Reversible random number generation

THOMAS BEWLEY*, UC San Diego, USA

The remarkable Permuted Congruential Generators (PCGs), Multiply With Carry (MWC) generators, and XOR/shift generators that have emerged in the last decade effectively render all pseudorandom number generators (PRNGs) proposed before them obsolete. They have small state size (2x to 4x the number of bits of each 32bit or 64bit integer output) and small code size (4 to 10 low-level C commands), they are fast (taking 1ns to 2ns per integer of output on modern hardware), and they are statistically excellent, passing every battery of statistical tests available today, including Big Crush in TestU01. This paper illustrates how the individual streams of all three of these modern families of carefully-optimized PRNGs may be marched exactly in reverse with codes of nearly the same simplicity (and, thus, speed, though certain cases require 128 bit arithmetic). This is valuable for many practical applications of such PRNGs, such as the variational (adjoint-based) analysis and optimization (of various control, identification, and estimation parameters) in Monte Carlo simulations, Ensemble Kalman Filters, and Particle Filters, in which statistically-good PRNGs are essential for generating appropriately-perturbed trajectories in forward-in-time simulations, and the inexpensive exact reproduction of the random excitations perturbing these trajectories in their retrospective (backward-in-time) analysis is required.

Additional Key Words and Phrases: Random number generation

ACM Reference Format:

Thomas Bewley. 2024. Reversible random number generation. 1, 1 (May 2024), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To provide cryptographic security, one can build a true random number generator (TRNG) that generates numbers from a physical (e.g., thermodynamic) process with entropy. However, both large-scale microprocessors (MPUs) in high-performance computing (HPC) servers and small-scale microcontrollers (MCUs) in embedded applications are useful in part because their behavior is entirely predictable. It is thus not obvious at first how to use an MPU or MCU appropriately to produce "adequately random" sequences for various 64-bit HPC and 32-bit embedded applications.

The development of deterministic pseudo random number generators (PRNGs) that produce sequences that are "effectively random" in application requires significant care. PRNGs quickly generate long sequences of unsigned integers $y_i \in [0, y_{\max}]$ that eventually repeat, usually with $y_{\max} = 2^b - 1$ for $b \in \{24, 32, 53, 64\}$. Good PRNGs can

(i) be initialized randomly (e.g., by using the number of microseconds since some epoch on the system clock),

(ii) be used to generate many, very long, statistically independent streams of unsigned integers, and

(iii) be postprocessed appropriately to generate the following three useful types of sequences:

A) Real numbers x_i with uniform distribution on an open interval (L, U) which, taking $b = 24$ or 53 for single or double precision resp., may be generated via $x_i = L + (U - L)(s_i/t)$ where $s_i = y_i/2.0 + 0.5$ and $t = 2^{b-1} + 0.5$.

B) Real numbers z_i with gaussian distribution, generated (for $\mu = 0$ and $\sigma = 1$) by applying a Box Muller transform [4] to a sequence $x_i \in (0, 1)$ (see above) via $z_i = \sqrt{-2\,\ln x_i}\,\cos(2\,\pi\,x_{i+1})$ and $z_{i+1} = \sqrt{-2\,\ln x_i}\,\sin(2\,\pi\,x_{i+1})$ for odd i.

Author's address: Thomas Bewley, tbewley@ucsd.edu, UC San Diego, Dept of MAE, La Jolla, CA, USA, 92093-0411.

Manuscript submitted to ACM

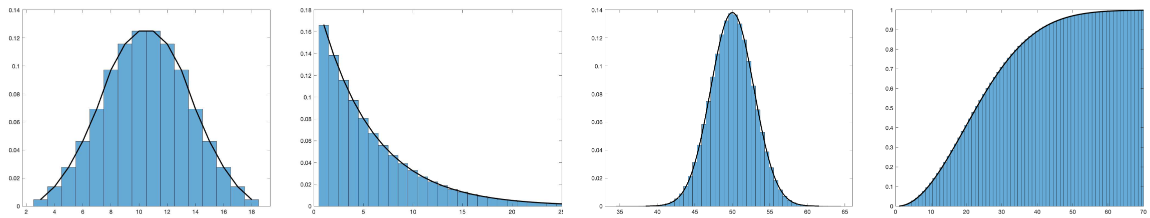

Fig. 1. (a) The (discrete) probability distribution generated by a model of the sum of three fair 6-sided dice. (b) The (discrete) exponential probability distribution generated by a model of the minimum $j > 0$ for which $x_i = x_{i+j}$ in the repeated rolling of a single 6-sided die. (c) A (discrete) histogram approximating the (continuous) gaussian probability distribution function (PDF) generated by a model of the sum of 100 real numbers uniformly distributed between 0 and 1. (d) The (discrete) cumulative distribution generated by a model of the birthday problem, indicating that in a random grouping of only 23 people, there is over a 50% chance that at least two have the same birthday, and with 50 people, there is a 97% chance. In all four subfigures, the bars indicate the statistics of millions of samples using a modern PRNG, and the solid curves represent the corresponding theoretical predictions.

C) Integers w_i with a discrete uniform distribution on a discrete interval $w_i \in [L, U]$, taking $b = 32$ or 64, which: (a) for $n = U - L + 1 = 2^s$, may be generated via $w_i = L + \lfloor y_i/N \rfloor$ with $N = 2^{b-s}$, or (b) for other n, may be generated via $w_i = L + \lfloor \tilde{y}_i/N \rfloor$ with $N = \lfloor y_{\max}/n \rfloor$ leveraging a PRNG sequence $\tilde{y}_i \in [0, \tilde{y}_{\max}]$, where $\tilde{y}_{\max} = n \cdot N - 1 < y_{\max} = 2^b - 1$, which itself may be generated from a standard $y_i \in [0, y_{\max}]$ PRNG sequence simply by eliminating all integer draws y_i with $y_i > \tilde{y}_{\max}$, thus ensuring the identical likelihood of each resulting integer $w_i \in [L, U]$.

Good PRNGs produce unsigned integer sequences that, primarily,

1) are characterized by *good statistical properties* (see, e.g., Figure 1), mimicking those of truly random processes,
2) have a *large period*, so in application they do not exhibit repeating patterns, and
3) are *fast to compute*, in a small memory footprint, when coded in a low-level language like C, C++, or Rust.

Note that, beyond property 1 above, the notion of *difficulty to predict* is sometimes also mentioned as a fourth desired property of a PRNG. However, none of the PRNGs considered in this work should be considered as cryptographically secure, and indeed most of them have already been "cracked"; that is, algorithms have been developed to determine their full internal state (and, thus, all their future outputs) from a relatively small number of their integer outputs. For the applications motivating this work (see the abstract), this potential fourth property is not of significant interest.

1.1 A brief survey of representative desired statistical properties of PRNGs

A fair 6-sided die (cf. loaded or shaved dice) rolls $\{1, 2, 3, 4, 5, 6\}$ with equal probability. The sum of two such dice will give a total of $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ with (discrete) probabilities $\{1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1\}/36$. Similarly, the sum of three such dice will give a total of 3 through 18 with probabilities $\{1, 3, 6, 10, 15, 21, 25, 27, 27, 25, 21, 15, 10, 6, 3, 1\}/216$ (see Figure 1a), etc; note that such probabilities are easily determined by the coefficients of the following polynomials (listed here as executable code in Matlab or Octave):

```
clear, syms z, for i=2:5, expand((z+z^2+z^3+z^4+z^5+z^6)^i), end
```

A good PRNG, adjusted to give integers on the interval $[1, n]$ for each n-sided die ($n = 6$ in the above example) with equal probability (type C above), should mimic such distributions over millions of trials.

