Multiple Streams of Random Numbers for Parallel Computers: Design and Implementation

Pierre L'Ecuyer

Université de Montréal, Canada
and
Inria–Rennes, France

CEA, Saclay, June 2014

What do we want?

Sequences of numbers that look random.

What do we want?

Sequences of numbers that look random.

Example: Bit sequence (head or tail):

011110100110110101001101100101000111?...

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

What do we want?

Sequences of numbers that look random.

Example: Bit sequence (head or tail):

01111?100110?1?10100110110010100011 1...

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

Uniformity and independance:
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.

What do 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 $1/2$.

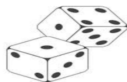
Uniformity and independance:
Example: 8 possibilities for the 3 bits ? ? ?:

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

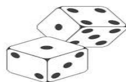
Want a probability of $1/8$ for each, independently of everything else.

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

Sequence of integers from 1 to 6:



Sequence of integers from 1 to 6:



Sequence of integers from 1 to 100: 31, 83, 02, 72, 54, 26, ...

Random permutation:

1 2 3 4 5 6 7

Random permutation:

1 2 3 4 5 6 7

1 2 3 4 6 7 5

Random permutation:

1 2 3 4 5 6 7	
1 2 3 4 6 7	5
1 3 4 6 7	5 2

Random permutation:

1 2 3 4 5 6 7	
1 2 3 4 6 7	5
1 3 4 6 7	5 2
3 4 6 7	5 2 1

Random permutation:

1 2 3 4 5 6 7
1 2 3 4 6 7   5
1 3 4 6 7   5 2
3 4 6 7   5 2 1

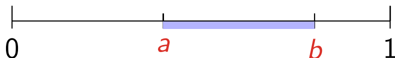
For n objets, choose an integer from 1 to n,
then an integer from 1 to $n-1$, then from 1 to $n-2$, ...
Each permutation should have the same probability.

To shuffle a deck of 52 cards: $52! \approx 2^{226}$ possibilities.

Uniform distribution over $(0, 1)$

For simulation in general, we want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$.

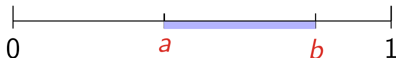
We want $\mathbb{P}[a \leq U_j \leq b] = b - a$.



Uniform distribution over $(0, 1)$

For simulation in general, we want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$.

We want $\mathbb{P}[a \le U_j \le b] = b - a$.



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

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

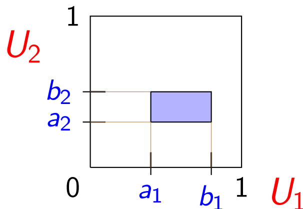
Example: If $F(x) = 1 - e^{-\lambda x}$, take $X = [-\ln(1 - U_j)]/\lambda$.

Independence:

For a random vector $(U_1, \ldots, U_s)$ in s dimensions, we want

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

We want this for any s and any choice of box.

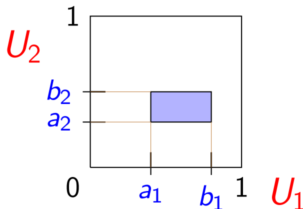


Independence:

For a random vector $(U_1, \ldots, U_s)$ in s dimensions, we want

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

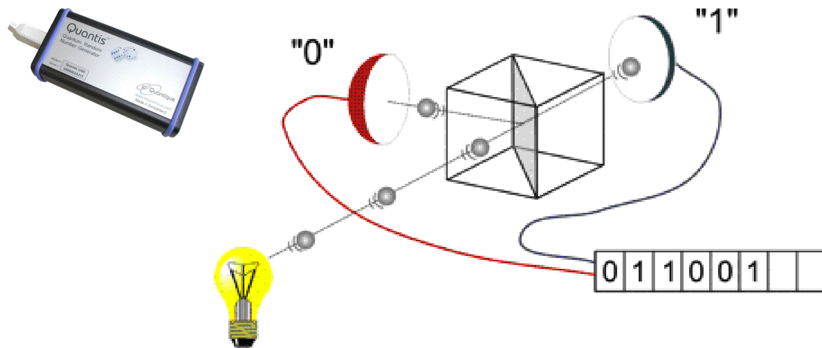
We want this for any s and any choice of box.



