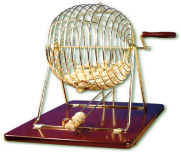
Générateurs de Nombres Aléatoires Parallèles

Pierre L'Ecuyer

Université de Montréal



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

Café Calcul, CNRS Mathématiques, France, Janvier 2025

We want sequences of numbers that look random

Example: Bit sequence (head or tail):

011110100110110101001101100101000111**?**...

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

We want sequences of numbers that look random

Example: Bit sequence (head or tail):

01111?100110?1?101001101100101000111...

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

Uniformity and independence:
Example: 8 possibilities for the 3 bits ? ? ?:

000, 001, 010, 011, 100, 101, 110, 111

Want a probability of 1/8 for each, independently of everything else.

We want sequences of numbers that look random

Example: Bit sequence (head or tail):

01111?100110?1?10100110110010100111...

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

Uniformity and independence:
Example: 8 possibilities for the 3 bits ? ? ?:

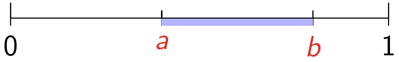
000, 001, 010, 011, 100, 101, 110, 111

Want a probability of 1/8 for each, independently of everything else.

For s bits, probability of $1/2^s$ for each of the 2^s possibilities.

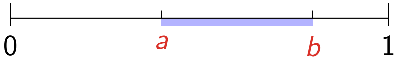
Uniform distribution over the interval $(0, 1)$

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



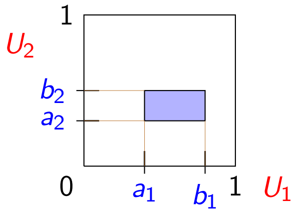
Uniform distribution over the interval $(0, 1)$

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



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

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



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

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).

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).

Non-uniform variates:

To generate X such that $\mathbb{P}[X \le x] = F(x)$:

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

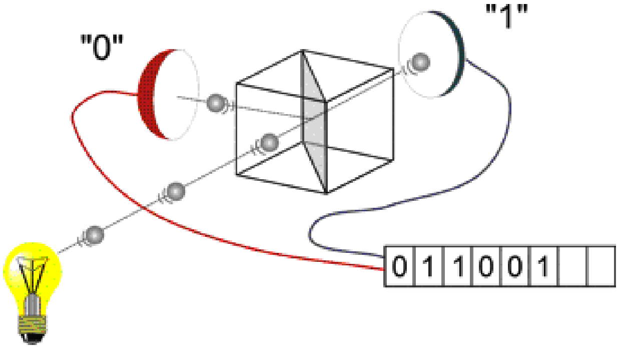
This is inversion.

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

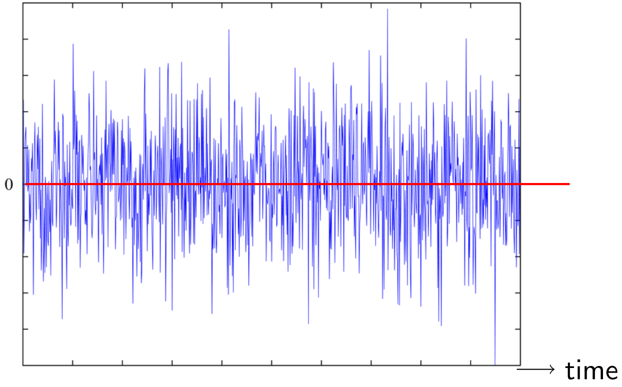
Also other methods such as rejection, etc., when F^{-1} is costly to compute.

Physical devices for computers

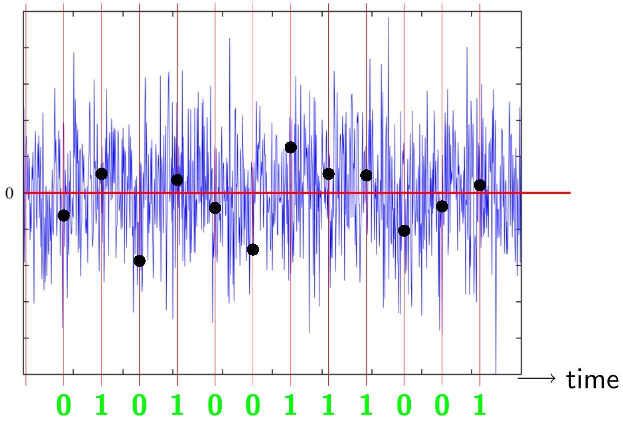
Photon trajectories (sold by **id-Quantique**):



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:

$$\underbrace{0\ 1}_{1}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 0}_{0}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 1}_{1}\ \underbrace{1\ 0}_{1}\ \underbrace{1\ 1}_{0}\ \underbrace{0\ 1}_{1}\ \underbrace{0\ 0}_{0}$$

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:

$$\underbrace{0\;1}_{1}\;\underbrace{1\;0}_{1}\;\underbrace{0\;0}_{0}\;\underbrace{1\;0}_{1}\;\underbrace{0\;1}_{1}\;\underbrace{1\;0}_{1}\;\underbrace{1\;1}_{0}\;\underbrace{0\;1}_{1}\;\underbrace{0\;0}_{0}$$

or (this eliminates the bias):

$$\underbrace{0\;1}_{0}\;\underbrace{1\;0}_{1}\;\underbrace{0\;0}_{}\;\underbrace{1\;0}_{1}\;\underbrace{0\;1}_{0}\;\underbrace{1\;0}_{1}\;\underbrace{1\;1}_{}\;\underbrace{0\;1}_{0}\;\underbrace{0\;0}_{}$$

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:

$$\underbrace{0\ 1}_{1}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 0}_{0}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 1}_{1}\ \underbrace{1\ 0}_{1}\ \underbrace{1\ 1}_{0}\ \underbrace{0\ 1}_{1}\ \underbrace{0\ 0}_{0}$$

or (this eliminates the bias):

$$\underbrace{0\ 1}_{0}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 0}_{}\ \underbrace{1\ 0}_{1}\ \underbrace{0\ 1}_{0}\ \underbrace{1\ 0}_{1}\ \underbrace{1\ 1}_{}\ \underbrace{0\ 1}_{0}\ \underbrace{0\ 0}_{}$$

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.

A 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.

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);

$$\begin{tikzcd}
s_0 \arrow[d, "g"'] \\
u_0
\end{tikzcd}$$

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);

$$\begin{tikzcd}
s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \\
u_0 &
\end{tikzcd}$$

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);

$$\begin{tikzcd} s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[d, "g"'] \\ u_0 & u_1 \end{tikzcd}$$

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);

$$\begin{tikzcd}
s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

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);

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

Period of $\{s_n,\ n \geq 0\}$: $\rho \leq$ cardinality of $\mathcal{S}$.

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);

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

Period of $\{s_n,\ n \geq 0\}$: $\rho \leq$ cardinality of $\mathcal{S}$.

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.

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

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

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

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

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

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

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

With random seed s_0, an RNG is a gigantic roulette wheel.
Selecting s_0 at random and generating s random numbers means spinning the wheel and taking $\mathbf{u} = (u_0, \ldots, u_{s-1})$.

$$\begin{tikzcd}
\cdots \arrow[r, "f"] & s_{\rho-1} \arrow[r, "f"] \arrow[d, "g"'] & s_0 \arrow[r, "f"] \arrow[d, "g"'] & s_1 \arrow[r, "f"] \arrow[d, "g"'] & \cdots \arrow[r, "f"] & s_n \arrow[r, "f"] \arrow[d, "g"'] & s_{n+1} \arrow[r, "f"] \arrow[d, "g"'] & \cdots \\
\cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots
\end{tikzcd}$$

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

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

With random seed s_0, an RNG is a gigantic roulette wheel.
Selecting s_0 at random and generating s random numbers means spinning the wheel and taking $\mathbf{u} = (u_0, \ldots, u_{s-1})$.

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

If we choose s_0 randomly in $\mathcal{S}$ and we generate s numbers, this corresponds to choosing a random point in the finite set

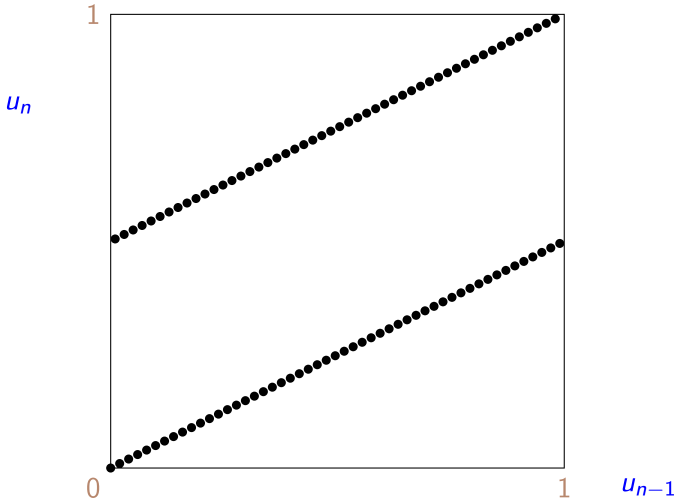
$$\Psi_s = \{\mathbf{u} = (u_0, \ldots, u_{s-1}) = (g(s_0), \ldots, g(s_{s-1})),\ s_0 \in \mathcal{S}\}.$$

We want to approximate "$\mathbf{u}$ has the uniform distribution over $[0, 1]^s$."

