POLYNOMIAL VERSUS MATRIX METHODS FOR LEAP-AHEAD IN SHIFT-REGISTER TYPE PSEUDORANDOM NUMBER GENERATORS

Michael Mascagni

Program in Scientific Computing
and
Department of Mathematics
The University of Southern Mississippi

Abstract. We compare the cost of polynomial and matrix methods for leaping ahead an arbitrary amount in the period of shift-register based pseudorandom number generators. It is well known that both methods are applicable in the binary shift-register case. However, for modular shift-registers with moduli other than 2, only the matrix method had been proposed. We present both methods for shift-registers with arbitrary moduli and compare their computational and memory costs.

1. Introduction.

The most common method for pseudorandom number generation still remains D. H. Lehmer's linear congruential method. The linear congruential generator (LCG) is based on the following modular first-order linear recursion

$$x_n = ax_{n-1} + b \pmod{M}. \tag{1}$$

A generalization of the homogeneous LCG is the modular shift-register given by the recursion

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_\ell x_{n-\ell} \pmod{M}. \tag{2}$$

This is the equation for a general ℓth order linear recursion modulo m. Since the use of equation (2) generally requires ℓ modular multiplications and $\ell - 1$ modular additions per iteration, it is common to use so-called lagged-Fibonacci recursions instead. These have the form

$$x_n = x_{n-k} + x_{n-\ell} \pmod{M}, \quad \ell > k. \tag{3}$$

A somewhat surprising fact about lagged-Fibonacci recursions is that by a judicious choice of k and ℓ, sequences may be obtained with the longest possible period achievable within the more general family defined by equation (2).

1991 *Mathematics Subject Classification.* 65C10, 65Y05.

Key words and phrases. random number generation, shift-register generator, fast leap-ahead, matrix method, polynomial method.

Typeset by $\mathcal{AMS}$-TeX

When $M = 2$ these are the recursions of binary shift-register sequences. In an implementation of a binary shift-register for use as a pseudorandom number generator one often needs to compute the value of the ℓ-bit shift-register at an arbitrary point in the period with respect to the shift-register's current (seed) value. Fast leap-ahead algorithms to accomplish this exist that are based on the manipulation of the underlying recursion via either matrix or polynomial arithmetic. By fast leap-ahead algorithm we mean an algorithm that transforms a seed at a particular point in pseudorandom number generator's cycle to a new point j steps away in $O(\log_2 j)$ "operations". Here an "operation" is any computation with complexity of the same order as a single step of the pseudorandom number generator. When $M \neq 2$, algorithms analogous to the binary case also exist. This paper explores the comparative efficiency of these two classes of fast leap-ahead algorithm.

The plan of the paper is as follows. In §2 we will provide some examples of when the ability to jump arbitrarily in a pseudorandom number generator's sequence is required. These examples are related to statistical independence and parallel computation. We then describe both the matrix and the polynomial fast leap-ahead algorithms in detail for the case of $M = 2$ in §3. Our descriptions will make it clear how these binary algorithms generalize for arbitrary M. In §4 we carefully compare the computational and memory cost of both these algorithms. Finally in §5 we conclude and briefly discuss the implications of these comparison.

2. Motivation for the Algorithm.

There are several common situations where the ability to leap ahead an arbitrary amount in the period of a pseudorandom number generator is desirable [5]. In fact, it is diappointing that very few of various pseudorandom number generation implementations include such a facility, [3, 7]. A simple situation often encountered when using a serial pseudorandom number generator is when a user wishes to perform again a given computation with a different set of pseudorandom numbers in order to obtain a new, statistically independent sample. Typically, the user supplies a new seed to the generator, and this new seed situates the user's new computation on a different starting value in the deterministic and periodic recursion used for pseudorandom number generation. In general, the user can only be sure that the new starting value is indeed different from the old starting value. However, there still may be substantial overlap between the first and second set of pseudorandom numbers used. A more sophisticated user will save the seed used by the pseudorandom number generator at the conclusion of the old computation to initialize the new computation. This is not always easy to do. One simple solution is to ask the user to supply an offset in the generator's period from a common starting value instead of just a seed. Proper offset values will insure that there is no overlap between sequences. The fast leap-ahead algorithm is required to implement such a facility.

In an implementation of a parallel version of pseudorandom number generator based on a single long periodic recursion, one must endeavor to start each processor's pseudorandom number generator at a point that minimizes the possibility of overlap among the sub-streams used by each process. While the appropriate strategy to achieve this varies with the application, any strategy requires a fast leap-ahead algorithm for efficient implementation. In fact, if one wishes to use such a pseudorandom number generator on both a parallel and serial machine to obtain identical results, e.g. for validation, one must use a fast leap-ahead algorithm in both the parallel and serial versions to get the same random numbers in both cases.