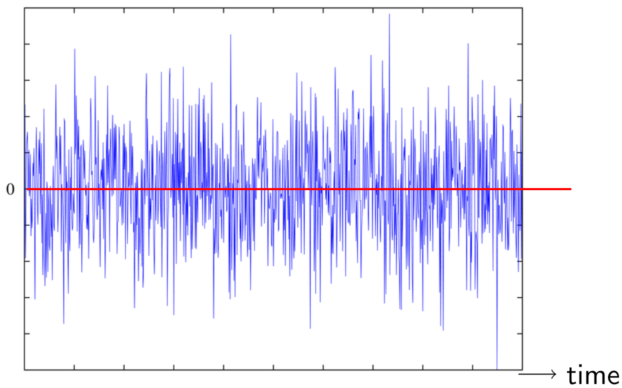
This notion of independent uniform random variables is only a mathematical abstraction. Perhaps it does not exist in the real world!

Physical devices for computers

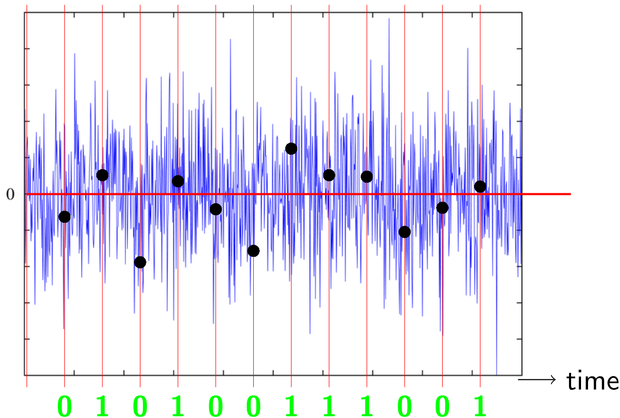
Photon trajectories (sold by **id-Quantique**):



Thermal noise in resistances of electronic circuits

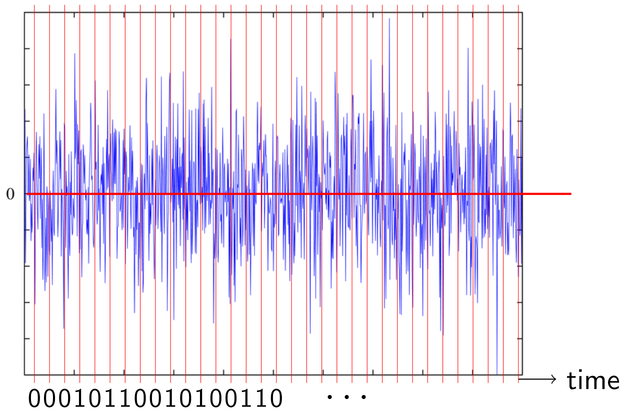


Thermal noise in resistances of electronic circuits



The signal is sampled periodically.

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 not for simulation.
Inconvenient, not reproducible, not always reliable, and no (or little) mathematical analysis.

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

1. Choose x_0 at random in $\{1, \ldots, 100\}$.
2. For $n = 1, 2, 3, ...$, return $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

1. Choose x_0 at random in $\{1, \ldots, 100\}$.
2. For $n = 1, 2, 3, \ldots$, return $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

For example, if $x_0 = 1$:

$$x_1 = (12 \times 1 \bmod 101) = 12,$$

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

1. Choose x_0 at random in $\{1, \ldots, 100\}$.
2. For $n = 1, 2, 3, \ldots$, return $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

For example, if $x_0 = 1$:

$$
\begin{aligned}
x_1 &= (12 \times 1 \bmod 101) = 12, \\
x_2 &= (12 \times 12 \bmod 101) = (144 \bmod 101) = 43,
\end{aligned}
$$

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

1. Choose x_0 at random in $\{1, \ldots, 100\}$.
2. For $n = 1, 2, 3, ...$, return $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

For example, if $x_0 = 1$:

$$\begin{aligned} x_1 &= (12 \times 1 \bmod 101) = 12, \\ x_2 &= (12 \times 12 \bmod 101) = (144 \bmod 101) = 43, \\ x_3 &= (12 \times 43 \bmod 101) = (516 \bmod 101) = 11, \quad \text{etc.} \\ x_n &= 12^n \bmod 101. \end{aligned}$$

Visits all numbers from 1 to 100 exactly once before returning to x_0.

Algorithmic (pseudorandom) generators

Baby-example: Want to imitate random numbers from 1 to 100.

