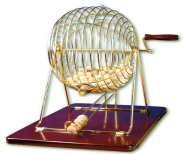
# Random number generation tools for parallel environments

**Pierre L'Ecuyer**

Université de Montréal




seed $\boldsymbol{x}_0$,
transition $\boldsymbol{x}_n = f(\boldsymbol{x}_{n-1})$,
output $u_n = g(\boldsymbol{x}_n)$

MODSIM 2025, Adelaide, Australia, November 2025

# We want sequences of numbers that look random

**Example: Bit sequence** (head or tail):


01111010011011010100110110010100011 1?...

Uniformity: Each bit is 1 with probability 1/2.

Uniformity and independence: For any $s$ bits, probability $1/2^s$ for each of the $2^s$ possibilities.

Common function types in RNG libraries: next64 or next32 return blocks of 64 or 32 bits.

# Uniform distribution over the interval $(0, 1)$

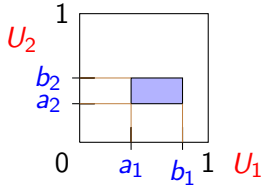
We want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$. We want $\mathbb{P}[a \leq U_j \leq b] = b - a$.

# Uniform distribution over the interval $(0, 1)$

We want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$. We want $\mathbb{P}[a \leq U_j \leq b] = b - a$.


Independence: For a random vector $\boldsymbol{U} = (U_1, \ldots, U_s)$, we want

$$\mathbb{P}[a_j \leq U_j \leq b_j \text{ for } j = 1, \ldots, s] = (b_1 - a_1) \cdots (b_s - a_s).$$

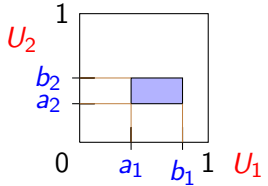


# Uniform distribution over the interval $(0, 1)$

We want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$. We want $\mathbb{P}[a \leq U_j \leq b] = b - a$.


Independence: For a random vector $\boldsymbol{U} = (U_1, \ldots, U_s)$, we want

$$\mathbb{P}[a_j \leq U_j \leq b_j \text{ for } j = 1, \ldots, s] = (b_1 - a_1) \cdots (b_s - a_s).$$




From independent random bits, one can easily approximate indep. $\mathcal{U}(0, 1)$ random variables.

For example: `double U = (next64() >> 11) * 2^-53`.

**Non-uniform variates:**
To generate $X$ such that $\mathbb{P}[X \leq x] = F(x)$:

$$X = F^{-1}(U_j) = \inf\{x : F(x) \geq U_j\}.$$

This is inversion.

Example: If $F(x) = 1 - e^{-\lambda x}$, take $X = [-\ln(1 - U_j)]/\lambda$.

Beware: It is usually important to avoid $U_j = 0$ or 1.

Also other methods such as rejection, etc., when $F^{-1}$ is costly to compute.

**Non-uniform variates:**
To generate $X$ such that $\mathbb{P}[X \leq x] = F(x)$:

$$X = F^{-1}(U_j) = \inf\{x : F(x) \geq U_j\}.$$

This is inversion.

Example: If $F(x) = 1 - e^{-\lambda x}$, take $X = [-\ln(1 - U_j)]/\lambda$.

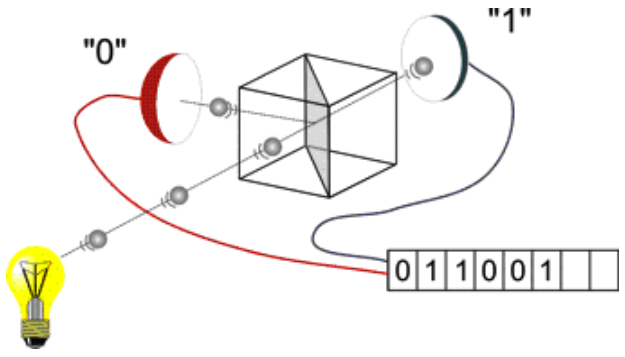
Beware: It is usually important to avoid $U_j = 0$ or 1.

Also other methods such as rejection, etc., when $F^{-1}$ is costly to compute.

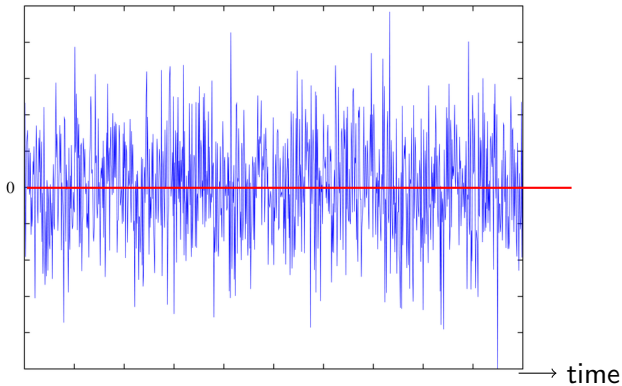
This notion of independent uniform random variables is only a mathematical abstraction. Perhaps it does not exist in the real world! We only wish to imitate it (approximately).

# Physical devices for computers

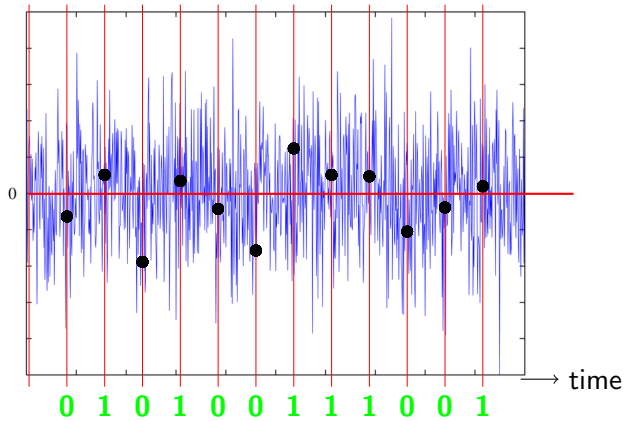
Photon trajectories (sold by **id-Quantique**):

# Thermal noise in resistances of electronic circuits



Thermal noise in resistances of electronic circuits



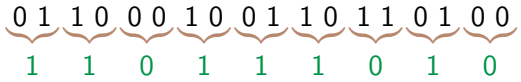

The signal is sampled periodically.

Several commercial devices on the market (and hundreds of patents!).

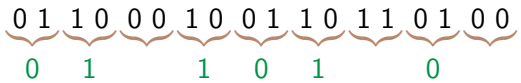
None is perfect.

Several commercial devices on the market (and hundreds of patents!).

None is perfect. Can reduce the bias and dependence by combining bits. E.g., with a XOR:

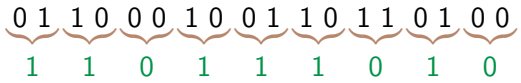



or (this eliminates the bias):

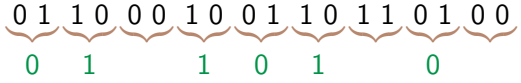


Several commercial devices on the market (and hundreds of patents!).

None is perfect. Can reduce the bias and dependence by combining bits. E.g., with a XOR:




or (this eliminates the bias):




Physical devices are essential for cryptology, lotteries, etc.

But for simulation, it is inconvenient, not always reliable, and benefits from no (or little) mathematical analysis. Much more important drawback: it is not reproducible.

# Reproducibility

Simulations are often required to be exactly replicable, and to always produce exactly the same results on different computers and architectures, sequential or parallel. Important for debugging and to replay exceptional events in more details, for better understanding.

Also essential when comparing systems with slightly different configurations or decision-making rules, by simulating them with common random numbers (CRNs). That is, to reduce the variance in comparisons, use the same random numbers at exactly the same places in all configurations of the system, as much as possible. Important for sensitivity analysis, derivative estimation, and effective stochastic optimization.

# Reproducibility

Simulations are often required to be exactly replicable, and to always produce exactly the same results on different computers and architectures, sequential or parallel. Important for debugging and to replay exceptional events in more details, for better understanding.

Also essential when comparing systems with slightly different configurations or decision-making rules, by simulating them with common random numbers (CRNs). That is, to reduce the variance in comparisons, use the same random numbers at exactly the same places in all configurations of the system, as much as possible. Important for sensitivity analysis, derivative estimation, and effective stochastic optimization.

**Algorithmic RNGs** permit one to replicate without storing the random numbers, which would be required for physical devices.

We will examine three types of algorithmic RNGs:
**(1) Recurrence-Based**, **(2) Counter-Based**, and **(3) Splittable**.

# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;

$f : \mathcal{S} \to \mathcal{S}$, transition function;

$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;

$s_0$, seed (initial state);

$s_0$

# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;

$f : \mathcal{S} \to \mathcal{S}$, transition function;

$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;

$s_0$, seed (initial state);

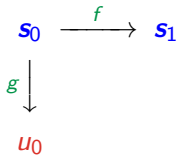
# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;

$f : \mathcal{S} \to \mathcal{S}$, transition function;

$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;

$s_0$, seed (initial state);



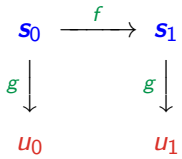
# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;

$f : \mathcal{S} \to \mathcal{S}$, transition function;

$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;

$s_0$, seed (initial state);



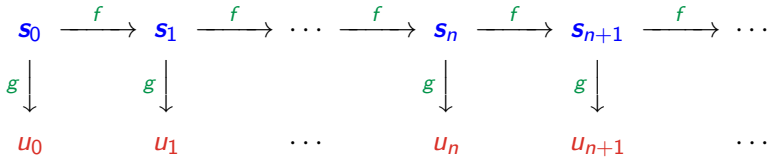
# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;

$f : \mathcal{S} \to \mathcal{S}$, transition function;

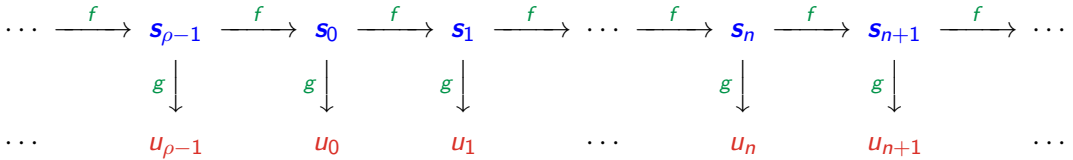
$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;

$s_0$, seed (initial state);



# Recurrence-based algorithmic generator

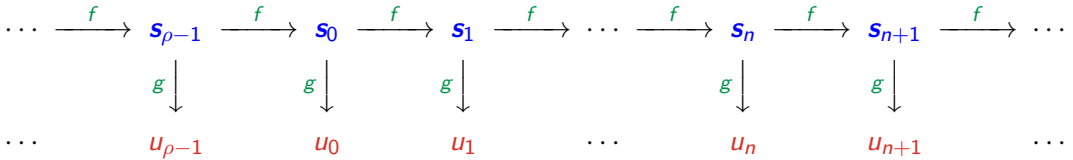
$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;
$s_0$, seed (initial state);




Period $\rho$ of $\{s_n,\ n \geq 0\}$ cannot exceed $|\mathcal{S}|$.

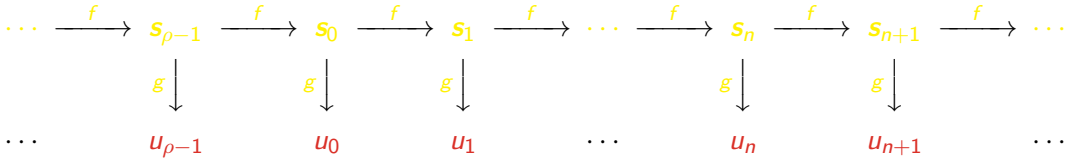
# Recurrence-based algorithmic generator

$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$ [or $g : \mathcal{S} \to \mathbb{Z}_m$], output function;
$s_0$, seed (initial state);

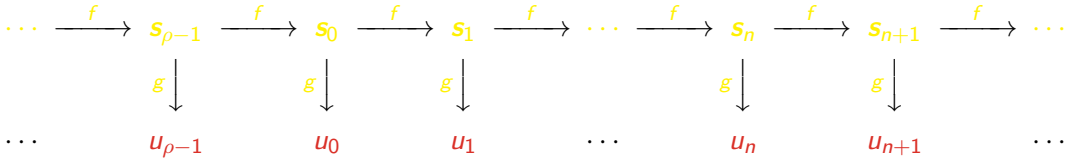



Period $\rho$ of $\{s_n,\ n \geq 0\}$ cannot exceed $|\mathcal{S}|$.

Classical setting: $f$ does a lot of work and $g$ does very little.
But it can be the opposite! For instance, $g$ can incorporate a hash function.
One could also have two (or more) transition functions $f_1$ and $f_2$, leading to a tree structure.




Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independent random variables over $(0, 1)$.



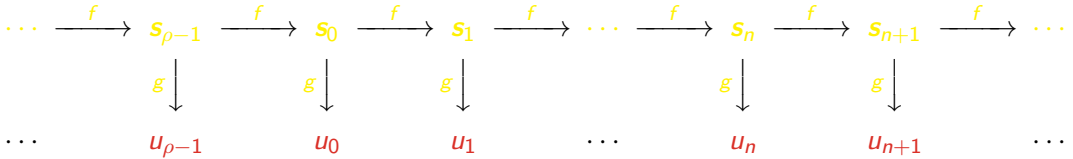

Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independent random variables over $(0, 1)$.

A long period is good, but far from sufficient!

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

Aside from empirical statistical tests, it is important to be able to measure the uniformity and independence by a mathematical analysis.



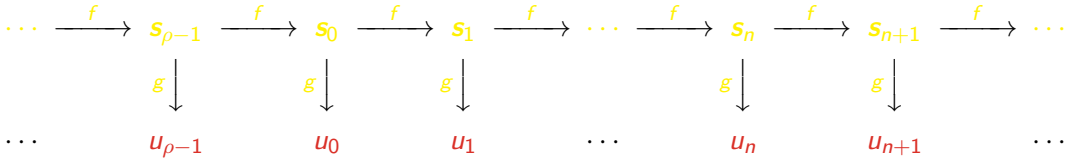

Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independent random variables over $(0, 1)$.

A long period is good, but far from sufficient!

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

Aside from empirical statistical tests, it is important to be able to measure the uniformity and independence by a mathematical analysis.




Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independent random variables over $(0, 1)$.

A long period is good, but far from sufficient!

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

Aside from empirical statistical tests, it is important to be able to measure the uniformity and independence by a mathematical analysis.

## Uniform distribution over $[0,1]^s$.

Choosing $\boldsymbol{s}_0$ randomly in $\mathcal{S}$ and generating $s$ numbers corresponds to picking a random point from the finite set

$$\Psi_s = \{\boldsymbol{u} = (u_0, \ldots, u_{s-1}) = (g(\boldsymbol{s}_0), \ldots, g(\boldsymbol{s}_{s-1})),\ \boldsymbol{s}_0 \in \mathcal{S}\}.$$

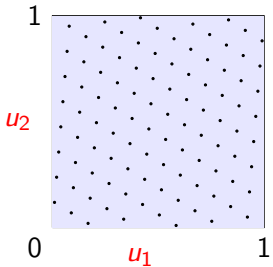
To approximate "$\boldsymbol{u}$ has the uniform distribution over $[0,1]^s$," $\Psi_s$ should cover $[0,1]^s$ very evenly.

## Uniform distribution over $[0,1]^s$.

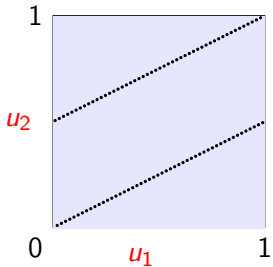
Choosing $\boldsymbol{s}_0$ randomly in $\mathcal{S}$ and generating $s$ numbers corresponds to picking a random point from the finite set

$$\Psi_s = \{\boldsymbol{u} = (u_0, \ldots, u_{s-1}) = (g(\boldsymbol{s}_0), \ldots, g(\boldsymbol{s}_{s-1})),\ \boldsymbol{s}_0 \in \mathcal{S}\}.$$

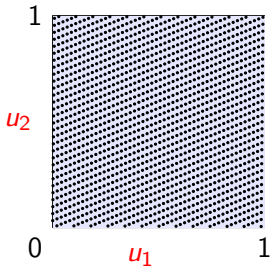
To approximate "$\boldsymbol{u}$ has the uniform distribution over $[0,1]^s$," $\Psi_s$ should cover $[0,1]^s$ very evenly. Baby illustration for $s = 2$: (Realistic case: $|\Psi_s| = 2^{256}$ or more)




$|\Psi_2| = 100$




$|\Psi_2| = 100$




$|\Psi_2| \approx 2000$

More general: can also consider non-successive indices $I = \{i_1, \ldots, i_s\}$:

$$\Psi_I = \{\boldsymbol{u} = (u_{i_1}, \ldots, u_{i_s}) = (g(\boldsymbol{s}_{i_1}), \ldots, g(\boldsymbol{s}_{i_s})),\ \boldsymbol{s}_0 \in \mathcal{S}\}.$$

Measure of quality (figure of merit): $\Psi_I$ must cover $[0,1]^s$ very evenly for a selected class $\mathcal{J}$ of index sets (projections) $I$.

RNG Design and selection:

1. Define a uniformity measure (figure of merit) that depend on $\{\Psi_I : I \in \mathcal{J}\}$, and which is computable without generating the points explicitly. Linear RNGs.
2. Choose a parameterized family (fast, long period, etc.)
   and search for parameters that "optimize" this measure.

# RNGs based on a linear recurrence

For most classical RNGs, the state $\boldsymbol{x}_n \in \mathbb{Z}_m^k$ obeys a linear recurrence

$$\boldsymbol{x}_n = \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod m,$$

for some integer $m$ (usually $m = 2$ or a large prime). Output at step $n$:

$$u_n = g(\boldsymbol{x}_n) \in (0, 1).$$

When $g$ is also linear mod $m$ in $\boldsymbol{x}_n$, $\Psi_s$ has a linear structure whose uniformity can be assessed mathematically by computable quantities. I view this as a big advantage. Computing these theoretical measures (when we can) is more important than empirical statistical testing.

Some constructions use a nonlinear $g$, but measuring the uniformity before the application of $g$ remains relevant.

# Linear multiple recursive generator (MRG)

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m, \qquad u_n = x_n/m.$$

# Linear multiple recursive generator (MRG)

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m, \qquad u_n = x_n/m.$$

State: $\boldsymbol{s}_n = \boldsymbol{x}_n = (x_{n-k+1}, \ldots, x_n)$,

$$\boldsymbol{x}_n = \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod m = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix} \boldsymbol{x}_{n-1} \bmod m.$$

Max. period: $\rho = m^k - 1$ if $m$ is prime.

