A Fast Jump Ahead Algorithm for Linear Recurrences in a Polynomial Space *

Hiroshi Haramoto[1], Makoto Matsumoto[1], and Pierre L'Ecuyer[2]

[1] Dept. of Math., Hiroshima University, Hiroshima 739-8526 JAPAN,
m-mat@math.sci.hiroshima-u.ac.jp,
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/eindex.html
[2] Département d'Informatique et de Recherche Opérationnelle,
Université de Montréal, Montréal, Canada
lecuyer@iro.umontreal.ca,
http://www.iro.umontreal.ca/~lecuyer

Abstract. Linear recurring sequences with very large periods are widely used as the basic building block of pseudorandom number generators. In many simulation applications, multiple streams of random numbers are needed, and these multiple streams are normally provided by jumping ahead in the sequence to obtain starting points that are far apart. For maximal-period generators having a large state space, this jumping ahead can be costly in both time and memory usage. We propose a new jump ahead method for this kind of situation. It requires much less memory than the fastest algorithms proposed earlier, while being approximately as fast (or faster) for generators with a large state space such as the Mersenne twister.

1 Introduction

Pseudorandom number generators (PRNGs) are widely used in many scientific areas, such as simulation, statistics, and cryptography. Generating multiple disjoint streams of pseudorandom number sequences is important for the smooth implementation of variance-reduction techniques on a single processor (see [1–3] for illustrative examples), as well as in conjunction with parallel computing. Generators with multiple streams and substreams have been already adopted, or are in the process of being adopted, in leading edge simulation software tools such as Arena, Automod, MATLAB, SAS, Simul8, SSJ, Witness, and ns2, for example. The substreams are normally obtained by splitting the sequence of a large-period generator into long disjoint subsequences whose starting points are equidistant in the original sequence, say J steps apart. To obtain the initial state (or starting point) of a new substream, we must jump ahead by J steps in the original sequence from the initial state of the most recently created substream.

* This work was supported in part by JSPS Grant-In-Aid #16204002, #18654021, #19204002, JSPS Core-to-Core Program No.18005, NSERC-Canada, and a Canada Research Chair to the third author.

In many cases, this must be done thousands of times (or more) in a simulation, so we need an efficient algorithm for this jump ahead. Unfortunately, for huge-period generators such as the Mersenne twister and the WELL [4, 5], for example, efficient jump ahead is difficult. For this reason, most current implementations use a base generator whose state space is not so large (e.g., 200 bits or so), and this limits the period length.

When the PRNG is based on a linear recurrence, a simple way to jump ahead is to express the recurrence in matrix form, precompute and store the J-th power of the relevant matrix, and jump ahead via a simple matrix-vector multiplication. But for large-period generators, this method requires an excessive amount of memory and is much too slow. Haramoto et al. [6] introduced a reasonably fast jumping-ahead algorithm based on a sliding-window method. One drawback of this method, however, is that it requires the storage of a large precomputed table, at least in its fastest version.

The new method proposed in this paper is based on a representation of the linear recurrence in a space of formal series, where jumping ahead corresponds to multiplying the series by a polynomial. It requires much less memory than the previous one, and is competitive in terms of speed. Under a certain condition on the output function, the speed is actually $O(k^{\log_3 2}) \approx O(k^{1.59})$ for a k-bit state space, compared with $O(k^2)$ for the previous method. For the Mersenne twister with period length $2^{19937} - 1$, this condition is satisfied and the new method turns out to be faster, according to our experiments.

The remainder is organized as follows. In Section 2, we define the setting in which these jumping-ahead methods are applied, and we briefly summarize the previously proposed techniques. The new method is explained in Section 3. Its application to the Mersenne twister is discussed in Section 5. Section 6 reports timing experiments.

2 Setting and Summary of Existing Methods

Many practical generators used for simulation are based on linear recurrences, because important properties such as the period and the high-dimensional distribution can then be analyzed easily by linear algebra techniques. For notational simplicity, our description in this paper is in the setting of a linear recurrence in a field of characteristic 2, i.e., we assume that the base field is the two-element field $\mathbb{F}_2$. However, the proposed method is valid for any finite field.

