Notes on Jumping PRNGs Ahead

Peter Occil

This version of the document is dated 2025-02-20.

Peter Occil

Some pseudorandom number generators (PRNGs) have an efficient way to advance their state as though a huge number of PRNG outputs were discarded. Notes on how they work are described in the following sections.

0.1 F_2-linear PRNGs

For some PRNGs, each bit of the PRNG's state can be described as a linear recurrence on its entire state. These PRNGs are called *F_2-linear PRNGs*, and they include the following:

- Linear congruential generators (LCGs) with a power-of-two modulus.
- Xorshift PRNGs.
- PRNGs in the xoroshiro and xoshiro families.
- Linear or generalized feedback shift register generators, including Mersenne Twister.

For an F_2-linear PRNG, there is an efficient way to discard a given (and arbitrary) number of its outputs (to "jump the PRNG ahead"). This jump-ahead strategy is further described in (Haramoto et al., 2008)[1]. See also (Vigna 2017)[2]. To calculate the jump-ahead parameters needed to advance the PRNG N steps:

1. Build `M`, an S×S matrix of zeros and ones that describes the linear transformation of the PRNG's state, where S is the size of that state in bits. For an example, see sections 3.1 and 3.2 of (Blackman and Vigna 2019)[3], where it should be noted that the additions inside the matrix are actually XORs.

2. Find the *characteristic polynomial* of `M`. This has to be done in the two-element field F_2, so that each coefficient of the polynomial is either 0 or 1.

   For example, SymPy's `charpoly()` method alone is inadequate for this purpose, since it doesn't operate on the correct field. However, it's easy to adapt that method's output for the field F_2: even coefficients become zeros and odd coefficients become ones.

   Note that for a linear feedback shift register (LFSR) generator, the characteristic polynomial's coefficients are 1 for each of its "taps" (and "tap" 0), and 0 elsewhere. For example, an LFSR generator with taps 6 and 8 has the characteristic polynomial $x^8 + x^6 + 1$.

   The section "Jump Parameters for Some PRNGs" shows characteristic polynomials for some PRNGs and one way their coefficients can be represented.

3. Calculate `powmodf2(2, N, CP)`, where `powmodf2` is a modular power function that calculates `2^N mod CP` in the field F_2, and `CP` is the characteristic polynomial. (`N` is the number of PRNG outputs to

[1]Haramoto, Matsumoto, Nishimura, Panneton, L'Ecuyer, "Efficient Jump Ahead for F_2-Linear Random Number Generators", *INFORMS Journal on Computing* 20(3), Summer 2008.

[2]Vigna, S., "Further scramblings of Marsaglia's xorshift generators", *Journal of Computational and Applied Mathematics* 315 (2017).

[3]Blackman, Vigna, "Scrambled Linear Pseudorandom Number Generators", 2019.

discard.) Ordinary modular power functions, such as BigInteger's `modPow` method, won't work here, even if the polynomial is represented in the manner described in "Jump Parameters for Some PRNGs".

4. The result is a *jump polynomial* for jumping the PRNG ahead N steps, that is, for discarding N outputs of the PRNG.

   An example of its use is found in the `jump` and `long_jump` functions in the **`xoroshiro128plus` source code**, which are identical except for the jump polynomial. In both functions, the jump polynomial's coefficients are packed into a 128-bit integer (as described in "Jump Parameters for Some PRNGs"), which is then split into the lower 64 bits and the upper 64 bits, in that order.

0.2 Counter-Based PRNGs

Counter-based PRNGs, in which their state is updated simply by incrementing a counter, can be trivially jumped ahead just by changing the seed, the counter, or both (Salmon et al. 2011)[4].

0.3 Multiple Recursive Generators

A *multiple recursive generator* (MRG) generates numbers by transforming its state using the following formula: `x(k) = (x(k-1)*A(1) + x(k-2)*A(2) + ... + x(k-n)*A(n)) mod modulus`, where `A(i)` are the *multipliers* and `modulus` is the *modulus*.