When determining integers x_i by rolling a single fair n-sided die, the odds that the next number rolled, x_{i+1}, is the same as x_i is $p(1) = 1/n$. Thus, defining $c = (n-1)/n$, the odds that the minimum $j > 0$ for which $x_i = x_{i+j}$ is equal to $\hat{j}$ is $p(\hat{j}) = c^{\hat{j}-1}/n$, generating what is called the (discrete) *exponential* distribution (see Figure 1b). A good PRNG (again, of type C above) measured in this manner should mimic this exponential distribution over millions of trials.

Consider a PRNG adjusted to give a real number $x \in (L, U) = (0, 1)$ (type A above); this *uniform* distribution has mean $\mu \triangleq \mathscr{E}\{x\} = (L + U)/2 = 1/2$ and variance $\sigma^2 \triangleq \mathscr{E}\{(x - \mu)^2\} = (U - L)^2/12 = 1/12$. Any sum of $n = 100$ consecutive real numbers x_i so generated, denoted $y_i = \sum_{j=0}^{n-1} x_{i+j}$, should total about $y_i \approx n \cdot \mu = 50$, but will sometimes be a bit higher, and sometimes a bit lower. It is a remarkable consequence of the celebrated Central Limit Theorem that a histogram of the computed values of this sum, normalized appropriately to approximate a (continuous) *probability distribution function* (PDF), will tend towards a *gaussian* distribution, with a mean of $n \cdot \mu = 50$ and a variance of $n \cdot \sigma^2 = 8.333$ (see Figure 1c). A good PRNG measured in this manner should mimic this gaussian distribution over millions of trials.

Finally, the odds that 2 people selected at random do not have the same birthday is 364/365. By the same logic, the odds that, in a random grouping of n people, none have the same birthday is $p_{\text{not}}(n) = \prod_{k=1}^{n-1}(365 - k)/365$. The odds this is false (that is, in a random group of n people, at least 2 *do* have the same birthday) is $p_{\text{do}}(n) = 1 - p_{\text{not}}(n)$. This (discrete) distribution, known as the *birthday problem*, approximates the (continuous) *cumulative distribution function* (CDF) of the *gamma* distribution (see Figure 1d). A good PRNG measured in this manner should mimic this type of distribution over millions of trials.

Unfortunately, theoretical analyses of PRNGs are only useful up to a point; most useful "randomness" tests are, like those surveyed above (and, many others), only statistical in nature, and require extensive computational testing to quantify the long-term statistical behavior of a PRNG, to verify that it is as expected. In particular, the PractRand [7] and TestU01 [15] suites of statistical tests for PRNGs evolved from substantial original analysis of the subject by Knuth [12]. Note that, for a given size PRNG state, such statistical tests will all *eventually* fail; the question is only how big a PRNG integer stream needs to be generated before undesired statistical correlations begin to become apparent; today, passing the Big Crush test suite (part of TestU01) is generally accepted as the "gold standard".

1.2 A brief review of LCGs and reversibility

Linear congruential generators (LCGs) [26] are the essential starting point. LCGs are PRNGs defined by a simple recurrance of the form

$$\text{forward march:} \quad x \leftarrow (a \cdot x + c) \bmod m, \tag{1a}$$

where the multiplier a, increment c, and modulus m are fixed unsigned integers, and the state x is updated at each iteration. An LCG with $c = 0$ is often called a Lehmer generator or multiplicative congruential generator (MCG) [16].

Two types of LCGs are of particular interest: (a) those with prime m, which (when taken on their own, for a given size of m) have the best statistics, and (b) those with $m = 2^b$ and odd c, where b is the total number of bits in the binary representation of the unsigned integers being used, which are generally much faster to compute (for a given size of m) when implemented in a language that wraps on integer overflow, and thus form our focus here (with $b = 32$, 64, or 128). When using either type of LCG, the trick is to select a well for a given m. Most choices of this parameter result in "bad" PRNGs, with short periods and/or bad statistics. Some choices, though, give fairly "good" PRNGs (in terms of properties 1, 2, and 3 itemized previously) given the simplicity of (1a). A starting point to find a good value for a in the $m = 2^b$ case, known as the Hull-Dobell Theorem [10], is to take $\text{mod}(a, 8) = 5$ [i.e., $a = 8k + 5$ for some k]; though this choice (together with odd c) generates PRNG sequences with full period (i.e., which repeat only after m elements, including $x = 0$), most values of a so generated in fact still do not have good statistics. Parameters leading to statistically good LCGs must be searched for exhaustively, and are well tabulated in the literature [13, 25]. Note that, for a given a, replacing c with any other odd integer produces a different PRNG sequence that is qualitatively similar.

As a (very small) example, take $a = 8 \cdot 19 + 5 = 157$, $c = 47$, and $m = 2^8 = 256$ in (1a). Starting from $x = 0$, this LCG generates every integer from 0 to $m - 1 = 255$, once, then repeats, as can be seen by executing the following simple line of code in Matlab or Octave:

```
a=157, c=47; m=256; x(1)=0; for i=2:m+5, x(i)=mod(a*x(i-1)+c,m); end
```

The first 9 integers x generated by this code, $\{0, 47, 2, 105, 148, 243, 54, 77, 104, \ldots\}$, are written in binary as

```
00000000 00101111 00000010 01101001 10010100 11110011 00110110 01001101 01101000
```

Note that the least significant bits (LSBs) alternate[1] between 0 and 1. The next significant bit follows the sequence 0,1,1,0,0,1,1,0,0,1,1, ... That is, the statistics of the lower-order bits in an $m = 2^b$ LCG follow very noticeable patterns, as the lower-order bits of such an LCG affect the evolution of the higher-order bits, but the higher-order bits do not affect the evolution of the lower-order bits. However, when m is large, several of these lower-order bits can easily be suppressed, or somehow "scrambled" or "permuted" or "filtered" with the higher-order bits, when outputting the result of the PRNG subroutine; it is effectively the "mod m" part of a (deterministic) LCG that, when a is a substantial fraction of m, makes the higher-order bits of the LCG appear to be "more random".

Now consider the following slight modification to the above code:

```
astar=181, c=47; m=256; y(1)=0; for i=2:m+5, y(i)=mod(astar*(y(i-1)-c),m); end
```

This modified code generates exactly the same sequence of integers in y, but in reverse order. This *reversibility* of LCGs is easily seen mathematically by writing (1a) more plainly[2], subtracting c from both sides, and multiplying the resulting equation by a^*:

$$xnew = a \cdot xold + c \quad \Rightarrow \quad a^* \cdot (xnew - c) = (a^* \cdot a) \cdot xold;$$

thus, if $mod(a^* \cdot a, m) = 1$ [that is, if a^* is the "modular inverse" of a, the existence and computation of which are discussed further in §2.1], we can reverse the order of the march in (1a), determining $xold$ from $xnew$, with

$$\text{backward march:} \quad x \leftarrow a^* \cdot (x - c), \tag{1b}$$

where mod m arithmetic (i.e., wrap on overflow) is now (and, henceforth) implicitly assumed in the notation used.

1.3 Improving PRNGs beyond LCGs

Good PRNGs are hard to find. A candidate PRNG may be relatively (a) fast to calculate, with (b) small memory footprint and (c) small code size and (d) long period, and may (e) satisfy many statistical randomness tests (see, e.g., Figure 1), only to fail some other randomness test (see, e.g., [3]). Though increasing the size of the internal state is certainly valuable (albeit, at increased computational cost), when restricted to using 32-bit, 64-bit, or 128-bit arithmetic, LCGs alone have proven to be insufficient for most applications.

Over the years, *many* PRNGs have been developed. Some are statistically adequate but unnecessarily complex in terms of both space usage and code size, including the Mersenne Twister and stream ciphers like RC4 and its modern successor ChaCha20. Some are simpler, but with inferior statistical properties, including IBM's once pervasive yet "truly horrible" [12] RANDU, Numerical Recipe's RanQ1, and unix's drand48, rand, and random implementations.