# Linear multiple recursive generator (MRG)

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m, \qquad u_n = x_n/m.$$

State: $\boldsymbol{s}_n = \boldsymbol{x}_n = (x_{n-k+1}, \ldots, x_n)$,

$$\boldsymbol{x}_n = \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod m = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix} \boldsymbol{x}_{n-1} \bmod m.$$

Max. period: $\rho = m^k - 1$ if $m$ is prime.

Numerous variants and implementations.

For $k = 1$: classical linear congruential generator (LCG).

For an MRG, $\Psi_s$ is the intersection of a lattice with the unit cube. We can use this to measure the uniformity, as we saw on the pictures.

# Linear recurrences modulo 2

$$
\begin{array}{lcllll}
\boldsymbol{x}_n & = & \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod 2 & = & (x_{n,0}, \ldots, x_{n,k-1})^{\mathrm{t}}, & \text{(state, } k \text{ bits)} \\
\boldsymbol{y}_n & = & \boldsymbol{B}\,\boldsymbol{x}_n \bmod 2 & = & (y_{n,0}, \ldots, y_{n,w-1})^{\mathrm{t}}, & (w \text{ bits of output)} \\
u_n & = & \sum_{j=1}^{w} y_{n,j-1} 2^{-j} & = & .y_{n,0}\; y_{n,1}\; y_{n,2}\; \cdots, & \text{(output)}
\end{array}
$$

# Linear recurrences modulo 2

$$
\begin{array}{lcll}
\boldsymbol{x}_n & = & \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod 2 & = (x_{n,0}, \ldots, x_{n,k-1})^{\mathrm{t}}, \quad \text{(state, } k \text{ bits)} \\
\boldsymbol{y}_n & = & \boldsymbol{B}\,\boldsymbol{x}_n \bmod 2 & = (y_{n,0}, \ldots, y_{n,w-1})^{\mathrm{t}}, \quad (w \text{ bits of output}) \\
u_n & = & \sum_{j=1}^{w} y_{n,j-1} 2^{-j} & = .y_{n,0}\, y_{n,1}\, y_{n,2}\, \cdots, \quad \text{(output)}
\end{array}
$$

Clever choice of $\boldsymbol{A}$: transition via shifts, XOR, AND, masks, etc., on blocks of bits. Very fast.

Examples: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.
With small nonlinear output filter: xoshiro, xoroshiro, etc.

# Linear recurrences modulo 2

$$\begin{array}{rcccll} \boldsymbol{x}_n & = & \boldsymbol{A}\,\boldsymbol{x}_{n-1} \bmod 2 & = & (x_{n,0}, \ldots, x_{n,k-1})^{\mathrm{t}}, & \text{(state, } k \text{ bits)} \\ \boldsymbol{y}_n & = & \boldsymbol{B}\,\boldsymbol{x}_n \bmod 2 & = & (y_{n,0}, \ldots, y_{n,w-1})^{\mathrm{t}}, & (w \text{ bits of output)} \\ u_n & = & \sum_{j=1}^{w} y_{n,j-1} 2^{-j} & = & .y_{n,0}\, y_{n,1}\, y_{n,2}\, \cdots, & \text{(output)} \end{array}$$

Clever choice of $\boldsymbol{A}$: transition via shifts, XOR, AND, masks, etc., on blocks of bits. Very fast.

Examples: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.
With small nonlinear output filter: xoshiro, xoroshiro, etc.

Each coordinate of $\boldsymbol{x}_n$ and of $\boldsymbol{y}_n$ follows the linear recurrence with characteristic polynomial

$$P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(\boldsymbol{A} - z\boldsymbol{I}).$$

Max. period: $\rho = 2^k - 1$ reached iff $P(z)$ is a primitive polynomial.

## Examples of fast transitions $x_n = A\,x_{n-1} \bmod 2$

32-bit LFSR (L 1999): State x changes from $x_{n-1}$ to $x_n$.

```
uint32_t z = (x << 6) ^ (x >> 13);
x = (x >> 1) << 1) << 18) ^ z;
```

## Examples of fast transitions $x_n = A\,x_{n-1} \bmod 2$

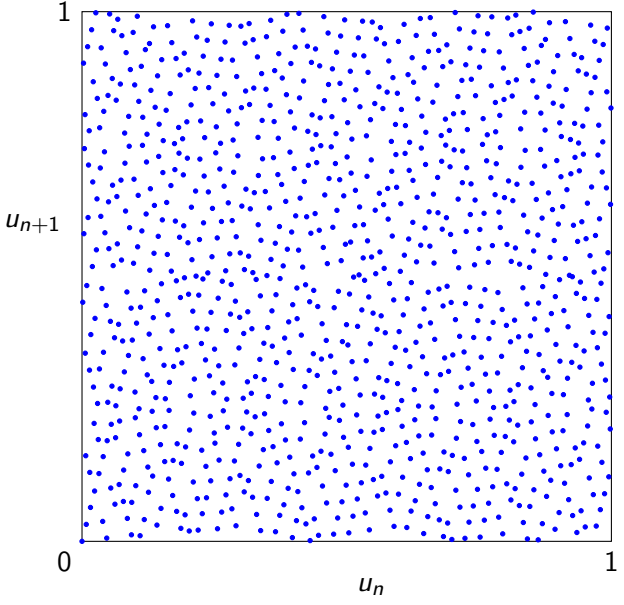
32-bit LFSR (L 1999): State x changes from $x_{n-1}$ to $x_n$.

```
uint32_t z = (x << 6) ^ (x >> 13);
x = (x >> 1) << 1) << 18) ^ z;
```

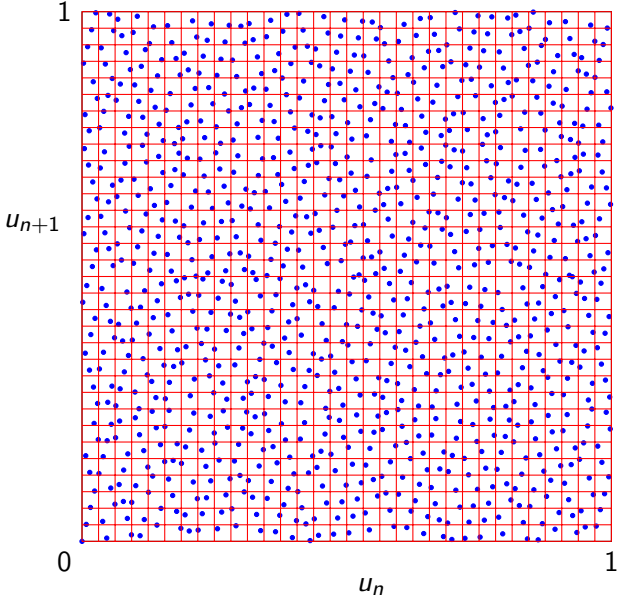
64-bit xoroshiro128+ (Blackman and Vigna 2021):
State s[0], s[1]: two 64-bit unsigned integers.

```
uint64_t s0 = s[0];
uint64_t s1 = s[1];
s1 ^= s0;
s[0] = (s0 << 24) | (s0 >> 40) ^ s1 ^ (s1 << 16);
s[1] = (s1 << 37) | (s1 >> 27);
```

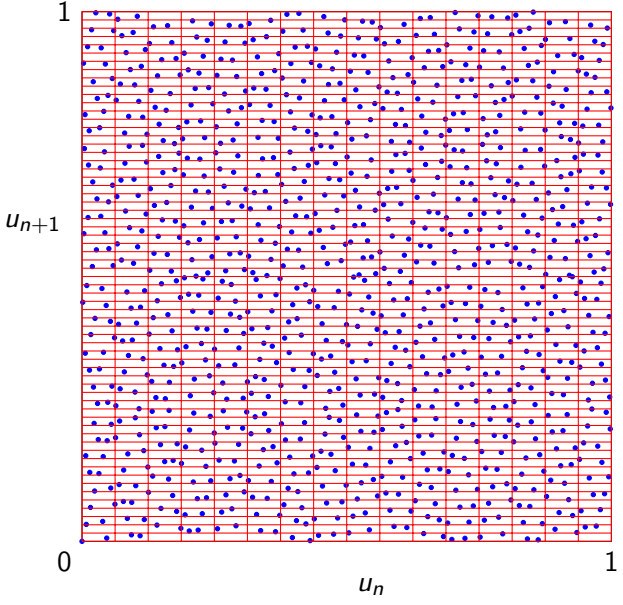
# Measures of uniformity. Example: $k = 10$, $2^{10} = 1024$ points



# Measures of uniformity. Example: $k = 10$, $2^{10} = 1024$ points



# Measures of uniformity. Example: $k = 10$, $2^{10} = 1024$ points



# A single RNG does not suffice.

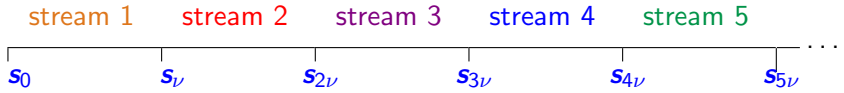
One often needs several independent streams of random numbers, for example:

1. To run simulations on parallel processors.
2. To compare systems with well synchronized common random numbers (CRNs). Can be complicated to implement and manage when different configurations do not need the same number of $U_j$'s.

