Efficient Jump Ahead for $\mathbb{F}_2$-linear Random Number Generators

Hiroshi Haramoto, Makoto Matsumoto
Department of Mathematics, Hiroshima University, Kagamiyama 1-3-1
Higashi-Hiroshima, Hiroshima 739-8526, Japan
{m-mat@math.sci.hiroshima-u.ac.jp}

Takuji Nishimura
Department of Mathematics, Yamagata University, Yamagata 990-8586, Japan
{nisimura@sci.kj.yamagata-u.ac.jp}

François Panneton and Pierre L'Ecuyer
Département d'Informatique et de Recherche Opérationnelle, Université de Montréal
C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada
{panneton@iro.umontreal.ca and lecuyer@iro.umontreal.ca}

The fastest long-period random number generators currently available are based on linear recurrences modulo 2. So far, software that provides multiple disjoint streams and substreams has not been available for these generators because of the lack of efficient jump-ahead facilities. In principle, it suffices to multiply the state (a k-bit vector) by an appropriate $k \times k$ binary matrix to find the new state far ahead in the sequence. However, when k is large (e.g., for a generator such as the popular Mersenne twister, for which $k = 19937$), this matrix-vector multiplication is slow and a large amount of memory is required to store the $k \times k$ matrix. In this paper, we provide a faster algorithm to jump ahead by a large number of steps in a linear recurrence modulo 2. The method uses much less than the k^2 bits of memory required by the matrix method. It is based on polynomial calculus modulo the characteristic polynomial of the recurrence and uses a sliding window algorithm for the multiplication.

Key words: simulation; random number generation; jumping ahead; multiple streams

1. Introduction

Random number generators (RNGs) with multiple disjoint streams and substreams are an important component of any good general-purpose simulation or statistical software. They are very handy, for example, to obtain parallel RNGs and to support the implementation of variance reduction techniques (Kelton, 2006; Law and Kelton, 2000; L'Ecuyer et al., 2002).

The most convenient way of getting these streams and substreams is to start with a backbone RNG having a huge period, and partition its output sequence into long disjoint subsequences and subsubsequences whose starting points are at equidistant lags. When a new stream is needed, we find its starting point by jumping ahead from the starting point of the current subsequence to the starting point of the next one. Substreams are obtained from subsequences in a similar way. To make sure that no overlap occurs, the streams and substreams must be very long, so that they cannot be exhausted even with days of computing time. To implement this, we need to know how to quickly jump ahead by large lags in the sequence of numbers produced by the generator.

Most generators used for simulation are based on linear recurrences. For these generators, the state $\mathbf{x}_n$ at step n is a vector of k integers in $\{0, \ldots, m-1\}$ for some integer m called the modulus, and it evolves as $\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} \bmod m$ where $\mathbf{A}$ is a $k \times k$ matrix with elements in $\{0, \ldots, m-1\}$. To jump ahead by ν steps from any state $\mathbf{x}_n$, regardless of how large is ν, it suffices to precompute the matrix $\mathbf{A}^\nu \bmod m$ (once for all) and then compute $\mathbf{x}_{n+\nu} = (\mathbf{A}^\nu \bmod m)\mathbf{x}_n \bmod m$ by a simple matrix-vector multiplication. This technique is used to provide streams and substreams in the random number package of L'Ecuyer et al. (2002), based on combined multiple recursive generators (CMRG), and which has been adopted in several simulation and statistical software products such as Arena, Automod, Witness, SSJ, SAS, etc.

There are faster generators than the CMRG, based on linear recurrences modulo 2, with extremely long periods and good statistical properties. The Mersenne twister and the WELL (Matsumoto and Nishimura, 1998; Panneton et al., 2006), for example, belong to that class. But efficient software that provides multiple disjoint streams and substreams for them is lacking. Since these generators are linear, the technique just described applies in principle (with $m = 2$). But the matrix-vector multiplication is slow if implemented naively and an excessive amount of memory is required to store the matrix when k is very large, which is typical. For example, the Mersenne twister generator has $k = 19937$. Then, the $k \times k$ binary matrix occupies 47.4 MBytes of memory!