[1]That is, odd entries in the sequence are even integers, and even entries in the sequence are odd integers, and thus the sum of any two consecutive integers in the sequence is odd, thereby failing the first test discussed in §1.1.

[2]That is, we rename the old and new values of x as $xold$ and $xnew$, and implicitly assume (as we do in the remainder of this paper) that all math is performed mod m where $m = 2^b$, which in code means "wrapping on integer overflow" in a b-bit representation (again, taking b = 32, 64, or 128).

In the following, we survey three modern families of PRNGs with 23-bit to 64-bit output, summarized in Figure 2 and briefly compared in Table 1, which effectively supersede all of the PRNGs mentioned above:

• **Permuted Congruential Generators** (PCGs) [21, 22, 24] propagate an internal LCG (1a) with $m = 2^b$, fixed odd c, and a carefully chosen multiplier a satisfying the Hull-Dobell Theorem, typically taking $b = 64$ or $b = 128$, and output integers of size $b/2$ bits at each iteration, generated with clever bit permutations on the internal LCG state x.

• **Multiply With Carry** (MWC) generators [11, 18, 28] propagate an internal state with $r+1$ integers $\{x_1, x_2, \ldots, x_r, c\}$ each with b bits, typically taking $r = 1$ to 3 and $b = 64$, using essentially the same LCG formula, taking a with b bits and generating an intermediate result t with $2b$ bits, with $r-1$ intermediate "lags", updating the full state at each iteration (including c, unlike LCGs/PCGs) such that, for example (see Figure 4 for the notation used throughout this discussion),

```
t←x1*a+c,  x1←x2,  x2←x3,  [c;x3]←t,
```

and output b-bit integers at each iteration by implementing clever bit permutations on x_1.

• **XOR/shift** generators [2, 19] propagate by taking cascaded XORs of bit-shifted versions of k state variables $\{s_1, s_2, \ldots, s_k\}$ each with b bits, typically with $k = 2$ to 4 and $b = 32$ or 64, such that, for example,

```
t=s1≪A,  s2←s2^s0,  s3←s3^s1,  s1←s1^s2,  s0←s0^s3,  s2←s2^t,  s3←s3⋘B,
```

and then output some combination of these state variables at each iteration with additional clever bit permutations.

After some further introduction below, this paper shows how all three of these modern families of high-quality (fast, small, statistically excellent) PRNGs are inexpensively reversible, as summarized in Figure 3 and derived in §2. Note in particular that the relative pros and cons of these three classes of schemes have been debated vigorously online (see, in particular, [23, 27], and elsewhere on reddit). This debate is worth a read, but does not form a significant focus here. (In short, amongst other things, the statistical correlation of separate PCG streams is one stated concern, and the behavior of MWC and XOR/shift generators when the state happens to reach a condition in which many of the state bits are zero, and how quickly these linear PRNGs can move away from such a condition, commonly referred to as the "escape from zeroland" problem, is another stated concern.) From a practical perspective, suffice it to say here that such stated concerns are likely of relatively minor significance when compared to the substantial improvements made by these PRNGs over all other PRNGs that came before them.

1.4 Multiple independent streams and jump functions

As mentioned in the abstract, many HPC applications of PRNGs (Monte Carlo simulations, Particle Filters, Ensemble Kalman filters, etc) involve the random excitation of many parallel numerical simulations in a statistically-appropriate manner. The statistical forcing in such applications accounts, in a sense, for the undersampling of the uncertainty distribution of the problem under consideration. In such applications, significant care is required to ensure that the random excitations of these parallel simulations are in fact statistically independent.

The PCG32 and PCG64 Permuted Congruential Generators support independent streams simply by selecting a different (odd, 64bit or 128bit) constant increment c (and, a different initialization; see §1.5) for each stream. Thus, PCG32 supports up to 2^{63} streams, each with period 2^{64}, and PCG64 supports up to 2^{127} streams, each with period 2^{128}. If far fewer streams than this are needed, which is typical, care should be taken that the values of the increment c are substantially different for each stream, which can be accomplished by determining these increment values themselves with a simple LCG (see §1.2) of the appropriate size, and with statistically good coefficients (see [13, 25]).

Multiply With Carry and XOR/shift generators, on the other hand, each support only 1 very long stream of length 2^z, with (a) $z = 128$ for the MWC128, xoshiro128, and xoroshiro128 generators, (b) $z = 192$ for the MWC192 generator, and (c) $z = 256$ for the MWC256 and xoshiro256 generators. However, with some effort, straightforward algorithms may be determined for "jumping" forward a pre-defined amount in each such stream, leveraging convenient precomputed "jump functions" for jumps of particular sizes of interest, as discussed further in [9, 20]. Thus, in each case, one can develop 2^x independent streams each of length 2^y, where $x + y = z$. Typical examples are illustrated in Table 1.

1.5 Seeding

All of the PRNGs discussed here need to be initialized. This can either be handled deterministically, in order to generate the same sequence of integers after initialization every time, for testing purposes, or in some sense randomly, for production runs. A single random nonzero "seed" may be generated in many different ways; one of the simplest is by determining, from the CPU's system clock, a 64-bit unsigned integer representing the number of microseconds since some system "epoch" (that is, from some reference date and time in the past). As there are almost as many 64-bit unsigned integers as there are microseconds in a million years, the seed so generated is guaranteed to be different every time the random number generator is initialized, which is sufficient.

The size of the internal state of the 14 modern PRNGs considered in this work is specified (immediately after the name of each PRNG) in Figure 2. Each of these internal states is a set of one to four 32-bit, 64-bit, or 128-bit integers. In the case of 64-bit and 128-bit state variables, a simple approach to PRNG initialization is to set the first of these large integers defining the initial PRNG state as the 64-bit seed described in the previous paragraph, and to set the remaining integers defining the initial PRNG state using a simple LCG of the appropriate size (see §1.2), with statistically good coefficients (see [13, 25]), itself initialized by this seed. Note that, in the case of the MWC schemes, c must always be smaller than a; one approach that satisfies this for the cases considered is to initialize c using the LCG approach described above, then setting to zero the MSB of this preliminary value of c, which is sufficient to ensure that $c < a$ regardless of the state of the LCG (recall that, in the MWC schemes, c changes to a different large 64-bit integer, with $c < a$, every iteration thereafter). In the case of XOR/shift PRNGs with 32-bit state variables, a reasonable approach is to set the first state variable equal to the upper 32-bits of the 64-bit seed mentioned in the previous paragraph, to set the second state variable equal to the lower 32-bits of this 64-bit seed, and to set the remaining integers defining the initial PRNG state using a simple LCG as before. After appropriate initialization as described here, which quickly generates a random initial state well away from the origin (also known, in the PRNG literature, as "zeroland" - see the last paragraph in §1.3), a modern PRNG is ready to be used immediately; no "warm-up" period is required.