We consider a PRNG with state space $S := \mathbb{F}_2^k$, for some integer $k > 0$, and (state) transition function $f : S \to S$, linear over $\mathbb{F}_2$. Thus, f can be identified with its representation matrix F, a binary matrix of size $k \times k$, and a state $s \in S$ is then a k-dimensional column vector. For a given initial state s_0, the state evolves according to the recurrence

$$s_{m+1} := f(s_m) = Fs_m, \qquad m = 0, 1, 2, \ldots. \tag{1}$$

An output function $o : S \to O$ is specified, where O is the set of output symbols, and $o(s)$ is the output when we are in state s. Thus, the generator produces a

sequence of elements $o(s_0), o(s_1), \ldots$, of O. For example, in the stream cipher, o is called the filter, and high nonlinearity is required. In principle, jumping ahead should depend only on f, and not on o. However, the algorithm introduced in this paper assumes that o has a specific (linear) form and that one can easily reconstruct the state from a sequence of k successive output values.

Our purpose is to provide an efficient procedure `Jump` that computes $f^J(s)$ for arbitrary states s, for a given huge integer J. Typically, J is fixed and taken large enough to make sure that a stream will never use more than J numbers in a simulation.

A naive implementation of `Jump` is to precompute the matrix power $A := F^J$ (in $\mathbb{F}_2$) and store it (this requires k^2 bits of memory). Then, $f^J(s)$ is just the matrix-vector multiplication As. However, modern generators with huge state spaces, such as the Mersenne Twister [4] and the WELL [7], for which $k = 19937$ or more, merely storing A requires too much memory, and the matrix-vector multiplication is very time-consuming.

The alternative proposed in [6] works as follows. Let $\varphi_F(t)$ be the minimal polynomial of F. (The method also works if we use the characteristic polynomial $\det(tI - F)$ instead of $\varphi_F(t)$.) First, we precompute and store the coefficients of

$$g(t) := t^J \bmod \varphi_F(t) = \sum_{i=0}^{k-1} a_i t^i. \tag{2}$$

This requires only k bits of memory. Then, `Jump` can be implemented using Horner's method:

$$F^J s_0 = g(F)s_0 = F(\cdots F(F(F(a_{k-1}s_0) + a_{k-2}s_0) + a_{k-3}s_0) + \cdots) + a_0 s_0. \tag{3}$$

An important remark is that the matrix-vector multiplication Fs corresponds to advancing the generator's state by one step as in (1). This operation is usually very fast: good generators are designed so that it requires only a few machine instructions. Thus, when computing the right side of (3), the addition of vectors dominates the computing effort. In this procedure, assuming that $g(t)$ is precomputed, k applications of F and approximately $k/2$ vector additions are required for implementing `Jump`. Since these are k-bit vectors, this means an $O(k^2)$ computing time.

The speed can be improved by a standard method called the sliding window algorithm, which precomputes a table that contains $h(F)s_0$ for all polynomials $h(t)$ of degree less than or equal to some constant q, and uses this table to compute (3) $(q+1)$ digits at a time. This requires $2^q k$ bits of memory for the table (this can be significant when k is huge), but the number of (time-consuming) vector additions is decreased to roughly

$$2^q + \lceil k/(q+1) \rceil. \tag{4}$$

The integer q can be selected to optimize the speed, while paying attention to the memory consumption of $2^q k$ bits; see [6] for the details. The new method proposed in the next section does not require such a large table.

3 Jumping by Fast Polynomial Multiplication

A linear recurrence over $\mathbb{F}_2$ can be represented in different spaces and it is not difficult (at least in principle) to switch from one representation to the other [8, 9]. The basic PRNG implementation usually represents the state as a k-bit vector and computes the matrix-vector product in (1) by just a few elementary operations. In other representations, used for example to verify maximal period conditions and to analyze the multidimensional uniformity of the output values, the state is represented as a polynomial or as a formal series [10, 9]. Here, we will use a formal series representation of the state, switch to that representation to perform the jumping ahead, and then recover the state in the original representation. A key practical requirement is the availability of an efficient method to perform this last step.

For our purpose, we assume that the output function o returns a single bit; that is, we have $o : S \to \mathbb{F}_2$. For the Mersenne twister, for example, the output at each step is a block of 32 bits and we can just pick up the most significant bit. We also assume that the mapping $S \to \mathbb{F}_2^k$ which maps the generator's state to the next k bits of output is one-to-one, so we can recover the state from k successive bits of output. This assumption is not restrictive: for example, if the period of this single-bit output is $2^k - 1$, which is usually the case in practice, then the assumption is satisfied by comparing the cardinality of the state space and the set of the k successive bits.

