+p+1-i}^{k} - \sum_{i=0}^{p}(-1)^i \frac{p!}{(p-i)!i!} Y_{n+p-i}^{k}\right)_{\text{mod } M} \\ &= \left(\sum_{i=0}^{p}(-1)^i \frac{p!}{(p-i)!i!} Y_{n+p+1-i}^{k} + \sum_{j=1}^{p+1}(-1)^j \frac{p!}{(p+1-j)!(j-1)!} Y_{n+p+1-j}^{k}\right)_{\text{mod } M} \\ &= \left(\sum_{i=0}^{p+1}(-1)^i \frac{(p+1)!}{(p+1-i)!i!} Y_{n+p+1-i}^{k}\right)_{\text{mod } M}. \end{aligned} \tag{11}$$

The second last line of Eq. (11) is obtained by putting $j = i + 1$ in the second summation, while the last line is obtained by regrouping the terms from the two summations into a single summation. Hence Eq. (10) holds also for $(p + 1)$ and we have demonstrated, by induction, the correctness of our assertion that Eq. (10) holds for each $p = 1, \ldots, k, k + 1$ and for every $n \geqslant 1$. In particular, putting $p = k$ in (10), and making use of Eq. (1) we get

$$Y_0^0 = Y_{n+k}^0 = \left(Y_{n+k}^1 - Y_{n+k-1}^1\right)_{\text{mod } M} = \left(\sum_{i=0}^{k}(-1)^i \frac{k!}{(k-i)!i!} Y_{n+k-i}^{k}\right)_{\text{mod } M}. \tag{12}$$

Finally, after rearranging Eq. (12) and setting $n + k = j$

$$Y_j^k = \left(\sum_{i=1}^{k}(-1)^{i+1} \frac{k!}{(k-i)!i!} Y_{j-i}^{k} + Y_0^0\right)_{\text{mod } M}. \tag{13}$$

The theorem now follows from a term by term comparison with Eq. (8). □

We observe in passing that the kth order ACORN generator might also be considered to be a $(k + 1)$th order multiple recursive generator. This can be shown by putting $p = k + 1$ in Eq. (10), leading to the result that

$$0 = \left(Y_{n+k+1}^0 - Y_{n+k}^0\right)_{\text{mod } M} = \left(\sum_{i=0}^{k+1}(-1)^i \frac{(k+1)!}{(k+1-i)!i!} Y_{n+k+1-i}^{k}\right)_{\text{mod } M}. \tag{14}$$

After rearranging Eq. (14) and setting $n + k + 1 = j$ this leads finally to

$$Y_j^k = \left(\sum_{i=1}^{k+1}(-1)^{i+1} \frac{(k+1)!}{(k+1-i)!i!} Y_{j-i}^{k}\right)_{\text{mod } M}. \tag{15}$$

This can be seen to be equivalent to a $(k+1)$th order multiple recursive generator by making a term by term comparison with Eq. (6).

3. Matrix generators

3.1. Definition

A k by k matrix generator can be defined (see, for example, [6,10]) by an equation of the form

$$\hat{\mathbf{y}}_j = (A_k \hat{\mathbf{y}}_{j-1} + \hat{\mathbf{c}})_{\text{mod } M}, \tag{16}$$

where $\hat{\mathbf{y}}_j$, $\hat{\mathbf{y}}_{j-1}$ and $\hat{\mathbf{c}}$ are vectors of length k and A_k is a k by k matrix. The constant vector $\hat{\mathbf{c}}$ is often chosen to be the zero-vector; however the more general form of non-zero $\hat{\mathbf{c}}$ is also of interest here.

3.2. The generalised multiple recursive generator as a special case of a matrix generator

It is well known that a multiple recursive generator can be considered as a special case of a matrix generator. This result can be extended to the generalised multiple recursive generator and stated formally as in the following theorem:

Theorem 2. *Every generalised multiple recursive generator of order k, defined by Eq.* (8), *is equivalent to a special case of a k by k matrix generator, Eq.* (16), *for which the components of the different* $\hat{\mathbf{y}}_j$ *form consecutive disjoint sub-sequences of the* y_j. *This matrix generator has matrix*

$$A_k = [G_k]^k \tag{17}$$

and constant

$$\hat{\mathbf{c}} = \left(\sum_{i=1}^{k} [G_k]^{i-1} \right) \mathbf{c}, \tag{18}$$

where

$$G_k = \begin{pmatrix} \mathbf{0}_{k-1} & I_{k-1} \\ a_k & \mathbf{a}_{k-1}^{\mathrm{T}} \end{pmatrix}. \tag{19}$$

Here $\mathbf{0}_{k-1}$ is a column vector of $(k-1)$ zeros; I_{k-1} represents the $(k-1)$ by $(k-1)$ identity matrix; $\mathbf{a}_{k-1}$ is a column vector such that the row vector $\mathbf{a}_{k-1}^{\mathrm{T}}$ is equal to $(a_{k-1}, a_{k-2}, \ldots, a_1)$; the vector $\mathbf{c}$ is equal to $(\mathbf{0}_{k-1}^{\mathrm{T}}, c)^{\mathrm{T}}$; and the superscript $^{\mathrm{T}}$ represents the transpose operation.