When generating many (say, N) statistically-independent streams in an HPC setting following the PCG approach, with each stream having a different increment c as discussed in §1.4, a (different) initialization is needed for each stream. All such streams might be generated at around the same time, and thus the approach described in the first paragraph of this section might unfortunately produce an identical initialization in multiple streams. In this case, one possible approach is to also incorporate the process ID (getpid in C) and/or the host ID (gethostid in C) in the generation of the seed for each of the N streams. An alternative initialization approach, which helps to ensure that each of the streams is initialized with substantially different values, is to use just one seed, for the first stream, and then to generate initial values for the states of the $(N - 1)$ other streams by applying a statistically good LCG to this single seed. Sharing the (single) seed with each (individually numbered) stream and leveraging the LCG "jumping" approach discussed in §2.6, this initialization approach (of both the state x and the increment c) can be efficiently run in parallel for each stream.

```
PCG32 XSH RR [21, 22]: 64bit state x (a=0x5851F42D4C957F2D, c=odd), 32bit output z
t←((x≫18)^x)≫27, r←x≫59, z←(t≫r)|((t≪-r)&31), x←x*a+c

PCG64 DXSM [8, 24]: 128bit state x (a=0xDA942042E4DD58B5, c=odd), 64bit output hi
[hi;lo]←x, hi←(hi^(hi≫32))*a, hi←(hi^(hi≫48))*lo, x←x*a+c
---------------------------------------------------------------------------
MWC128 [18, 28]: 2x64bit state {x,c} (a=0xFFEBB71D94FCDAF9), 64bit output z
z←x^(x≪32), t←x*a+c, [c;x]←t

MWC192 [18, 28]: 3x64bit state {x,y,c} (a=0xFFA04E67B3C95D86), 64bit output y
t←x*a+c, x←y,          [c;y]←t

MWC256 [18, 28]: 4x64bit state {x,y,z,c} (a=0xFFF62CF2CCC0CDAF), 64bit output z
t←x*a+c, x←y, y←z,     [c;z]←t
---------------------------------------------------------------------------
xoshiro128++ [2, 28]: 4x32bit state {s0,s1,s2,s3}, 32bit output z←((s0+s3)⋘7)+s0
t←s1≪9, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘11

xoshiro128** [2, 28]: 4x32bit state {s0,s1,s2,s3}, 32bit output z←((s1*5)⋘7)*9
t←s1≪9, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘11

xoshiro128+ [2, 28]: 4x32bit state {s0,s1,s2,s3}, 24bit output z←(s0+s3)≫8
t←s1≪9, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘11
---------------------------------------------------------------------------
xoshiro256++ [2, 28]: 4x64bit state {s0,s1,s2,s3}, 64bit output z←((s0+s3)⋘23)+s0
t←s1≪17, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘45

xoshiro256** [2, 28]: 4x64bit state {s0,s1,s2,s3}, 64bit output z←((s1*5)⋘7)*9
t←s1≪17, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘45

xoshiro256+  [2, 28]: 4x64bit state {s0,s1,s2,s3}, 53bit output z←(s0+s3)≫11
t←s1≪17, s2←s2^s0, s3←s3^s1, s1←s1^s2, s0←s0^s3, s2←s2^t, s3←s3⋘45
---------------------------------------------------------------------------
xoroshiro128++  [2, 28]: 2x64bit state {s0,s1}, 64bit output z←((s0+s1)⋘17)+s0
s1←s1^s0, s0←(s0⋘49)^s1^(s1≪21), s1←s1⋘28

xoroshiro128**  [2, 28]: 2x64bit state {s0,s1}, 64bit output z←((s0*5)⋘7)*9
s1←s1^s0, s0←(s0⋘24)^s1^(s1≪16), s1←s1⋘37

xoroshiro128+  [2, 28]: 2x64bit state {s0,s1}, 53bit output z←(s0+s1)≫11
s1←s1^s0, s0←(s0⋘24)^s1^(s1≪16), s1←s1⋘37
```

Fig. 2. Complete specification of 14 modern (fast, small, statistically excellent) PRNGs; see Figure 4 for pseudocode notation.

```
PCG32_rev:  state x (a*=0xC097EF87329E28A5), output z
x←a*∗(x−c), t←((x≫18)^x)≫27, r←x≫59, z←(t≫r)|((t≪−r)&31)

PCG64_rev:  state x (a*={0x0CD365D2CB1A6A6C, 0x8B838D0354EAD59D}), output hi
x←a*∗(x−c), [hi;lo]←x, hi←(hi^(hi≫32))∗a, hi←(hi^(hi≫48))∗lo
--------------------------------------------------------------------------------
MWC128_rev:  state {x,c} (a=0xFFEBB71D94FCDAF9), output z
t←[c;x],           [x,c]←t/a, z←x^(x≪32)

MWC192_rev:  state {x,y,c} (a=0xFFA04E67B3C95D86), output y
t←[c;y], y←x,      [x,c]←t/a,

MWC256_rev:  state {x,y,z,c} (a=0xFFF62CF2CCC0CDAF), output z
t←[c;z], z←y, y←x, [x,c]←t/a,
--------------------------------------------------------------------------------
xoshiro128_rev++:  state {s0,s1,s2,s3}, output z←((s0+s3)⋘7)+s0
s3←s3⋙11, q←s1, r←s1^s2, s0←s0^s3, s1←Shift32(r), s2←q^s1^s0, s3←s3^s1

xoshiro128_rev**:  state {s0,s1,s2,s3}, output z←((s1∗5)⋘7)∗9
s3←s3⋙11, q←s1, r←s1^s2, s0←s0^s3, s1←Shift32(r), s2←q^s1^s0, s3←s3^s1

xoshiro128_rev+:   state {s0,s1,s2,s3}, output z←(s0+s3)≫8
s3←s3⋙11, q←s1, r←s1^s2, s0←s0^s3, s1←Shift32(r), s2←q^s1^s0, s3←s3^s1
--------------------------------------------------------------------------------
xoshiro256_rev++:  state {s0,s1,s2,s3}, output z←((s0+s3)⋘23)+s0
s3←s3⋙45, q←s1, r←s1^s2, s0←s0^s3, s1←Shift64(r), s2←q^s1^s0, s3←s3^s1

xoshiro256_rev**:  state {s0,s1,s2,s3}, output z←((s1∗5)⋘7)∗9
s3←s3⋙45, q←s1, r←s1^s2, s0←s0^s3, s1←Shift64(r), s2←q^s1^s0, s3←s3^s1

xoshiro256_rev+:   state {s0,s1,s2,s3}, output z←(s0+s3)≫11
s3←s3⋙45, q←s1, r←s1^s2, s0←s0^s3, s1←Shift64(r), s2←q^s1^s0, s3←s3^s1
--------------------------------------------------------------------------------
xoroshiro128_rev++:  state {s0,s1}, output z←((s0+s1)⋘17)+s0
s1←s1⋙28, s0←(s0^s1^(s1≪21))⋙49, s1←s1^s0

xoroshiro128_rev**:  state {s0,s1}, output z←((s0∗5)⋘7)∗9
s1←s1⋙37, s0←(s0^s1^(s1≪16))⋙24, s1←s1^s0

xoroshiro128_rev+:   state {s0,s1}, output z←(s0+s1)≫11
s1←s1⋙37, s0←(s0^s1^(s1≪16))⋙24, s1←s1^s0
```

Fig. 3. The (new) efficient and exact reversal of the 14 modern PRNGs summarized in Figure 2, as derived in §2 and verified in [1].

(1) a≪k and a≫k denote, resp., the leftshift and rightshift of the bitwise representation of a by k bits; the bit positions vacated by ≫ and ≪ are filled with zeros.
(2) a⋘k and a⋙k denote, resp., the *periodic* leftshift and rightshift of the bitwise representation of a by k bits, where a ⋘ k scoots the k bits moved off the left in the bitwise representation of a into the k vacated positions on the right, and may be implemented in C, for unsigned integers represented with b=32 or b=64 bits, as (x<<k)|(x>>(b−k)); ditto for a ⋙ k, which may be implemented in C as (x>>k)|(x<<(b−k)).
(3) a&b, a^b, and a|b denote, resp., the logical AND, XOR, and OR of the bitwise representations of a and b.
(4) ~a and −a=~a+1 denote, resp., bitwise negation, and the two's complement representation of negative a.
(5) [hi;lo]←x denotes the splitting of an unsigned integer x, represented with b bits, into its high-order and low-order parts, hi and lo, each represented with b/2 bits; this may be implemented in C as hi=x>>32, lo=x&0xFFFFFFFF for b=32, and as hi=x>>64, lo=x&0xFFFFFFFFFFFFFFFF for b=64.
(6) x←[hi; lo] denotes the joining of hi and lo, each represented with b/2 bits, into x, represented with b bits.
(7) [q,r]=a/b denotes the computation of the quotient q and remainder r such that a=b∗q+r where r<b; see §2.2.