Ψ_s must cover $[0, 1]^s$ very evenly.

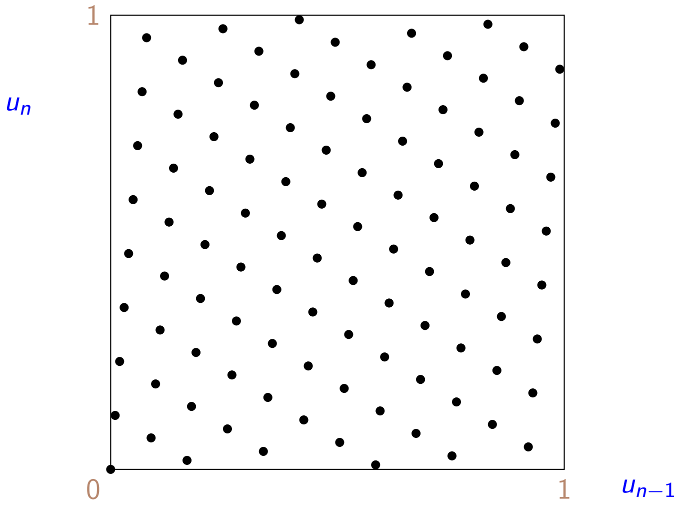
Example: Tiny Linear Congruential Generator (LCG) with $s = 2$:

$x_n = 51\,x_{n-1} \bmod 101$; $u_n = x_n/101$. All numbers from 1 to 100 appear once in a cycle.



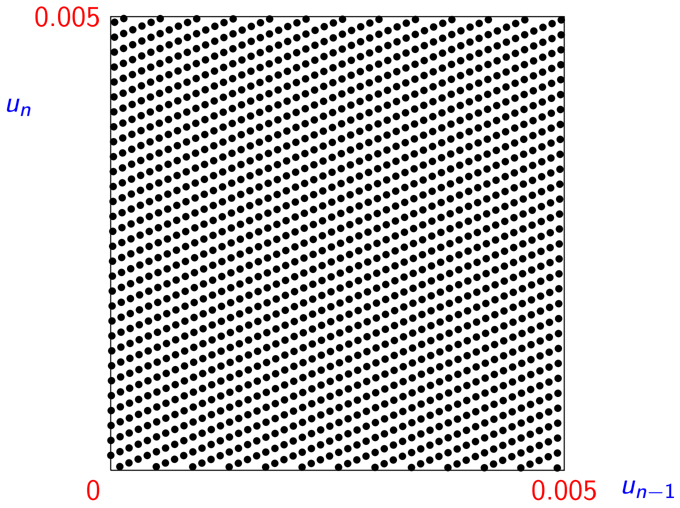
Ψ_2 contains the 101 black dots. Good uniformity in one dimension, but not in two!

Tiny LCG with $s = 2$: $x_n = 12\,x_{n-1} \bmod 101$; $u_n = x_n/101$.
All numbers from 1 to 100 appear exactly once in a cycle.



Here, Ψ_2 covers the space much more evenly in two dimensions.

Another LCG example: $x_n = 4809922\, x_{n-1} \bmod 60466169$, $u_n = x_n/60466169$



With more points, we can do much better!

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

If we choose s_0 randomly in $\mathcal{S}$ and we generate s numbers, this corresponds to choosing a random point in the finite set

$$\Psi_s = \{\mathbf{u} = (u_0, \dots, u_{s-1}) = (g(s_0), \dots, g(s_{s-1})),\ s_0 \in \mathcal{S}\}.$$

We want to approximate "$\mathbf{u}$ has the uniform distribution over $[0,1]^s$."

Measure of quality: Ψ_s must cover $[0,1]^s$ very evenly.

Design and analysis:

1. Define a uniformity measure for Ψ_s, 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.

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

If we choose s_0 randomly in $\mathcal{S}$ and we generate s numbers, this corresponds to choosing a random point in the finite set

$$\Psi_s = \{\mathbf{u} = (u_0, \ldots, u_{s-1}) = (g(s_0), \ldots, g(s_{s-1})),\ s_0 \in \mathcal{S}\}.$$

We want to approximate "$\mathbf{u}$ has the uniform distribution over $[0,1]^s$."

Measure of quality: Ψ_s must cover $[0,1]^s$ very evenly.

Design and analysis:

1. Define a uniformity measure for Ψ_s, 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.

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

$$\Psi_I = \{\mathbf{u} = (u_{i_1}, \ldots, u_{i_s}) = (g(s_{i_1}), \ldots, g(s_{i_s})),\ s_0 \in \mathcal{S}\}.$$

Myth 1. After over 70 years of study and thousands of articles, this problem is certainly solved and RNGs available in popular software must be reliable.

Myth 1. After over 70 years of study and thousands of articles, this problem is certainly solved and RNGs available in popular software must be reliable.

No.

Myth 2. I use a fast RNG with period length $> 2^{1000}$, so it is certainly excellent!

Myth 1. After over 70 years of study and thousands of articles, this problem is certainly solved and RNGs available in popular software must be reliable.

No.

Myth 2. I use a fast RNG with period length $> 2^{1000}$, so it is certainly excellent!

No.

Example: $u_n = (n/2^{1000}) \bmod 1$ for $n = 0, 1, 2, \ldots$.

Myth 1. After over 70 years of study and thousands of articles, this problem is certainly solved and RNGs available in popular software must be reliable.

No.

Myth 2. I use a fast RNG with period length $> 2^{1000}$, so it is certainly excellent!

No.

Example: $u_n = (n/2^{1000}) \bmod 1$ for $n = 0, 1, 2, ...$.

Other examples: Subtract-with-borrow, lagged-Fibonacci, xorwow, etc.
Were designed to be very fast: simple with very few operations.
They have bad uniformity in higher dimensions.

RNGs based on a linear recurrence

For most classical RNGs, the state follows a linear recurrence of the form

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m,$$

or more generally the state is $\mathbf{x}_n \in \mathbb{Z}_m^k$ and

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

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

$$u_n = g(\mathbf{x}_n).$$

When g has the same form of linearity, the sets Ψ_s have a linear structure whose uniformity can be assessed mathematically by computable quantities (e.g., the spectral test, measures of equidistribution, etc.). Computing these theoretical measures (when we can) is more important than empirical statistical testing.

RNGs based on a linear recurrence

For most classical RNGs, the state follows a linear recurrence of the form

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m,$$

or more generally the state is $\mathbf{x}_n \in \mathbb{Z}_m^k$ and

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

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

$$u_n = g(\mathbf{x}_n).$$

When g has the same form of linearity, the sets Ψ_s have a linear structure whose uniformity can be assessed mathematically by computable quantities (e.g., the spectral test, measures of equidistribution, etc.). Computing these theoretical measures (when we can) is more important than empirical statistical testing.

There are also constructions for which g is nonlinear or has a different form of linearity than f. For those, measuring the uniformity before the application of g is also relevant.

Linear multiple recursive generator (MRG)

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

State: $s_n = (x_{n-k+1}, \ldots, x_n)$. 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: $s_n = (x_{n-k+1}, \ldots, x_n)$. Max. period: $\rho = m^k - 1$ if m is prime.

Numerous variants and implementations.

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

Structure of the points Ψ_s:

$x_0, \ldots, x_{k-1}$ can take any value from 0 to $m-1$, then $x_k, x_{k+1}, \ldots$ are determined by the linear recurrence. Thus, $(x_0, \ldots, x_{k-1}) \mapsto (x_0, \ldots, x_{k-1}, x_k, \ldots, x_{s-1})$ is a linear mapping.

It follows that Ψ_s is a linear space; it is the intersection of a lattice with the unit cube.

RNGs based on linear recurrences modulo 2

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

RNGs based on linear recurrences modulo 2

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

Clever choice of **A**: transition via shifts, XOR, AND, masks, etc., on blocks of bits. Very fast.

Examples: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, . . .

RNGs based on linear recurrences modulo 2

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

Clever choice of **A**: transition via shifts, XOR, AND, masks, etc., on blocks of bits. Very fast.

Examples: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, ...