Proof. Observe first that a kth order generalised multiple recursive generator can be written in matrix form as

$$\mathbf{y}_j = (G_k \mathbf{y}_{j-1} + \mathbf{c})_{\bmod M}, \tag{20}$$

where the vector $\mathbf{y}_j$ is equal to $(y_{j-k+1}, y_{j-k+2}, \ldots, y_j)^{\mathrm{T}}$ and $\mathbf{y}_{j-1}$ is equal to $(y_{j-k}, y_{j-k+1}, \ldots, y_{j-1})^{\mathrm{T}}$. Thus the first $(k-1)$ components of $\mathbf{y}_j$ are equal to the last $(k-1)$ components of $\mathbf{y}_{j-1}$ and the components of the different $\mathbf{y}_j$ form overlapping subsequences of the y_j. This result (which has been noted previously by other authors, for example, L'Ecuyer [4]) can be demonstrated by substituting from (19) into (20) and considering the resulting equations for each of the components of the y_j in turn. The first $(k-1)$ of these equations are trivially true while a term by term comparison of the last of these with Eq. (8) leads immediately to the desired result. Applying Eq. (20) k times we obtain the equation

$$\mathbf{y}_j = \left([G_k]^k \mathbf{y}_{j-k} + \left\{ \sum_{i=1}^{k} [G_k]^{i-1} \right\} \mathbf{c} \right)_{\bmod M} = (A_k \mathbf{y}_{j-k} + B_k \mathbf{c})_{\bmod M} = (A_k \mathbf{y}_{j-k} + \mathbf{b}_k c)_{\bmod M}. \tag{21}$$

Here $A_k = [G_k]^k$, $B_k = \sum_{i=1}^{k} [G_k]^{i-1}$ and $\mathbf{b}_k$ represents the final column of B_k.

If we now define $\hat{\mathbf{y}}_{j-p} = \mathbf{y}_{j-pk}$ for each integer p (positive, negative or zero) the components of the different $\hat{\mathbf{y}}_j$ form consecutive disjoint sub-sequences of the y_j. Furthermore $\hat{\mathbf{y}}_j = \mathbf{y}_j$, $\hat{\mathbf{y}}_{j-1} = \mathbf{y}_{j-k}$ and it is clear, from a comparison with (16), that Eq. (21) represents a matrix generator for which the constant vector $\hat{\mathbf{c}}$ is given by Eq. (18) (and which is also equal both to $B_k \mathbf{c}$ and to $\mathbf{b}_k c$). □

3.3. The ACORN generator as a special case of a matrix generator

Corollary (*to Theorems 1 and 2*). *Every k*th *order ACORN generator, defined by Eqs.* (1) *and* (2), *is equivalent to a particular case of a k by k matrix generator* (*Eq.* (16)) *for which the components of the different* $\hat{\mathbf{y}}_j$ *form consecutive disjoint sub-sequences of the sequence* Y_j^k. *This matrix generator has matrix*

$$A_k = [G_k]^k = \begin{pmatrix} \mathbf{0}_{k-1} & \boldsymbol{I}_{k-1} \\ a_k & \mathbf{a}_{k-1}^{\mathrm{T}} \end{pmatrix}^k \tag{22}$$

and constant

$$\hat{\mathbf{c}} = \left\{ \sum_{j=1}^{k} [G_k]^{j-1} \right\} \mathbf{c} = B_k \mathbf{c} = \mathbf{b}_k c, \tag{23}$$

where the terms appearing in these equations are as defined in the statement and proof of Theorem 2.

Proof. Theorem 1 has shown that any ACORN generator of order k is equivalent to a generalised multiplicative recursive generator of order k, while Theorem 2 has shown that any generalised multiplicative recursive generator of order k is equivalent to a matrix generator having a k by k matrix and (in general) a non-zero constant vector. Eqs. (22) and (23) follow immediately from Eq. (9) and Eqs. (17)–(19) on making use of the fact that $c = Y_0^0$ (which follows from Theorem 1). □

We will define the resulting matrix generator to be the kth order ACORN matrix generator; the matrix A_k is then the kth order ACORN matrix while the corresponding constant vector is equal to the product of the vector $\mathbf{b}_k$ with the constant Y_0^0, calculated modulo M.

3.4. Some properties of the kth order ACORN matrix

In the following section we make some observations concerning the form of the matrices G_k and A_k and of the vectors $\mathbf{b}_k$ together with some of their properties. At this stage it is not clear how useful these observations will be in practice, although they are of interest from a mathematical point of view; however, it is possible that some of them may provide useful starting points for further theoretical analysis of the ACORN algorithm. All of the observations can be verified for values of k between 2 and 12 by reference to Tables 1–4. Mathematical proofs of the results for arbitrary k are possible, although we will not include these here.

We note first that the kth order ACORN matrix is a k by k matrix. Making use of Eq. (22) we can calculate G_k and A_k for any values of k. Table 1 shows the matrices G_k for values of k between 2 and 12. Table 2 shows the matrices A_k for values of k between 2 and 12, together with the corresponding vectors $\mathbf{b}_k$ (which can be calculated using Eq. (23)).

We observe by inspection that the row sum is 1 for each row of G_k (and also for each row of A_k). We observe further that the determinant of G_k is always equal to 1 (in every case, expanding the determinant using the first column of G_k gives the result $(-1)^{2k}$). Now consider the matrix H_k which is defined by

$$H_k = \begin{pmatrix} \tilde{\mathbf{a}}_{k-1}^{\mathrm{T}} & a_k \\ I_{k-1} & \mathbf{0}_{k-1} \end{pmatrix}, \tag{24}$$

where the vector $\tilde{\mathbf{a}}_{k-1}$ is equal to $(a_1, a_2, \ldots, a_{k-1})^{\mathrm{T}}$. We note that this is the same as the vector $\mathbf{a}_{k-1}$ that appears in Eq. (19), but with the order of the components reversed. It is straightforward to verify, by calculating the products $G_k H_k$ and $H_k G_k$ and making use of the observation that $a_k = (-1)^{k+1}$, $a_{k-s} = (-1)^k a_s = -a_k a_s$ for each positive s, that each of the products is equal to the k by k identity matrix, and hence that H_k is the inverse of G_k. Table 3 shows the inverse of each of the matrices G_k for values of k between 2 and 12.