3. Matrix and Polynomial Methods.

Equation (3) with $M = 2$ along with an initial vector $\mathbf{x}_0 = [x_0, x_{-1}, \ldots, x_{-\ell-1}]^T$ of length ℓ defines a binary shift-register sequence. One can define a vector recursion as

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} \pmod 2, \tag{4}$$

where the matrix $\mathbf{A}$ has the form

$$\mathbf{A} = \begin{array}{c} \begin{array}{ccccccccccc} & & & & & k & & & & & \ell \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & 0 & 1 \\ 1 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 1 & 0 \end{pmatrix} \end{array}. \tag{5}$$

This is an $\ell \times \ell$ matrix with elements defined modulo 2. Given the initial (seed) vector $\mathbf{x}_0$, we can compute the jth vector in the sequence defined by equation (4) as

$$\mathbf{x}_j = \mathbf{A}^j\mathbf{x}_0 \pmod 2. \tag{6}$$

This algorithm for leaping j elements in the shift-register's sequence, which we will call the matrix fast leap-ahead (MFLAP) algorithm, requires two steps:

(i) computation of $\mathbf{A}^j \pmod 2$ *and*

(ii) the subsequent evaluation of $\mathbf{A}^j\mathbf{x}_0 \pmod 2$.

Step (i) can be accomplished with $O(\log_2 j)$ matrix-matrix multiplications with the well-known square and multiply algorithm while step (ii) requires a single matrix-vector multiplication.

The other common algorithm for computing the jth element of a shift-register sequence is based on polynomial algebra instead of matrix algebra, [2, 1, 6]. Thus we call this algorithm the the polynomial fast leap-ahead (PFLAP) algorithm. The algorithm can be best explained without proof by means of an illustrative example. Let us refer to x_0 by the following polynomial: $r(x) = 1 \pmod 2$. It is important to note that $r(x)$ is a polynomial with binary coefficients. In an analogous manner, we denote x_1 by $r(x) = x \pmod 2$, and so on. Thus we symbolically multiply $r(x)$ by the monomial x to reflect the result of a single step in the iteration given by equation (3). This procedure continues until the ℓth iteration produces $r(x) = x^\ell \pmod 2$. Since $x^\ell \pmod 2$ represents x_ℓ, and $x_\ell = x_{\ell-k} + x_0 \pmod 2$ we have that $x^\ell = x^{\ell-k} + 1 \pmod 2$ and so $r(x) = x^{\ell-k} + 1 \pmod 2$. A common notation for this is to write $r(x) = x^\ell \pmod{f(x)}$ with $f(x) = x^\ell - x^{\ell-k} - 1 \equiv x^\ell + x^{\ell-k} + 1 \pmod 2$. The notation $s(x) \pmod{f(x)}$ means that we replace the higher powers in $s(x)$ by the identity $f(x) = 0 \pmod 2$ until the resulting polynomial has degree less than the degree of $f(x)$. Thus in our case we replace x^ℓ by $x^{\ell-k} + 1 \pmod 2$ until the resulting degree of $s(x)$ is less than ℓ.

Suppose that at some point in this procedure we have $r(x) = x^j \pmod{f(x)} = \sum_{i=0}^{\ell-1} c_i x^i$, and we wish to use this to evaluate x_j. With the initial values $x_{\ell-1}, x_{\ell-2}, \ldots, x_0$ we have that

$$x_j = \sum_{i=0}^{\ell-1} c_i x_i. \tag{7}$$

This fact is easy to prove by induction.

Thus the PFLAP algorithm consists of the following two steps:

(a) evaluate $r(x) = x^j \pmod{f(x)}$ *and*
(b) evaluate x_j using equation (7).

Step (a) can be accomplished in $O(\log_2 j)$ polynomial-polynomial multiplications, while step (b) requires the application of equation (7).

It is obvious that the MFLAP algorithm can be applied for an arbitrary modulus, M. While not as obvious, the PFLAP algorithm too is valid with an arbitrary modulus. Since there is no division involved in either FLAP algorithm, one need not have all the operations in a field, one can work over the ring of integers modulo M with no complications. Moreover, in the PFLAP algorithm the operation of forming $r(x) \pmod{f(x)}$ can be accomplished in general with a polynomial version of Euclid's greatest common divisor algorithm, which requires no division. Thus it makes perfect sense to consider both the FLAP algorithms for any modulus.

4. Computational and Memory Costs.

We will now attempt to compute carefully the cost of both the MFLAP and PFLAP algorithms. Let us assume that we have the initial values $x_{\ell-1}, x_{\ell-2}, \ldots, x_0$ of residues modulo M and that we are given j, the amount we wish to leap ahead. There are two different computations that we wish to compare, these are

(A) evaluation of x_j *and*
(B) evaluation of $x_{\ell+j-1}, x_{\ell+j-2}, \ldots, x_j$.