We propose a more efficient technique to perform this multiplication. It uses a representation of the recurrence in a space of polynomials. For a given step size ν, the state $\mathbf{x}_{n+\nu}$ is expressed as a polynomial in $\mathbf{A}$ of degree less than k, say $g(\mathbf{A})$, multiplied by $\mathbf{x}_n$. A key ingredient is that we use the implementation of the original recurrence (i.e., of the generator) to compute the product $g(\mathbf{A})\mathbf{x}_n$. The most expensive operations in this computation turn out to be the k-bit vector additions modulo 2. We use a sliding window technique to reduce the number of these additions, e.g., by a factor of about 4 when $k = 19937$. For this particular value of k, with the proposed method, the generator can jump ahead for an arbitrary lag in less than 5 milliseconds on a 32-bit 3.0 GHz computer.

The next section gives a framework for $\mathbb{F}_2$-linear generators and states the jump-ahead problem. In Section 3, we examine how to jump ahead, explain our proposed technique, and analyze its computational efficiency. The algorithm is stated in Section 3.4 and some timings are given in Section 3.5.

2. $\mathbb{F}_2$-Linear Generators

All along this paper, arithmetic operations are assumed to be performed in $\mathbb{F}_2$, the finite field with two elements, represented as 0 and 1. This corresponds to doing arithmetic modulo 2. Note that in $\mathbb{F}_2$, subtraction and addition are equivalent; so we can always write "+" instead of "−" and we do so everywhere in this paper. The RNGs considered obey the general $\mathbb{F}_2$-linear recurrence

$$\mathbf{x}_n \quad = \quad \mathbf{A}\mathbf{x}_{n-1}, \tag{1}$$

where $\mathbf{x}_n = (x_{n,0}, \ldots, x_{n,k-1})^{\mathsf{t}} \in \mathbb{F}_2^k$ is the k-bit *state vector* at step n and $\mathbf{A}$ is the $k \times k$ *transition matrix* with elements in $\mathbb{F}_2$. The *output* can be defined by any transformation $\mathbf{x}_n \mapsto u_n$; the exact form of this transformation is irrelevant for the remainder of the paper. Usually, the output $u_n \in [0, 1)$ at step n is defined by $u_n = \sum_{\ell=1}^{w} y_{n,\ell-1} 2^{-\ell}$ for some positive integer w, where $\mathbf{y}_n = (y_{n,0}, \ldots, y_{n,w-1})^{\mathsf{t}} = \mathbf{B}\mathbf{x}_n$ and $\mathbf{B}$ is a $w \times k$ matrix with elements in $\mathbb{F}_2$. Several types of RNGs fit this framework, including the Tausworthe or linear feedback shift register (LFSR), the generalized feedback shift register (GFSR), the twisted GFSR (TGFSR), the Mersenne twister, the WELL, and xorshift generators (Tezuka, 1995; Matsumoto and Nishimura, 1998; L'Ecuyer and Panneton, 2005; Panneton and L'Ecuyer, 2005; Panneton et al., 2006).

Our aim is to compute

$$\mathbf{x}_{n+\nu} \quad = \quad \mathbf{A}^{\nu}\mathbf{x}_n \tag{2}$$

for a large value of ν, say larger than 2^{100} or even more. We assume that ν is fixed in advance and that (2) must be computed for several arbitrary vectors $\mathbf{x}_n$ unknown in advance. This is what we need to implement multiple streams and substreams. The algorithm also works if ν is not fixed, but then the computationally expensive setup must be repeated each time.

3. Jumping Ahead

3.1. Matrix Method

A first method to jump ahead is the standard one, described in the introduction: We start by precomputing the matrix $\mathbf{J} = \mathbf{A}^{\nu}$ in $\mathbb{F}_2$. By a standard square-and-multiply exponentiation technique (Knuth, 1998), this requires $O(k^3 \log \nu)$ operations, and we need k^2 bits to store $\mathbf{J}$. Then, whenever jumping-ahead is required from state $\mathbf{x}$, we compute the vector $\mathbf{Jx}$. To obtain the ith element of $\mathbf{Jx}$, we compute the componentwise product of the ith row of $\mathbf{J}$ by the (transposed) vector $\mathbf{x}$, by a bitwise AND, and add the bits of the resulting vector, modulo 2. A straightforward implementation of this on a w-bit computer requires $k\lceil k/w \rceil$ AND operations, followed by k^2 operations to count the bits.