## RNG with multiple streams

From a single RNG, one can create multiple "`random stream`" **objects** that behave as "independent" virtual RNGs (or streams of random numbers).

Simple approach: partition the entire sequence into disjoint segments (streams) of length $\nu$.




Jumping ahead by $\nu$ steps is easy when $f$ is linear:

$$\boldsymbol{x}_{n+1} \;=\; f(\boldsymbol{x}_n) \;=\; \boldsymbol{A}\boldsymbol{x}_n \bmod m$$

where the state $\boldsymbol{x}_n$ is a vector and $\boldsymbol{A}$ a matrix. Then

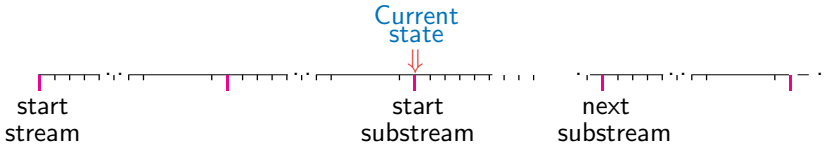
$$\boldsymbol{x}_{n+\nu} \;=\; (\boldsymbol{A}^\nu \bmod m)\boldsymbol{x}_n \bmod m$$

with the matrix $(\boldsymbol{A}^\nu \bmod m)$ precomputed once for all.

## RNG with multiple streams and substreams

The **RngStreams** software (L et al. 2000) offers an implementation with multiple streams and substreams. The streams are further partitioned in substreams (which are not objects).

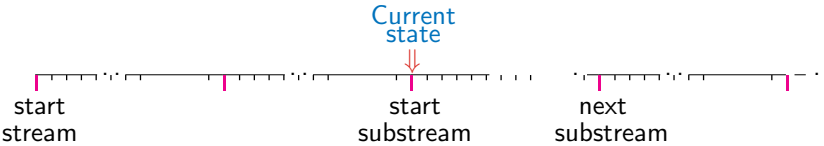
One stream:



## RNG with multiple streams and substreams

The **RngStreams** software (L et al. 2000) offers an implementation with multiple streams and substreams. The streams are further partitioned in substreams (which are not objects).

One stream:




Original RngStreams was based on the MRG32k3a generator, with period $\approx 2^{191}$.
Streams start $\nu = 2^{127}$ values apart and substreams have length $2^{76}$.
Has been implemented in C, C++, FORTRAN, Java, R, Cuda, etc. In C:

```
RngStream stream1 = createStream ();
double u = randU01 (stream1);
int i = randInt (stream1, 1, 6);

ResetStartSubstream (stream1);
ResetNextSubstream (stream1);
ResetStartStream (stream1);
```

# Comparing systems with common random numbers: a simple inventory example

$X_j$ = inventory level in morning of day $j$;
$D_j$ = **random demand** on day $j$;
Orders follow a $(s, S)$ policy: If $Y_j < s$, order $S - Y_j$ items.
Each order **arrives (random)** for next morning with probability $p$.
Revenue for day $j$: sales − inventory costs − order costs.

Goal: Simulate $m$ days of operation, repeat $n$ times independently ($n$ runs), for several choices of $(s, S)$ to find the best one.

Want one RandomStream for each usage, one substream for each simulation run.
Want same streams and substreams for all policies $(s, S)$.

## Comparing $p$ policies with CRNs (using a single processor)

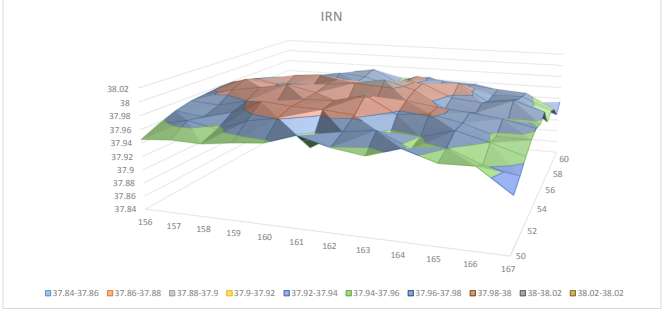
```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
RngStream* stream_demand = CreateStream();
RngStream* stream_order  = CreateStream();
for (int k = 0; k < p; k++) {   // for each policy
   for (int i = 0; i < n; i++) {    // perform n runs
      profit[k, i] = inventorySimulateOneRun (m, s[k], S[k], stream_demand, stream_order);
      // Realign starting points to use same substreams for all policies
      ResetNextSubstream (stream_demand);
      ResetNextSubstream (stream_order);
   }
   ResetStartStream (stream_demand);
   ResetStartStream (stream_order);
}

// Print and plot results ...
   ...
```

Only two streams suffice for the entire simulation experiment. If we use different streams for the $n$ different runs, we would need $2n$ stream objects instead. Would be less efficient.

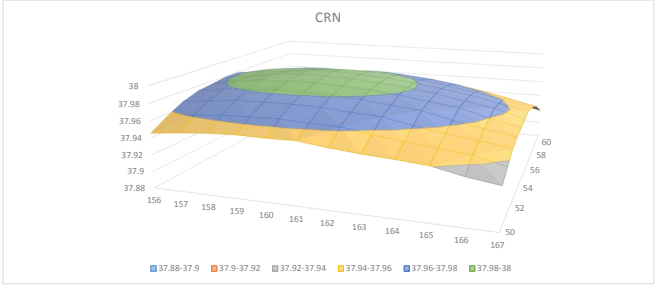
# Comparison with independent random numbers

| | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 37.94537 | 37.94888 | 37.94736 | 37.95314 | 37.95718 | 37.97194 | 37.95955 | 37.95281 | 37.96711 | 37.95221 | 37.95325 | 37.92063 |
| 51 | 37.9574 | 37.9665 | 37.95732 | 37.97337 | 37.98137 | 37.94273 | 37.96965 | 37.97573 | 37.95425 | 37.96074 | 37.94185 | 37.93139 |
| 52 | 37.96725 | 37.96166 | 37.97192 | 37.99236 | 37.98856 | 37.98708 | 37.98266 | 37.94671 | 37.95961 | 37.97238 | 37.95982 | 37.94465 |
| 53 | 37.97356 | 37.96999 | 37.97977 | 37.97611 | 37.98929 | 37.99089 | 38.00219 | 37.97693 | 37.98191 | 37.97217 | 37.95713 | 37.95575 |
| 54 | 37.97593 | 37.9852 | 37.99233 | 38.00043 | 37.99056 | 37.9744 | 37.98008 | 37.98817 | 37.98168 | 37.97703 | 37.97145 | 37.96138 |
| 55 | 37.97865 | 37.9946 | 37.97297 | 37.98383 | 37.99527 | 38.00068 | **38.00826** | 37.99519 | 37.96897 | 37.96675 | 37.9577 | 37.95672 |
| 56 | 37.97871 | 37.9867 | 37.97672 | 37.9744 | 37.9955 | 37.9712 | 37.96967 | 37.99717 | 37.97736 | 37.97275 | 37.97968 | 37.96523 |
| 57 | 37.97414 | 37.97797 | 37.98816 | 37.99192 | 37.9678 | 37.98415 | 37.97774 | 37.97844 | 37.99203 | 37.96531 | 37.97226 | 37.93934 |
| 58 | 37.96869 | 37.97435 | 37.9625 | 37.96581 | 37.97331 | 37.95655 | 37.98382 | 37.97144 | 37.97409 | 37.96631 | 37.96764 | 37.94759 |
| 59 | 37.95772 | 37.94725 | 37.9711 | 37.97905 | 37.97504 | 37.96237 | 37.98182 | 37.97656 | 37.97212 | 37.96762 | 37.96429 | 37.93976 |
| 60 | 37.94434 | 37.95081 | 37.94275 | 37.95515 | 37.98134 | 37.95863 | 37.96581 | 37.95548 | 37.96573 | 37.93949 | 37.93839 | 37.9203 |
| 61 | 37.922 | 37.93006 | 37.92656 | 37.93281 | 37.94999 | 37.95799 | 37.96368 | 37.94849 | 37.954 | 37.92439 | 37.90535 | 37.93375 |



# Comparison with common random numbers

| | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 37.94537 | 37.94888 | 37.95166 | 37.95319 | 37.95274 | 37.95318 | 37.94887 | 37.94584 | 37.94361 | 37.94074 | 37.93335 | 37.92832 |
| 51 | 37.9574 | 37.96169 | 37.96379 | 37.96524 | 37.96546 | 37.96379 | 37.96293 | 37.95726 | 37.95295 | 37.94944 | 37.94536 | 37.93685 |
| 52 | 37.96725 | 37.97117 | 37.97402 | 37.97476 | 37.97492 | 37.97387 | 37.971 | 37.96879 | 37.96184 | 37.95627 | 37.95154 | 37.94626 |
| 53 | 37.97356 | 37.97852 | 37.98098 | 37.98243 | 37.98187 | 37.98079 | 37.97848 | 37.97436 | 37.97088 | 37.96268 | 37.95589 | 37.94995 |
| 54 | 37.97593 | 37.98241 | 37.98589 | 37.98692 | 37.98703 | 37.98522 | 37.9829 | 37.97931 | 37.97397 | 37.96925 | 37.95986 | 37.95186 |
| 55 | 37.97865 | 37.98235 | 37.9874 | 37.9894 | **37.98909** | 37.9879 | 37.98483 | 37.98125 | 37.97641 | 37.96992 | 37.96401 | 37.95343 |
| 56 | 37.97871 | 37.98269 | 37.98494 | 37.98857 | 37.98917 | 37.98757 | 37.98507 | 37.98073 | 37.97594 | 37.96989 | 37.96227 | 37.95519 |
| 57 | 37.97414 | 37.98035 | 37.98293 | 37.98377 | 37.98603 | 37.98528 | 37.98239 | 37.97858 | 37.97299 | 37.96703 | 37.95981 | 37.95107 |
| 58 | 37.96869 | 37.97207 | 37.97825 | 37.97944 | 37.97895 | 37.97987 | 37.97776 | 37.97358 | 37.96848 | 37.9617 | 37.95461 | 37.94622 |
| 59 | 37.95772 | 37.96302 | 37.9663 | 37.97245 | 37.97234 | 37.97055 | 37.9701 | 37.96664 | 37.96122 | 37.95487 | 37.94695 | 37.93871 |
| 60 | 37.94434 | 37.94861 | 37.95371 | 37.95691 | 37.96309 | 37.96167 | 37.9586 | 37.95678 | 37.95202 | 37.9454 | 37.93785 | 37.92875 |
| 61 | 37.922 | 37.93169 | 37.93591 | 37.94085 | 37.94401 | 37.95021 | 37.94751 | 37.94312 | 37.94 | 37.93398 | 37.92621 | 37.91742 |



## Larger and more complicated systems

May require thousands of different streams, even for a simulation on a single CPU.