1. Choose x_0 at random in $\{1, \ldots, 100\}$.
2. For $n = 1, 2, 3, \ldots$, return $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

For example, if $x_0 = 1$:

$$
\begin{aligned}
x_1 &= (12 \times 1 \bmod 101) = 12, \\
x_2 &= (12 \times 12 \bmod 101) = (144 \bmod 101) = 43, \\
x_3 &= (12 \times 43 \bmod 101) = (516 \bmod 101) = 11, \quad \text{etc.} \\
x_n &= 12^n \bmod 101.
\end{aligned}
$$

Visits all numbers from 1 to 100 exactly once before returning to x_0.

For real numbers between 0 and 1:

$$
\begin{aligned}
u_1 &= x_1/101 = 12/101 \approx 0.11881188\ldots, \\
u_2 &= x_2/101 = 43/101 \approx 0.42574257\ldots, \\
u_3 &= x_3/101 = 11/101 \approx 0.10891089\ldots, \quad \text{etc.}
\end{aligned}
$$

A Larger Linear Recurrence

Choose 3 integers x_{-2}, x_{-1}, x_0 in $\{0, 1, \ldots, 4294967086\}$ (not all 0).
For $n = 1, 2, \ldots,$ let

$$\begin{aligned} x_n &= (1403580 x_{n-2} - 810728 x_{n-3}) \mod 4294967087, \\ u_n &= x_n / 4294967087. \end{aligned}$$

A Larger Linear Recurrence

Choose 3 integers x_{-2}, x_{-1}, x_0 in $\{0, 1, \ldots, 4294967086\}$ (not all 0).
For $n = 1, 2, \ldots,$ let

$$\begin{aligned} x_n &= (1403580 x_{n-2} - 810728 x_{n-3}) \mod 4294967087, \\ u_n &= x_n / 4294967087. \end{aligned}$$

The sequence $x_0, x_1, x_2, \ldots$ is periodic, with cycle length $4294967087^3 - 1 \approx 2^{96}$, and (x_{n-2}, x_{n-1}, x_n) visits each of the $4294967087^3 - 1$ nonzero triples exactly once when n runs over a cycle.

1. Computer games for kids: the “look” suffices.

1. Computer games for kids: the “look” suffices.

2. Stochastic simulation (Monte Carlo):
Simulate a mathematical model of the behavior of a complex system (hospital, call center, logistic system, financial market, etc.). Must reproduce the relevant statistical properties of the mathematical model.
Algorithmic generators.

1. Computer games for kids: the “look” suffices.

2. Stochastic simulation (Monte Carlo):
Simulate a mathematical model of the behavior of a complex system (hospital, call center, logistic system, financial market, etc.). Must reproduce the relevant statistical properties of the mathematical model.
Algorithmic generators.

3. Lotteries, casino machines, Internet gambling, etc.
It should not be possible (or practical) to make an inference that provides an advantage in guessing the next numbers. Stronger requirements than for simulation.
Algorithmic generators + physical noise.

1. Computer games for kids: the “look” suffices.

2. Stochastic simulation (Monte Carlo):
Simulate a mathematical model of the behavior of a complex system (hospital, call center, logistic system, financial market, etc.). Must reproduce the relevant statistical properties of the mathematical model.
Algorithmic generators.

3. Lotteries, casino machines, Internet gambling, etc.
It should not be possible (or practical) to make an inference that provides an advantage in guessing the next numbers. Stronger requirements than for simulation.
Algorithmic generators + physical noise.

4. Cryptology: Even stronger requirements. Observing any part the output should not help guessing (with reasonable effort) any other part.
Often: very limited computational power and memory.
Nonlinear algorithmic generators with random parameters.

Algorithmic generator

$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$, output function.

s_0, germe (état initial);

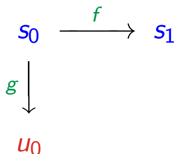
s_0

Algorithmic generator

$\mathcal{S}$, finite state space; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$, output function.

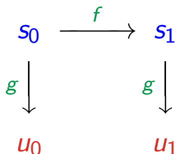
Algorithmic generator

$\mathcal{S}$, finite state space; s_0, germe (état initial);
$f : \mathcal{S} \rightarrow \mathcal{S}$, transition function;
$g : \mathcal{S} \rightarrow [0, 1]$, output function.



Algorithmic generator

$\mathcal{S}$, finite state space; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$, output function.