Finally we observe that the determinant of A_k is also equal to 1 (since the determinant of a product is equal to the product of the determinants) and that the inverse of A_k can be written down as

$$(A_k)^{-1} = (G_k^{-1})^k = (H_k)^k = \begin{pmatrix} \tilde{\mathbf{a}}_{k-1}^{\mathrm{T}} & a_k \\ I_k & \mathbf{0}_{k-1} \end{pmatrix}^k. \tag{25}$$

The fact that the matrices A_k and G_k can be inverted means that we can re-write Eq. (16) in the following form, demonstrating that it is possible to cycle backwards as well as forwards through the ACORN sequences.

$$\hat{\mathbf{y}}_{j-1} = ([A_k]^{-1} \hat{\mathbf{y}}_j - [A_k]^{-1} \hat{\mathbf{c}})_{\mathrm{mod}\, M}. \tag{26}$$

We will define the 'matrix rotate' operation (which we will denote by a superscript $^{\mathrm{R}}$) to be a 180° rotation about the mid-point of the matrix (in effect, reversing the order of both rows and columns); thus if $M = (m_{pq})$ represents

Table 1
Matrix G_k, for values of k between 2 and 12

	Matrix G_k									
Order, $k = 2$	0	1								
	−1	2								
Order, $k = 3$	0	1	0							
	0	0	1							
	1	−3	3							
Order, $k = 4$	0	1	0	0						
	0	0	1	0						
	0	0	0	1						
	−1	4	−6	4						
Order, $k = 5$	0	1	0	0	0					
	0	0	1	0	0					
	0	0	0	1	0					
	0	0	0	0	1					
	1	−5	10	−10	5					
Order, $k = 6$	0	1	0	0	0	0				
	0	0	1	0	0	0				
	0	0	0	1	0	0				
	0	0	0	0	1	0				
	0	0	0	0	0	1				
	−1	6	−15	20	−15	6				
Order, $k = 7$	0	1	0	0	0	0	0			
	0	0	1	0	0	0	0			
	0	0	0	1	0	0	0			
	0	0	0	0	1	0	0			
	0	0	0	0	0	1	0			
	0	0	0	0	0	0	1			
	1	−7	21	−35	35	−21	7			
Order, $k = 8$	0	1	0	0	0	0	0	0		
	0	0	1	0	0	0	0	0		
	0	0	0	1	0	0	0	0		
	0	0	0	0	1	0	0	0		
	0	0	0	0	0	1	0	0		
	0	0	0	0	0	0	1	0		
	0	0	0	0	0	0	0	1		
	−1	8	−28	56	−70	56	−28	8		
Order, $k = 9$	0	1	0	0	0	0	0	0	0	
	0	0	1	0	0	0	0	0	0	
	0	0	0	1	0	0	0	0	0	
	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	1	0	0	0	
	0	0	0	0	0	0	1	0	0	
	0	0	0	0	0	0	0	1	0	
	0	0	0	0	0	0	0	0	1	
	1	−9	36	−84	126	−126	84	−36	9	
Order, $k = 10$	0	1	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	1
	−1	10	−45	120	−210	252	−210	120	−45	10

Table 1 (*Contd*)

	Matrix G_k											
Order, $k = 11$	0	1	0	0	0	0	0	0	0	0	0	
	0	0	1	0	0	0	0	0	0	0	0	
	0	0	0	1	0	0	0	0	0	0	0	
	0	0	0	0	1	0	0	0	0	0	0	
	0	0	0	0	0	1	0	0	0	0	0	
	0	0	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	0	0	1	0	0	0	
	0	0	0	0	0	0	0	0	1	0	0	
	0	0	0	0	0	0	0	0	0	1	0	
	1	−11	55	−165	330	−462	462	−330	165	−55	11	
Order, $k = 12$	0	1	0	0	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	1	0	0	0	0
	0	0	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	0	0	1
	−1	12	−66	220	−495	792	−924	792	−495	220	−66	12

any k by k matrix (so that m_{pq} represents the entry in the pth row and the qth column of the matrix) then we can write

$$M^{\mathrm{R}} = (m_{(k+1-p)(k+1-q)}). \tag{27}$$

It is clear from the above discussion that each of the matrices G_k and A_k have the unusual and quite remarkable property that

$$(G_k)^{-1} = G_k^{\mathrm{R}}, \quad k = 1, 2, 3, \ldots, \tag{28}$$

$$(A_k)^{-1} = A_k^{\mathrm{R}}, \quad k = 1, 2, 3, \ldots . \tag{29}$$

From an inspection of Table 2 we observe that the vector $\mathbf{b}_k$, which is equal to the final column of the matrix B_k, has entries that are all positive and larger in modulus (by a factor of between 1 and 2) compared to the corresponding entries in the first column of the matrix A_k.

Finally we observe that all the entries in each column of A_k are of the same sign, with the final column being positive in each case. Within each column the absolute values of the terms increase as the row number increases. As the order increases the magnitude of the entry in the matrix having the largest absolute value also increases; this is illustrated by Table 4, which shows the magnitude of the term with largest absolute value (respectively $g_{\max}, a_{\max}$) in each of the matrices G_k andA_k for values of k between 2 and 12. Table 4 also shows the logarithm (base 2) of these numbers—this allows us to see how quickly the magnitude of the terms in the matrices grow compared to a typical modulus of $M = 2^{60}$ for a practical implementation of the ACORN random number generator.

4. Discussion

In this paper we have shown that the kth order ACORN generator is a particular case of a generalised multiple recursive generator of order k (and also of a matrix generator having a k by k matrix). The ACORN generators are a family of generators having a prescribed form for each value of k, together with an extremely simple implementation (no more than a few tens of lines in any high-level computing language, as will be illustrated in Section 4.1) that can be

Table 2
Matrix A_k and corresponding vector b_k, for values of k between 2 and 12