Fig. 4. Pseudocode notation of the bitwise operations on unsigned integers used in Figures 2 and 3 (mostly in C; note that ≪ is written in ASCII as <<).

generator	useful bits of output	state size	independent streams	recommended default use [1, 28]	other properties
PCG32	32 bits	64 bits	2^{63} streams of period 2^{64}		jumps easy for
PCG64	64 bits	128 bits	2^{127} streams of period 2^{128}		any k (see §2.6)
MWC128	64 bits	2x64 bits	2^{32} streams of length 2^{96}		fast, iff 128bit
MWC192	64 bits	3x64 bits	2^{48} streams of length 2^{144}		math available
MWC256	64 bits	4x64 bits	2^{64} streams of length 2^{192}		(see §2.2)
xoshiro128++	32 bits	4x32 bits	2^{32} streams of length 2^{96}	32-bit integers	
xoshiro128**	32 bits	4x32 bits	2^{32} streams of length 2^{96}		
xoshiro128+	24 bits	4x32 bits	2^{32} streams of length 2^{96}	single-precision reals	
xoshiro256++	64 bits	4x64 bits	2^{64} streams of length 2^{192}	64-bit integers	
xoshiro256**	64 bits	4x64 bits	2^{64} streams of length 2^{192}		
xoshiro256+	53 bits	4x64 bits	2^{64} streams of length 2^{192}	double-precision reals	
xoroshiro128++	64 bits	2x64 bits	2^{32} streams of length 2^{96}		reduced
xoroshiro128**	64 bits	2x64 bits	2^{32} streams of length 2^{96}		memory
xoroshiro128+	53 bits	2x64 bits	2^{32} streams of length 2^{96}		footprint

Table 1. Some properties of the modern PRNGs given in Figure 2. Each of them: (a) execute in 1ns to 2ns per integer output when implemented efficiently in C on a modern CPU, (b) have zero failures when tested in PractRand and TestU01, including Big Crush, and (c) are efficiently reversible (as derived in §2, summarized in Figure 3, and implemented in [1]).

1.6 Rare events, and "Smart Shuffling"

"Rare" events in random sequences happen with perhaps surprising frequency. For example, π is conjectured, but not proven, to be a "normal" number (that is, its decimal digits, discretely distributed on $[0, 9]$, have statistics like those generated by a good PRNG, as discussed in §1.1); however, within the first 1000 decimal digits of π, the subsequence 999999 appears. When using a PRNG to randomly "shuffle" a list of songs or jokes, such occasional repeats might be unwanted. In such situations, it is a straightforward matter to keep a running list of the last M integers produced by the PRNG, and to reject any new integer produced by the PRNG that repeats one of these recent values. Such post-processing, actually, substantially reduces the "randomness" of the resulting integer sequence, as quantified in §1.1 (with dice, eliminating the possibility of rolling "snake eyes", etc), but in certain applications can make such sequences "seem" more random to a human user [17].

2 REVERSIBILITY OF THE MODERN CLASSES OF PRNGS

As mentioned in the abstract and introduced in §1.4, Monte Carlo simulations, Particle Filters, Ensemble Kalman filters, and the like use the random excitation of many parallel numerical simulations to account, in a sense, for the undersampling of the uncertainty distribution in large-scale simulation problems. In such applications, it is often desired to perform retrospective (backward-in-time) analyses, to see how the spread of a set of perturbed simulations changes when (a) initial conditions, (b) system parameters, and/or (c) control variables are changed. In order to decouple the (numerical) effects of the random forcing from the (physically interesting) effects that (a), (b), and/or (c) have on this spread of the set of perturbed simulations, it is sometimes needed [6] to reproduce the (many) random excitations used on the forward (state) marches when revisiting these simulations in the corresponding reverse (adjoint/costate/dual/Lagrange multiplier) calculations. The results of the present paper make this task inexpensive to accomplish.

The discussion below focuses on the efficient reversal of the 14 specific PRNGs listed in Figure 2, the (simple) results of which are listed in Figure 3. The approaches taken to reverse these 14 schemes should extend immediately to any new PRNGs in these three general classes (e.g., in [11]) that are inevitably developed.

2.1 PCG32 and PCG64

Using mod m arithmetic, PCG32 (with $m = 2^{64}$) and PCG64 (with $m = 2^{128}$) both propagate via $x \leftarrow x \cdot a + c$, where $a =$ 0x5851F42D4C957F2D for PCG32 and $a =$ 0xDA942042E4DD58B5 for PCG64, and, in any given stream, c is taken as an odd constant [21]. As discussed in §1.2, the trick to reversing this propagation, in either case, is to determine an a^* such that, using mod m arithmetic, $a^* \cdot a = 1$ (that is, where a^* is the "modular inverse" of a, which is guaranteed to exist if m is a power of 2 and a satisfies the Hull-Dobell Theorem, and is therefore odd), from which it follows that the reverse propagation is given by $x \leftarrow a^* \cdot (x - c)$. The calculation of a^* such that $\mathrm{mod}(a^* \cdot a, m) = 1$ can be accomplished by solving Bezout's equation $k \cdot m + a^* \cdot a = 1$ for the integers k and a^* using the extended Euclidean algorithm. Denoting the Euclidean division of m/a as $m = q \cdot a + r$ (that is, as giving a quotient q and remainder r, both nonnegative integers), this computation proceeds by first solving for the GCD g of m and a (which is $g = 1$, since m and a are coprime) using the standard Euclidean algorithm over the integers:

$$m = q_1\, a + r_1, \quad a = q_2\, r_1 + r_2, \quad r_1 = q_3\, r_2 + r_3 \quad \longrightarrow \quad r_{n-4} = q_{n-2}\, r_{n-3} + r_{n-2}, \quad r_{n-3} = q_{n-1}\, r_{n-2} + g, \tag{2a}$$

where $r_{n-2} = q_n\, g + 0$. The extended Euclidean algorithm then work backwards through the relations in (2a), solving each relation for its last term:

$$g = r_{n-3} - q_{n-1}\, r_{n-2}, \quad r_{n-2} = r_{n-4} - q_{n-2}\, r_{n-3} \quad \longrightarrow \quad r_3 = r_1 - q_3\, r_2, \quad r_2 = a - q_2\, r_1, \quad r_1 = m - q_1\, a. \tag{2b}$$

Starting with the first expression in (2b), substituting in the second to eliminate r_{n-2}, substituting in the next to eliminate r_{n-3}, etc., ultimately leads to $g = k\, m + a^*\, a$, where k and a^* are linear combinations of the integers q_i appearing in (2a). This can all be implemented in executable code, called as [k, astar]=Bezout(m,a), as follows:

```
function [g,q,n]=GCD(a,b)
n=0, rm=a, r=b
while r~=0
   n=n+1, [q{n},rn]=rm/r, rm=r, r=rn
end, g=rm
```

```
function [x,y]=Bezout(a,b)
[g,q,n] = GCD(a,b)
x=0, y=1, for j=n-1:-1:1
   t=x, x=y, y=t-q{j}*y
end
```

A challenge arises when implementing the GCD algorithm using unsigned integers represented using only b bits, where $m = 2^b$, as in this case m exceeds (that is, is exactly one larger than) the maximum unsigned integer representable with b bits. This challenge may be circumvented by replacing the first step of (2a) with $(m - a) = \tilde{q}_1\, a + r_1$, where $q_1 = \tilde{q}_1 + 1$, noting that, as opposed to the integer $m = 2^b$, the integer $(m - a)$ [that is, the integer $-a$ represented in twos complement notation] *is* representable with b bits. Determining q_1 from $\tilde{q}_1$ in this manner, the rest of the standard Euclidean algorithm (2a) [in code, GCD], to compute the q_i, followed by the extended Euclidean algorithm (2b) [in code, Bezout], to compute a^*, then proceeds as before.