But the work to add the bits modulo 2 can be reduced as follows. Observe that we only need the *parity* of the sum of bits in the vector, which can be obtained by xoring all its bits. This can be achieved as follows: partition the k-bit vector into w-bit blocks (this is how it is stored), xor all these blocks together (for a given vector, this requires $\lceil k/w \rceil$ xor operations), then xor the bits in the resulting w-bit block (w operations). The total number of operations with this approach is $2k\lceil k/w \rceil + kw$.

Nevertheless, for $k = 19937$, for instance, storing $\mathbf{J}$ takes around 47.4 megabytes of memory, and computing it by squaring and multiplying the binary matrices is impractical (each squaring takes $O(k^3)$ time).

3.2. Using the Polynomial Representation

A more efficient approach, when k is large, works with the polynomial representation of the recurrence, as follows. Write the *characteristic polynomial* of the matrix $\mathbf{A}$ as

$$p(z) = \det(z\mathbf{I} + \mathbf{A}) = z^k + \alpha_1 z^{k-1} + \cdots + \alpha_{k-1} z + \alpha_k,$$

where $\mathbf{I}$ is the identity matrix and $\alpha_j \in \mathbb{F}_2$ for each j, and recall that

$$p(\mathbf{A}) = z^k + \alpha_1 \mathbf{A}^{k-1} + \cdots + \alpha_{k-1} \mathbf{A} + \alpha_k \mathbf{I} = 0$$

(this is a fundamental property of the characteristic polynomial). Let

$$g(z) = z^\nu \bmod p(z) = a_1 z^{k-1} + \cdots + a_{k-1} z + a_k. \tag{3}$$

This $g(z)$ can be computed (once for all) in $O(k^2 \log \nu)$ time by the square-and-multiply method (Knuth, 1998) in the space of polynomials modulo $p(z)$. Observe that $g(z) = z^\nu + q(z)p(z)$ for some polynomial $q(z)$. Combining this with the fact that $p(\mathbf{A}) = 0$, we see that $g(\mathbf{A}) = \mathbf{A}^\nu$ and thus

$$\mathbf{J} = \mathbf{A}^\nu = g(\mathbf{A}) = a_1 \mathbf{A}^{k-1} + \cdots + a_{k-1}\mathbf{A} + a_k \mathbf{I}.$$

Therefore, $\mathbf{Jx}$ can be computed by

$$\begin{aligned} \mathbf{Jx} &= (a_1 \mathbf{A}^{k-1} + \cdots + a_{k-1}\mathbf{A} + a_k \mathbf{I})\mathbf{x} \\ &= \mathbf{A}(\cdots \mathbf{A}(\mathbf{A}(\mathbf{A} a_1 \mathbf{x} + a_2 \mathbf{x}) + a_3 \mathbf{x}) + \cdots + a_{k-1}\mathbf{x}) + a_k \mathbf{x}, \end{aligned} \tag{4}$$

where the latter represents Horner's method for polynomial evaluation. To compute this, we can simply advance the RNG by $k-1$ steps from state $\mathbf{x}$ and add (by bitwise exclusive-or) the states obtained at the steps that correspond to the nonzero a_j's. This computation requires running the RNG for $k-1$ steps and adding at most $(k-1)$ k-bit vectors. For a random polynomial $g(z)$ (whose coefficients $a_1, \ldots, a_k$ are drawn uniformly over the set of all 2^k possibilities), there is on average $k/2$ nonzero coefficients, so $(k/2)-1$ vector additions are required. This computation still demands $O(k^2)$ operations, but only k bits of storage are needed for the coefficients of $g(z)$.

To examine more closely the cost of this implementation, let us suppose that the computer has w-bit words and that $g(z)$ has $k/2$ nonzero coefficients. Each k-bit vector addition requires $\eta = \lceil k/w \rceil$ XOR operations on the computer, so we need $((k/2)-1)\eta \approx k^2/(2w)$ operations to add the vectors if we use a "standard" method.

It is important to recall here that the large-period $\mathbb{F}_2$-linear RNGs are normally designed so that $\mathbf{Ax}$ can be computed with only a handful of binary operations (such as XORs, shifts, and bit masks). Suppose our RNG needs c such operations at each step. Then we need $(k-1)c$ operations to advance the RNG by $k-1$ steps. The total jump-ahead cost is thus $(k-1)c + ((k/2)-1)\eta$ operations.