	Matrix A_k								Vector b_k
Order, $k = 2$	−1	2							1
	−2	3							3
Order, $k = 3$	1	−3	3						1
	3	−8	6						4
	6	−15	10						10
Order, $k = 4$	−1	4	−6	4					1
	−4	15	−20	10					5
	−10	36	−45	20					15
	−20	70	−84	35					35
Order, $k = 5$	1	−5	10	−10	5				1
	5	−24	45	−40	15				6
	15	−70	126	−105	35				21
	35	−160	280	−224	70				56
	70	−315	540	−420	126				126
Order, $k = 6$	−1	6	−15	20	−15	6			1
	−6	35	−84	105	−70	21			7
	−21	120	−280	336	−210	56			28
	−56	315	−720	840	−504	126			84
	−126	700	−1575	1800	−1050	252			210
	−252	1386	−3080	3465	−1980	462			462
Order, $k = 7$	1	−7	21	−35	35	−21	7		1
	7	−48	140	−224	210	−112	28		8
	28	−189	540	−840	756	−378	84		36
	84	−560	1575	−2400	2100	−1008	210		120
	210	−1386	3850	−5775	4950	−2310	462		330
	462	−3024	8316	−12320	10395	−4752	924		792
	924	−6006	16380	−24024	20020	−9009	1716		1716
Order, $k = 8$	−1	8	−28	56	−70	56	−28	8	1
	−8	63	−216	420	−504	378	−168	36	9
	−36	280	−945	1800	−2100	1512	−630	120	45
	−120	924	−3080	5775	−6600	4620	−1848	330	165
	−330	2520	−8316	15400	−17325	11880	−4620	792	495
	−792	6006	−19656	36036	−40040	27027	−10296	1716	1287
	−1716	12936	−42042	76440	−84084	56056	−21021	3432	3003
	−3432	25740	−83160	150150	−163800	108108	−40040	6435	6435

Table 2 (*Contd*)

	Matrix A_k												Vector b_k
Order, $k = 9$	1	−9	36	−84	126	−126	84	−36	9				1
	9	−80	315	−720	1050	−1008	630	−240	45				10
	45	−396	1540	−3465	4950	−4620	2772	−990	165				55
	165	−1440	5544	−12320	17325	−15840	9240	−3168	495				220
	495	−4290	16380	−36036	50050	−45045	25740	−8580	1287				715
	1287	−11088	42042	−91728	126126	−112112	63063	−20592	3003				2002
	3003	−25740	97020	−210210	286650	−252252	140140	−45045	6435				5005
	6435	−54912	205920	−443520	600600	−524160	288288	−91520	12870				11440
	12870	−109395	408408	−875160	1178100	−1021020	556920	−175032	24310				24310
Order, $k = 10$	−1	10	−45	120	−210	252	−210	120	−45	10			1
	−10	99	−440	1155	−1980	2310	−1848	990	−330	55			11
	−55	540	−2376	6160	−10395	11880	−9240	4752	−1485	220			66
	−220	2145	−9360	24024	−40040	45045	−34320	17160	−5148	715			286
	−715	6930	−30030	76440	−126126	140140	−105105	51480	−15015	2002			1001
	−2002	19305	−83160	210210	−343980	378378	−280280	135135	−38610	5005			3003
	−5005	48048	−205920	517440	−840840	917280	−672672	320320	−90090	11440			8008
	−11440	109395	−466752	1166880	−1884960	2042040	−1485120	700128	−194480	24310			19448
	−24310	231660	−984555	2450448	−3938220	4241160	−3063060	1432080	−393822	48620			43758
	−48620	461890	−1956240	4849845	−7759752	8314020	−5969040	2771340	−755820	92378			92378
Order, $k = 11$	1	−11	55	−165	330	−462	462	−330	165	−55	11		1
	11	−120	594	−1760	3465	−4752	4620	−3168	1485	−440	66		12
	66	−715	3510	−10296	20020	−27027	25740	−17160	7722	−2145	286		78
	286	−3080	15015	−43680	84084	−112112	105105	−68640	30030	−8008	1001		364
	1001	−10725	51975	−150150	286650	−378378	350350	−225225	96525	−25025	3003		1365
	3003	−32032	154440	−443520	840840	−1100736	1009008	−640640	270270	−68640	8008		4368
	8008	−85085	408408	−1166880	2199120	−2858856	2598960	−1633632	680680	−170170	19448		12376
	19448	−205920	984555	−2800512	5250960	−6785856	6126120	−3818880	1575288	−388960	43758		31824
	43758	−461890	2200770	−6235515	11639628	−14965236	13430340	−8314020	3401190	−831402	92378		75582
	92378	−972400	4618900	−13041600	24249225	−31039008	27713400	−17054400	6928350	−1679600	184756		167960
	184756	−1939938	9189180	−25865840	47927880	−61108047	54318264	−33256080	13430340	−3233230	352716		352716
Order, $k = 12$	−1	12	−66	220	−495	792	−924	792	−495	220	−66	12	1
	−12	143	−780	2574	−5720	9009	−10296	8580	−5148	2145	−572	78	13
	−78	924	−5005	16380	−36036	56056	−63063	51480	−30030	12012	−3003	364	91
	−364	4290	−23100	75075	−163800	252252	−280280	225225	−128700	50050	−12012	1365	455
	−1365	16016	−85800	277200	−600600	917280	−1009008	800800	−450450	171600	−40040	4368	1820
	−4368	51051	−272272	875160	−1884960	2858856	−3118752	2450448	−1361360	510510	−116688	12376	6188
	−12376	144144	−765765	2450448	−5250960	7916832	−8576568	6683040	−3675672	1361360	−306306	31824	18564
	−31824	369512	−1956240	6235515	−13302432	19953648	−21488544	16628040	−9069840	3325608	−739024	75582	50388
	−75582	875160	−4618900	14671800	−31177575	46558512	−49884120	38372400	−20785050	7558200	−1662804	167960	125970
	−167960	1939938	−10210200	32332300	−68468400	101846745	−108636528	83140200	−44767800	16166150	−3527160	352716	293930
	−352716	4064632	−21339318	67387320	−142262120	210882672	−224062839	170714544	−91454220	32829720	−7113106	705432	646646
	−705432	8112468	−42493880	133855722	−281801520	416440024	−440936496	334639305	−178474296	63740820	−13728792	1352078	1352078