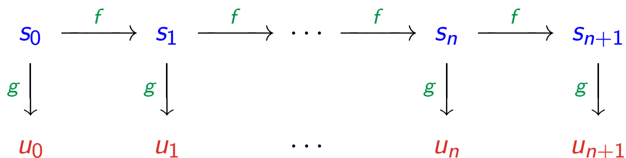
Algorithmic generator

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

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

$g : \mathcal{S} \to [0, 1]$, output function.

s_0, germe (état initial);



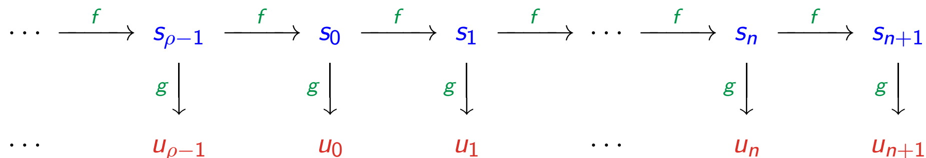
Algorithmic generator

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

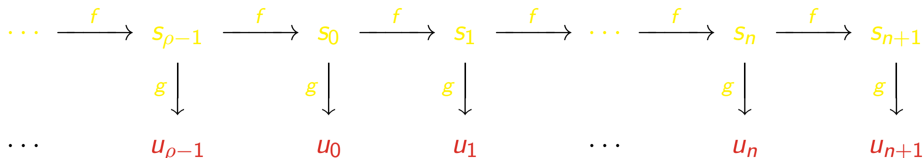
$f : \mathcal{S} \to \mathcal{S}$, transition function;

$g : \mathcal{S} \to [0, 1]$, output function.

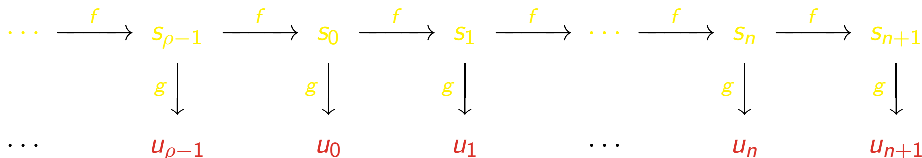
s_0, germe (état initial);



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



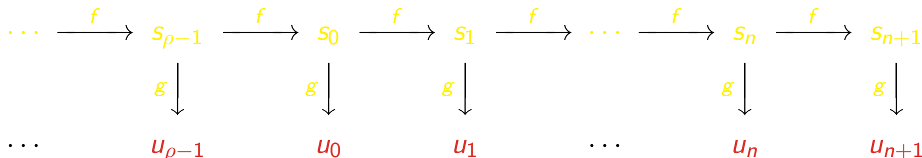
Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independant random variables over $(0, 1)$.



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

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.



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

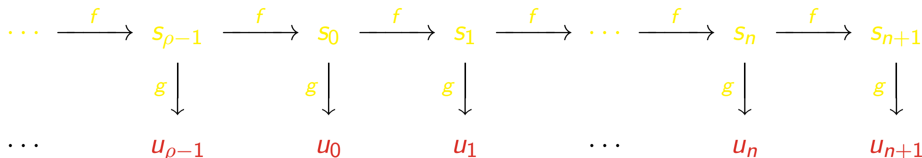
Utopia: passes all statistical tests. Impossible!

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 $\mathbf{u} = (u_0, \ldots, u_{s-1})$.

Number of possibilities cannot exceed $\mathrm{card}(\mathcal{S})$. Ex.: shuffling 52 cards.

Lottery machines: modify the state s_n frequently.



Goal: if we observe only $(u_0, u_1, \ldots)$, difficult to distinguish from a sequence of independant 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 $\mathbf{u} = (u_0, \ldots, u_{s-1})$.

Number of possibilities cannot exceed $\mathrm{card}(\mathcal{S})$. Ex.: shuffling 52 cards.

Lottery machines: modify the state s_n frequently.

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 $[0,1]^s$."

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.