Each coordinate of $\mathbf{x}_n$ and of $\mathbf{y}_n$ follows the linear recurrence

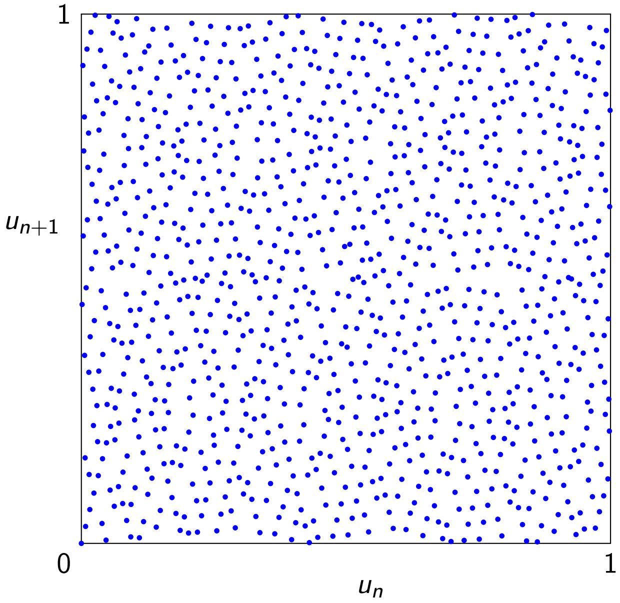
$$x_{n,j} = (\alpha_1 x_{n-1,j} + \cdots + \alpha_k x_{n-k,j}),$$

with characteristic polynomial

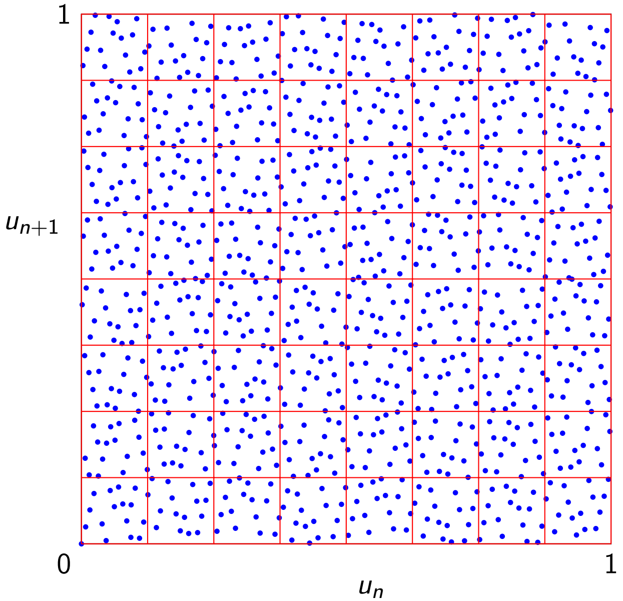
$$P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(\mathbf{A} - z\mathbf{I}).$$

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

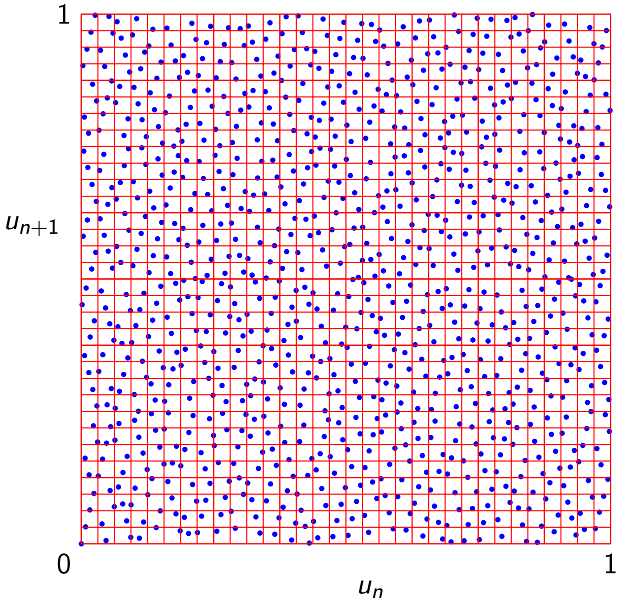
Uniformity measures. Example: $k = 10$, $2^{10} = 1024$ points



Uniformity measures. Example: $k = 10$, $2^{10} = 1024$ points



Uniformity measures. Example: $k = 10$, $2^{10} = 1024$ points



Uniformity measures based on equidistribution.

For each $n \geq 0$, we have $\mathbf{y}_n = \mathbf{B}\mathbf{A}^n\mathbf{x}_0 \bmod 2$.

Suppose we partition $[0,1)^s$ in 2^{ℓ} equal intervals.
Gives $2^{s\ell}$ cubic boxes.

For each s and ℓ, the $s\ell$ bits that determine the box are the first ℓ bits of $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{s-1}$.
These bits can be written as $\mathbf{M}\,\mathbf{x}_0 \bmod 2$.

Each box contains exactly $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. This is possible only if $s\ell \leq k$. We then say that those points are equidistributed for ℓ bits in s dimensions.

Uniformity measures based on equidistribution.

For each $n \geq 0$, we have $\mathbf{y}_n = \mathbf{B}\mathbf{A}^n\mathbf{x}_0 \bmod 2$.

Suppose we partition $[0, 1)^s$ in 2^{ℓ} equal intervals.
Gives $2^{s\ell}$ cubic boxes.

For each s and ℓ, the $s\ell$ bits that determine the box are the first ℓ bits of $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{s-1}$.
These bits can be written as $\mathbf{M}\,\mathbf{x}_0 \bmod 2$.

Each box contains exactly $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. This is possible only if $s\ell \leq k$. We then say that those points are equidistributed for ℓ bits in s dimensions.

If this holds for all s and ℓ such that $s\ell \leq k$, the RNG is called maximally equidistributed.

Uniformity measures based on equidistribution.

For each $n \geq 0$, we have $\mathbf{y}_n = \mathbf{B}\mathbf{A}^n\mathbf{x}_0 \bmod 2$.

Suppose we partition $[0,1)^s$ in 2^ℓ equal intervals.
Gives $2^{s\ell}$ cubic boxes.

For each s and ℓ, the $s\ell$ bits that determine the box are the first ℓ bits of $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{s-1}$.
These bits can be written as $\mathbf{M}\,\mathbf{x}_0 \bmod 2$.

Each box contains exactly $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. This is possible only if $s\ell \leq k$. We then say that those points are equidistributed for ℓ bits in s dimensions.

If this holds for all s and ℓ such that $s\ell \leq k$, the RNG is called maximally equidistributed.

Can be generalized to rectangular boxes: take ℓ_j bits for coordinate j, with
$\ell_0 + \cdots + \ell_{s-1} \leq k$.

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Evolution:

$x_{n-1} =$ |00010100101001101100110110100101|

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Evolution:

$$(x_{n-1} \ll 6) \text{ XOR } x_{n-1}$$

$$\begin{array}{rr|l|}
x_{n-1} = & & 00010100101001101100110110100101 \\
 & 100101 & 00101001101100110110100101 \\
 & & 00111101000101011010010011100101
\end{array}$$

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32}-1\}$ (32 bits). Evolution:

$$B \quad = \quad ((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13$$

```
x_{n-1} =       00010100101001101100110110100101
          10010100101001101100110110100101
                00111101000101011010010011100101
B =                          00111101000101011010010011100101
```

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Evolution:

$$
\begin{aligned}
B &= ((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13 \\
x_n &= (((x_{n-1} \text{ with last bit at } 0) \ll 18) \text{ XOR } B).
\end{aligned}
$$

```
x_{n-1} =         00010100101001101100110110100101
            10010100101001101100110110100101
                  00111101000101011010010011100101
B =                            00111101000101011010010011100101
x_{n-1}           00010100101001101100110110100100
00010100101001101100110110100100
```

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Evolution:

$$\begin{aligned} B &= ((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13 \\ x_n &= (((x_{n-1} \text{ with last bit at } 0) \ll 18) \text{ XOR } B). \end{aligned}$$

```
x_{n-1} =         00010100101001101100110110100101
            10010100101001101100110110100101
                  00111101000101011010010011100101
B =                            00111101000101011010010011100101
x_{n-1}           00010100101001101100110110100100
00010100101001101100110110100100
x_n =             00110110100100011110100010101101
```

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32}-1\}$ (32 bits). Evolution:

$$\begin{aligned} B &= ((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13 \\ x_n &= (((x_{n-1} \text{ with last bit at 0}) \ll 18) \text{ XOR } B). \end{aligned}$$

```
x_{n-1} =         00010100101001101100110110100101
            10010100101001101100110110100101
                  00111101000101011010010011100101
B =                            00111101000101011010010011100101
x_{n-1}           00010100101001101100110110100100
00010100101001101100110110100100
x_n =             00110110100100011110100010101101
```

This implements $\mathbf{x}_n = \mathbf{A}\,\mathbf{x}_{n-1} \bmod 2$ for a certain $\mathbf{A}$.
The first $k = 31$ bits of $x_1, x_2, x_3, \ldots,$ visit all integers from 1 to 2147483647 $(= 2^{31}-1)$ exactly once before returning to x_0. Output in $(0,1)$: $u_n = x_n \times 2^{-31}$.

Example of fast RNG: operations on blocks of bits.