Table 3
Inverse of matrix G_k, for values of k between 2 and 12

	Inverse of matrix G_k									
Order, $k = 2$	2	−1								
	1	0								
Order, $k = 3$	3	−3	1							
	1	0	0							
	0	1	0							
Order, $k = 4$	4	−6	4	−1						
	1	0	0	0						
	0	1	0	0						
	0	0	1	0						
Order, $k = 5$	5	−10	10	−5	1					
	1	0	0	0	0					
	0	1	0	0	0					
	0	0	1	0	0					
	0	0	0	1	0					
Order, $k = 6$	6	−15	20	−15	6	−1				
	1	0	0	0	0	0				
	0	1	0	0	0	0				
	0	0	1	0	0	0				
	0	0	0	1	0	0				
	0	0	0	0	1	0				
Order, $k = 7$	7	−21	35	−35	21	−7	1			
	1	0	0	0	0	0	0			
	0	1	0	0	0	0	0			
	0	0	1	0	0	0	0			
	0	0	0	1	0	0	0			
	0	0	0	0	1	0	0			
	0	0	0	0	0	1	0			
Order, $k = 8$	8	−28	56	−70	56	−28	8	−1		
	1	0	0	0	0	0	0	0		
	0	1	0	0	0	0	0	0		
	0	0	1	0	0	0	0	0		
	0	0	0	1	0	0	0	0		
	0	0	0	0	1	0	0	0		
	0	0	0	0	0	1	0	0		
	0	0	0	0	0	0	1	0		
Order, $k = 9$	9	−36	84	−126	126	−84	36	−9	1	
	1	0	0	0	0	0	0	0	0	
	0	1	0	0	0	0	0	0	0	
	0	0	1	0	0	0	0	0	0	
	0	0	0	1	0	0	0	0	0	
	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	1	0	0	0	
	0	0	0	0	0	0	1	0	0	
	0	0	0	0	0	0	0	1	0	
Order, $k = 10$	10	−45	120	−210	252	−210	120	−45	10	−1
	1	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	1	0

Table 3 (*Contd*)

	Inverse of matrix G_k										
Order, $k = 11$	11	−55	165	−330	462	−462	330	−165	55	−11	1	
	1	0	0	0	0	0	0	0	0	0	0	
	0	1	0	0	0	0	0	0	0	0	0	
	0	0	1	0	0	0	0	0	0	0	0	
	0	0	0	1	0	0	0	0	0	0	0	
	0	0	0	0	1	0	0	0	0	0	0	
	0	0	0	0	0	1	0	0	0	0	0	
	0	0	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	0	0	1	0	0	0	
	0	0	0	0	0	0	0	0	1	0	0	
	0	0	0	0	0	0	0	0	0	1	0	
Order, $k = 12$	12	−66	220	−495	792	−924	792	−495	220	−66	12	−1
	1	0	0	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	1	0	0	0	0
	0	0	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	0	0	1	0

Table 4
Magnitude of term with the largest absolute value in the matrices G_k and A_k

	$g_{\max}$	$\log_2(g_{\max})$	$a_{\max}$	$\log_2(a_{\max})$
Order, $k = 2$	2	1.0	3	1.6
Order, $k = 3$	3	1.6	15	3.9
Order, $k = 4$	6	2.6	84	6.4
Order, $k = 5$	10	3.3	540	9.1
Order, $k = 6$	20	4.3	3465	11.8
Order, $k = 7$	35	5.1	24024	14.6
Order, $k = 8$	70	6.1	163800	17.3
Order, $k = 9$	126	7.0	1178100	20.2
Order, $k = 10$	252	8.0	8314020	23.0
Order, $k = 11$	462	8.9	61108047	25.9
Order, $k = 12$	924	9.9	440936496	28.7

generalised to arbitrarily large order k and modulus M. By making an appropriate choice of k the resulting sequence approximates to uniformity in up to k dimensions. A kth order ACORN generator can be selected with a period in excess of any prescribed value by making an appropriate choice of the modulus M. Simple conditions can be prescribed for the initialisation of the generator with very few restrictions on the choice of initial conditions. Provided the initial conditions are chosen to satisfy these conditions the ACORN sequences that result all appear to have uniformly good statistical properties. This overcomes a key limitation which applies as a general rule both to multiple recursive and matrix generators—that it is not straightforward to determine what would constitute a good set of initial conditions. By implementing an ACORN generator in integer arithmetic it also becomes possible to generate identical sequences on any machine (to within the accuracy of the machine representation of real numbers in the unit interval).

4.1. Implementation

The ACORN algorithm is very simple to implement in any high-level programming language, requiring no more than a few tens of lines of code. There follows an example implementation in Fortran 77 which will work for a modulus of 2^{60} and for any order less than or equal to 120. It is easy to generalise this implementation to larger values of the modulus (for example 2^{90} or 2^{120}) and order, or to convert to other programming languages such as C. It is worth noting that increasing the modulus from 2^{60} to 2^{120} (i.e. squaring the modulus, which also gives a comparable increase in the period length) leads only to a doubling of the computational effort required to generate each variate; on the other hand doubling the order of the generator also leads to a doubling of the computational effort.

```
      DOUBLE PRECISION FUNCTION ACORNJ(XDUMMY)

C
C        Fortran implementation of ACORN random number generator
C        of order less than or equal to 120 (higher orders can be
C        obtained by increasing the parameter value MAXORD) and
C        modulus less than or equal to 2^60.
C
C        After appropriate initialization of the common block /IACO2/
C        each call to ACORNJ generates a single variate drawn from
C        a uniform distribution over the unit interval.
C
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      PARAMETER (MAXORD=120,MAXOP1=MAXORD+1)
      COMMON /IACO2/ KORDEJ,MAXJNT,IXV1(MAXOP1),IXV2(MAXOP1)
      DO 7 I=1,KORDEJ
        IXV1(I+1)=(IXV1(I+1)+IXV1(I))
        IXV2(I+1)=(IXV2(I+1)+IXV2(I))
        IF (IXV2(I+1).GE.MAXJNT) THEN
          IXV2(I+1)=IXV2(I+1)-MAXJNT
          IXV1(I+1)=IXV1(I+1)+1
        ENDIF
        IF (IXV1(I+1).GE.MAXJNT) IXV1(I+1)=IXV1(I+1)-MAXJNT
    7 CONTINUE
      ACORNJ=(DBLE(IXV1(KORDEJ+1))
     1               +DBLE(IXV2(KORDEJ+1))/MAXJNT)/MAXJNT
      RETURN
      END
```