Substreams can be used for the independent replications, as we just saw. Very convenient.

My students have used that a lot for simulation and optimization of service systems such as call centers, reliability models, and also financial contracts and systems.

One may also think of factories, transportation networks, logistic systems, supply chains, etc.

# Using multiple streams on parallel processors

We may need many streams on each processor, not only one per processor.

Single RNG with equally-spaced starting points for streams and for substreams:
Recommended when possible. The streams are usually created in succession, one after the other. This creation process is inherently sequential.
For highly-parallel computations, we would prefer to create the streams in parallel.

# Using multiple streams on parallel processors

We may need many streams on each processor, not only one per processor.

Single RNG with equally-spaced starting points for streams and for substreams:
Recommended when possible. The streams are usually created in succession, one after the other. This creation process is inherently sequential.
For highly-parallel computations, we would prefer to create the streams in parallel.

Can be done by picking a random seed for each stream. Or use a counter-based generator that returns a pseudorandom seed for each stream number. Risk of overlap!

For period length $\rho$ and $s$ streams (or substreams) of length $\ell$,
$\mathbb{P}[\text{overlap somewhere}] = P_o \leq s^2\ell/\rho$.

| $\rho$ | $s$ | $\ell$ | $P_o \leq$ |
|---|---|---|---|
| $2^{64}$ | $2^{20}$ | $2^{20}$ | $2^{-4}$ |
| $2^{128}$ | $2^{20}$ | $2^{20}$ | $2^{-68}$ |
| $2^{128}$ | $2^{30}$ | $2^{30}$ | $2^{-38}$ |
| $2^{256}$ | $2^{30}$ | $2^{30}$ | $2^{-166}$ |

**Example:** Tensor Processing Units (TPUs) at Google use the `xorshift128+` RNG from Vigna (2016), based on a 128-bit $\mathbb{F}_2$-linear recurrence with an ordinary addition at the output. Each TPU chip can run 64 copies in parallel, implemented in hardware.

To determine the 64 initial states, the user provides two 64-bit integers which are used to transform the 32-bit numbers $1, 2, \ldots, 128$ in some elaborate (deterministic) manner to obtain the 64 128-bit seeds.

A naive idea could be to just apply one round of `xorshift128+` to the successive integers, but this bring insufficient dispersion. Google software does more, to be on the safe side.

# Counter-Based RNGs (CBRNGs)

In traditional RNGs, the transition function $f$ does most of the transformation and the output function $g$ is very simple. CBRNGs do the opposite: $f$ just increases a counter by 1 and $g$ does most of the work.

# Counter-Based RNGs (CBRNGs)

In traditional RNGs, the transition function $f$ does most of the transformation and the output function $g$ is very simple. CBRNGs do the opposite: $f$ just increases a counter by 1 and $g$ does most of the work.

In most versions, the state is a pair of integers (key, counter) $= (k, i) \in \mathbb{Z}_2^{m+c}$.
The key $k$ is a $m$-bit integer and the counter $i$ is a $c$-bit integer.
Transition function: $f(k, i) = (k, i + 1 \bmod 2^c)$. Jumping ahead is trivial.
Each value of $k$ may correspond to a stream, starting at $(k, 0)$.

The counter can also be multidimensional.
Can be seen as a huge direct-access table of random numbers!

# Counter-Based RNGs (CBRNGs)

In traditional RNGs, the transition function $f$ does most of the transformation and the output function $g$ is very simple. CBRNGs do the opposite: $f$ just increases a counter by 1 and $g$ does most of the work.

In most versions, the state is a pair of integers (key, counter) $= (k, i) \in \mathbb{Z}_2^{m+c}$.
The key $k$ is a $m$-bit integer and the counter $i$ is a $c$-bit integer.
Transition function: $f(k, i) = (k, i + 1 \bmod 2^c)$. Jumping ahead is trivial.
Each value of $k$ may correspond to a stream, starting at $(k, 0)$.

The counter can also be multidimensional.
Can be seen as a huge direct-access table of random numbers!

The keys $k$ for the successive streams can be defined in many ways:
(1) randomly (not reproducible);
(2) using a key schedule determined by a second RNG (inherently sequential);
(3) just use $k = 0, 1, 2, \ldots$ for the successive keys;
(4) let the user select the keys.
One can also use a second CBRNG to "hash" these keys into new keys (parallel-friendly).

## Collisions between keys

If $s$ random keys are drawn uniformly and independently among the $2^m$ possibilities, the probability that they are not all distinct (at least one collision) is $\approx s^2/2^{m+1}$.
For $s = 2^{20}$ and $m = 64$, this is approx. $2^{-25}$.

## Collisions between keys

If $s$ random keys are drawn uniformly and independently among the $2^m$ possibilities, the probability that they are not all distinct (at least one collision) is $\approx s^2/2^{m+1}$.
For $s = 2^{20}$ and $m = 64$, this is approx. $2^{-25}$.

## Dispersion requirement

In case (3), the output function $g$ must apply enough transformation so that there is no easily detectable similarity between $g(k, i)$ and $g(k, i+1)$ or $g(k+1, i)$. This requirement could make the CBRNG slower than a fast recurrence-based RNG.

**CBRNGs** were introduced as one mode of operation (the counter mode) for block ciphers in cryptography. They were included in the advanced encryption standard (**AES**) in 2001.

Hellekalek and Wegenkittl (2003) proposed and tested AES in counter mode for simulation. They found no statistical weakness. But AES is slow, unless implemented in hardware.

**CBRNGs** were introduced as one mode of operation (the counter mode) for block ciphers in cryptography. They were included in the advanced encryption standard (**AES**) in 2001.

Hellekalek and Wegenkittl (2003) proposed and tested AES in counter mode for simulation. They found no statistical weakness. But AES is slow, unless implemented in hardware.

Salmon et al. (2011) proposed faster (simplified) block ciphers named **ARS**, **Threefry**, and **Philox**, selected based on empirical statistical testing, to be used as CBRNGs.
Philox is fastest on GPUs while Threefry is fastest on CPUs.

**CBRNGs** were introduced as one mode of operation (the counter mode) for block ciphers in cryptography. They were included in the advanced encryption standard (**AES**) in 2001.

Hellekalek and Wegenkittl (2003) proposed and tested AES in counter mode for simulation. They found no statistical weakness. But AES is slow, unless implemented in hardware.

Salmon et al. (2011) proposed faster (simplified) block ciphers named **ARS**, **Threefry**, and **Philox**, selected based on empirical statistical testing, to be used as CBRNGs.
Philox is fastest on GPUs while Threefry is fastest on CPUs.

On GPUs, **Tensorflow** uses `Philox-4×32-10`, with $m = 64$ and $c = 128$, implemented in the `C++` class `PhiloxRandom`. The function $g$ makes 10 rounds of bijective transformations to the counter $i$, parameterized by the key $k$, and returns four 32-bit integers as output.
For each $k$, each 128-bit output appears once when the counter goes from 0 to $2^{128} - 1$.

**CBRNGs** were introduced as one mode of operation (the counter mode) for block ciphers in cryptography. They were included in the advanced encryption standard (**AES**) in 2001.

Hellekalek and Wegenkittl (2003) proposed and tested AES in counter mode for simulation. They found no statistical weakness. But AES is slow, unless implemented in hardware.

Salmon et al. (2011) proposed faster (simplified) block ciphers named **ARS**, **Threefry**, and **Philox**, selected based on empirical statistical testing, to be used as CBRNGs.
Philox is fastest on GPUs while Threefry is fastest on CPUs.

On GPUs, **Tensorflow** uses `Philox-4×32-10`, with $m = 64$ and $c = 128$, implemented in the C++ class `PhiloxRandom`. The function $g$ makes 10 rounds of bijective transformations to the counter $i$, parameterized by the key $k$, and returns four 32-bit integers as output.
For each $k$, each 128-bit output appears once when the counter goes from 0 to $2^{128} - 1$.

One small weakness: The 10 rounds must be done sequentially (cannot be parallelized).
I believe that there are faster good alternatives, based on more standard RNGs.