Mini-Example: Choose $x_0 \in \{2, \ldots, 2^{32}-1\}$ (32 bits). Evolution:

$$\begin{aligned} B &= ((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13 \\ x_n &= (((x_{n-1} \text{ with last bit at } 0) \ll 18) \text{ XOR } B). \end{aligned}$$

```
x_{n-1} =         00010100101001101100110110100101
            10010100101001101100110110100101
                  00111101000101011010010011100101
B =                            00111101000101011010010011100101
x_{n-1}           00010100101001101100110110100100
00010100101001101100110110100100
x_n =             00110110100100011110100010101101
```

This implements $\mathbf{x}_n = \mathbf{A}\,\mathbf{x}_{n-1} \bmod 2$ for a certain $\mathbf{A}$.
The first $k = 31$ bits of $x_1, x_2, x_3, \ldots,$ visit all integers from 1 to 2147483647 $(= 2^{31}-1)$ exactly once before returning to x_0. Output in $(0,1)$: $u_n = x_n \times 2^{-31}$.

A single RNG does not suffice.

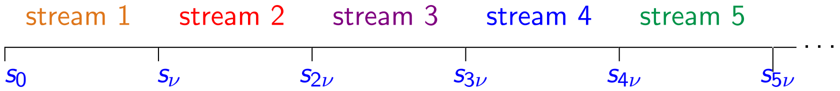
One often needs several independent streams of random numbers, for example:

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 ν.



Jumping ahead by ν steps is easy when f is linear:

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

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

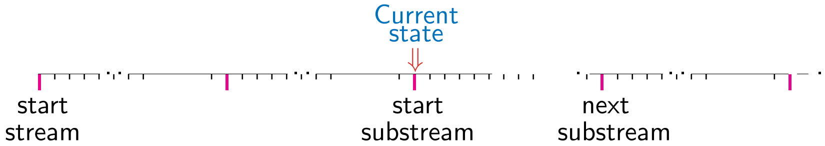
$$\mathbf{x}_{n+\nu} = (\mathbf{A}^\nu \bmod m)\mathbf{x}_n \bmod m$$

with the matrix $(\mathbf{A}^\nu \bmod m)$ 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).

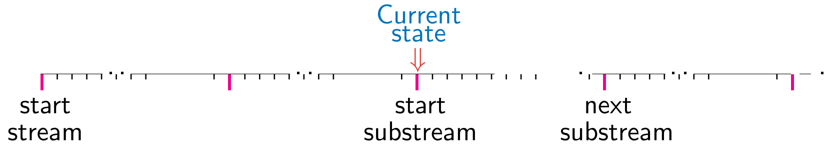
One stream:



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:



RngStreams is based on the MRG32k3a generator, with period $\approx 2^{191}$.
Streams start $\nu = 2^{127}$ values apart and substreams have length 2^{76}.
It 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 = **demand (random)** on day j, uniform over $\{0, 1, \ldots, L\}$;
$\min(D_j, X_j)$ sales on day j;
$Y_j = \max(0, X_j - D_j)$ inventory at end of 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
$= c \cdot \min(D_j, X_j) - h \cdot Y_j - (K + k \cdot (S - Yj)) \cdot \mathbb{I}[\text{an order arrives}]$.

Goal: Simulate n times m days for several choices of (s, S) to find the best one.
Number of calls to RNG for order arrivals is random!

Want two streams of random numbers, one substream for each simulation run.
Want same streams and substreams for all policies (s, S).

Inventory example: C code to simulate *m* days with two streams

```
double inventorySimulateOneRun (int m, int s, int S,
       RngStream *stream_demand, RngStream *stream_order) {
   // Simulates inventory model for m days, with the (s,S) policy.
   int Xj = S, Yj;         // Stock Xj in morning and Yj in evening.
   double profit = 0.0;    // Cumulated profit.
   for (int j = 0; j < m; j++) {
      // Generate and subtract the demand for the day.
      Yj = Xj - RandInt (stream_demand, 0, L);
      if (Yj < 0) Yj = 0; // Lost demand.
      profit += c * (Xj - Yj) - h * Yj;
      if ((Yj < s) && (RandU01 (stream_order) < p)) {
         // We have a successful order, we pay for it.
         profit -= K + k * (S - Yj);
         Xj = S;
      } else
         Xj = Yj;          // Order not received.
   }
   return profit / m;      // Return average profit per day.
}
```

Parameters for an experiment

Model parameters:

$L = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$, and $p = 0.95$.

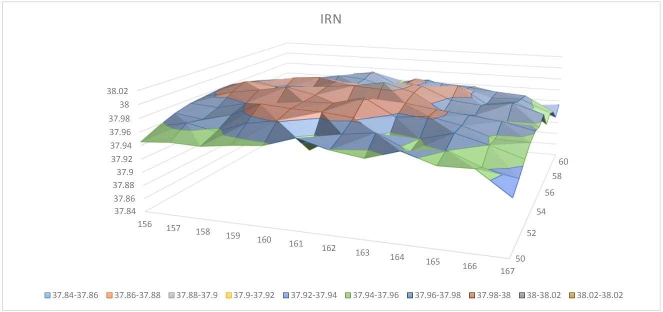
Experiment parameters:

$m = 100$ days per simulation run, $n = 2^{18}$ runs (or replications).

$p = 144$ different policies (s, S): $50 \le s \le 61$ and $156 \le S \le 167$.

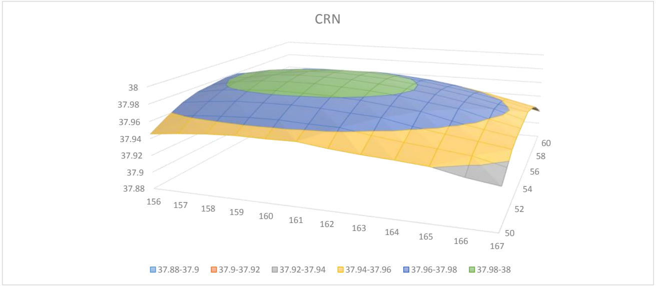
Comparison with independent random numbers

	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



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
      stat_profit[k, i] = inventorySimulateOneRun (m, s[k], S[k],
                                    stream_demand, stream_order);
      // Realign starting points so they are the same 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.

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.

Multiple streams for parallel processors

To run n replications using n threads (or work items) on parallel processors (e.g., on a GPU), in our example we would need $2n$ stream objects (two per thread) to store the RNG states.

Since the substreams in `RngStreams` are not objects, we cannot use one substream per thread, unless we transform them into objects to memorize the substream states.

Multiple streams for parallel processors

To run n replications using n threads (or work items) on parallel processors (e.g., on a GPU), in our example we would need $2n$ stream objects (two per thread) to store the RNG states.

Since the substreams in `RngStreams` are not objects, we cannot use one substream per thread, unless we transform them into objects to memorize the substream states.

Simple approach: create and use $2n$ distinct streams and use them in place of substreams. Drawback: creating many more streams brings significant overhead. But if we really want n parallel threads, we have no choice.

Multiple streams for parallel processors

To run n replications using n threads (or work items) on parallel processors (e.g., on a GPU), in our example we would need $2n$ stream objects (two per thread) to store the RNG states.

Since the substreams in `RngStreams` are not objects, we cannot use one substream per thread, unless we transform them into objects to memorize the substream states.

Simple approach: create and use $2n$ distinct streams and use them in place of substreams. Drawback: creating many more streams brings significant overhead. But if we really want n parallel threads, we have no choice.

If the number n of replications is very large, it makes sense to use $n_1 \ll n$ threads, and have each thread run n_2 independent replications, with $n = n_1 n_2$. Then it suffices to create $2n_1$ stream objects, and let each thread use n_2 substreams.

For the inventory example, we have observed a speedup factor around 3 or 4, compared to creating $2n$ stream objects, by doing this with $n_1 = 2^{16}$ and $n_2 = 64$ for $n = 2^{22}$.
(This was on an oldish AMD Radeon HD 7900 Series GPU using clRNG in OpenCL.)

How to use the multiple streams

Why not use a single stream of random numbers for all threads?
Very bad because (1) too much overhead for transfer and (2) non reproducible.

How to use the multiple streams

Why not use a single stream of random numbers for all threads?
Very bad because (1) too much overhead for transfer and (2) non reproducible.

We need multiple streams.
One stream per processor? One per thread? One per subtask? No.

For reproducibility and effective use of CRNs, streams must be assigned at a **logical level (hardware-independent)**. It should be possible to have many streams in any given thread.

How to use the multiple streams

Why not use a single stream of random numbers for all threads?
Very bad because (1) too much overhead for transfer and (2) non reproducible.

We need multiple streams.
One stream per processor? One per thread? One per subtask? No.

For reproducibility and effective use of CRNs, streams must be assigned at a **logical level (hardware-independent)**. It should be possible to have many streams in any given thread.

There are also simplified settings in which one stream per thread is enough; e.g., if the task is just to fill up a large tensor or random numbers in a given order.

How to define and construct the streams?

A **different RNG** (or parameters) for each stream?

Inconvenient and limited: would be hard to handle millions of streams.

How to define and construct the streams?

A **different RNG** (or parameters) for each stream?
Inconvenient and limited: would be hard to handle millions of streams.

Single RNG with **equally-spaced starting points** for streams and for substreams:
Recommended when possible. Streams can start ν steps apart, as we saw earlier.
Normally requires fast computing of $s_{n+\nu} = f^{\nu}(s_n)$ for large ν, and a single monitor to create all the streams sequentially, one after the other.