An even more general version of the ACORN algorithm which is implemented in Fortran 90 and which works for arbitrarily large modulus and order is available to researchers for download from Numerical Algorithms Group Ltd., Oxford, United Kingdom, via their web site (http://www.nag.co.uk/nagware/Examples/Acorn.asp). The NAG version also includes an example implementation of the stride algorithm, which allows you to take steps of arbitrary length through an ACORN sequence; in the example implementation the stride length is restricted to be a power of 2 while the modulus is a larger power of 2. The application of the stride algorithm to the parallelisation of Monte-Carlo simulations, using the ACORN generator as a source of pseudo-random numbers, has been considered in some detail in a previous paper [13] and will not be discussed further here.

4.2. Initialisation

In order to initialise a kth order ACORN generator it is necessary to specify $k+1$ parameters (the seed Y_0^0 and the k initial values Y_0^i, $i = 1, \ldots k$). In order to initialise a kth order multiple recursive or matrix generator it is necessary to specify the first k values in the sequence and the additive constant. This has been put forward as a difficulty in using these latter generators, as it is not always clear how best to choose an appropriate initial condition. For the ACORN generator, implemented using integer arithmetic, the choice of initialisation is straightforward—for modulus M, choose a seed relatively prime with M and an arbitrary set of k initial values (note that this really does mean arbitrary—the generator will perform equally well if the initial values are all set to zero, to 1 or to some other constant value, or if they are set to take different values such as 12345, 9876, 24680, 99321, etc.). For the equivalent multiple recursive or matrix generators, the problem of initialisation is now easily solved—since we could simply choose an appropriate

seed and initial values for the ACORN generator, calculate the first k values in the ACORN sequence and then use these values together with the seed to define initial conditions for the equivalent generalised multiple recursive or matrix generators. In practice it is much more efficient (particularly with large values of the modulus) to make use of the ACORN algorithm for generating the sequences, rather than to use either the generalised multiplicative recursive or matrix forms of the generator.

For the implementation shown in Section 4.1, initialisation is as follows. The parameters KORDEJ (the order of the generator which must be less than or equal to MAXORD), (IXV1(i), IXV2(i), $i = 1, \ldots,$ KORDEJ + 1) and MAXJNT (which should be set equal to 2^{30}) in the common block must be initialised by the user before the first call to ACORNJ; after initialisation, the common block should not be further modified by the user. Each subsequent call to the function ACORNJ then generates a single random variate which approximates to being uniformly distributed on the unit interval. The seed and initial values are integers modulo 2^{60}, each represented by a pair of integers modulo 2^{30}. Thus the condition that the seed is odd is equivalent to initialising IXV2(1) to an arbitrary odd integer and IXV1(1) to an arbitrary integer, each less than 2^{30}, while the arbitrary choice of initial values is equivalent to initialising IXV2(i) and IXV1(i), $i = 1, \ldots,$ KORDEJ to arbitrary integer values each less than 2^{30}. The value taken by the seed is then given by

$$Y_0^0 = 2^{30}\mathrm{IXV1}(1) + \mathrm{IXV2}(1) \tag{30}$$

while the ith initial value (where $1 \leqslant i \leqslant k$) is given by

$$Y_0^i = 2^{30}\mathrm{IXV1}(1+i) + \mathrm{IXV2}(1+i). \tag{31}$$

4.3. Periodicity

Wikramaratna [12] has shown that the period length of an ACORN sequence with modulus equal to a power of two will be an integer multiple of the modulus, provided only that the seed is chosen to be odd. For the implementation considered above, initialised according to the specified criteria, this gives a period length in excess of 2^{60}. If this is insufficient for a particular application then the period length of the sequence can be increased, effectively without limit, simply by increasing the value of the modulus to a suitably large power of 2 and again choosing the seed to take an odd value. There is an obvious and extremely straightforward generalisation of the implementation in Section 4.1 for modulus 2^{90} or 2^{120}, giving access to sequences with period lengths in excess of these values.

It appears that it is in fact possible to derive some much more general results concerning the periodicity of ACORN generators. Based on the results of numerical experiments that were undertaken with a wide range of choices of modulus, seed and initial value, we propose the following conjecture (for which we have not as yet derived a rigorous proof) concerning the period length of ACORN generators with prime power modulus:

Conjecture 1. *Let X_n^k be a kth order ACORN generator, defined by Eqs.* (1)–(3), *with modulus equal to a prime power, say $M = q^t$, where q is a prime and suppose that the seed and modulus are chosen to be relatively prime. Then the sequence X_n^k, $k = 1, \ldots, n$ will have a period length equal to $q^i M = q^{i+t}$, where i is the largest integer such that $q^i \leqslant k$.*

We make a number of observations concerning this conjecture:

(i) The condition that the seed and modulus are chosen to be relatively prime is equivalent to the seed not being a multiple of q.
(ii) If q is equal to 2, then this condition reduces to the requirement that the seed should take an odd value.
(iii) The result holds regardless of the choice of initial values, provided that the specified restriction is placed on the value taken by the seed.
(iv) The restriction on the value taken by the seed is necessary—consider for example the case with seed equal to q and initial values equal to zero; in this case the first order generator has a shorter period length, equal to (M/q) rather than M.