More specifically, let

$$G(s,t) = \sum_{i=1}^{\infty} o(s_{i-1})t^{-i},$$

which is the generating function of the output sequence when the initial state is $s_0 = s$. Note that $G(s_1,t) = t\,G(s_0,t) \pmod{\mathbb{F}_2[t]}$, so that

$$G(s_J,t) = t^J G(s_0,t) \pmod{\mathbb{F}_2[t]} = g(t)G(s_0,t) \pmod{\mathbb{F}_2[t]},$$

because $\varphi_F(t) \in \mathbb{F}_2[t]$. To recover the state s_J, we only need the coefficients of $t^{-1}, \ldots, t^{-k}$ in $G(s_J,t) = g(t)G(s_0,t)$, i.e., the truncation of $G(s_J,t)$ to its first k terms. This means that we can replace $G(s_0,t)$ by its truncation to its first $2k$ terms, or equivalently by the truncation of $t^{2k}G(s_0,t)$ to its first $2k$ terms, which gives the polynomial

$$h(s_0,t) = \sum_{i=0}^{2k-1} o(s_i)t^{2k-1-i}.$$

We can then compute the polynomial product $g(t)h(s_0,t)$, and observe that the coefficients of $t^{2k-1}, \ldots, t^k$ in this polynomial are exactly the output bits $o(s_J), \ldots, o(s_{J+k-1})$, from which we can recover the state s_J.

With the classical (standard) method, we need $O(k^2)$ bit operations just to multiply the polynomials $g(t)$ and $h(s_0,t)$, so we are doing no better than with

the method of [6]. Polynomial multiplication can be done with only $O(k \log k)$ bit operations using fast Fourier transforms, but the hidden constants are larger and the corresponding algorithm turns out to be slower when implemented, for the values of k that we are interested in. A third approach, implemented in the NTL library [11], is Karatsuba's algorithm (see, e.g., [12]), which requires $O(k^{\log_2 3}) \approx O(k^{1.59})$ bit operations. This algorithm is faster than the classical method even for moderate values of k, and this is the one we adopt for this step of our method.

The last ingredient we need is a fast method to compute the inverse image of the mapping

$$o^{(k)} : s \mapsto (o(s), o(Fs), o(F^2 s), \ldots, o(F^{k-1} s)),$$

to be able to recover the state s_J from the coefficients of $g(t)h(s_0, t)$. For important classes of PRNGs such as the twisted GFSR and Mersenne twister, there is a simple algorithm to compute this inverse image in $O(k)$ time. Then, our entire procedure works in $O(k^{\log_2 3}) \approx O(k^{1.59})$ time.

The procedure is summarized in Algorithm 1. It assumes that $g(t)$ has been precomputed in advance.

Algorithm 1 Jump ahead by polynomial multiplication

Input the state $s = s_0$;
Compute the polynomial $h(s_0, t)$ by advancing the generator for $2k$ steps;
Compute the product $g(t)h(s_0, t)$ and extract the coefficients $o(s_J), \ldots, o(s_{J+k-1})$;
Compute the state s_J from the bits $o(s_J), \ldots, o(s_{J+k-1})$;
Return s_J.

4 Illustrative Example by LFSR

The Linear Feedback Shift Register (LFSR) is a most classical and widely spread generator. Here, we use the term LFSR in the following limited sense (see [13]), although sometimes LFSR refers to a wider class of generators.

The state space is the row vector space $S := \mathbb{F}_2^k$, and the state transition function is defined by

$$(x_0, \ldots, x_{k-1}) \mapsto (x_1, x_2, \ldots, x_{k-1}, \sum_{i=0}^{k-1} a_i x_i),$$

where $a_0, \ldots, a_{k-1}$ are constants in $\mathbb{F}_2$. If we choose $o : (x_0, \ldots, x_{k-1}) \mapsto x_0$, then it directly follows that $o^{(k)} : S \to \mathbb{F}_2^k$ is the identity function. Thus, we can skip the computation of its inverse.

Proposition 1. *The computational complexity of PM-Jump for LFSR is the same with that for the polynomial multiplications of degree $2k$. As a result, jumping ahead can be done in $O(k^{1.59})$ time if we use Karatsuba's polynomial multiplication, and in $O(k \log k)$ time if we use a fast Fourier transform.*

Note that it is irrelevant to use $(x_i, \ldots, x_{i+k-1})$ as the i-th output k-bit integer of the pseudorandom number generator, since the consecutive outputs are overlapped. However, such an LFSR is used in stream cipher (pseudorandom bit generator), with suitably chosen nonlinear output function $o : S \to \mathbb{F}_2$, see for example [14].