How to define and construct the streams?

A **different RNG** (or parameters) for each stream?
Inconvenient and limited: would be hard to handle millions of streams.

Single RNG with **equally-spaced starting points** for streams and for substreams:
Recommended when possible. Streams can start ν steps apart, as we saw earlier.
Normally requires fast computing of $s_{n+\nu} = f^{\nu}(s_n)$ for large ν, and a single monitor to create all the streams sequentially, one after the other.
Drawback: this creation process seems inherently sequential.

How to define and construct the streams?

A **different RNG** (or parameters) for each stream?
Inconvenient and limited: would be hard to handle millions of streams.

Single RNG with **equally-spaced starting points** for streams and for substreams:
Recommended when possible. Streams can start ν steps apart, as we saw earlier.
Normally requires fast computing of $s_{n+\nu} = f^{\nu}(s_n)$ for large ν, and a single monitor to create all the streams sequentially, one after the other.
Drawback: this creation process seems inherently sequential.

We would rather prefer being able to create many streams in parallel!

Can we compute all the starting points in parallel, efficiently?
For example, given s_n, compute $s_{n+j\nu}$ for $j = 1, \ldots, 2^{16}$ all in parallel on a GPU?

How to define and construct the streams?

A **different RNG** (or parameters) for each stream?
Inconvenient and limited: would be hard to handle millions of streams.

Single RNG with **equally-spaced starting points** for streams and for substreams:
Recommended when possible. Streams can start ν steps apart, as we saw earlier.
Normally requires fast computing of $s_{n+\nu} = f^{\nu}(s_n)$ for large ν, and a single monitor to create all the streams sequentially, one after the other.
Drawback: this creation process seems inherently sequential.

We would rather prefer being able to create many streams in parallel!

Can we compute all the starting points in parallel, efficiently?
For example, given s_n, compute $s_{n+j\nu}$ for $j = 1, \ldots, 2^{16}$ all in parallel on a GPU?
Yes. But requires additional storage for jump-ahead matrices, and more work per jump than for a unique jump length ν. This is implemented in cuRand (Cuda), for instance.

Use a function that jumps ahead by $2^k\nu$ for any $k = 0, 1, 2, \ldots$, with a precomputed matrix for each k. Decompose j into a sum of powers of 2, and jump ahead for each power.

Streams with random starting points.

Can be used if jumping ahead is deemed too expensive.

For each stream, we draw a random seed uniformly over $\mathcal{S}$, independently.
There is a risk of overlap between streams, but it can be made very small.

Streams with random starting points.

Can be used if jumping ahead is deemed too expensive.

For each stream, we draw a random seed uniformly over $\mathcal{S}$, independently.
There is a risk of overlap between streams, but it can be made very small.

For period length ρ and s streams of length ℓ, $\mathbb{P}[\text{overlap somewhere}] = P_o \le s^2\ell/\rho$.

ρ	s	ℓ	$P_o \le$
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}

Streams with random starting points.

Can be used if jumping ahead is deemed too expensive.

For each stream, we draw a random seed uniformly over $\mathcal{S}$, independently.
There is a risk of overlap between streams, but it can be made very small.

For period length ρ and s streams of length ℓ, $\mathbb{P}[\text{overlap somewhere}] = P_o \le s^2\ell/\rho$.

ρ	s	ℓ	$P_o \le$
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}

Drawback: generating truly random and independent starting points is not cheap and easy.
It also makes the results not reproducible!

Pseudorandom starting points

Use a second RNG (or similar mechanism) to provide the starting points for the streams. This second RNG provides a sequence of states for the first RNG, which we call a seed schedule. These are the seeds of the successive streams.

Pseudorandom starting points

Use a second RNG (or similar mechanism) to provide the starting points for the streams. This second RNG provides a sequence of states for the first RNG, which we call a seed schedule. These are the seeds of the successive streams.

To cover all seed possibilities for the streams, the state of the second RNG must be at least as large as the state of the first.

For example, if the first RNG has a 128-bit state and the second provides only 64 bits, then at most 2^{64} of the 2^{128} possible seeds can be reached. This increases the probability of having identical streams! I have seen this often in popular software.

Pseudorandom starting points

Use a second RNG (or similar mechanism) to provide the starting points for the streams. This second RNG provides a sequence of states for the first RNG, which we call a seed schedule. These are the seeds of the successive streams.

To cover all seed possibilities for the streams, the state of the second RNG must be at least as large as the state of the first.

For example, if the first RNG has a 128-bit state and the second provides only 64 bits, then at most 2^{64} of the 2^{128} possible seeds can be reached. This increases the probability of having identical streams! I have seen this often in popular software.

Chances of a collision: If we pick s seeds at random uniformly from a pool of size r, the probability that the seeds are not all distinct is approximately $s^2/2r$ when this number is small. For $r = 2^{64}$ and $s = 2^{30}$, this gives $1/32$. This could be acceptable for some applications.

To reduce the chance of bad interactions, the structures of two RNGs should be different. Otherwise, it could create detectable dependence between the streams.

If the second RNG is recurrence-based, then this approach remains inherently sequential.

User provides explicit starting points

When creating new streams, the users can provide their starting points (seeds) as inputs.

Typically, a one-to-one hash function (or a simplified block cipher) would be applied automatically to those inputs, to transform them.

If done right, the user may create 2^{20} streams with "seeds" $1, 2, \ldots, 2^{20}$, for example, without worrying about the dependence between those streams.

User provides explicit starting points

When creating new streams, the users can provide their starting points (seeds) as inputs.

Typically, a one-to-one hash function (or a simplified block cipher) would be applied automatically to those inputs, to transform them.

If done right, the user may create 2^{20} streams with "seeds" $1, 2, \ldots, 2^{20}$, for example, without worrying about the dependence between those streams.

A variant: the user provides a single seed to create a large block of n streams. The system may automatically combine a counter with this seed, then hash the values to obtain the seeds of the n streams.

User provides explicit starting points

When creating new streams, the users can provide their starting points (seeds) as inputs.

Typically, a one-to-one hash function (or a simplified block cipher) would be applied automatically to those inputs, to transform them.

If done right, the user may create 2^{20} streams with “seeds” $1, 2, \dots, 2^{20}$, for example, without worrying about the dependence between those streams.

A variant: the user provides a single seed to create a large block of n streams. The system may automatically combine a counter with this seed, then hash the values to obtain the seeds of the n streams.

Advantage: a large block of streams can be created in parallel.
Drawback: The users must remember the seeds (for reproducibility), and make sure that the blocks of streams created at different places all use different seeds!

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.

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.

Jumping ahead by ν steps could be done by multiplying the 128-bit state by an appropriate 128×128 binary matrix modulo 2, but this is not implemented.

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.

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.

Jumping ahead by ν steps could be done by multiplying the 128-bit state by an appropriate 128×128 binary matrix modulo 2, but this is not implemented.

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 $(\text{key, counter}) = (k, i) \in \mathbb{Z}_2^{m+c}$.
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 $(k, 0)$.

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 $(\text{key}, \text{counter}) = (k, i) \in \mathbb{Z}_2^{m+c}$.
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`$\times$`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.

Stateless and stateful functions for `PhiloxRandom` in Tensorflow.

Stateless functions to generate n random numbers:

No memory. The user provides a pair of 32-bit or 64-bit integers as a seed.
These are transformed to get the 64-bit key and 64 most significant bits of the counter.
Then the low bits of the counter go from 0 to $n-1$.
Further calls with the same seeds return the same random numbers.

Stateless and stateful functions for `PhiloxRandom` in Tensorflow.

Stateless functions to generate n random numbers:
No memory. The user provides a pair of 32-bit or 64-bit integers as a seed.
These are transformed to get the 64-bit key and 64 most significant bits of the counter.
Then the low bits of the counter go from 0 to $n - 1$.
Further calls with the same seeds return the same random numbers.

Stateful functions: Provide multiple streams with their own states (memory).
A global seed set at the beginning affects all streams.
Then an operational seed given as a parameter acts as a stream identifier.
When a new one is used, it defines a new stream.
Whenever one is reused, the counter for that stream just continues to increase.
When no operational seed is given, the system uses a default one.

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 ℓ is identified by a ℓ-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.

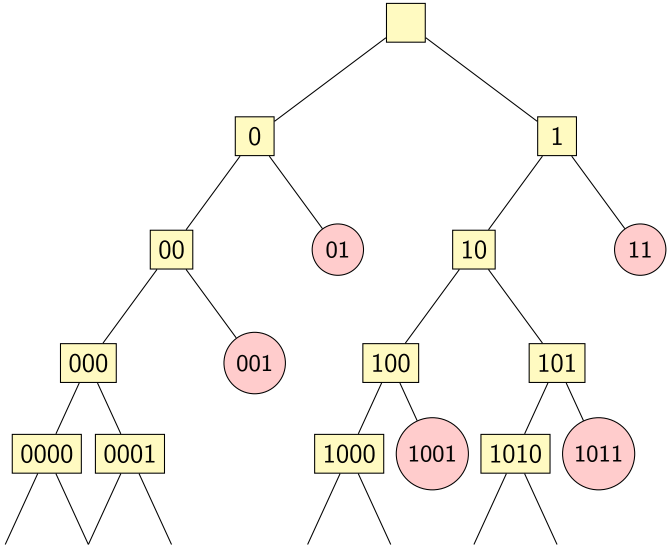
In their design, a tree node can either produce an output or create a split, but not both.