Table 5 summarises what Conjecture 1 means in practice when the modulus is a power of certain specific primes. As concrete examples, taking the modulus equal to 2^{60} and order equal to 10 gives rise to a sequence having a period

Table 5
Relationship between modulus, order and period length for ACORN generators with prime power modulus (and with seed and modulus relatively prime)

$M = 2^t$		$M = 3^t$		...	$M = q^t$, q prime	
Order k	Period	Order k	Period		Order k	Period
$k = 1$	M	$1 \leqslant k < 3$	M		$1 \leqslant k < q$	M
$2 \leqslant k < 4$	$2M$	$3 \leqslant k < 9$	$3M$		$q \leqslant k < q^2$	qM
$4 \leqslant k < 8$	$4M$	$9 \leqslant k < 27$	$9M$		$q^2 \leqslant k < q^3$	$q^2 M$
...		...		...	...	
$2^i \leqslant k < 2^{i+1}$	$2^i M$	$3^i \leqslant k < 3^{i+1}$	$3^i M$		$q^i \leqslant k < q^{i+1}$	$q^i M$

Table 6
Time (in seconds) to generate one million random variates using ACORN generator with modulus 2^{30p} and different orders

p	Order $k = 5$	Order $k = 10$	Order $k = 15$	Order $k = 20$
1	1.05	2.00	2.87	3.82
2	1.47	2.87	4.25	5.62
3	1.98	3.86	5.69	7.51
4	2.39	4.79	7.13	9.36
5	2.82	5.72	8.48	11.25
6	3.43	7.00	10.45	13.97

length of 2^{63}, while taking the modulus equal to 2^{90} and order equal to 16 gives rise to a sequence with period length of 2^{94} (provided only that in each case the seed is chosen to be odd).

4.4. Computational performance

Table 6 compares the computational performance of ACORN generators of order k equal to 5, 10, 15 and 20 and with modulus equal to 2^{30p} for values of p between 1 and 6. In each case, the table shows the time in seconds taken to generate one million variates. The timing comparisons were made using the generalised implementation available from Numerical Algorithms Group Ltd (referenced at the end of Section 4.1) run under Windows 2000 Professional on a Pentium III 600 MHz processor with 128 Mb memory, using a Compaq Visual Fortran 6 compiler. It is clear from the table that the computational time scales approximately both with k and with p.

For comparison, the equivalent time to generate one million variates for the subroutine G05LGF from the NAG Fortran Library [9] is 1.62 s. G05LGF is an implementation of a linear congruential generator which has a period length of 2^{57}. The routine was set to generate a single variate rather than a vector of variates at each call to allow a meaningful comparison to be made.

We can conclude that the computational performance of a 10th order ACORN generator with modulus 2^{60} is comparable, to within a factor of 2, with that of a linear congruential generator having a similar period length. Owing to the simplicity of the algorithm it may be feasible to further improve the computational speed of the ACORN algorithm, for example by coding it in-line or by making modifications to the implementation to tune it to specific hardware. However, it is considered unlikely in practice that the generation of the random numbers will consume more than a small proportion of the total execution time in a Monte-Carlo calculation so that the benefits of doing this are likely to be marginal. We observe in passing that as the modulus increases beyond 2^{60}, the efficiency of the ACORN algorithm results in increasing performance benefits for the ACORN approach when compared with multiplicative generators having similar modulus—consider for example an ACORN generator with modulus 2^{120}, which requires only a doubling of the computational effort per random variate compared with an equivalent ACORN generator with modulus 2^{60}.

4.5. Results of some further empirical tests

As stated in the introduction to this paper, further empirical tests have been carried out recently by the author, making use of the Diehard statistical test suite, Marsaglia [8]. In this section we will briefly review the approach taken together

with the key results and conclusions. It is not our intention to document these tests in detail in the present paper since the primary focus here is on the theoretical results in Sections 2 and 3.

The Diehard tests work on a file of 32-bit integers written in binary and require some 2^{23} 32-bit integers in order to run the full set of tests. The procedure used to test the ACORN random number generator was as follows. A test program was written, in Fortran, to generate an ASCII file containing 32-bit integers written ten integers per line in hexadecimal with eight hexadecimal digits per integer and no intervening spaces. The ASCII file was then converted to a binary file in appropriate format using the program ASC2BIN.EXE (from the Diehard download). Finally the program DIEHARD.EXE was used to run the full Diehard test suite using the binary file as input. The results of testing were inspected to check the p-values calculated for each of the tests (where we have taken the criterion for failure on any particular test to be a p-value equal either to one or zero, to at least four decimal places).

Initially tests were carried out on bits 1 to 32 of sequences of random variates generated using the ACORN algorithm with modulus 2^{60} and a range of values for the order, between 10 and 100. Results of testing on the leading 32 bits showed that the ACORN generator passed all the tests, irrespective of the order of generator (between 10 and 100) or the particular initialisation selected, provided only that the seed was chosen to be odd.

In order to provide a direct comparison and to allow a preliminary assessment of the significance of these results, the same tests have been applied in an identical way to a linear congruential generator with modulus 2^{59} and multiplier 13^{13}. This is a standard generator that has been used, for example by NAG [9] in the routine G05CAF from their mathematical software libraries, as a basis for generating single-precision uniform variates in [0, 1). This generator (which has a period length that is known to be greater than 2^{57}, provided that it is initialised with an odd value) also passed all of the Diehard tests when they were applied to the high-order bits 1–32 of the resulting variates.

Although in principal the Diehard test suite is capable of generalisation to test more than 32 consecutive bits, such a generalisation is not currently available to us. However, by applying the Diehard tests separately to the 32-bit strings formed by bits 17–48 as well as to those formed by bits 1–32, the Diehard tests can in effect be applied to variates with more than 32 bits without needing to make any modifications to the Diehard test suite.

Results of testing on bits 17–48 showed that the ACORN generators typically failed some of the tests on the lower order bits (44–48) but in general passed all the tests that were restricted to bits up to about 43. Increasing the order led to a small increase in the number of bits that passed all the tests (perhaps one or two extra bits as a result of increasing the order from 10 to 100).