Following this process, it is readily determined that a^* for PCG32_rev, which is representable using $b = 64$ bits, and a^* for PCG64_rev, which requires $b = 128$ bits to represent, are both as indicated in Figure 3.

2.2 128 bit arithmetic and integer division

Note that full hardware implementations of arithmetic on 128-bit unsigned integers (and, on combinations of 128-bit and 64-bit unsigned integers) are not broadly available today, especially on MCUs, and in many situations must be built up in software from several smaller (64-bit or 32-bit) arithmetic operations $\{+, -, \times\}$. This is entirely straightforward for unsigned integer addition, subtraction, and multiplication. However, it is difficult to implement, in software, general 128-bit unsigned integer division from (hardware) arithmetic operations on smaller integers; it generally turns out to be more efficient (though, still quite slow) to perform integer division following a bitwise approach in software, like the nonrestoring division algorithm outlined below.

The nonrestoring division algorithm [12, 29] is a simple and convenient approach for performing unsigned integer division from scratch when necessary (e.g., on unsigned integers of size 128 bits or larger). This algorithm computes the quotient $q = a/b$ and the remainder r from the dividend a and the divisor b such that, as is standard[3,4], $a = b \cdot q + r$, where $r < b$. For completeness, this algorithm is listed in (Matlab-like, for clarity) pseudocode below.

```
function [q,r]=div128(a,b)
if b>a,       r=a,   q=0, return   % <-- solve the trivial cases directly
elseif b>a-b, r=a-b, q=1, return
else
  q=a, r=0
  for n=128:-1:1
    s=bitget(r,128), r=bitsll(r,1), r=bitset(r,1,bitget(q,128)), q=bitsll(q,1)
    if s, r=r+b, else, r=r-b, end
    if bitget(r,128), q=bitset(q,1,0); else, q=bitset(q,1,1), end
  end
  if bitget(r,128), r=r+b, end
end
```

For convenience, in [1], we provide (amongst other things) both efficient stand-alone functions and convenient new class definitions for the simple arithmetic operations $\{+, -, \times, /\}$, relationals $\{<, >, ==, \ldots\}$, and bitwise operations

[3]This is the modern definition for unsigned integer division, as used by Ada, C/C++, Fortran, Go, Mathematica, Python, R, Ruby, Rust, SQL, Swift, and many other computer languages. Unfortunately, as of this writing, the / operator for unsigned integer division in Matlab rounds to the *nearest* integer, instead of rounding towards zero (a.k.a. truncated division) or (equivalently, for unsigned integers) towards $-\infty$ (a.k.a. floored division [12]), and thus does not behave in this standard manner; in most applications using Matlab's built-in integer data types, be certain to use idivide instead.

[4]Note also that Matlab's built-in integer data types saturate instead of wrap upon integer overflow, rendering them inconveient for testing PRNGs.

$\{\ll, \gg, \lll, \ggg, \&, \wedge, |, \sim, -, ...\}$ on unsigned integer data types, from 8-bit to 1024-bit, that both wrap on integer overflow, and (unlike Matlab's builtin / operator) implement the standard definition of unsigned integer division and remainder $[q, r] = a/b$ as discussed above (that is, $a = b \cdot q + r$ where $r < b$), thus facilitating the easy derivation and testing of PRNGs, like those discussed herein, in Matlab.

2.3 MWC128, MWC192, MWC256

As stated previously, MWC128, MWC192, MWC256 all propagate (using mod m arithmetic, where $m = 2^b$) via

$$t \leftarrow x_1 \cdot a + c, \quad x_1 \leftarrow x_2, \quad \ldots \quad x_{r-1} \leftarrow x_r, \quad [x_r; c] \leftarrow t. \tag{3a}$$

To reverse the direction of this propagation, we effectively need to invert each relation in (3a) and compute them in the opposite order. Note in particular that the inversion of the multiply/add operation in the first step above (to determine t) is the quotient/remainder operation in the last step below (to determine the quotient x_1 and remainder c).

$$t \leftarrow [c; x_r], \quad x_r \leftarrow x_{r-1}, \quad \ldots, \quad x_2 \leftarrow x_1, \quad [x_1, c] \leftarrow t/a. \tag{3b}$$

2.4 The xoshiro128 and xoshiro256 families

Xoshiro128++, xoshiro128**, xoshiro128+, xoshiro256++, xoshiro256**, and xoshiro256+ all propagate via

$$\mathtt{t=s1{\ll}A,\ \ s2{\leftarrow}s2\,\hat{}\,s0,\ \ s3{\leftarrow}s3\,\hat{}\,s1,\ \ s1{\leftarrow}s1\,\hat{}\,s2,\ \ s0{\leftarrow}s0\,\hat{}\,s3,\ \ s2{\leftarrow}s2\,\hat{}\,t,\ \ s3{\leftarrow}s3{\lll}B,} \tag{4a}$$

with the output z=f(·) of the ++, **, and + variants of these schemes computed, respectively, according to

$$\mathtt{f1=((s0+s3){\lll}D)+s0,\ \ f2=((s1*D){\lll}E)*F,\quad or\quad f3=s0+s3,} \tag{4b}$$

where the constants {A,B,D,E,F} have been optimized for each of the schemes in [2] (see Figure 2). We focus here specifically on the propagation relations in (4a), which may equivalently be reordered into the form:

$$\mathtt{s2{\leftarrow}s2\,\hat{}\,s0,\ \ s3{\leftarrow}s3\,\hat{}\,s1,\ \ t=s1{\ll}A,\ \ s1{\leftarrow}s1\,\hat{}\,s2,\ \ s2{\leftarrow}s2\,\hat{}\,t,\ \ s0{\leftarrow}s0\,\hat{}\,s3,\ \ s3{\leftarrow}s3{\lll}B.} \tag{4c}$$

We again seek to reverse the direction of this propagation, inverting each relation in (4c) and computing them in the opposite order. The first two and last two relations in (4c) are easily inverted, to find the value of the variable on the LHS before each update from the value of that variable after the update. For example, writing the last relation in (4c) as (s3new=s3old⋘B)⋙B, it follows immediately that s3old=s3new⋙B. Similarly, writing the first relation in (4c) as (s2new=s2old^s0)^s0, and noting the associativity and commutativity of the XOR (^) operation, and that s0^s0=0, it follows that s2old=s2new^s0. We thus focus on the three relations in the middle of (4c), which can not be individually inverted, by first writing them in the form

$$\mathtt{t=s1old{\ll}A,\ \ s1new=s1old\,\hat{}\,s2old,\ \ s2new=s2old\,\hat{}\,t.} \tag{5a}$$

To proceed, we first write the middle relation in (5a) as (s1new=s1old^s2old)^t, from which it follows that

$$\mathtt{s1new\,\hat{}\,(s1old{\ll}A)=s1old\,\hat{}\,s2new}\quad\Rightarrow\quad\mathtt{r\,\hat{}\,(s1old{\ll}A)=s1old}\ \ \text{where}\ \ \mathtt{r=s1new\,\hat{}\,s2new.} \tag{5b}$$

The relation at right in (5b), for r, is easily computed from s1new and s2new. The relation in the middle of (5b) then needs to be solved for s1old, given r and A. Once s1old is determined, the value of s2old is easily calculated from the relation in the middle of (5a), written in the form s2old=s1new^s1old.