Claessen and Palka (2013) designed a splittable RNG defined by a binary tree. Any node at level ℓ is identified by a ℓ-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

Empirical statistical Tests

Hypothesis $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \ldots\}$ are i.i.d. $U(0,1)$ r.v.'s".
We know that $\mathcal{H}_0$ is false, but can we detect it ?

Empirical statistical Tests

Hypothesis $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \ldots\}$ are i.i.d. $U(0,1)$ r.v.'s".
We know that $\mathcal{H}_0$ is false, but can we detect it ?

Test:
— Define a statistic T, function of the u_i, whose distribution under $\mathcal{H}_0$ is known (or approx.).
— Reject $\mathcal{H}_0$ if value of T is too extreme. If suspect, can repeat.

Different tests detect different deficiencies.

Empirical statistical Tests

Hypothesis $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \ldots\}$ are i.i.d. $U(0,1)$ r.v.'s".
We know that $\mathcal{H}_0$ is false, but can we detect it ?

Test:
— Define a statistic T, function of the u_i, whose distribution under $\mathcal{H}_0$ is known (or approx.).
— Reject $\mathcal{H}_0$ if value of T is too extreme. If suspect, can repeat.

Different tests detect different deficiencies.

Utopian ideal: T mimics the r.v. of practical interest. Not easy.

Ultimate dream: Build an algorithmic RNG that passes all the tests? Formally impossible.

Empirical statistical Tests

Hypothesis $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \dots\}$ are i.i.d. $U(0,1)$ r.v.'s".
We know that $\mathcal{H}_0$ is false, but can we detect it ?

Test:
— Define a statistic T, function of the u_i, whose distribution under $\mathcal{H}_0$ is known (or approx.).
— Reject $\mathcal{H}_0$ if value of T is too extreme. If suspect, can repeat.

Different tests detect different deficiencies.

Utopian ideal: T mimics the r.v. of practical interest. Not easy.

Ultimate dream: Build an algorithmic RNG that passes all the tests? Formally impossible.

Compromise: Build an RNG that passes most reasonable tests.
Tests that fail are hard to find.
Formalization: computational complexity framework.

Example: the serial test

To challenge $\mathcal{H}_0$, we construct n disjoint vectors of t successive outputs:

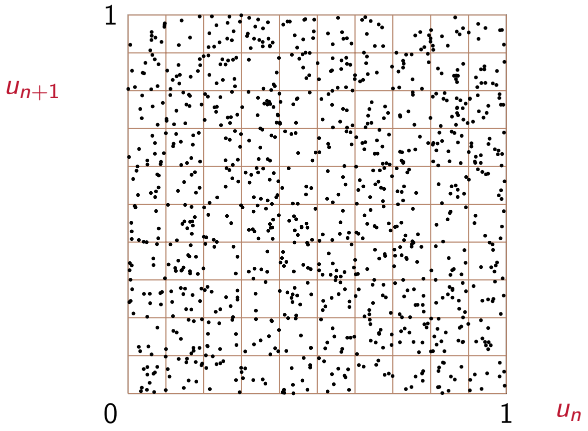
$$(u_0, \dots, u_{t-1}), (u_t, \dots, u_{2t-1}), \dots, (u_{(n-1)t}, \dots, u_{nt-1}).$$

Then partition the unit cube $(0,1)^t$ into k rectangular boxes of equal size $1/k$, count how many points fall in each box, compute the value x of the corresponding chi-square test statistic, and report its p-value

$$p(x) = \mathbb{P}[\chi^2 > x \mid \mathcal{H}_0.]$$

We reject $\mathcal{H}_0$ if $p(x)$ is too close to 0 (x is too large, lack of uniformity).

Example: the serial test in $t = 2$ dimensions



Throw $n = 1000$ points in $k = 100$ boxes. Compute chi-square and p-value.

Serial test: improvements

Overlapping test. To save work, we can construct the points with (circular) overlapping:

$$(u_0, \ldots, u_{t-1}), (u_1, \ldots, u_t), (u_2, \ldots, u_{t+1}), \ldots, (u_{n-2}, u_{n-1}, u_0, \ldots, u_{t-3}), (u_{n-1}, u_0, \ldots, u_{t-2}).$$

The statistic no longer has a chi-square distribution, but we can compute the p-value.

Serial test: improvements

Overlapping test. To save work, we can construct the points with (circular) overlapping:

$$(u_0, \ldots, u_{t-1}), (u_1, \ldots, u_t), (u_2, \ldots, u_{t+1}), \ldots, (u_{n-2}, u_{n-1}, u_0, \ldots, u_{t-3}), (u_{n-1}, u_0, \ldots, u_{t-2}).$$

The statistic no longer has a chi-square distribution, but we can compute the p-value.

Collision test. Permits one to have much more boxes or fewer points. Test statistic:

$$C = \text{number of collisions}.$$

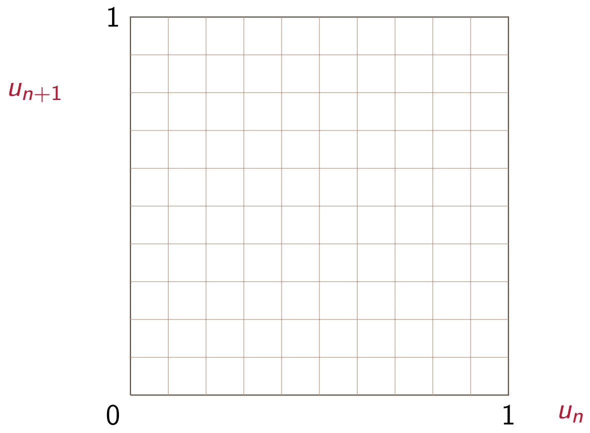
Under $\mathcal{H}_0$, $C \approx$ Poisson of mean $\lambda = n^2/(2k)$, if k is large and λ is small.

If we observe c collisions, we compute the p-values:

$$p^+(c) = \mathbb{P}[X \geq c \mid X \sim \text{ Poisson}(\lambda)], \qquad p^-(c) = \mathbb{P}[X \leq c \mid X \sim \text{ Poisson}(\lambda)].$$

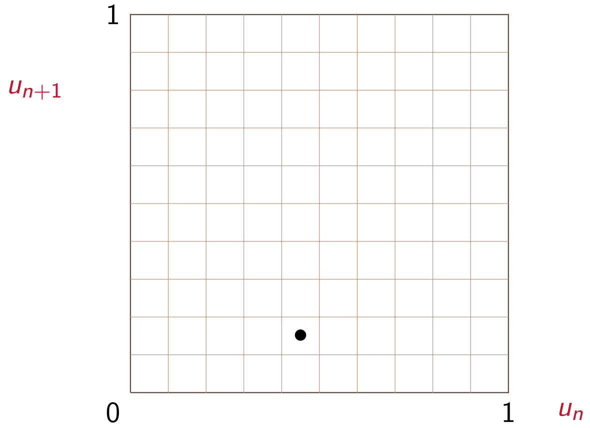
We reject $\mathcal{H}_0$ if $p^+(c)$ is near 0 or $p^-(c)$ is near 1.

Example: A collision test



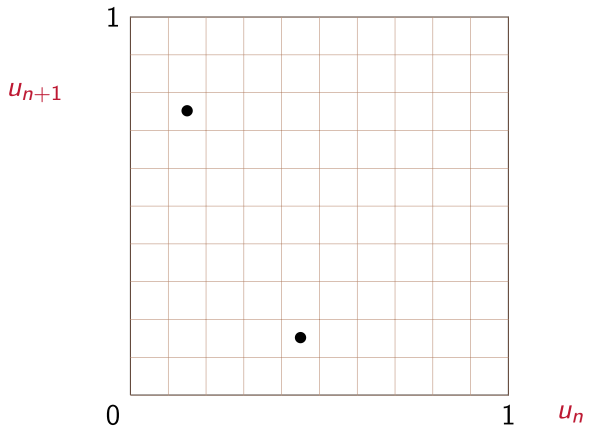
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



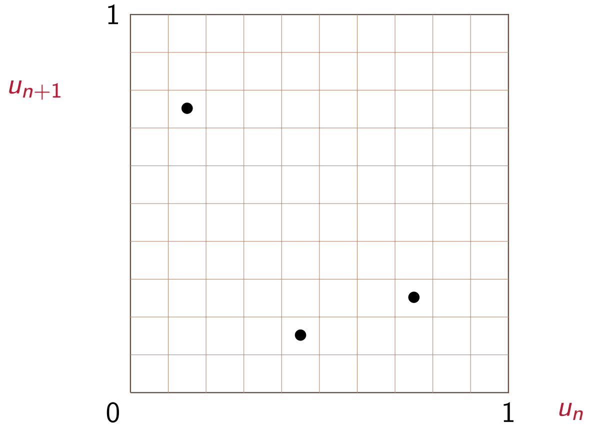
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