Computation **(A)** just provides the jth element of the sequence while computation **(B)** gives the ℓ starting values for initializing the shift-register j steps from the given seed. In the case of the MFLAP algorithm computations **(A)** and **(B)** are identical, while for the PFLAP algorithm once one has computed **(A)** there is still a bit more work to do before **(B)** is obtained.

Both the MFLAP and PFLAP algorithms rely on their particular version of the square and multiply algorithm for efficient powering, [4]. Because of this fact we will briefly describe the square and multiply algorithm here. Suppose we have an integer reduced modulo M, a, and we wish to compute $a^j \pmod M$ for some large power j. The integer j will require $N = n + 1 = \lceil \log_2 j \rceil + 1$ bits of storage and can be written in base 2 as $j = (b_n b_{n-1} b_{n-2} \ldots b_1 b_0)_2$. The square and multiply algorithm then is given by the following piece of pseudo-code. The input is a and $j = (b_n b_{n-1} b_{n-2} \ldots b_1 b_0)_2$, and the output is z $= a^j \pmod{M}$:

```
z = $a$;
for (i = n-1 ; i >= 0 ; i-- ) {
     if ($b_i$==0) then z = z*z $\pmod{M}$; /* SQUARE */
     else z = z*$a$ $\pmod{M}$; /* MULTIPLY */
}
```

It is clear from this pseudocode that in $N = \lceil \log_2 j \rceil + 1$ modular multiplications we obtain z, thus the complexity is $O(\log_2 j)$ multiplications. We have used the example of modular multiplication in the pseudocode. However this same version of the square and multiply algorithm is suitable for our purposes with the modular multiplication replaced by matrix-matrix multiplication for the MFLAP algorithm and by polynomial-polynomial multiplication for the PFLAP algorithm.

Since the square and multiply algorithm is the key to computation **(A)** in both algorithms, and its computational complexity is $\lceil \log_2 j \rceil + 1$ multiplications. Let us recall the complexity of both matrix-matrix multiplication (MMM) and polynomial-polynomial multiplication (PPM) with coefficients that are reduced residues modulo M. At this point we should distinquish between two further cases. We imagine that most shift-register implementations will endeavor to minimize the cost per step by choosing a two term additive lagged-Fibonacci recursion. However, some other circumstances may determine the use of a more general recursion as in equation (2). It is well known that MMM of two $\ell \times \ell$ matricies with no special structure requires ℓ^3 modular multiplications and $\ell^3 - \ell^2$ modular additions. Similarly the PPM of two degree $\ell - 1$ polynomials requires ℓ^2 modular multiplications and ℓ^2 modular additions. Now we have computed a polynomial of degree $2\ell - 2$ and must further reduce this to a polynomial of degree $\ell - 1$ by using the identity $f(x) \equiv 0 \pmod{M}$. This will require at most $\ell - 1$ reductions, each of which will require at most 2 modular additions in the two-term case and $\ell - 1$ modular additions in the general case. This makes the overall cost at most ℓ^2 modular multiplications and $\ell^2 + 2\ell - 2$ modular additions in the two-term case and generally $2\ell^2 - 2\ell + 1$ modular additions.

We have not taken into account some very simple optimizations that will reduce the cost of both algorithms. During the **MULTIPLY** phase of the square and multiply algorithm, one of the matricies is simply $\mathbf{A}$ in the MFLAP algorithm while in the PFLAP algorithm one of the polynomials is simple monomial x. The cost of multiplying an arbitrary matrix by $\mathbf{A}$ requires less work than the multiplication of two arbitrary matrixcies. In the two-term case no modular multiplications and only ℓ modular additions are required. In the general case, only ℓ^2 modular multiplications and ℓ^2 modular additions are required. Similarly, multiplication of an arbitrary polynomial by x requires no modular multiplications and at most 2 modular additions in the two-term case and $\ell - 1$ modular additions in the general case. Here the modular additions corresponding to at most a single reduction by $f(x)$ in both cases.