As a typical illustration, take $w = 32$, $k = 19937$, and $c = 10$. Then $(k-1)c \approx 2.0 \times 10^5$ whereas $((k/2)-1)\eta \approx 6.2 \times 10^6$, so the vector additions dominate the cost. Our next improvement will reduce this number of additions in exchange for some additional storage.

3.3. Improvement via Decomposition and a Sliding Window

We choose a small positive integer q, say somewhere from 4 to 10. Let $\mathbb{T}_q$ be the set of polynomials with coefficients in $\mathbb{F}_2$ and of degree exactly q, i.e., of the form $h(z) = z^q + b_1 z^{q-1} + \cdots + b_q$ where the b_j's are in $\mathbb{F}_2$. This set has cardinality 2^q. We decompose $g(z)$ as

$$g(z) \quad = \quad h_1(z)z^{d_1} + \cdots + h_m(z)z^{d_m} + h_{m+1}(z) + z^q \tag{5}$$

where $h_j(z) \in \mathbb{T}_q$ for $j = 1, \ldots, m+1$, $0 \le d_m < \cdots < d_1 < k$, and m is as small as possible. This decomposition is obtained as follows. We write the coefficients of $g(z)$ in a sequence:

$$a_1 \; a_2 \; a_3 \; \cdots a_{k-1} \; a_k.$$

If $i_1 = \min\{i > 0 : a_i = 1\}$ is the index of the first nonzero coefficient in the sequence, we set $d_1 = k - q - i_1$, we define

$$h_1(z) = z^q + a_{i_1+1} z^{q-1} + \cdots + a_{i_1+q},$$

and we remove $a_1, \ldots, a_{i_1+q}$ from the sequence. Then we repeat the same process with the sequence that starts with a_{i_1+q+1} to define d_2 and $h_2(z)$, and so on. In general, if $i_j = \min\{i > i_{j-1} + q : a_i = 1\}$ is the index of the first nonzero element in the sequence $a_{i_{j-1}+q+1}, \ldots, a_k$ and if $k - i_j \ge q$, we put $d_j = k - q - i_j$ and

$$h_j(z) = z^q + a_{i_j+1} z^{q-1} + \cdots + a_{i_j+q}.$$

As soon as $k - i_j < q$, we put $m = j - 1$ and

$$h_{m+1}(z) = z^q + a_{i_j} z^{k-i_j} + \cdots + a_k.$$

This completes the decomposition (5). Note that this decomposition does not depend on $\mathbf{x}$.

Now, from (5), we can rewrite

$$\begin{aligned} \mathbf{Jx} \quad &= \quad g(\mathbf{A})\mathbf{x} \\ &= \quad \mathbf{A}^{d_m}(\cdots(\mathbf{A}^{d_2-d_3}(\mathbf{A}^{d_1-d_2}h_1(\mathbf{A})\mathbf{x} + h_2(\mathbf{A})\mathbf{x}) + h_3(\mathbf{A})\mathbf{x}) + \cdots \\ &\qquad + h_m(\mathbf{A})\mathbf{x}) + h_{m+1}(\mathbf{A})\mathbf{x} + \mathbf{A}^q\mathbf{x}. \end{aligned} \tag{6}$$

To compute $\mathbf{Jx}$, we first compute the vectors $h(\mathbf{A})\mathbf{x}$ for all 2^q polynomials $h(z)$ in $\mathbb{T}_q$, and store these vectors in a table. Then, we start the RNG from state $h_1(\mathbf{A})\mathbf{x}$, advance it by

$d_1 - d_2$ steps, add $h_2(\mathbf{A})\mathbf{x}$ to its state, advance it by $d_2 - d_3$ steps, $\ldots$, add $h_m(\mathbf{A})\mathbf{x}$ to the state, advance the RNG by d_m steps, and finally add $h_{m+1}(\mathbf{A})\mathbf{x} + \mathbf{A}^q\mathbf{x}$. We still need to advance the RNG by a total of $k-1$ steps, but $m+1$ vector additions now suffice instead of $k/2$, where $m \leq \lceil k/(q+1) \rceil$.

This method is a direct adaptation of the *sliding window algorithm* used for exponentiation in a group (Möller, 2005). We illustrate it by an example.