For an MRG, the following matrix (`M`) describes the state transition `[x(k-n), ..., x(k-1)]` to `[x(k-n+1), ..., x(k)]` (mod `modulus`):

```
| 0    1    0   ...  0   |
| 0    0    1   ...  0   |
| .    .    .   ... ...  |
| 0    0    0   ...  1   |
|A(n)A(n  A(n   ... A(1) |
|      -1) -2)           |
```

To calculate the parameter needed to jump the MRG ahead N steps, calculate `M` ^{N} mod `modulus`; the result is a *jump matrix* `J`.

Then, to jump the MRG ahead N steps, calculate `J * S` mod `modulus`, where `J` is the jump matrix and `S` is the state in the form of a column vector; the result is a new state for the MRG.

This technique was mentioned (but for binary matrices) in Haramoto, in sections 1 and 3.1. They point out, though, that it isn't efficient if the transition matrix is large. See also (L'Ecuyer et al., 2002)[5].

0.3.1 Example

A multiple recursive generator with a modulus of 1449 has the following transition matrix:

```
|  0   1   0  |
|  0   0   1  |
| 444 342 499 |
```

To calculate the 3 × 3 jump matrix to jump 100 steps from this MRG, raise this matrix to the power of 100 then take the result's elements mod 1449. One way to do this is the "square-and-multiply" method, described by D. Knuth in *The Art of Computer Programming*: Set J to the identity matrix, N to 100, and M to a copy of the transition matrix, then while N is greater than 0:

[4]Salmon, John K., Mark A. Moraes, Ron O. Dror, and David E. Shaw. "Parallel random numbers: as easy as 1, 2, 3." In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12. 2011.

[5]L'Ecuyer, Simard, Chen, Kelton, "An Object-Oriented Random-Number Package with Many Long Streams and Substreams", *Operations Research* 50(6), 2002.

1. If N is odd[6], multiply J by M then take J's elements mod 1449.
2. Divide N by 2 and round down, then multiply M by M then take M's elements mod 1449.

The resulting J is a *jump matrix* as follows:

```
| 156    93  1240 |
| 1389 1128   130 |
| 1209  930   793 |
```

Transforming the MRG's state with J (and taking its elements mod 1449) will transform the state as though 100 outputs were discarded from the MRG.

0.4 Linear Congruential Generators

A *linear congruential generator* (LCG) generates numbers by transforming its state using the following formula: `x(k) = (x(k-1)*a + c) mod modulus`, where `a` is the *multiplier*, `c` is the additive constant, and `modulus` is the *modulus*.

An efficient way to jump an LCG ahead is described in (Brown 1994)[7]. This also applies to LCGs that transform each `x(k)` before giving out it, such as M.O'Neill's PCG32 and PCG64.

An MRG with only one multiplier expresses the special case of an LCG with `c = 0` (also known as a *multiplicative* LCG). For `c` other than 0, the following matrix describes the state transition `[x(k-1), 1]` to `[x(k), 1]` (mod `modulus`):

```
 | a   c |
 | 0   1 |
```

Jumping the LCG ahead can then be done using this matrix as described in the previous section.

0.5 Multiply-with-Carry, Add-with-Carry, Subtract-with-Borrow

There are implementations for jumping a multiply-with-carry (MWC) PRNG ahead, but **only in source-code form**[8]. I am not aware of an article or paper that describes how jumping an MWC PRNG ahead works.

I am not aware of any efficient ways to jump an add-with-carry or subtract-with-borrow PRNG ahead an arbitrary number of steps.

0.6 Combined PRNGs

A combined PRNG can be jumped ahead N steps by jumping each of its components ahead N steps.