Simplifying the notation a bit, what remains is to develop an efficient algorithm to solve s1=r^(s1≪A) for s1, given r and A. Since the last A bits of (s1≪A) are zero, it is seen that the first A bits of s1 are just

```
s1(1:A)=r(1:A)                                         (6a)
```

where r(1) denote the MSB of r. Given this initialization, and denoting as b the number of bits in the discretization, we can then loop through to compute the remaining bits of s as follows:

```
for i=A+1:b,  s1(i)=r(i)^s1(i-A),  end                  (6b)
```

To accelerate its execution, the loop in (6b) can be manually unrolled in chunks of size A. In particular, for the xoshiro128 schemes, we have $b = 32$ and $A = 9$, whereas for the xoshiro256 schemes, we have $b = 64$ and $A = 17$; we thus define the following two simple functions to compute (6a)-(6b) in these two cases:

```
function s1=Shift32(r)
s1(1:9)=r(1:9)
s1(10:18)=r(10:18)^r(1:9)
s1(19:27)=r(19:27)^s1(10:18)
s1(28:32)=r(28:32)^s1(19:23)
```

```
function s1=Shift64(r)
s1(1:17)=r(1:17)
s1(18:34)=r(18:34)^r(1:17)
s1(35:51)=r(35:51)^s1(18:34)
s1(52:64)=r(52:64)^s1(35:47)
```

To summarize, applying a minor bit of additional reordering to improve instruction-level parallelism, the six variants of xoshiro128 and xoshiro256 mentioned previously may all be marched in reverse via the equations shown in the corresponding rows of Figure 3.

2.5 The xoroshiro128 family

Similarly, but much more simply, xoroshiro128++, xoroshiro128**, and xoroshiro128+ all propagate via

```
z=f(·),  s1←s1^s0,  s0←(s0⋘A)^s1^(s1≪B),  s1←s1⋘C,      (7)
```

with the output z=f(·) of the ++, **, and + variants of these schemes computed according to (4b) as before, where again the constants have been optimized for each of the schemes in [2] (see Figure 2).

Applying similar logic as before, reverse propagation of these three variants of xoroshiro128 may be achieved via

```
s1←s1⋙C,  s0←(s0^s1^(s1≪B))⋙A,  s1←s1^s0,
```

with z=f(·) computed after each step as before.

2.6 An accelerated approach for jumping LCGs (and, thus, PCGs) in reverse

As noted in (1a), LCGs (and, thus, PCGs) propagate any individual stream (that is, for a given value of c) according to $x_{n+1} = a \cdot x_n + c$. Thus, as in [5], writing k in binary as $k = \sum_{i=1}^{i_{max}} \bar{k}_i\, 2^{i-1}$ where the individual $\bar{k}_i$ are bits (zero or one), LCGs and PCGs can easily be jumped forward k steps, for any k, by calculating, using mod $m = 2^b$ arithmetic,

$$x_{n+k} = A \cdot x_n + C \quad \text{where} \quad \begin{array}{l} A = a^k = a^{\left[\sum_{i=1}^{i_{max}} \bar{k}_i\, 2^{i-1}\right]} = \prod_{i=1}^{i_{max}} a^{(2^{i-1})^{\bar{k}_i}}, \\ C = c\,[a^{k-1} + a^{k-2} + \ldots + a + 1] = c[a^k - 1]/[a-1]. \end{array} \tag{8a}$$

Thus, streamlining the algorithm proposed in [5], A and C above can be computed quickly as follows:

```
function [A,C]=Function_A_C(a,c,k)
A=1, kbar=dec2bin(k), imax=length(kbar), h=a
for i=imax:-1:1, if kbar(i), A=A*h, end, h=h*h, end
C=c*(A-1)/(a-1)
```

where, as in Matlab, the command kbar=dec2bin(k) converts the integer k into a minimal-length vector of bits kbar, where kbar(1) is the MSB and kbar(imax) is the LSB, where imax=length(kbar).

As noted in (1b), LCGs and PCGs propagate in reverse according to $x_{n-1} = a^* \cdot (x_n - c)$. Thus, LCGs and PCGs can be jumped in reverse k steps by calculating, using mod $m = 2^b$ arithmetic,

$$x_{n-k} = A^* \cdot x_n - C^* \quad \text{where} \quad A^* = (a^*)^k, \quad C^* = c\, a^* [(a^*)^k - 1]/[a^* - 1]. \tag{8b}$$

Thus, for reverse shifts, A^* and C^* can be computed quickly via simple modification of Function_A_C:

```
function [Astar, Cstar]=Function_Astar_Cstar(astar, c, k)
Astar=1, kbar=dec2bin(k), imax=length(kbar), h=astar
for i=imax:-1:1, if kbar(i), Astar=Astar*h, end, h=h*h, end
C=c*astar*(Astar -1)/(astar -1)
```

To illustrate, consider the PCG32 algorithm, using b=64 bit arithmetic. Applying Function_Astar_Cstar to develop a scheme to back up the PRNG stream k=200 steps results in imax=8, with 3 nonzero values of kbar, whereas applying Function_A_C with, as suggested in [5], a two's complement form of −k, given by 0xFFFFFFFFFFFFFF38 = 18446744073709551416, results in imax=64, with 59 nonzero values of kbar. It is seen that the new approach, leveraging Function_Astar_Cstar whenever k is negative, is much more computationally efficient. This difference is even more pronounced when considering the PCG64 algorithm, in which the minimal-length binary form of k is unchanged, but the two's complement form of −k for k=200 results in imax=128, with 123 nonzero values of kbar.

3 COMPARISONS OF COMPLEXITY, AND CONCLUSIONS

The 14 PRNG schemes summarized in Figure 2, and the reverse of each of these 14 schemes in Figure 3, are each provided in executable Matlab code in the PRNG section of our Renaissance Repository [1], verifying numerically that the latter exactly reverse the former.

Despite their name, LCGs [$x \leftarrow (a \cdot x + c) \bmod m$; see (1a)] are *affine*, not linear, in the state. The fact that LCGs/PCGs with $m = 2^b$ and odd c and odd $a = 8\,k + 5$ for some k are periodic and jumpable (and, thus, reversible), as reviewed in §1.2 and §1.4, is well known [5]. MWC generators with $r - 1$ intermediate "lags" [see (3a), with multiplier a and using mod m arithmetic where $m = 2^b$] are equivalent to Lehmer generators (i.e., LCGs that are actually linear, with $c = 0$) of the form $x \leftarrow (b \cdot x) \bmod p$ where $p = a\, b^r - 1$ [18], and are thus, by similar reasoning[5], also reversible. Further, the fact that $\mathbb{F}_2$-linear transformations, such as those used by the XOR/shift generators reviewed here, are periodic and jumpable (and, thus, reversible) is also well known [9, 14]. The focus of this paper is thus not on whether or not the inverses of these underlying transformations exist, but rather on how to calculate these inverses *efficiently*. We thus now compare the complexity of the forward and reverse schemes considered in this paper.

[5] Note that, when $b = 64$, p is odd, and thus their GCD is 1, and the modular inverse b^* exists, and can be computed via the machinery laid out in §2.1.

The XSH RR variant of the PCG32 scheme propagates forward with one 64-bit multiplication followed by one 64-bit addition, whereas the PCG32_rev scheme propagates this PRNG in reverse with one 64-bit subtraction followed by one 64-bit multiplication. These two schemes thus have identical computational cost.