Example 1 Let $k = 18$, $q = 3$, and suppose that the coefficients of $g(z)$ are the following:

$$\overset{a_1}{0}\quad \overset{a_2}{0}\quad \underbrace{\overset{a_3}{1}\quad \overset{a_4}{1}\quad \overset{a_5}{1}\quad \overset{a_6}{1}}_{h_1(z)}\quad \overset{a_7}{0}\quad \underbrace{\overset{a_8}{1}\quad \overset{a_9}{0}\quad \overset{a_{10}}{0}\quad \overset{a_{11}}{0}}_{h_2(z)}\quad \underbrace{\overset{a_{12}}{1}\quad \overset{a_{13}}{1}\quad \overset{a_{14}}{1}\quad \overset{a_{15}}{0}}_{h_3(z)}\quad \overset{a_{16}}{0}\quad \overset{a_{17}}{0}\qquad \underbrace{\overset{a_{18}}{1}}_{h_4(z)+z^3}$$

In this case, we have $i_1 = 3$, $d_1 = 15 - i_1 = 12$, $h_1(z) = z^3 + z^2 + z + 1$, $i_2 = 8$, $d_2 = 15 - i_2 = 7$, $h_2(z) = z^3$, $i_3 = 12$, $d_3 = 15 - i_3 = 3$, $h_3(z) = z^3 + z^2 + z$, and $h_4(z) = z^3 + 1$.

We also need an efficient method to compute the 2^q vectors $h(\mathbf{A})\mathbf{x}$, because this has to be redone for each new vector $\mathbf{x}$. These 2^q vectors can be computed efficiently by using a Gray code (Savage, 1997) to represent the elements of $\mathbb{T}_q$: we enumerate these elements as $t_0(z), t_1(z), \ldots, t_{2^q-1}(z)$ so that $t_0(z) = z^q$ and any two successive elements in the sequence differ by a single coefficient. This is Gray code enumeration. The first vector $t_0(\mathbf{A})\mathbf{x} = \mathbf{A}^q\mathbf{x}$ is computed automatically when we advance the RNG, and then each vector $t_i(\mathbf{A})\mathbf{x}$ is computed from the previous one, $t_{i-1}(\mathbf{A})\mathbf{x}$, by adding or subtracting (in $\mathbb{F}_2$ this is the same) a single vector of the form $\mathbf{A}^j\mathbf{x}$ for $0 \leq j < q$. These q vectors are precomputed when we advance the generators by q steps.

In the next subsection, we put together these ingredients to define our algorithm.

3.4. Algorithm

The algorithm has two parts: (a) a one-time setup for each jump size ν and (b) the jumping from $\mathbf{x}_n$ to $\mathbf{x}_{n+\nu}$. It is described in Figure 1.

To summarize the computing costs in the second part, we need $(k-1)c$ operations to advance the RNG by $k-1$ steps, then $2^q - 1$ vector additions to compute all the vectors $t_i(\mathbf{A})\mathbf{x}$, and a further $m + 1 \leq 1 + \lceil k/(q+1) \rceil$ additions to compute (6). Since each vector addition requires η operations, the total cost is at most

$$(k-1)c + (2^q - 1 + m + 1)\eta \;\leq\; (k-1)c + (2^q + \lceil k/(q+1) \rceil)\eta$$

```
Preliminary setup for a given $\nu$.
Compute the polynomial $g(z)$ in (3) and store its coefficients in an array.
Select $q > 0$ and choose a Gray code to enumerate the polynomials of $\mathbb{T}_q$
    (equivalently, the integers $0, 1, \ldots, 2^q - 1$).
Compute $m, d_1, \ldots, d_m, h_1(z), \ldots, h_m(z), h_{m+1}(z), c_1, \ldots, c_m, c_{m+1}$,
    where $c_j$ is the Gray code of $h_j(z)$, i.e., $h_j(z) = t_{c_j}(z)$, for each $j$.

Jump-ahead by $\nu$ steps, from state $\mathbf{x}$.
Compute $\mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \ldots, \mathbf{A}^q\mathbf{x}$ by running the RNG for $q$ steps.
$\mathbf{y}_0 \leftarrow t_0(\mathbf{x}) = \mathbf{A}^q\mathbf{x}$.
For $i = 1, \ldots, 2^q - 1$ do
    Compute $\mathbf{y}_{i+1} = t_{i+1}(\mathbf{A})\mathbf{x}$ from $\mathbf{y}_i = t_i(\mathbf{A})\mathbf{x}$; this requires a single vector XOR.
$\mathbf{x}' \leftarrow \mathbf{y}_{c_1}$.
For $j = 2, \ldots, m$ do
    $\mathbf{x}' \leftarrow \mathbf{A}^{d_{j-1}-d_j}\mathbf{x}' + \mathbf{y}_{c_j}$.
$\mathbf{x}' \leftarrow \mathbf{A}^{d_m}\mathbf{x}' + \mathbf{y}_{c_{m+1}} + \mathbf{y}_0$.
Return $\mathbf{x}'$.
```