For computation **(A)** we still have to compute the final matrix-vector product for the MFLAP. Of course, this costs ℓ^2 modular multiplications and ℓ^2 modular additions in both cases. We now know the cost of computation **(A)** for the PFLAP algorithms, and as we said above, this is also the cost of computation **(B)** for the MFLAP aglorithm. However, for the PFLAP algorithm once we have computed $x^j \pmod{f(x)}$ we still must compute x_j for computation **(A)** and we have two more things to compute for computation **(B)**. We must (1) compute $x^i \pmod{f(x)}$ for $i = j+1, j+2, \ldots, j+\ell-1$ and we must (2) use these ℓ polynomials to compute x_i for $i = j, j+1, \ldots, j+\ell-1$. Evaluation of x_j requires the application of equation (7), which can be thought of as the inner product of two vectors of length ℓ. This costs ℓ modular multiplications and ℓ modular additions. Each computation in (1) is just a PPM with one polynomial being x. As computed above, each of these requires 2 modular additions in the two-term case and $\ell - 1$ modular additions in general. Thus only $2\ell-2$ modular additions are need in the two-term with $\ell^2-2\ell+1$ modular additions generally required for (1). Each of the ℓ steps of (2) requires the application of equation (7). This makes the overall cost ℓ^2 modular multiplications and $\ell^2 - \ell$ modular additions.

Finally we must calculate how much memory is required for the two algorithms. For the MFLAP algorithm we must store the current matrix iterate, the matrix $\mathbf{A}$, and a temporary location to store intermediate results. Thus $3\ell^3$ storage is required. If we use the optimization for multiplication by $\mathbf{A}$, we can dispense with its storage and reduce to $2\ell^2$ storage in both the two-term and general case. For the PFLAP algorithm we must store the current polynomial and have a temporary location large enough to square such a polynomial. Since we can overlap these two functions, only $2\ell - 1$ storage is required.

Below in Table 1 we summarize these complexity and storage results. In Table 1 the "general" method assumes optimization appropriate to the while the "two-term" method assumes we save work and space as mentioned above. As in the rest of the text, **(A)** and **(B)** refer to the corresponding computations in the text. Instead of expressing the table values in terms of "big-oh" notation, we have tabulated the exact leading term for all entries.

	Algorithm	
Quantitiy Analyzed	MFLAP	PFLAP
Complextity (general) **(A)**	$\ell^3 \log_2 j$	$\ell^2 \log_2 j$
Complextity (two-term) **(A)**	$\frac{1}{2}\ell^3 \log_2 j$	$\frac{1}{2}\ell^2 \log_2 j$
Complextity (general) **(B)**	0	ℓ
Complextity (two-term) **(B)**	0	ℓ^2
Memory (general)	$2\ell^2$	2ℓ
Memory (two-term)	$2\ell^2$	2ℓ

Table 1. Leading terms in the computational and memory costs. These are associated with the two-term and general versions of the MFLAP and PFLAP algorithms for both computations **(A)** and **(B)**. For **(B)** we only include the increment to go from bf (A) to **(B)**. See text for more details.

5. Conclusions.

In this paper we have described the matrix and polynomial based fast leap-ahead algorithm for shift-register based pseudorandom number generators. We have also shown that the well known binary polynomial fast leap-ahead algorithm is applicable for general moduli. Both these fast leap-ahead algorithms are based on the application of the square and multiply algorithm. As such, their computational complexity is ultimately based on the complexity of matrix-matrix multiplication in one case and polynomial-polynomial multiplication in the other. In general, the computational cost as well as the memory costs of the polynomial based algorithms are a factor of ℓ smaller than the corresponding values for the matrix based algorithms. Here ℓ is the length of the shift-register in the pseudorandom number generator. This difference in costs is rather inconsequential until the length of the shift-register becomes quite large. While there are many pseudorandom number generators based on shift-registers with modest lengths, it is not unheard of to consider lagged-Fibonacci or binary shift-register generators with long registers. In these cases it is clear that the polynomial methods are clearly advantageous.

Another factor to consider is that we save a factor of ℓ in work and storage with the polynomial methods on each processor of a a multiprocessor system. Thus if a particular computation requires many leap-aheads be stored per processor, there may be a considerable agregate savings in memory by using the polynomial methods.

References

1. B. J. Collings and G. B. Hembree, *Initializing generalized feedback shift register pseudorandom number generators*, J. of the ACM **33** (1986), 706–711.
2. I. Deák, *Uniform random number generators for parallel computers*, Parallel Comput. **15** (1990), 155–164.
3. IMSL, *Library User's Manual*, Edition 9.2, IMSL, Inc., Houston, Texas, 1984.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second edition*, Addison-Wesley, Reading, Massachusetts, 1981.
5. P. L'Ecuyer and S. Côté, *Implementing a random number package with splitting facilities*, ACM Trans. on Math. Soft. **17** (1991), 98–111.
6. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, Cambridge, London, New York, 1986.
7. SAS, *User's Guide*, Basics, Version 5, SAS Institute, Inc., Cary, NC, 1985.

Box 10057 Southern Station; University of Southern Mississippi Hattiesburg, Mississippi 39406-0057 **USA**

E-mail address: Michael.Mascagni@usm.edu, URL: www.ncsa.uiuc.edu/Apps/CMP/RNG/mascagni