# Splittable RNGs

Sometimes, we want to be able to split dynamically (and unpredictably) any stream in two or more streams during execution.
That is, we have a random tree of streams. Can also be a tree of states.

Example: in particle physics (or computer graphics), we would use one stream for each particle (or ray of light). When a particle splits, we need to split the stream. Using a central monitor for this is not acceptable: too slow and not fully reproducible, because the order in which splits occur would depend on the relative speed of processors.

We want an efficient hardware-independent and fully reproducible solution.

# Splittable RNGs

Sometimes, we want to be able to split dynamically (and unpredictably) any stream in two or more streams during execution.
That is, we have a random tree of streams. Can also be a tree of states.

Example: in particle physics (or computer graphics), we would use one stream for each particle (or ray of light). When a particle splits, we need to split the stream. Using a central monitor for this is not acceptable: too slow and not fully reproducible, because the order in which splits occur would depend on the relative speed of processors.

We want an efficient hardware-independent and fully reproducible solution.

An algorithmic RNG can cover this as follows. Instead of a single transition function $f$, define two transitions functions $f_1$ and $f_2$ (or $f_1, \ldots, f_d$ if we allow $d$-fold splitting for $d > 2$).
When in state $s$, if we split the stream in two, the two new states will be $f_1(s)$ and $f_2(s)$.

Claessen and Palka (2013) designed a splittable RNG defined by a binary tree. Any node at level $\ell$ is identified by a $\ell$-bit string that represents the path to that node: When going to the next level, we add a 0 when going left and a 1 when going right. This gives an infinite binary tree in which each node has a distinct label. See picture.

One problem: these labels can grow too long. Solution: hash (compress) them periodically into shorter $b$-bit strings, e.g., each time the string reaches $2b$ bits, say for $b = 64$. For this, they apply a block cipher that takes the first $b$ bits as a key, the next $b$ bits as input, and outputs a new $b$-bit block which is the next key.
They also use the same block cipher as an output function.

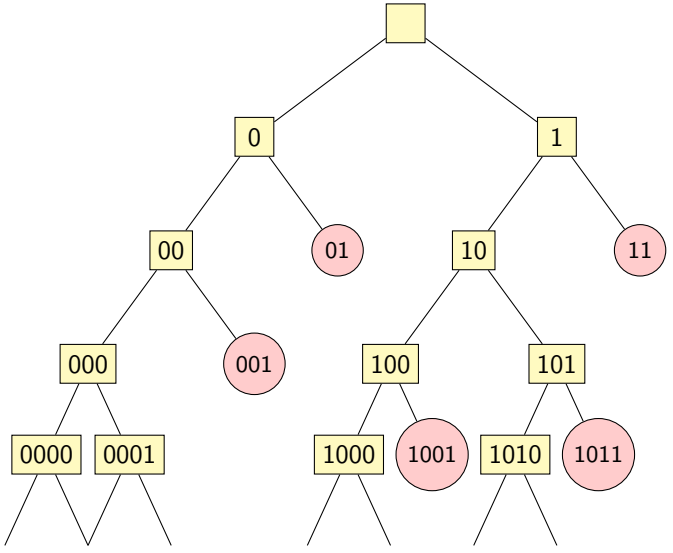
In their design, a tree node can either produce an output or create a split, but not both.

Claessen and Palka (2013) designed a splittable RNG defined by a binary tree. Any node at level $\ell$ is identified by a $\ell$-bit string that represents the path to that node: When going to the next level, we add a 0 when going left and a 1 when going right. This gives an infinite binary tree in which each node has a distinct label. See picture.

One problem: these labels can grow too long. Solution: hash (compress) them periodically into shorter $b$-bit strings, e.g., each time the string reaches $2b$ bits, say for $b = 64$. For this, they apply a block cipher that takes the first $b$ bits as a key, the next $b$ bits as input, and outputs a new $b$-bit block which is the next key.
They also use the same block cipher as an output function.

In their design, a tree node can either produce an output or create a split, but not both.

This scheme is implemented in **JAX**, using `Threefry-2×32-20` as a hash function, and $b = 64$. For each node, one can either call the "`split`" function to make a split, or the "`rand`" function to generate a (large) tensor of random numbers.
For a tensor of size $n$, the high 64 bits are used as the key and a counter goes from 0 to $n - 1$.
This is essentially as fast as using `Threefry` as an RNG.


0
1
00
01
10
11
000
001
100
101
0000
0001
1000
1001
1010
1011

# Multiply-with-Carry (MWC) RNGs

Full-period MRGs must have a prime modulus $m$. But operations modulo a large prime number are slow. Can be much faster if $m$ is a power of 2, such as $2^{64}$ or $2^{32}$, but then the MRG cannot have a long period and a good structure.

MWC generators solve this problem by adding a carry to the recurrence.
MWC of order $k$ in base $b = 2^e$ has state $(x_{n-k+1}, \ldots, x_n, c_n)$ and recurrence:

$$\begin{aligned} \tau_n &= a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}, \\ x_n &= \tau_n \bmod b, \\ c_n &= \tau_n \operatorname{div} b. \qquad \text{// Integer division} \end{aligned}$$

# Multiply-with-Carry (MWC) RNGs

Full-period MRGs must have a prime modulus $m$. But operations modulo a large prime number are slow. Can be much faster if $m$ is a power of 2, such as $2^{64}$ or $2^{32}$, but then the MRG cannot have a long period and a good structure.

MWC generators solve this problem by adding a carry to the recurrence.
MWC of order $k$ in base $b = 2^e$ has state $(x_{n-k+1}, \ldots, x_n, c_n)$ and recurrence:

$$\begin{aligned} \tau_n &= a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}, \\ x_n &= \tau_n \bmod b, \\ c_n &= \tau_n \text{ div } b. \qquad \text{// Integer division} \end{aligned}$$

Period can be almost $b^{k+1}/2$. Example: if $k = 3$ and $b = 2^{64}$, period $\rho \approx 2^{255}$. C Code:

```
uint64 x1, x2, x3, c;
uint128 tau = a1*(uint128)x1 + a2*(uint128)x2 + a3*(uint128)x3 + c;
x3 = x2;   x2 = x1;   x1 = tau;   c = tau >> 64;
```

Goresky and Klapper (2003): More general version with odd $a_0 < -1$; period can be almost $b^{k+1}$, but slower and $b$ cannot be an even power of 2:

$$\begin{aligned} c_n b - a_0 x_n &= \tau_n \stackrel{\text{def}}{=} a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}, \\ x_n &= (-a_0)^* \tau_n \bmod b, \\ c_n &= (\tau_n + a_0 x_n) \operatorname{div} b. \end{aligned}$$

To construct a real number in $[0, 1)$:

$$u_n = x_n \, b^{-\ell} \quad \text{or more generally} \quad u_n = \sum_{\ell=1}^{L} x_{n+1-\ell} \, b^{-\ell}$$

for some integer $L \geq 1$. Our mathematical analysis assumes $L = \infty$, but in practice, if $b = 2^{64}$ and $u_n$ is in "`double`", taking $L = 1$ is practically equivalent to $L = \infty$ since it already gives 64 bits for $u_n$. For a "`double`", we only want 53 bits for $u_n$.

# Equivalence between MWC and an LCG with large modulus $m$

$$\text{Let} \qquad m = \sum_{j=0}^{k} a_j b^j$$

where $a_0 = -1$ in the simplified case, and let $b^*$ be the multiplicative inverse of $b$ modulo $m$. It was shown by Couture, L, Tezuka (1993, 1997) that the MWC sequence $\{u_n,\ n \geq 0\}$ is the same as that produced by an LCG with modulus $m$ and multiplier $b^*$:

$$y_n = b^* y_{n-1} \bmod m \qquad \text{and} \qquad \tilde{u}_n = y_n/m.$$

Same LCG sequence in reverse order:

$$y_n = b y_{n+1} \bmod m \qquad \text{and} \qquad \tilde{u}_n = y_n/m.$$

# Equivalence between MWC and an LCG with large modulus $m$

$$\text{Let} \qquad m = \sum_{j=0}^{k} a_j b^j$$

where $a_0 = -1$ in the simplified case, and let $b^*$ be the multiplicative inverse of $b$ modulo $m$. It was shown by Couture, L, Tezuka (1993, 1997) that the MWC sequence $\{u_n,\ n \geq 0\}$ is the same as that produced by an LCG with modulus $m$ and multiplier $b^*$:

$$y_n = b^* y_{n-1} \bmod m \qquad \text{and} \qquad \tilde{u}_n = y_n/m.$$

Same LCG sequence in reverse order:

$$y_n = b y_{n+1} \bmod m \qquad \text{and} \qquad \tilde{u}_n = y_n/m.$$

Thus, the MWC is a clever way to implement a LCG with huge modulus.

One-to-one mapping $\phi : \mathbb{S} \to \mathbb{Z}_m$ between MWC state $\boldsymbol{x} = (x_0, \ldots, x_{k-1}, c)^{\mathsf{t}}$ and corresponding LCG state $y$:

$$y \ = \ \phi(\boldsymbol{x}) \ = \ cb^k - \sum_{j=0}^{k-1} b^j \sum_{i=0}^{j} a_i x_{j-i}.$$

# LCG representation is useful for:

1. Period length: smallest $\rho > 1$ such that $b^\rho \bmod m = 1$. Must be a divisor of $m - 1$.
Max possible period in general is $m - 1$, if $m$ is a prime number.
When $a_0 = -1$, max period is $(m - 1)/2$, achieved when both $m$ and $(m - 1)/2$ are prime.