The DXSM variant of the PCG64 scheme propagates forward with one 128-bit by 64-bit multiplication followed by one 128-bit addition, whereas the PCG64_rev scheme propagates this PRNG in reverse with one 128-bit subtraction followed by one 128-bit by 128-bit multiplication. If the CPU being used fully implements 128 bit unsigned integer multiplication, which today is uncommon, both of these schemes likely execute in about the same amount of time. However, if these 128 bit integer operations are being emulated in software using smaller 64-bit integer operations (addition and multiplication), the 128-bit by 128-bit multiplication step in the (reverse) PCG64_rev scheme will be about twice as expensive as the corresponding 128-bit by 64-bit multiplication step in the (forward) PCG64 scheme, and both PCG64 and PCG64_rev will be relatively slow as compared with the PCG32 and XOR/shift schemes listed in Figures 2 and 3.

The MWC schemes propagate forward with one 64-bit by 64-bit multiplication (generating both a 64-bit product and a 64-bit carry) followed by one 128-bit by 64-bit addition, plus $k-1$ lag operations (for $k=1$, 2, or 3), whereas the MWC_rev schemes propagate these PRNGs in reverse with one 128-bit by 128-bit division (generating both a 64-bit quotient x and a 64-bit remainder c), plus $k-1$ lag operations. If the CPU being used fully implements 128 bit unsigned integer multiplication and division, which today is uncommon, both of these schemes likely execute in about the same amount of time. However, if these 128 bit integer operations are being emulated in software (in particular, if 128-bit integer division needs to be emulated in software using something like the nonrestoring division algorithm reviewed in §2.2), the division operation in the (reverse) MWC_rev schemes will be *substantially* slower than the corresponding multiplication operation in the (forward) MWC schemes, and both the MWC and MWC_rev schemes will be relatively slow as compared with the PCG32 and XOR/shift schemes listed in Figures 2 and 3.

The (forward) xoshiro128, xoshiro256, and xoroshiro128 families of schemes are all very similar in computational complexity to their (reverse) xoshiro128_rev, xoshiro256_rev, and xoroshiro128_rev counterparts, and should thus execute in nearly the same amount of time. The only substantial difference is that the bitshift operations required to deduce s1 from r in the (reverse) xoshiro128_rev and xoshiro256_rev families of schemes need to be calculated in four distinct chunks, as shown in the Shift32 and Shift64 algorithms developed in §2.4, and thus these reverse schemes will be slightly slower than their corresponding (forward) xoshiro128 and xoshiro256 counterparts.

Finally, leveraging knowledge of a^*, a new algorithm for jumping LCGs and PCGs in reverse was proposed in §2.6. As discussed further there, this revised algorithm reduces the number of computations required for small reverse jumps, as compared to the algorithm proposed in [5], by an order of magnitude or more.

As an aside, also recall the "escape from zeroland" discussion in §1.3. The question of how many steps it takes for the state of a (linear) MWC or XOR/shift generator to move from some particular near-zero condition to a specified distance away from the origin may be addressed by forward PRNG marches. However, the questions of whether and how that particular problematical near-zero condition can actually be reached (given that good PRNG initialization schemes take specific steps to avoid zeroland, as suggested in §1.5) is best addressed by reverse PRNG marches. The latter significant question is distinct from the former, and is facilitated by the present work.

Overall, it is seen that the reverse PRNG schemes summarized in Figure 3 are structurally similar to the corresponding PCG, MWC, and XOR/shift families of modern PRNGs themselves, as summarized in Figure 2, and thus should execute at similar speeds (in certain cases, requiring 128-bit arithmetic for efficiency) when implemented appropriately at a low level. This should be valuable for the applications of interest in this paper, as described in the abstract.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge financial support of the ERDC Construction Engineering Research Laboratory (CERL) via a subcontract with GTI Energy.

REFERENCES

[1] Bewley, T (2024) *The Renaissance Repository*. Available at https://github.com/tbewley/RR/. See in particular the Renaissance Robotics directory, and the chap02/PRNGs and chap02/special_functions and chapAA/classes subdirectories contained therein.

[2] Blackman, D and Vigna, S (2021) Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.*, **47**: 1-32.

[3] Blackman, D and Vigna, S (2022) A new test for Hamming-weight dependencies. *ACM Trans. Model. Comput. Simul.*, **32** (3): 1-19.

[4] Box, G.E.P., and Muller, M.E. (1958) A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics.* **29** (2): 610?611.

[5] Brown, F.B. (1994) Random number generation with arbitrary strides. *Transactions of the American Nuclear Society* **71**, CONF-941102-(1994).

[6] Cessna, J (2010) *The Hybrid Ensemble Smoother (HEnS) & Noncartesian Computational Interconnects* PhD Thesis, Dept of MAE, UC San Diego.

[7] Doty-Humphrey, C. (2010) *Practically random: C++ library of statistical tests for rngs.* Available at https://pracrand.sourceforge.net/

[8] Finch, T (2023) *PCG64 DXSM random number generator.* Available at https://dotat.at/@/2023-06-21-pcg64-dxsm.html

[9] Haramoto, H, Matsumoto, M, Nishimura, T, Panneton, F, L'Ecuyer, P (2008) Efficient Jump Ahead for F2-Linear Random Number Generators, *INFORMS Journal on Computing* **20** (3): 385-390.

[10] Hull, T.E. and Dobell, A.R. (1962) Random Number Generators. *SIAM Review*, **4** (3): 230-254.

[11] Kaitchuck, T (2021) Designing a new PRNG. Available at https://tom-kaitchuck.medium.com/designing-a-new-prng-1c4ffd27124d.

[12] Knuth, D. (1968-2022) *The Art of Computer Programming, vol. 2*. Addison-Wesley.

[13] L'Ecuyer, P. (1999) Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation* **68** (225): 249-260.

[14] L?Ecuyer, P., Panneton, F. (2009). F2-Linear Random Number Generators. In: Alexopoulos, C., Goldsman, D., Wilson, J. (eds) *Advancing the Frontiers of Simulation.* International Series in Operations Research & Management Science, **133**. Springer.

[15] L'Ecuyer, P. and Simard, R. (2007) TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software* **33** (4): 22. Available at https://simul.iro.umontreal.ca/testu01/tu01.html

[16] Lehmer, D.H. (1951) Mathematical methods in large-scale computing units. *Proceedings of 2nd Symposium on Large-Scale Digital Calculating Machinery*, 141-146.

[17] Levy, S (2017) *Requiem for a shuffle.* Available at https://www.wired.com/story/requiem-for-the-ipod-shuffle/

[18] Marsaglia, G (2003). Random number generators. *Journal of Modern Applied Statistical Methods.* **2** (1): 2-13.

[19] Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*, **8** (14): 1-6.

[20] Occil, P (2023) *Notes on Jumping PRNGs Ahead.* Available at https://peteroupc.github.io/jump.pdf

[21] O'Neill, M.E. (2014) PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Harvey Mudd College technical report HMC-CS-2014-0905.

[22] O'Neill, M.E. (2018-2024) *PCG, A Family of Better Random Number Generators.* Available at https://www.pcg-random.org/

[23] O'Neill, M.E. (2018) *On Vigna's PCG Critique.* Available at https://www.pcg-random.org/posts/on-vignas-pcg-critique.html

[24] O'Neill, M.E. (2019) *PCG64 DXSM.*. Announced at https://github.com/numpy/numpy/issues/13635#issuecomment-506088698.

[25] Steele, G.L. and Vigna, S (2022) Computationally easy, spectrally good multipliers for congruential pseudorandom number generators. *Software: Practice and Experience*, **52** (2): 443-458.

[26] Thomson, W.E. (1958). A Modified Congruence Method of Generating Pseudo-random Numbers. *The Computer Journal* **1** (2): 83.

[27] Vigna, S (2018) *The wrap-up on PCG generators.* Available at https://pcg.di.unimi.it/pcg.php

[28] Vigna, S (2021-24) *xoshiro / xoroshiro generators and the PRNG shootout.* Available at https://prng.di.unimi.it/

[29] Warren, H (2012) *Hacker's Delight*, 2nd Edition. Addison-Wesley.

Received May 31, 2024