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

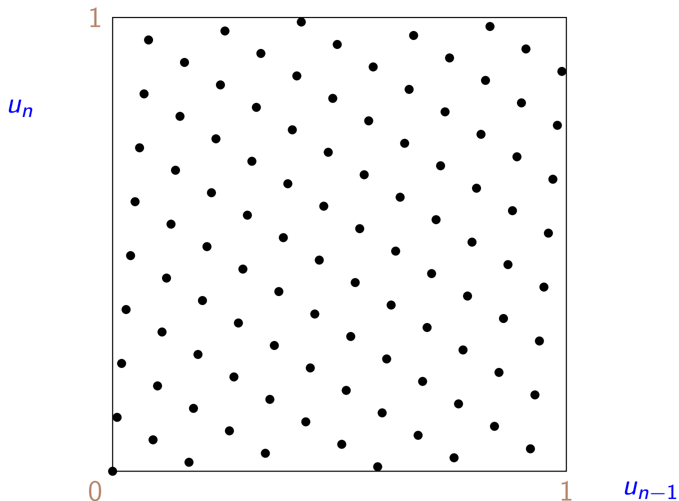
$$\Psi_s = \{\mathbf{u} = (u_0, \ldots, u_{s-1}) = (g(s_0), \ldots, g(s_{s-1})),\ s_0 \in \mathcal{S}\}.$$

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

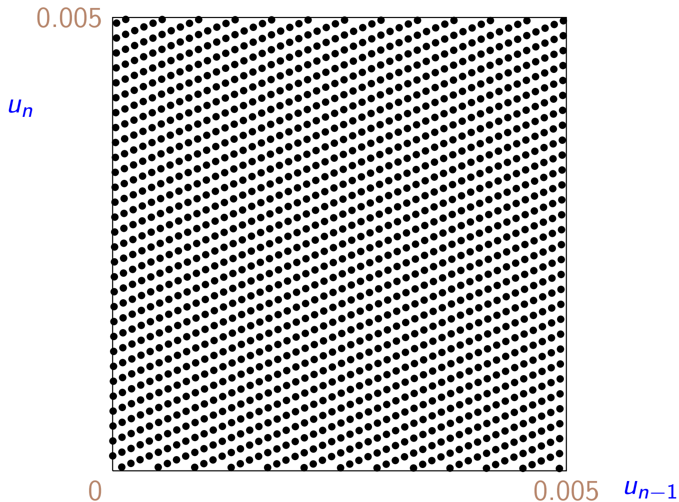
Measure of quality: Ψ_s must cover $[0,1]^s$ very evenly.

Design and analysis:

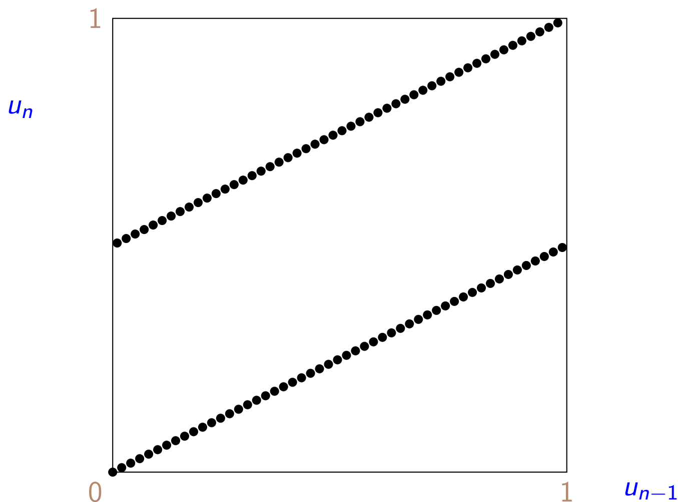
1. Define a uniformity measure for Ψ_s, computable without generating the points explicitly. Linear RNGs.
2. Choose a parameterized family (fast, long period, etc.) and search for parameters that "optimize" this measure.



$$x_n = 12\, x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



$x_n = 4809922\, x_{n-1} \bmod 60466169$ and $u_n = x_n / 60466169$



$x_n = 51\, x_{n-1} \bmod 101; \qquad u_n = x_n/101.$

Good uniformity in one dimension, but not in two!

Myth 1. After 60 years of study and thousands of articles, this problem is certainly solved and RNGs available in popular software must be reliable.

Myth 1. After 60 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 60 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 1. $u_n = (n/2^{1000}) \bmod 1$ for $n = 0, 1, 2, \ldots$.

Exemple 2. Subtract-with-borrow.

A single RNG does not suffice.

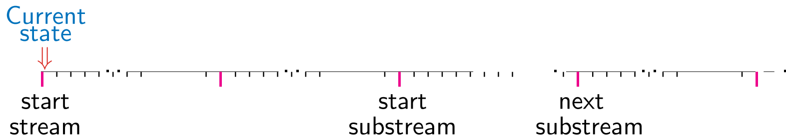
One often needs several independent streams of random numbers, e.g., to:

- Run a simulation on parallel processors.
- Compare similar systems with well synchronized common random numbers (for sensitivity analysis, derivative estimation, optimization). The idea is to simulate the two configurations with the same uniform random numbers U_j used at the same places, as much as possible. This requires good synchronization of the random numbers. Can be complicated to implement and manage when the two configurations do not need the same number of U_j's.