When the same tests were applied to bits 17–48 from the NAG routine G05CAF the results were very different, with the generator failing at least one of the tests for all of the bits 33–48. This suggests that while the leading 32 bits are random and uniformly distributed, subsequent bits are not. While the generator appears to be adequate for generating single-precision variates in [0, 1), as in the NAG application, it would have limitations if it were to be used as a basis for generation of double-precision variates. We note that this routine is in fact scheduled for replacement in the next release of the NAG libraries; however, we include these results as an example to indicate what might be expected from a good linear congruential generator with similar period to the ACORN generators that were being tested.

The results described above suggest that the ACORN algorithm with modulus 2^{60} can be used with confidence for generation of double precision variates in [0, 1), which will be uniform and random to the available precision of 13 or so decimal places (equivalent to 43 binary digits). If this was considered to be insufficient for any particular application (for example an application using quadruple-precision arithmetic for which it was considered important to achieve uniformity over the full number of available digits), then variates with a larger number of random digits could be obtained by simply increasing the modulus of the generator to a larger power of two. For most practical implementations a modulus equal to 2^{30p} for a small integer value of p (for example $p = 2, 3$ or 4) combined with a order of at least 10 or more (for example, choosing the order equal to 15) appears to be a reasonable choice.

The most remarkable observation from these results is that the ACORN generators appear to give uniformly good results over the whole range of possible sequences (subject only to a few very straightforward constraints on the modulus, seed, and initial values) and that the quality of the results is as good as (or better than) that for a sequence, with comparable period length, obtained using what is considered to be among the ‘best’ choices of generally available linear congruential generators. Increasing the modulus of the ACORN sequence appears to increase the number of digits that are ‘random’, at the same time as increasing the period length. Further work is required to establish exactly how the number of random digits scales with increasing modulus.

These results contrast with those obtained for linear congruential generators where only a very small proportion of possible combinations of seed, multiplier and modulus yield good distribution properties–so that identifying the small

proportion of good generators entails a large amount of effort in testing different combinations and rejecting most of them. If an increased period length or an increase in the number of 'random' digits is required then it is necessary to increase the modulus and then start testing multipliers all over again. We observe that a good multiplier with one value of the modulus might quite possibly be a poor multiplier for a different value of the modulus, so that the search for a good multiplier really does need to start again from scratch for every different modulus that is considered.

It may be thought that making a comparison like this between a kth order ACORN generator and a multiplicative congruential generator having a similar modulus is unfair, given that it has now been demonstrated that the kth order ACORN generator is in fact a special case of a kth order multiple recursive generator. The justification for making the comparison in this way is that both generators require similar computational effort to generate each random variate (this is a consequence of the exceptionally efficient computational algorithm that can be implemented for the ACORN generator and, as we have observed elsewhere in this paper, the computational benefits that can be gained from the ACORN approach increase even further with increasing modulus and order). In addition, the ACORN generator is being considered as an alternative to (and possibly an eventual replacement for) this particular linear congruential generator, which remains very widely used in practice. From both these points of view we consider that the comparison that we have made is perfectly valid.

5. Conclusions

This paper has shown that the ACORN generator can be viewed as a particular form either of a generalised multiple recursive generator or of a generalised matrix generator that is particularly simple to compute for arbitrarily large modulus and order. In each case, the matrix is a full matrix with no zeroes, while increasing the order gives increasingly large matrix elements, with alternating positive and negative sign. Use of the ACORN algorithm gives rise to a computationally efficient method of calculation without the restriction of requiring a sparse matrix with only a few small non-zero entries. In addition, there is no practical limit on the modulus used, allowing sequences to be generated efficiently with effectively unlimited period length.

Previous papers have shown that the ACORN generators are amenable to theoretical analysis, which can be used to demonstrate various properties (for example long period, approximation to k-distribution, etc.) that are desirable in a source of pseudo-random numbers. By following the approach outlined in this paper and viewing the ACORN generator as a particular case of either a multiple recursive or matrix generator, it may be possible in the future to apply some aspects of the analysis of these generators to further improve our understanding of and build additional confidence in the use of the ACORN algorithm.

References

[1] L. Deng, D.K.J. Lin, Random number generation for the new century, Amer. Stat. 54 (2000) 145–150.
[2] C.V. Deutsch, Geostatistical Reservoir Modelling, Oxford University Press, Oxford, UK, 2002.
[3] C.V. Deutsch, A.G. Journel, GSLIB Geostatistical Software Library and User's Guide (Applied Geostatistics Series), second ed., Oxford University Press, Oxford, UK, 1997.
[4] P. L'Ecuyer, Random numbers for simulation, Commun. ACM 33 (1990) 86–97.
[5] P. L'Ecuyer, F. Blouin, R. Couture, A search for good multiple recursive random number generators, ACM Trans. Modell. Comput. Simul. 3 (1993) 87–98.
[6] J.E. Gentle, Random Number Generation and Monte-Carlo Methods, second ed., Springer, Berlin, 2003.
[7] D. Knuth, The Art of Computer Programming, Volume 2-Seminumerical Algorithms, second ed., Addison-Wesley, Reading, MA, 1998.
[8] G. Marsaglia, The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, Florida State University, FL, USA, ⟨http://stat.fsu.edu/pub/diehard⟩, 1995.
[9] NAG, Numerical Algorithms Group (NAG) Fortran Library Mark 21, Numerical Algorithms Group Ltd, Oxford, UK, 2004. ⟨www.nag.co.uk⟩.
[10] H. Niederreiter, Random Number Generation and Quasi-Monte Carlo Methods, CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA, USA, 1992.
[11] R.S. Wikramaratna, ACORN—a new method for generating sequences of uniformly distributed pseudo-random numbers, J. Comput. Phys. 83 (1989) 16–31.
[12] R.S. Wikramaratna, Theoretical background for the ACORN random number generator, Report AEA-APS-0244, AEA Technology, Winfrith, Dorset, UK, 1992.
[13] R.S. Wikramaratna, Pseudo-random number generation for parallel processing—a splitting approach, SIAM News 33 (9) (2000).