Figure 1: The jump-ahead algorithm

operations. The value of q can be chosen to minimize this number, i.e., minimize the upper bound $n_{\mathrm{a}}(k, q) = 2^q + \lceil k/(q+1) \rceil$ on the number of vector additions (since the term $(k-1)c$ does not depend on q). As an illustration, Table 1 gives the value of $n_{\mathrm{a}}(k, q)$ as a function of q for $k = 19937$ and $w = 32$. The minimum is attained for $q = 8$. The case $q = 0$ refers to the computation via the ordinary Horner method given in (4), for which the table gives the expected number of vector additions for a random polynomial. With $q = 8$, the number of additions is reduced approximately by a factor of 4.

This algorithm can still be applied when the value of ν is not fixed, but then the (costly) preliminary setup must be repeated each time.

Table 1: Value of $n_{\mathrm{a}}(k, q)$ as a function of q for $k = 19937$.

q	0	...	4	5	6	7	8	9	10	11	12
$n_{\mathrm{a}}(k, q)$	9968	...	4004	3355	2913	2621	2472	2506	2837	3710	5630

3.5. Timings

We made the following experiment to measure the CPU time required by the proposed algorithm to jump ahead by an arbitrary number of steps, from an arbitrary state, for

Table 2: Required memory size and average CPU time (in milliseconds) for a random jump ahead, for $k = 19937$, on Pentium 4 (32-bit) and AMD Athlon (64-bit) computers.

	q	0	4	5	6	7	8	9	10
	Memory (Kb)	2.5	39	78	156	312	624	1248	2496
32-bit Pentium	MT19937, $w = 32$	15.9	5.8	5.1	4.7	4.3	4.5	5.1	6.8
	WELL19937, $w = 32$	15.9	6.3	5.4	4.9	4.6	4.7	5.4	7.4
	MT19937-64, $w = 64$	19.8	7.1	6.2	5.7	5.2	5.3	6.1	8.4
64-bit Athlon	MT19937, $w = 32$	9.0	3.9	3.6	3.3	3.2	3.3	4.5	6.7
	WELL19937, $w = 32$	9.7	4.0	3.8	3.5	3.4	3.5	4.8	7.1
	MT19937-64, $w = 64$	5.6	2.4	2.4	2.3	2.1	2.3	2.9	5.0

Table 3: Required memory size and average CPU time (in milliseconds) for a random jump ahead, for $k = 1024$, on Pentium 4 (32-bit) and AMD Athlon (64-bit) computers.

	q	0	3	4	5	6	7
	Memory (Kb)	0.13	1.0	2.0	4.0	8.0	16.0
32-bit Pentium	WELL1024, $w = 32$	0.098	0.069	0.068	0.075	0.077	0.092
64-bit Athlon	WELL1024, $w = 32$	0.096	0.060	0.062	0.068	0.071	0.086

$k = 19937$ with both the Mersenne twister and a WELL generator, $k = 1024$ with a WELL generator, and various values of q. We generated a polynomial $g(z)$ and a state $\mathbf{x}$ at random, uniformly over the set of possible values, and measured the time to compute $g(\mathbf{A})\mathbf{x}$. Thus, $g(z)$ was a polynomial of degree $\leq k$ and $\mathbf{x}$ was a k-bit vector. We replicated this 1000 times and computed the average CPU time for jumping ahead, in milliseconds (msec). Table 2 reports these CPU times and the required memory size for each method, on the following computers: (a) An Intel Pentium 4 at 3.0 GHz, with 1.0 GB of memory, using the `gcc` compiler with the `-O2` option, under Linux; and (b) a 64-bit AMD-Athlon 64 3200+, with 2.0 GB of memory, also using Linux and the same compiler. The tested generators are the 32-bit MT19937 (Matsumoto and Nishimura, 1998), the 32-bit WELL19937 (Panneton et al., 2006), and a 64-bit Mersenne Twister named MT19937-64 (Nishimura, 2000).