A solution: RNG with multiple streams and substreams.

Can create `RandomStream` objects at will, behave as "independent' streams viewed as virtual RNGs. Can be further partitioned in substreams.

Example: With MRG32k3a generator, streams start 2^{127} values apart, and each stream is partitioned into 2^{51} substreams of length 2^{76}.



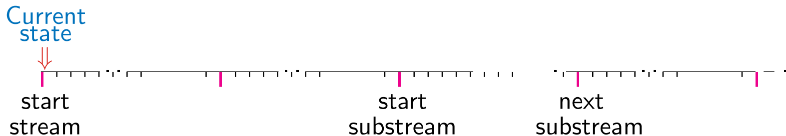
A solution: RNG with multiple streams and substreams.

Can create `RandomStream` objects at will, behave as “independent’ streams viewed as virtual RNGs. Can be further partitioned in substreams.

Example: With MRG32k3a generator, streams start 2^{127} values apart, and each stream is partitioned into 2^{51} substreams of length 2^{76}.

```
RandomStream stream1 = new MRG32k3a();
RandomStream stream2 = new MRG32k3a();
double u = stream1.nextDouble(); ....
double z = NormalGen.nextDouble (stream1, 0.0, 1.0);

stream1.resetNextSubstream(); ....
stream1.resetStartStream();
```



Example of “poor” multiple streams (visible dependence):
Image synthesis on GPUs.
(Thanks to Steve Worley, from Worley Laboratories).

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: $\rho = m^k - 1$.

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: $\rho = m^k - 1$.

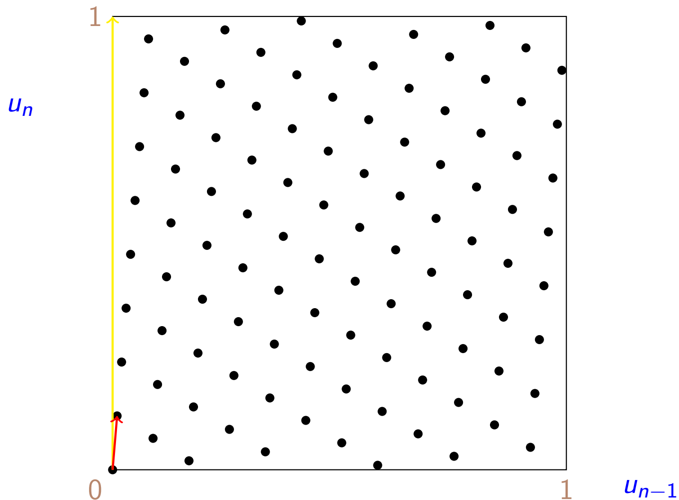
Numerous variants and implementations.

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

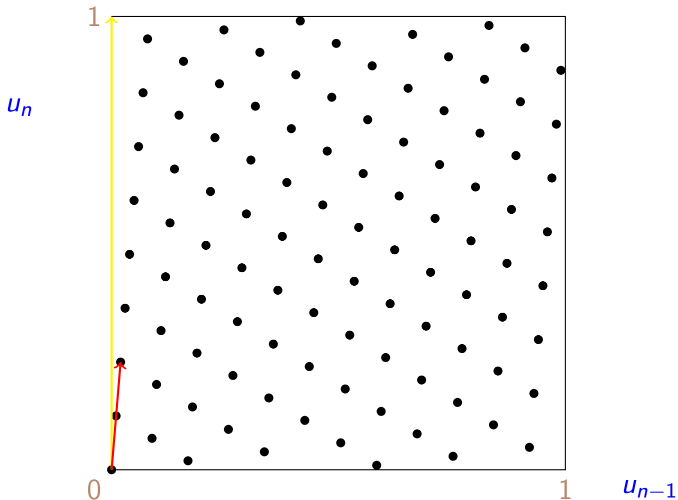
Structure of the points Ψ_s:

$x_0, \ldots, x_{k-1}$ 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.