# LCG representation is useful for:

1. Period length: smallest $\rho > 1$ such that $b^\rho \bmod m = 1$. Must be a divisor of $m - 1$.
Max possible period in general is $m - 1$, if $m$ is a prime number.
When $a_0 = -1$, max period is $(m - 1)/2$, achieved when both $m$ and $(m - 1)/2$ are prime.

2. To jump ahead or back by $\nu$ steps for the LCG:

$$y_{n-\nu} = (b^\nu \bmod m) y_n \bmod m.$$

This requires computations with large integers. For the MWC:

$$y_n = \phi(\boldsymbol{x}_n); \qquad y_{n-\nu} = (b^\nu \bmod m) y_n \bmod m; \qquad \boldsymbol{x}_{n-\nu} = \phi^{-1}(y_{n-\nu}).$$

# LCG representation is useful for:

1. Period length: smallest $\rho > 1$ such that $b^\rho \bmod m = 1$. Must be a divisor of $m - 1$.
Max possible period in general is $m - 1$, if $m$ is a prime number.
When $a_0 = -1$, max period is $(m - 1)/2$, achieved when both $m$ and $(m - 1)/2$ are prime.

2. To jump ahead or back by $\nu$ steps for the LCG:

$$y_{n-\nu} = (b^\nu \bmod m) y_n \bmod m.$$

This requires computations with large integers. For the MWC:

$$y_n = \phi(\boldsymbol{x}_n); \qquad y_{n-\nu} = (b^\nu \bmod m) y_n \bmod m; \qquad \boldsymbol{x}_{n-\nu} = \phi^{-1}(y_{n-\nu}).$$

3. To measure the multivariate uniformity of the MWC: examine the lattice structure of the corresponding LCG.

# Lattice structure

For an LCG, each set $\Psi_s$ is the intersection of a lattice $L_s$ with the unit hypercube $[0,1)^s$. This means that all the points of $\Psi_s$ lie in a limited number of equidistant parallel hyperplanes. And similarly for each $\Psi_I$.

We know how to compute the number $n_s$ of hyperplanes and the distance $d_s$ between them: $d_s = 1/\ell_s$ where $\ell_s$ is the Euclidean length of a shortest non-zero vector in the dual lattice

$$L_s^* = \{\boldsymbol{w} \in \mathbb{Z}^s : \boldsymbol{w} \cdot \boldsymbol{v} \in \mathbb{Z} \text{ for all } \boldsymbol{v} \in L_s\}$$

and $n_s = \ell_s^{(1)} - 1$, where $\ell_s^{(1)}$ is the length of a shortest nonzero vector with the $L^1$ norm.

We have a C++ software named `LatMRG` (currently in renovation) to compute these values.

# A Figure of Merit for LCGs and MWC Generators

We can compute the distance $d_I$ between hyperplanes for a collection $\mathcal{J}$ of subsets $I$ of coordinates. But how to combine these values into a single measure?

For a lattice with density $m$ in $s$ dimensions, there is a general upper bound $\ell_s^*(m)$ on the length of the shortest nonzero dual vector. We can standardize $\ell_s$ to $S_s = \ell_s/\ell_s^*(m)$, which is a real number between 0 and 1, and we want it to be close to 1. For projections $I$, we define $S_I = \ell_I/\ell_{|I|}^*(m)$. The we can look at the worst-case figure of merit $M_{\mathcal{J}} = \min_{I\in\mathcal{J}} S_I$.

# A Figure of Merit for LCGs and MWC Generators

We can compute the distance $d_I$ between hyperplanes for a collection $\mathcal{J}$ of subsets $I$ of coordinates. But how to combine these values into a single measure?

For a lattice with density $m$ in $s$ dimensions, there is a general upper bound $\ell_s^*(m)$ on the length of the shortest nonzero dual vector. We can standardize $\ell_s$ to $S_s = \ell_s/\ell_s^*(m)$, which is a real number between 0 and 1, and we want it to be close to 1. For projections $I$, we define $S_I = \ell_I/\ell_{|I|}^*(m)$. The we can look at the worst-case figure of merit $M_{\mathcal{J}} = \min_{I\in\mathcal{J}} S_I$.

In the following, we will take $\mathcal{J}$ as the collection of subsets
$I = \{0, \ldots, s-1\}$ for $s = k+1, \ldots, t_1$, and
$I = \{0 = i_1 < \cdots < i_s\}$ with $i_s < t_s$, for $k+1 \le s \le 5$.

We take $(t_1, \ldots, t_5) = (12, 12, 10, 10, 8)$.
Then $\mathcal{J}$ contains 162 subsets $I$ for $k = 2$, and 126 subsets for $k = 3$.

Some inequalities proved in forthcoming paper (L 2025):

(i) For any $s \geq 2$, $\ell_s^2 \leq b^2 + 1$ and $n_s \leq b$.
(ii) For any $s \geq k+1$, $\ell_s^2 \leq \sum_{j=0}^{k} a_j^2$ and $n_s \leq -1 + \sum_{j=0}^{k} |a_j|$.
(iii) For any $s \geq k+2$,

$$\begin{aligned} \ell_s^2 &\leq 1 + a_0^2 + \sum_{j=1}^{k} \left((b - a_j)\mathbb{I}[a_j \neq 0] - \mathbb{I}[a_{j-1} \neq 0]\right)^2, \\ n_s &\leq |a_0| + \sum_{j=1}^{k} \left|(b - a_j)\mathbb{I}[a_j \neq 0] - \mathbb{I}[a_{j-1} \neq 0]\right|. \end{aligned}$$

(iv) If $a_0 = -1$, $a_j = a_k$ or $0$ for $j \geq 1$, and $s \geq k+2$, then

$$\ell_s^2 \leq (3 + 4|\{0 < j < k : a_j \neq 0\}|)b.$$

(v) If only $a_0$ and $a_k$ are nonzero and $s \geq k+2$, then $n_s \leq \sqrt{3b}$.

Some inequalities proved in forthcoming paper (L 2025):

(i) For any $s \geq 2$, $\ell_s^2 \leq b^2 + 1$ and $n_s \leq b$.
(ii) For any $s \geq k+1$, $\ell_s^2 \leq \sum_{j=0}^{k} a_j^2$ and $n_s \leq -1 + \sum_{j=0}^{k} |a_j|$.
(iii) For any $s \geq k+2$,

$$\begin{aligned} \ell_s^2 &\leq 1 + a_0^2 + \sum_{j=1}^{k} ((b - a_j)\mathbb{I}[a_j \neq 0] - \mathbb{I}[a_{j-1} \neq 0])^2, \\ n_s &\leq |a_0| + \sum_{j=1}^{k} |(b - a_j)\mathbb{I}[a_j \neq 0] - \mathbb{I}[a_{j-1} \neq 0]|. \end{aligned}$$

(iv) If $a_0 = -1$, $a_j = a_k$ or $0$ for $j \geq 1$, and $s \geq k+2$, then

$$\ell_s^2 \leq (3 + 4|\{0 < j < k : a_j \neq 0\}|)b.$$

(v) If only $a_0$ and $a_k$ are nonzero and $s \geq k+2$, then $n_s \leq \sqrt{3b}$.

These inequalities also hold if we replace $b$ by $b^*$.

There are similar inequalities for $\ell_I$ and $n_I$, for subsets $I$ of coordinates.

# Some Previously-Proposed MWC Generators

All of them were proposed without looking at the lattice structure.

**Marsaglia and Zaman (1991): AWC and SWB.**

$a_0 = -1$, $a_k = \pm 1$, and $a_r = \pm 1$ for some integers $0 < r < k$.

Bad structure: all points $(u_{n-k}, u_{n-r}, u_n)$ are in two planes at distance $1/\sqrt{3}$ apart.

# Some Previously-Proposed MWC Generators

All of them were proposed without looking at the lattice structure.

**Marsaglia and Zaman (1991): AWC and SWB.**
$a_0 = -1$, $a_k = \pm 1$, and $a_r = \pm 1$ for some integers $0 < r < k$.
Bad structure: all points $(u_{n-k}, u_{n-r}, u_n)$ are in two planes at distance $1/\sqrt{3}$ apart.

**Marsaglia (1994)**:
$b = 2^{32}$, $k = 2$, $(a_0, a_1, a_2) = (-1, 1111111464, 1111111464)$, period $\rho \approx 2^{94}$.
Gives $d_4 \approx 3.4 \times 10^{-5}$, which is relatively large, and $S_4 \approx 0.0021$.

These defects (and others) were shown by Couture, L, and Tezuka (1993, 1994, 1995, 1997).

From then, one of my projects was to construct better MWC generators, but I never had time to get into it until now.