0.7 Jump Parameters for Some PRNGs

The following table shows the characteristic polynomial and jump polynomials for some PRNG families. In the table:

- Each number before the colon in the jump polynomial column is the number of PRNG outputs discarded when the corresponding jump polynomial is used.
- Each polynomial's coefficients are zeros and ones, so the table shows them as a base-16 integer that stores the coefficients as individual bits: the least significant bit is the degree-0 coefficient, the next

[6]"x is odd" means that x is an integer and not divisible by 2. This is true if $x - 2$*floor($x/2$) equals 1, or if x is an integer and the least significant bit of abs(x) is 1.

[7]Brown, F., "Random Number Generation with Arbitrary Strides", *Transactions of the American Nuclear Society* Nov. 1994.

[8]https://github.com/rsaucier/Random/blob/3a7981bd6a8ac6d4507e9630393303b18e8967ca/kiss.h

bit is the degree-1 coefficient, and so on. For example, the integer 0x23 stores the coefficients of the polynomial $x^5 + x + 1$.

- Each characteristic polynomial's highest coefficient is x^n, where n is the PRNG's state size. Thus, the table shows it as a base-16 integer with n plus one bits.
- "'Period'/ φ" means the PRNG's maximum cycle length divided by the golden ratio, and rounded to the closest odd integer; this jump parameter is chosen to avoid overlapping number sequences as much as possible (see also **NumPy documentation**[9]).

PRNG	Characteristic Polynomial	Jump Polynomials
xoroshiro64	0x1053be9da6e2286c1	2^{32}: 0x4cbf99bd77fcd1a02 2^{48}: 0xb4e7e4633f1f8b95 "Period"/ φ : 0x751f355609af0e3b
xoshiro128	0x100fc65a2006254b11b489db6de18fc01	2^{32}: 0xf8aed94730b948df3be07b8f7afe1082 2^{48}: 0xdeaa4ca2dec5bb9a87a4583dcb56667c2 2^{64}: 0x77f2db5b6fa035c3f542d2d38764000b2 2^{96}: 0x1c580662ccf5a0ef0b6f099fb523952e "Period"/ φ : 0x338b58d0590169928fda8fd5d1cf96b6
xoroshiro128 (except ++)	0x10008828e513b43d5095b8f76579aa001	2^{32}: 0xd4e95eef9edbdbc6fad843622b252c782 2^{48}: 0x9b19ba6b3752065ad769cfc9028deb782 2^{64}: 0x170865df4b3201fcdf900294d8f554a52 2^{96}: 0xd-ddf9b1090aa7ac1d2a98b26625eee7b "Period"/ φ : 0xc1c620fd7bf598c34a2828365a7df3e0
xoroshiro128++	0x10031bcf2f855d6e58dae70779760b081	2^{32}: 0x2e1bcf52f1051044fcceec21d5c306d92 2^{48}: 0xc8462a08ab3d7f9b99030a888c8679392 2^{64}: 0x992ccaf6a6fca052bd7a6a6e99c2ddc2 2^{96}: 0x9c6e6877736c46e3360fd5f2cf8d5d99 "Period"/ φ : 0x1b4c7a8989405b16d3e4e127a6a11513
xoshiro256	0x10003c03c3f3ecb1904b4edcf26259f85-0280002bcefd1a5e9d116f2bb0f0f001	2^{32}: 0xe055d3520fdb9d7214fafc0fbdbc2087d8d0632bd08 0x5f728be2c97e9066474579292f705634f825539dee5e 0x12e4a2fbfc19bff934faff184785c20ab60d6c5b8c78f 0x31eebb6c82a9615fb27c05962ea56a13cdb45d7def4 0x39abdc4529b1661ca9582618e03fc9aad5a61266f0c 0xf567382197055bf04823b45b89dc689c69e6e6e431a 0x39109bb02acbe63577710069854ee241c5004e441c5 0xa2b5d83a373c7ac2f31d2e03157bc387d317530723a φ : 0x294e2bac089b06c7d4ce5d1a031b6cf8787f49127b3

[9]https://docs.scipy.org/doc/numpy/reference/random/parallel.html

0.8 Acknowledgments

Sebastiano Vigna reviewed this page and gave comments.

1 Notes