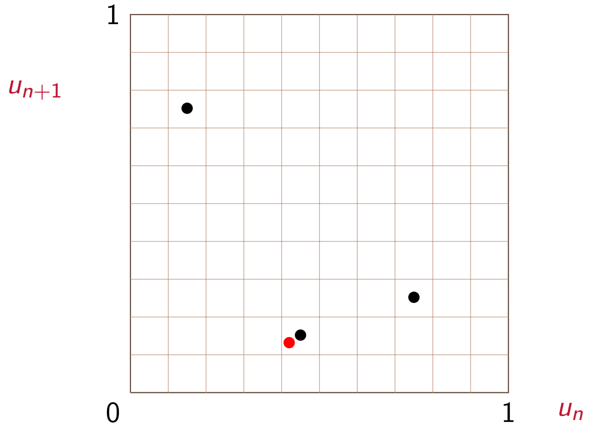
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



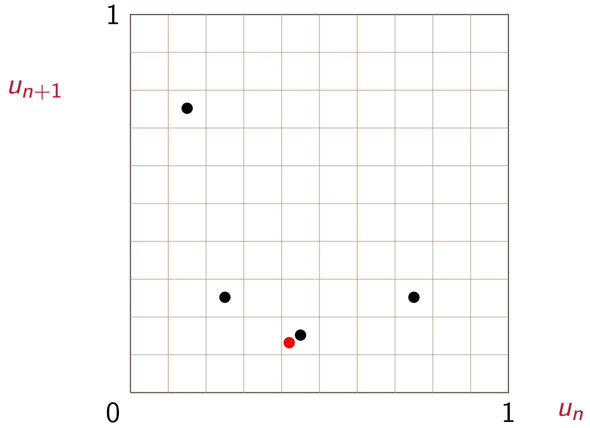
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



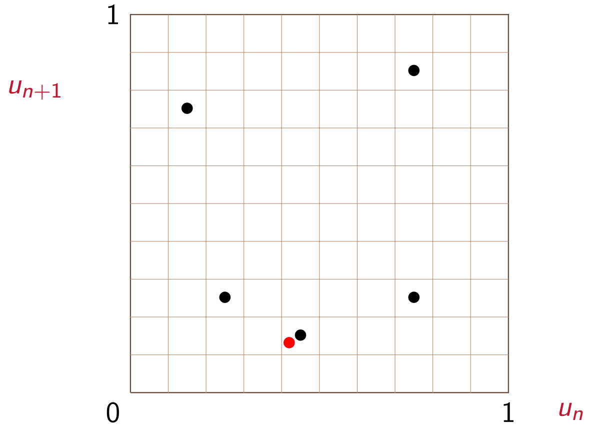
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



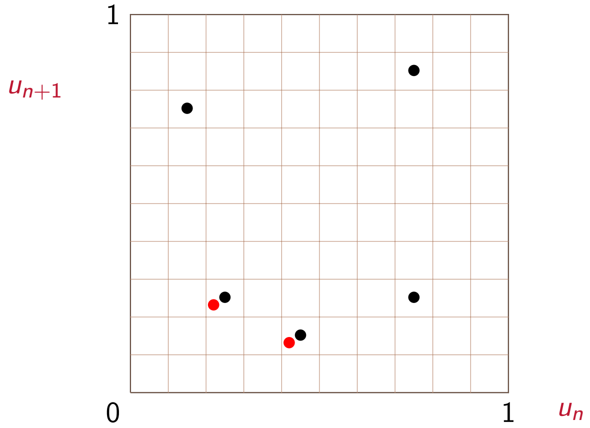
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



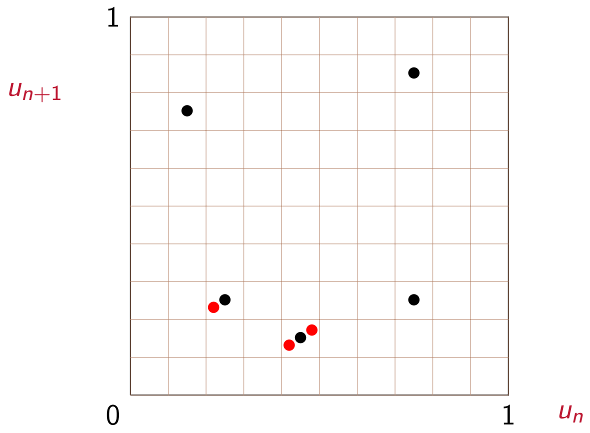
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



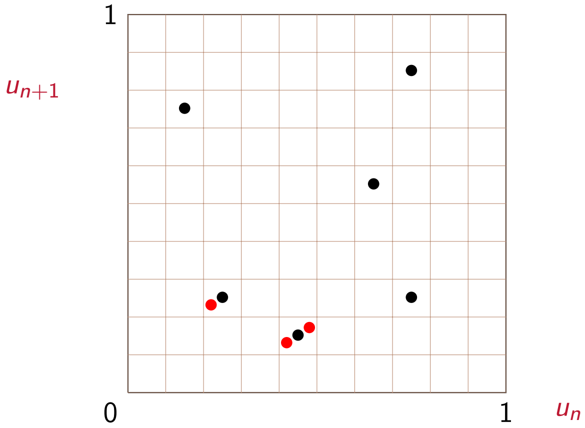
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



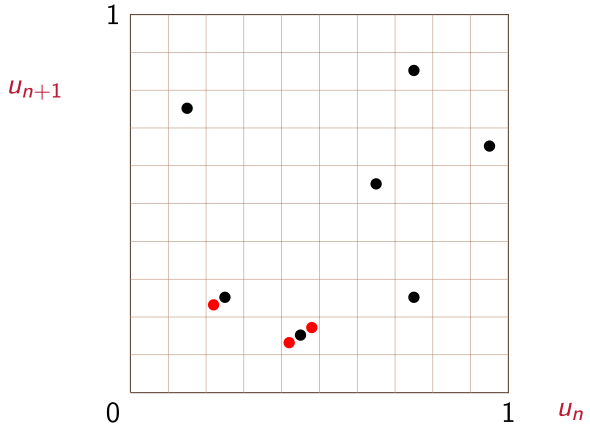
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



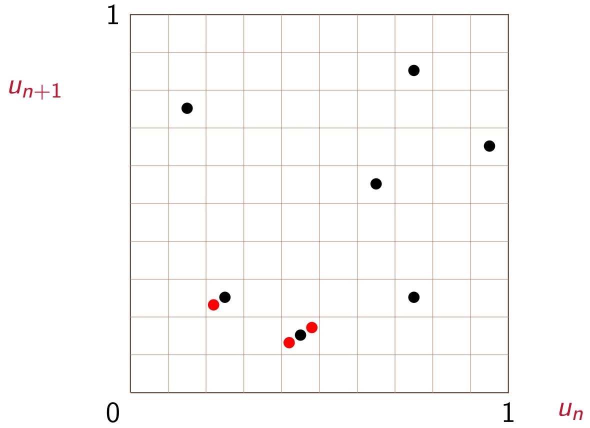
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



Throw $n = 10$ points in $k = 100$ boxes.

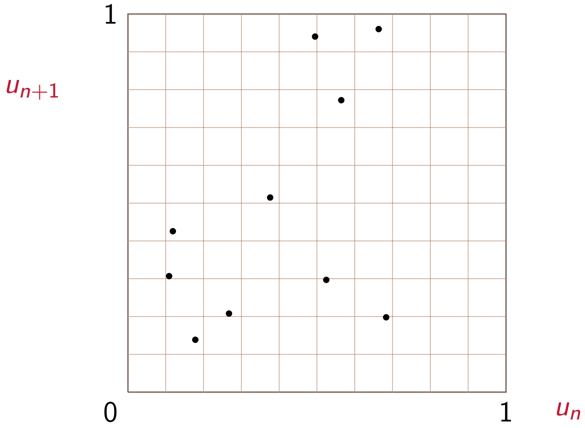
Example: A collision test



Throw $n = 10$ points in $k = 100$ boxes.

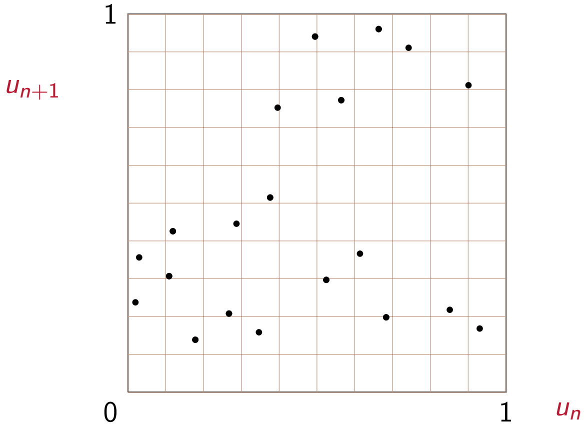
Here we observe 3 collisions. $\mathbb{P}[C \geq 3 \mid \mathcal{H}_0] \approx 0.144$.