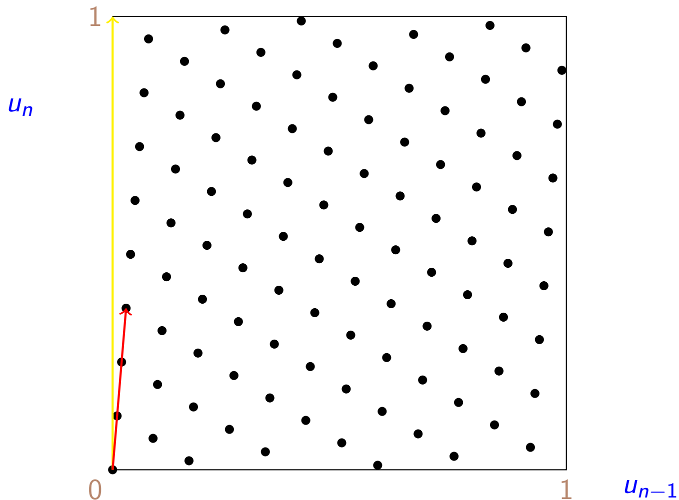
It follows that Ψ_s is a linear space; it is the intersection of a lattice with the unit cube.



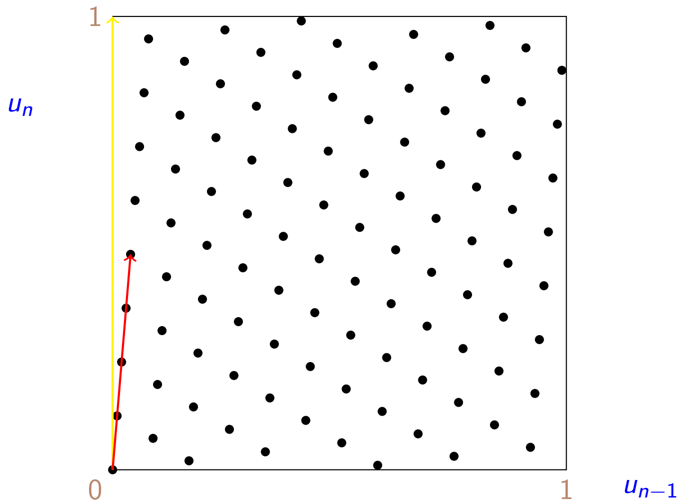
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



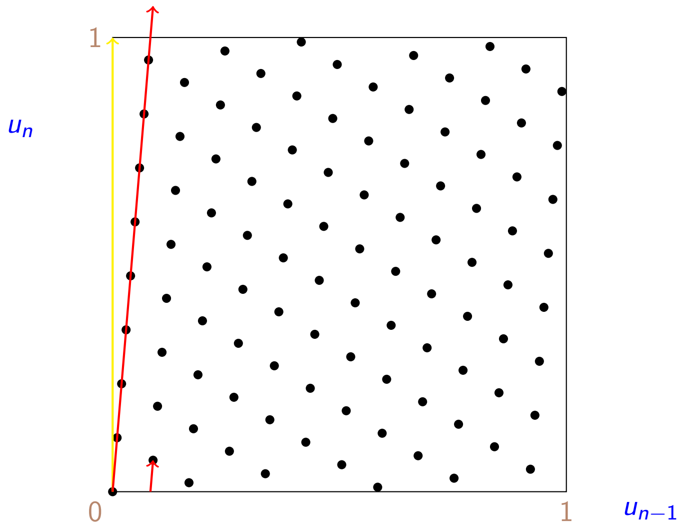
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



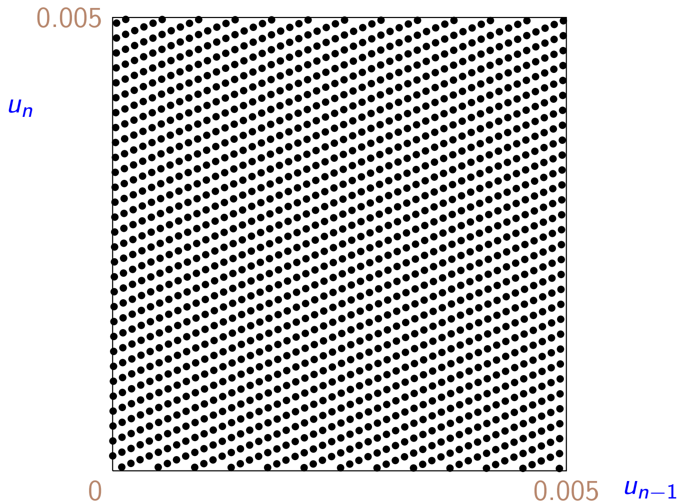
$x_n = 12\, x_{n-1} \bmod 101; \qquad u_n = x_n/101$