The timings show that the proposed jump-ahead algorithm is viable even with $q = 0$. For $k = 19937$, the sliding window with a good value of q provides a speedup by a factor of about 3 on a 32-bit computer. It also requires 312 Kb of memory, but for most practical applications this is not a serious drawback given the memory sizes currently available. There is more improvement on the Pentium than on the AMD Athlon, and this especially true for the MT19937-64 generator. Note that the speed does not necessarily double when going from a 32-bit to a 64-bit processor, for several reasons (memory access is not twice faster, the Pentium and Athlon are different, etc.). Even for $k = 1024$ (a small value) the sliding window remains advantageous.

A similar experiment with a clever implementation of the matrix method of Section 3.1 gave the following results: For $k = 19937$, the jump ahead took 24.5 msec on the 32-bit Pentium and 17.0 msec on the 64-bit Athlon, on average. For $k = 1024$, the timings were 0.117 msec on the 32-bit Pentium and 0.096 msec on the 64-bit Athlon, on average. This is roughly five times slower than the proposed method for $k = 19937$ and 50% slower for $k = 1024$.

We also estimated the time to precompute $z^\nu \bmod p(z)$ using C++ and the NTL library (`http://www.shoup.net/ntl`) for the polynomial calculations. For this, we generated 1000 values of ν randomly and uniformly in $\{0, 1, \ldots, 2^{64} - 1\}$, and measured the average CPU time to compute $z^\nu \bmod p(z)$. The average was 239 milliseconds for $k = 19937$ and 1.9 milliseconds for $k = 1024$. This is much more than the time required to jump ahead.

4. Conclusions

We have developed a viable jump-ahead algorithm for large linear RNGs over $\mathbb{F}_2$. With this technique, one can easily implement RNG packages with multiple streams and substreams, based on long-period generators such as the Mersenne twister and the WELL with period length of $2^{19937} - 1$. For these generators, jumping ahead takes a few milliseconds with the proposed method. This is still significantly slower than for the `MRG32k3a` generator in L'Ecuyer et al. (2002), whose jump-ahead time is a few microseconds. On the other hand, `MRG32k3a` is slower to generate its numbers, by a factor of 2 or 3 on common 32-bit computers, and has a much shorter period length. For applications where jumping ahead is not required too frequently and where a fast long-period RNG is desired, the new jump-ahead algorithm comes very handy.

Acknowledgments

This study is partially supported by JSPS/Ministry of Education Grant-in-Aid for Scientific Research No. 18654021 and 16204002, JSPS Core-to-Core Program No. 18005, NSERC-Canada grant No. ODGP0110050, and a Canada Research Chair to the last author.

References

Kelton, W. D. 2006. Implementing representations of uncertainty. S. G. Henderson, B. L. Nelson, eds., *Simulation*. Handbooks in Operations Research and Management Science, Elsevier, Amsterdam, The Netherlands, 181–191. Chapter 7.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Addison-Wesley, Reading, Mass.

Law, A. M., W. D. Kelton. 2000. *Simulation Modeling and Analysis*. 3rd ed. McGraw-Hill, New York.

L'Ecuyer, P., F. Panneton. 2005. Fast random number generators based on linear recurrences modulo 2: Overview and comparison. *Proceedings of the 2005 Winter Simulation Conference*. IEEE Press, 110–119.

L'Ecuyer, P., R. Simard, E. J. Chen, W. D. Kelton. 2002. An object-oriented random-number package with many long streams and substreams. *Operations Research* **50** 1073–1075.

Matsumoto, M., T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* **8** 3–30.

Möller, B. 2005. Sliding window exponentiation. H. C. A. van Tilborg, ed., *Encyclopedia of Cryptography and Security*. Springer-Verlag, New York, 588–590.

Nishimura, T. 2000. Tables of 64-bit Mersenne twisters. *ACM Transactions on Modeling and Computer Simulation* **10** 348–357.

Panneton, F., P. L'Ecuyer. 2005. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation* **15** 346–361.

Panneton, F., P. L'Ecuyer, M. Matsumoto. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software* **32** 1–16.

Savage, C. 1997. A survey of combinatorial Gray codes. *SIAM Review* **39** 605–629.

Tezuka, S. 1995. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Publishers, Norwell, Mass.