5 Application to the Mersenne Twister

The Mersenne Twister (MT) generator can be described as follows [4]. Let w be the word size of the machine (e.g., $w = 32$). The row vector space $W := \mathbb{F}_2^w$ is identified with the set of words. For fixed integers $n > m$, MT generates a sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots$ of elements of W by the following recurrence:

$$\mathbf{x}_{j+n} := \mathbf{x}_{j+m} \oplus (\mathbf{x}_j^{w-r} | \mathbf{x}_{j+1}^{r}) A, \qquad j = 0, 1, \ldots,$$

where $(\mathbf{x}_j^{w-r} | \mathbf{x}_{j+1}^{r})$ denotes the concatenation of the upper $(w-r)$ bit $(\mathbf{x}_j^{w-r})$ of $\mathbf{x}_j$ and the lower r bit $(\mathbf{x}_{j+1}^{r})$ of $\mathbf{x}_{j+1}$, and the $w \times w$ matrix A is defined indirectly as follows: For any w-dimensional row vector $\mathbf{x}$,

$$\mathbf{x}A = \begin{cases} \text{shiftright}(\mathbf{x}) & \text{if the LSB of } \mathbf{x} = 0, \\ \text{shiftright}(\mathbf{x}) \oplus \mathbf{a} & \text{if the LSB of } \mathbf{x} = 1, \end{cases}$$

where LSB means the least significant bit (i.e., the rightmost bit), and $\mathbf{a}$ is a suitably chosen constant. This generator has the state transition function

$$f(\mathbf{x}_0^{w-r}, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = (\mathbf{x}_1^{w-r}, \mathbf{x}_2, \ldots, \mathbf{x}_n),$$

where $\mathbf{x}_n$ is determined by the above recursion with $j = 0$, and the state space is $S = \mathbb{F}_2^{nw-r}$.

The most popular instance, named MT19937, uses the parameters $n = 624$, $w = 32$, $r = 31$. Its sequence has a maximal period equal to the Mersenne prime $2^{19937} - 1$. Because of its high speed and good distribution property, MT19937 is widely used as a standard PRNG. However, is had been lacking an efficient jumping-ahead method, and this was a motivation for the work of [6].

Proposition 2. *For the MT generator, if we choose the output function*

$$o : (\mathbf{x}_0^{w-r}, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) \mapsto \textit{the LSB of } \mathbf{x}_1,$$

then the inverse image by $o^{(k)}$ is computable with time complexity $O(k)$. As a result, jumping ahead can be done in $O(k^{1.59})$ time if we use Karatsuba's polynomial multiplication, and in $O(k \log k)$ time if we use a fast Fourier transform.

Proof. A tricky algorithm that does that is described in [4, Section 4.3, Proposition 4.2].

The twisted GFSR generator [15] is based on the same construction as MT, but with $r = 0$. Thus, the proposition also applies to it.

6 Timings

We made an experiment to compare the speeds of three jumping-ahead methods: the one that directly implements Horner's method (3) (Horner), the method of [6] with a sliding window with parameter q (SW), and our new method based on polynomial multiplication with Karatsuba's algorithm (PM). For the latter, we used the NTL implementation [11]. We applied these methods to MT generators with the Mersenne exponents $k = 19937$, 21701, 23209, 44497, 86243, 110503, 132049.

For each value of k, we repeated the following 1000 times, on two different computers: We generated a random polynomial $g(t)$ uniformly over the polynomials of degree less than k in $\mathbb{F}_2[t]$ and a random state s uniformly in $S = \mathbb{F}_2^k$, then we computed $f^J(s)$ by each of the three algorithms, and we measured the required CPU time. We then summed those CPU times over the 1000 replications. The results are given in Table 1 for the Intel Core Duo 32-bit processor and in Table 2 for the AMD Athlon 64 3800+ 64-bit processor. The total CPU times are in seconds. For the SW method, we selected the parameter q that gave the highest speed; this parameter is listed, together with the required amount of memory in Kbytes.

We see that for the 32-bit computer, PM is faster than SW when $k \geq 44497$, whereas for the 64-bit machine, PM is faster for $k \geq 19937$. The results agree with the computational complexity approximations, which are $O(k^2)$ for SW and $O(k^{1.59})$ for PM.

