

Notes on Jumping PRNGs Ahead

This version of the document is dated 2020-08-28.

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:

- 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 $S \times S$ 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 $\text{powmodf2}(2, N, CP)$, where powmodf2 is a modular power function that calculates $2^N \bmod CP$ in the field F_2 , and CP is the characteristic polynomial. Regular 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.

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: $x(k) = (x(k-1)*A(1) + x(k-2)*A(2) + \dots + x(k-n)*A(n)) \bmod \text{modulus}$, where $A(i)$ are the *multipliers* and *modulus* is the *modulus*.

For an MRG, the following matrix (M) describes the state transition $[x(k-n), \dots, x(k-1)]$ to $[x(k-n+1), \dots, x(k)] \bmod \text{modulus}$:

	0	1	0	...	0	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	
	
	0	0	0	...	1	
	0	1	0	...	0	
	
	0	0	1	...	0	
	
	0	0	0	...	1	
	A(n)A(n-1)...	A(n-1)A(n-2)...	A(n-2)A(n-3)...	...	A(1)1	

identity matrix, N to 100, and M to a copy of the transition matrix, then while N is greater than 0:

1. If N is odd, multiply J by M then reduce J's elements mod 1449.
2. Divide N by 2 and round down, then multiply M by M then reduce 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 reducing 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: $x(k) = (x(k-1)*a + c) \bmod \text{modulus}$, 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)⁽⁶⁾. This also applies to LCGs that transform each $x(k)$ before outputting 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 $[x(k), 1] \bmod \text{modulus}$:

```
| 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 ([ref. 1](#)). 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 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 0-order coefficient, the next bit is the first-order coefficient, and so on. For example, the integer 0x23 stores the coefficients of the polynomial $x^5 + x + 1$.
- "Period"/q 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](#)).

PRNG	Characteristic Polynomial	Jump Polynomials
xoroshiro64	0x1053be9da6e2286c1	2^{32} : 0x4cbf99bd77fcd1a0
		2^{48} : 0xb4e7e4633f1f8b95
		"Period"/q: 0x751f355609af0e3b
xoshiro128	0x100fc65a2006254b11b489db6de18fc01	2^{32} : 0xf8aed94730b948df3be07b8f7afe108
		2^{48} : 0xdeaa4ca2dec5bb9a87a4583dcb56667c
		2^{64} : 0x77f2db5b6fa035c3f542d2d38764000b
xoroshiro128 (except ++)	0x10008828e513b43d5095b8f76579aa001	2^{96} : 0x1c580662ccf5a0ef0b6f099fb523952e
		"Period"/q: 0x338b58d0590169928fda8fd5d1cf96b6
		2^{32} : 0xd4e95eef9edbdbc6fad843622b252c78
xoroshiro128++	0x10031bcf2f855d6e58dae70779760b081	2^{48} : 0x9b19ba6b3752065ad769cfc9028deb78
		2^{64} : 0x170865df4b3201fcd900294d8f554a5
		2^{96} : 0xdddf9b1090aa7ac1d2a98b26625eee7b
xoroshiro128++	0x10031bcf2f855d6e58dae70779760b081	"Period"/q: 0xc1c620fd7bf598c34a2828365a7df3e0
		2^{32} : 0x2e1bcf52f1051044fcceec21d5c306d9
		2^{48} : 0xc8462a08ab3d7f9b99030a888c867939
xoshiro256	0x10003c03c3f3ecb1904b4edcf26259f85-	2^{64} : 0x992ccaf6a6fca052bd7a6a6e99c2ddc
		2^{96} : 0x9c6e6877736c46e3360fd5f2cf8d5d99
		"Period"/q: 0x1b4c7a8989405b16d3e4e127a6a11513
xoshiro256	0x10003c03c3f3ecb1904b4edcf26259f85-	2^{32} : 0xe055d3520fdb9d7214fafc0fdbc2087d8d0632bd08e6ac58120d583c112f69
		2^{48} : 0x5f728be2c97e9066474579292f705634f825539dee5e4763f11fb4faea62c7f1
		2^{64} : 0x12e4a2fbfc19bff934faff184785c20ab60d6c5b8c78f106b13c16e8096f0754
xoshiro256	0x10003c03c3f3ecb1904b4edcf26259f85-	2^{96} : 0x31eebb6c82a9615fb27c05962ea56a13cdb45d7def42c317148c356c3114b7a9
		2^{128} :

0280002bcefd1a5e9d116f2bb0f0f001	0x39abdc4529b1661ca9582618e03fc9aad5a61266f0c9392c180ec6d33cfd0aba 2 ¹⁶⁰ ; 0xf567382197055bf04823b45b89dc689c69e6e6e431a2d40bc04b4f9c5d26c200 2 ¹⁹² ; 0x39109bb02acbe63577710069854ee241c5004e441c522fb376e15d3efefdcbbf 2 ²²⁴ ; 0xa2b5d83a373c7ac2f31d2e03157bc387d317530723ab526a0c7840cbc3b121ad "Period"/ φ : 0x294e2bac089b06c7d4ce5d1a031b6cf8787f49127b37f506ac1c9e5f5f53046c
----------------------------------	---

0.8 Acknowledgments

Sebastiano Vigna reviewed this page and gave comments.

1 Notes

- ⁽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.
- ⁽⁴⁾ Salmon, J.K.; Moraes, M.A.; et al., "Parallel Random Numbers: As Easy as 1, 2, 3", 2011.
- ⁽⁵⁾ L'Ecuyer, Simard, Chen, Kelton, "An Object-Oriented Random-Number Package with Many Long Streams and Substreams", *Operations Research* 50(6), 2002.
- ⁽⁶⁾ Brown, F., "Random Number Generation with Arbitrary Strides", *Transactions of the American Nuclear Society* Nov. 1994.