**Goresky and Klapper (2003)**: They suggested taking $a_0 \neq -1$.

| name | $b$ | $k$ | $m$ | $\sum_{j=0}^{k} a_j^2$ | $n_{k+1}$ | $S_{k+1}$ |
|---|---|---|---|---|---|---|
| GK24a | $2^{24}$ | 48 | $2(b^{48} - b^{46} - b^{38} - b^{14}) + 3$ | 25 | 10 | 1.54e-7 |
| GK24b | $2^{24}$ | 41 | $2(b^{41} - b^{38} - 2b^{14}) + 3$ | 33 | 10 | 2.25e-7 |
| GK25 | $2^{25}$ | 22 | $2(b^{22} - b^{20} - b^{17} - b^{16} + b^{15} + b^{11} - b^{6} + b^{4}) + 3$ | 41 | 18 | 2.03e-7 |
| GK32 | $2^{32}$ | 33 | $4(b^{33} - b^{20} - b^{14} - b^{11} - b^{4}) + 5$ | 105 | 24 | 2.14e-9 |

**Posted by Vigna (2021)**:

| name | $b$ | $k$ | $(a_0, \ldots, a_k)$ only $a_0$ and $a_k \neq 0$ | $m \approx$ | sec | $S_{k+2}$ | $M_{\mathcal{J}}$ |
|---|---|---|---|---|---|---|---|
| MWC192 | $2^{64}$ | 2 | $(-1, 0, 1841980868325024499 8)$ | $2^{192}$ | 0.42 | 1.52e-5 | 2.47e-10 |
| MWC256 | $2^{64}$ | 3 | $(-1, 0, 0, 18443978745271340463)$ | $2^{256}$ | 0.45 | 1.67e-6 | 9.71e-17 |
| GMWC256 | $2^{64}$ | 3 | $(23859240299902735, 0, 0, 18416972077401671842)$ | $2^{256}$ | 1.85 | 1.41e-3 | 2.05e-7 |

Column "sec" gives the time to generate $10^9$ 64-bit integers.

# New Proposals

Take $b = 2^{64}$, $k = 2, 3$, some $a_j$'s set to 0. In each case, we sampled 10 million sets of coefficients $a_j$, computed the FOM $M_{\mathcal{J}}$ for each case where both $m$ and $(m-1)/2$ was prime, and retained the one with the largest $M_{\mathcal{J}}$.

| name | $k$ | $(a_0, \ldots, a_k)$ | sec | $d_{k+2}$, $d_l$ | $S_{k+2}$, $S_l$ |
|---|---|---|---|---|---|
| mwc64k2a1 | 2 | $(-1, 0, 82961845397085)$ | 0.41 | 2.56e-10 | 2.54e-4 |
| | | $l = \{1, 3, 4\}$ | | 2.56e-10 | 1.14e-8 |
| mwc64k2a2 | 2 | $(-1, 193154555888013165, 1966812196490295)$ | 0.81 | 1.63e-13 | 0.180 |
| | | $l = \{1, 2, 3\}$ | | 2.35e-17 | 0.113 |
| mwc64k3a1 | 3 | $(-1, 0, 0, 369074847843357)$ | 0.44 | 2.52e-10 | 1.08e-5 |
| | | $l = \{1, 2, 4, 5\}$ | | 2.52e-10 | 2.70e-9 |
| mwc64k3a2 | 3 | $(-1, 0, 184698970548483715, 6028691832887)$ | 0.68 | 1.78e-13 | 3.50e-2 |
| | | $l = \{1, 3, 4, 5\}$ | | 1.78e-13 | 1.07e-5 |
| mwc64k3a3 | 3 | $(-1, 42677777384320164, 224559258625446056, 89699660373453)$ | 1.12 | 5.55e-15 | 0.653 |
| | | $l = \{1, 2, 5, 7, 8\}$ | | 1.68e-14 | 0.216 |
| mwc64k3a1Xor | 3 | output $x_n \oplus x_{n-2}$ | 0.49 | | |
| mwc64k3a2Xor | 3 | output $x_n \oplus x_{n-2}$ | 0.73 | | |

For comparison, Mersenne Twister and PCG in Python take more than 1.10 sec.

To generate uniform random numbers in "double", can use

`u = inv53 * (x >> 11)` $= 2^{-53}(\mathrm{x} * 2^{-11})$ or perhaps `u = inv63 * (x >> 1)`

With the first method, each real number $i \times 2^{-53}$ for $i = 0, 1, \ldots, 2^{53} - 1$ has the same probability $2^{-53}$ and is represented exactly as a `double`.

Pretty much the best we can do to approximate $U(0, 1)$, except that $u$ can be 0.
To avoid 0, just skip it when it occurs. Function code:

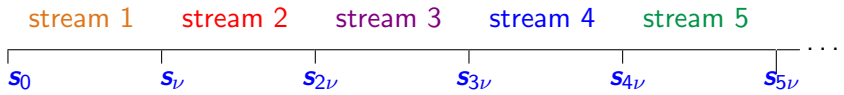
```
double mwc64k3a2U01 {
   uint64 block53 = (mwc64k3a2() >> 11);
   if (block53 == 0) return mwc64k3a2U01();
   return inv53 * block53; }
```

Time (seconds) to generate $10^9$ $U(0, 1)$ random numbers:

| | | inv53 | inv63 | inv53 skip zero |
|---|---|---|---|---|
| mwc64k2a2 | $k = 2$ | 0.85 | 0.84 | 1.00 |
| mwc64k3a1Xor | $k = 3$ | 0.67 | 0.67 | 0.80 |
| mwc64k3a2 | $k = 3$ | 0.87 | 0.87 | 1.01 |
| mwc64k3a2Xor | $k = 3$ | 0.94 | 0.95 | 1.01 |
| mwc64k3a3 | $k = 3$ | 1.09 | 1.07 | 1.20 |

# Creating new streams

Disjoint streams that start $\nu$ steps apart:




Initial state $\boldsymbol{s}_{i\nu}$ of stream $i$ corresponds to LCG state $y_{i\nu}$.
Precompute $b_1 = (b^\nu \bmod m)$ and/or its inverse $b_1^*$.
Memorize the LCG seed $y_{i\nu}$ of the last stream that was created.

Jump ahead: new LCG seed will be $y_{(i+1)\nu} = b_1^* y_{i\nu} \bmod m$.
Jump backwards: new LCG seed will be $y_{(i-1)\nu} = b_1 y_{i\nu} \bmod m$.

Convert to MWC state: $\boldsymbol{x}_{n+\nu} = \phi^{-1}(y_{n+\nu})$.

Time to make $10^6$ large jumps with `mwc64k3a2`, using NTL library to handle large integers:

LCG jump only: 0.05 seconds
Jump + conversion: 0.24 seconds. This gives 4 million streams per second.

# Conclusion

- A flurry of computer applications require random numbers.
- Multiple streams of random numbers are required for parallel computing but also for other settings, such as comparing systems and simulation-based optimization.
  Efficiency and repeatability are important.
- Creating the streams in parallel is highly desirable when we need many.
  This can be implemented in many ways, including jumping ahead, counter-based RNGs, or just numbering the streams sequentially and hashing the numbers to get the seeds.
- MWC recurrences can provide very fast and reliable generators for simulation.
  They can provide multiple streams and could also be used to define hashing functions for counter-based generators.

## Some References



Blackman, D., and S. Vigna. 2021.
"Scrambled Linear Pseudorandom Number Generators".
*ACM Transactions on mathematical Software*. 47(4):Article 36.


Claessen, K., and M. H. Pałka. 2013.
"Splittable pseudorandom number generators using cryptographic hashing".
In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell'13, 47–58: ACM.


Couture, R., and P. L'Ecuyer. 1997.
"Distribution Properties of Multiply-with-Carry Random Number Generators".
*Mathematics of Computation* 66(218):591–607.


Goresky, M., and A. Klapper. 2003.
"Efficient Multiply-with-Carry Random Number Generators with Maximal Period".
*ACM Transactions on Modeling and Computer Simulation* 13(4):310–321.


L'Ecuyer, P. 1999.
"Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators".
*Operations Research* 47(1):159–164.


L'Ecuyer, P. 1999.
"Tables of Maximally Equidistributed Combined LFSR Generators".
*Mathematics of Computation* 68(225):261–269.


L'Ecuyer, P. 2012.
"Random Number Generation".
In *Handbook of Computational Statistics* (second ed.)., edited by J. E. Gentle, W. Haerdle, and Y. Mori, 35–71. Berlin: Springer-Verlag.


L'Ecuyer, P. 2015.
"Random Number Generation with Multiple Streams for Sequential and Parallel Computers".
In *Proceedings of the 2015 Winter Simulation Conference*, 31–44.


L'Ecuyer, P. 2026.
"Fast and Reliable Multiply-With-Carry Random Number Generators". Forthcoming.


P. L'Ecuyer, D. Munger, and N. Kemerchou. 2015.
"clRNG: A Random Number API with Multiple Streams for OpenCL," technical report (user guide), `https://www-labs.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf`.


L'Ecuyer, P., D. Munger, B. Oreshkin, and R. Simard. 2017.
"Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs".
*Mathematics and Computers in Simulation* 135:3–17.


L'Ecuyer, P., Nadeau-Chamard, O., Chen, Y.-F., and Lebar J. 2021.
"Multiple Streams with Recurrence-Based, Counter-Based, and Splittable Random Number Generators".
In *Proceedings of the 2021 Winter Simulation Conference*: IEEE Press, 1–16.


L'Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. 2002.

"An Object-Oriented Random-Number Package with Many Long Streams and Substreams". *Operations Research* 50(6):1073–1075.

Marsaglia, G. 1994.
"Yet Another RNG". Posted to the electronic billboard `sci.stat.math`, August 1.

Tezuka, S., P. L'Ecuyer, and R. Couture. 1993.
"On the Add-with-Carry and Subtract-with-Borrow Random Number Generators". *ACM Transactions of Modeling and Computer Simulation* 3(4):315–331.

Vigna, S. 2021.
"A PRNG Shootout". `https://prng.di.unimi.it/`.