Table 1. Comparison of CPU time (in seconds) for 1000 jumps ahead for MT generators of various sizes, with the Horner, SW, and PM methods. This experiment was done on an Intel Core Duo (2.0 GHz) with 1 Gbytes of Memory

k	Horner	SW			PM
	CPU (sec)	q	memory (KB)	CPU (sec)	CPU (sec)
19937	17.730	7	312	5.129	6.692
21701	20.830	8	679	5.853	7.438
23209	23.691	8	726	6.537	7.930
44497	83.990	8	1391	20.444	19.765
86243	309.268	8	2696	71.731	52.288
110503	445.387	9	6908	113.588	71.098
132049	648.121	9	8254	158.290	89.833

7 Conclusion

The proposed jump ahead algorithm based on polynomial multiplication is advantageous over the sliding window method when the dimension k of state space

Table 2. The same experiment as in Table 1, but on a 64-bit Athlon 64 3800+ processor with 1 Gbyte of memory

k	Horner	SW			PM
	CPU (sec)	q	memory (KB)	CPU (sec)	CPU (sec)
19937	8.573	7	312.000	2.727	2.219
21701	10.070	7	339.500	3.135	2.442
23209	11.435	7	363.000	3.519	2.578
44497	40.371	6	347.750	11.651	5.968
86243	148.330	6	674.000	44.842	14.509
110503	242.219	5	431.750	73.232	19.694
132049	344.972	5	515.875	106.179	24.743

is large enough, since the new PM method has time complexity $O(k^{1.59})$, compared with $O(k^2)$ for the sliding window method. Our empirical experiments confirm this and show that this large enough k corresponds roughly to the value of k used in the most popular implementation of MT. Much more importantly, the new PM method has space complexity of $O(k)$, which is much smaller than that of the sliding window method, namely $O(k2^q)$. The main limitation is that the new method requires the availability of an efficient algorithm to compute the inverse of $o^{(k)}$.

References

1. Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis. Third edn. McGraw-Hill, New York, NY (2000)
2. L'Ecuyer, P., Buist, E.: Simulation in Java with SSJ. In: Proceedings of the 2005 Winter Simulation Conference, IEEE Press (2005) 611–620
3. L'Ecuyer, P.: Pseudorandom number generators. In Platen, E., Jaeckel, P., eds.: Simulation Methods in Financial Engineering. Encyclopedia of Quantitative Finance. Wiley (2008) Forthcoming.
4. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8**(1) (1998) 3–30
5. Panneton, F., L'Ecuyer, P., Matsumoto, M.: Improved long-period generators based on linear recurrences modulo 2. ACM Transactions on Mathematical Software **32**(1) (2006) 1–16
6. Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., L'Ecuyer, P.: Efficient jump ahead for $\mathbf{F}_2$-linear random number generators. INFORMS Journal on Computing (2008) to appear.
7. Panneton, F., L'Ecuyer, P.: Infinite-dimensional highly-uniform point sets defined via linear recurrences in $\mathbb{F}_{2^w}$. In Niederreiter, H., Talay, D., eds.: Monte Carlo and Quasi-Monte Carlo Methods 2004, Berlin, Springer-Verlag (2006) 419–429
8. L'Ecuyer, P.: Uniform random number generation. Annals of Operations Research **53** (1994) 77–120
9. L'Ecuyer, P., Panneton, F.: $\mathbf{F}_2$-linear random number generators. In Alexopoulos, C., Goldsman, D., eds.: Advancing the Frontiers of Simulation: A Festschrift in Honor of George S. Fishman. Spinger-Verlag, New York (2007) To appear.

10. Couture, R., L'Ecuyer, P.: Lattice computations for random numbers. Mathematics of Computation **69**(230) (2000) 757–765
11. Shoup, V.: NTL: A Library for doing Number Theory. Courant Institute, New York University, New York, NY. (2005) Available at `http://shoup.net/ntl/`.
12. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, Cambridge, U.K. (2003)
13. Golomb, S.W.: Shift-Register Sequences. Holden-Day, San Francisco (1967)
14. Rueppel, R.A.: Analysis and Design of Stream Ciphers. Springer-Verlag (1986)
15. Matsumoto, M., Kurita, Y.: Twisted GFSR generators II. ACM Transactions on Modeling and Computer Simulation **4**(3) (1994) 254–266