Example: LCG with $m = 101$ and $a = 12$:



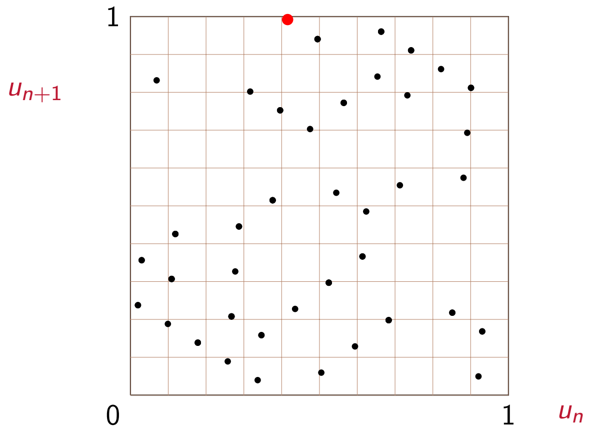
n	λ	C	$p^-(C)$
10	1/2	0	0.6281

Example: LCG with $m = 101$ and $a = 12$:



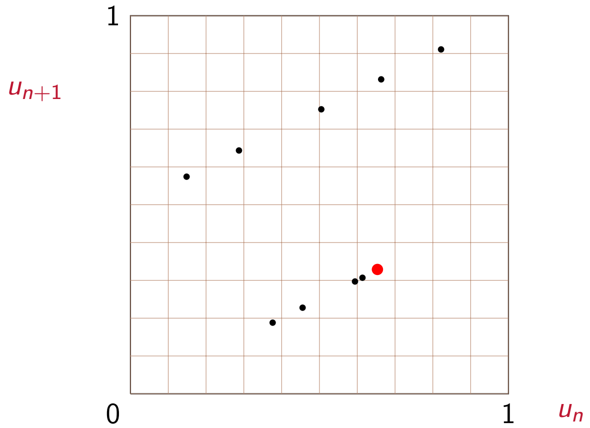
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304

Example: LCG with $m = 101$ and $a = 12$:



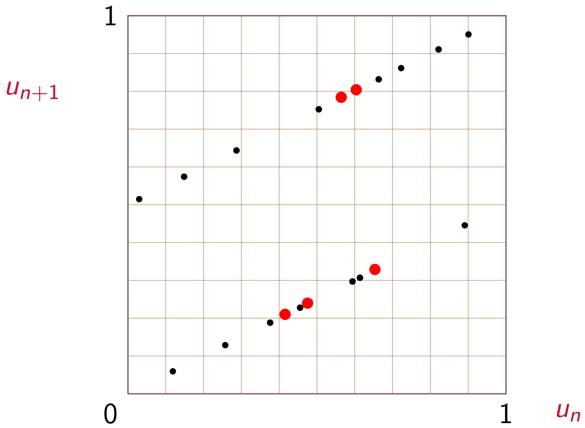
n	λ	C	$p^-(C)$
10	$1/2$	0	0.6281
20	2	0	0.1304
40	8	1	0.0015

LCG with $m = 101$ and $a = 51$ (collisions are not in red):



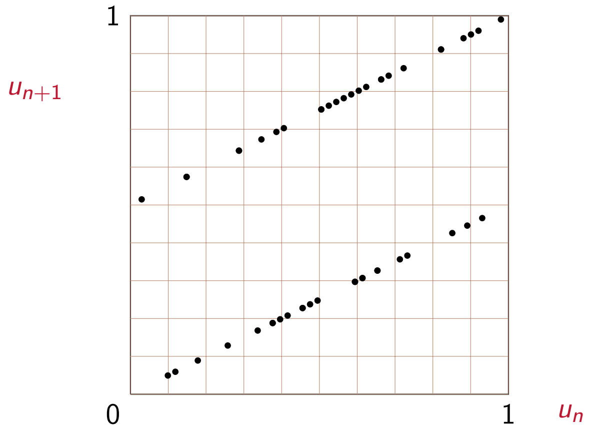
n	λ	C	$p^+(C)$
10	1/2	1	0.3718

LCG with $m = 101$ and $a = 51$ (collisions are not in red):



n	λ	C	$p^+(C)$
10	1/2	1	0.3718
20	2	5	0.0177

LCG with $m = 101$ and $a = 51$ (collisions are not in red):



n	λ	C	$p^+(C)$
10	1/2	1	0.3718
20	2	5	0.0177
40	8	20	2.2×10^{-9}

Subtract-with-borrow RNG in (older) Mathematica

For this RNG, there is a very strong dependence between the coordinates of the points $(U_i, U_{i+20}, U_{i+24})$. It turns out that all these points fall on only two planes in $[0,1]^3$! This can be detected by the collision test, as follows.

Subtract-with-borrow RNG in (older) Mathematica

For this RNG, there is a very strong dependence between the coordinates of the points $(U_i, U_{i+20}, U_{i+24})$. It turns out that all these points fall on only two planes in $[0,1]^3$! This can be detected by the collision test, as follows.

For the unit cube $[0,1)^3$, divide each axis in $d = 100$ equal intervals.
This gives $k = 100^3 = 1$ million boxes.

Generate $n = 10\,000$ vectors in 25 dimensions: $(U_0, \ldots, U_{24})$.
For each, note the box where (U_0, U_{20}, U_{24}) falls.
Here, $\lambda = n^2/(2k) = 50$.

Subtract-with-borrow RNG in (older) Mathematica

For this RNG, there is a very strong dependence between the coordinates of the points $(U_i, U_{i+20}, U_{i+24})$. It turns out that all these points fall on only two planes in $[0,1]^3$! This can be detected by the collision test, as follows.

For the unit cube $[0,1)^3$, divide each axis in $d = 100$ equal intervals.
This gives $k = 100^3 = 1$ million boxes.

Generate $n = 10\,000$ vectors in 25 dimensions: $(U_0, \ldots, U_{24})$.
For each, note the box where (U_0, U_{20}, U_{24}) falls.
Here, $\lambda = n^2/(2k) = 50$.

Results with SWB: $C = 2070,\ 2137,\ 2100,\ 2104,\ 2127,\ \ldots$.

Subtract-with-borrow RNG in (older) Mathematica

For this RNG, there is a very strong dependence between the coordinates of the points $(U_i, U_{i+20}, U_{i+24})$. It turns out that all these points fall on only two planes in $[0,1]^3$! This can be detected by the collision test, as follows.

For the unit cube $[0,1)^3$, divide each axis in $d = 100$ equal intervals.
This gives $k = 100^3 = 1$ million boxes.

Generate $n = 10\,000$ vectors in 25 dimensions: $(U_0, \ldots, U_{24})$.
For each, note the box where (U_0, U_{20}, U_{24}) falls.
Here, $\lambda = n^2/(2k) = 50$.

Results with SWB: $C = 2070,\ 2137,\ 2100,\ 2104,\ 2127,\ \ldots$.

With MRG32k3a: $C = 41,\ 66,\ 53,\ 50,\ 54,\ \ldots$.

Other examples of tests

Nearest pairs of points in $[0,1)^s$.

Sorting card decks (poker, etc.).

Rank of random binary matrix.

Linear complexity of binary sequence.

Measures of entropy.

Complexity measures based on data compression.

Etc.

For a given class of applications, the most relevant tests would be those that mimic the behavior of what we want to simulate.

The TestU01 software

[L'Ecuyer et Simard, ACM Trans. on Math. Software, 2007].

- A library with a large variety of statistical tests.
  For both algorithmic and physical RNGs.
  Widely used. On my web page.
- Some predefined batteries of tests:
  SmallCrush: quick check, 15 seconds;
  Crush: 96 test statistics, 1 hour;
  BigCrush: 144 test statistics, 6 hours;
  Rabbit: for bit strings.
- Many widely-used generators fail these batteries unequivocally.
- **A new 64-bit version is currently under development (on and off...).**
  Test batteries for multiple streams, for counter-based RNGs, etc.
  Separate batteries for $\mathcal{U}(0,1)$ and for binary sequences.

Some fast recommendable RNGs

Good RNGs for 64-bit CPU

Good implementations exploit the fact that CPUs can perform some operations in parallel.
Combined MRG and combined LFSR generators: MRG32k3a, MRG31k3p, LFSR258 (L 1999)
xoshiro and xoroshiro generators (Blackman and Vigna 2021)

Good RNGs for 32-bit GPU

One processing element does not perform operations in parallel.
Simple 32-bit MRG and multiply-with-carry RNGs.
xoshiro-128+, xoshiro128++.

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.
- **Ongoing and future work:**
  Fast and reliable multi-streams packages for modern 64-bit CPUs in many languages.
  Faster counter-based and splittable RNGs.
  Improved 64-bit version of TestU01, with test batteries for multiple streams, counter-based, and splittable RNGs.

Some References

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.

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.

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.

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., 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.