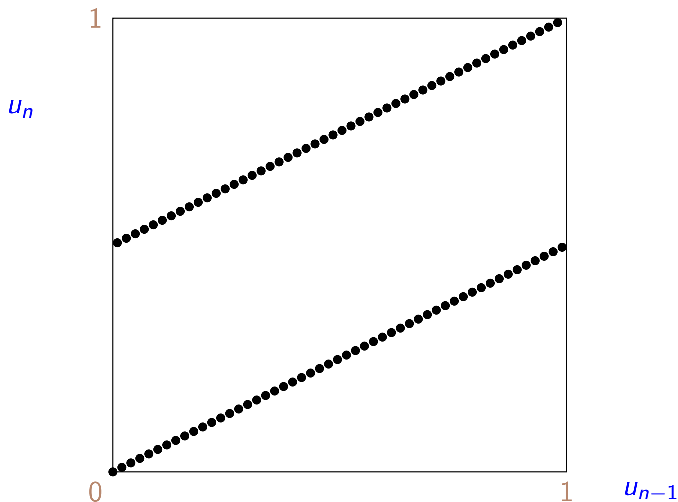
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



$$x_n = 12\, x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



$$x_n = 4809922\, x_{n-1} \bmod 60466169 \text{ and } u_n = x_n/60466169$$



$x_n = 51\, x_{n-1} \bmod 101; \qquad u_n = x_n/101.$

Good uniformity in one dimension, but not in two!

Example: lagged-Fibonacci

$$x_n = (x_{n-r} + x_{n-k}) \mod m.$$

Example: lagged-Fibonacci

$$x_n = (x_{n-r} + x_{n-k}) \mod m.$$

Very fast, but bad. All points $(u_n, u_{n+k-r}, u_{n+k})$ belong to only two parallel planes in $[0,1)^3$.

Example: subtract-with-borrow (SWB)

State $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ where $x_n \in \{0, \ldots, 2^{31} - 1\}$ and $c_n \in \{0, 1\}$:

$$
\begin{aligned}
x_n &= (x_{n-8} - x_{n-48} - c_{n-1}) \bmod 2^{31}, \\
c_n &= 1 \ \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \qquad c_n = 0 \text{ otherwise}, \\
u_n &= x_n / 2^{31},
\end{aligned}
$$

Period $\rho \approx 2^{1479} \approx 1.67 \times 10^{445}$.

Example: subtract-with-borrow (SWB)

State $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ where $x_n \in \{0, \ldots, 2^{31}-1\}$ and $c_n \in \{0, 1\}$:

$$
\begin{aligned}
x_n &= (x_{n-8} - x_{n-48} - c_{n-1}) \bmod 2^{31}, \\
c_n &= 1 \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \qquad c_n = 0 \text{ otherwise}, \\
u_n &= x_n / 2^{31},
\end{aligned}
$$

Period $\rho \approx 2^{1479} \approx 1.67 \times 10^{445}$.

In Mathematica versions ≤ 5.2:
modified SWB with output $\tilde{u}_n = x_{2n}/2^{62} + x_{2n+1}/2^{31}$.

Great generator?

Example: subtract-with-borrow (SWB)

State $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ where $x_n \in \{0, \ldots, 2^{31} - 1\}$ and $c_n \in \{0, 1\}$:

$$
\begin{aligned}
x_n &= (x_{n-8} - x_{n-48} - c_{n-1}) \bmod 2^{31}, \\
c_n &= 1 \ \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \qquad c_n = 0 \text{ otherwise}, \\
u_n &= x_n / 2^{31},
\end{aligned}
$$

Period $\rho \approx 2^{1479} \approx 1.67 \times 10^{445}$.

In Mathematica versions ≤ 5.2:
modified SWB with output $\tilde{u}_n = x_{2n}/2^{62} + x_{2n+1}/2^{31}$.

Great generator? No, not at all; very bad...

All points $(u_n, u_{n+40}, u_{n+48})$ belong to only two parallel planes in $[0,1)^3$.

Ferrenberg et Landau (1991). "Critical behavior of the three-dimensional Ising model: A high-resolution Monte Carlo study."

Ferrenberg, Landau et Wong (1992). "Monte Carlo simulations: Hidden errors from "good" random number generators."

All points $(u_n, u_{n+40}, u_{n+48})$ belong to only two parallel planes in $[0,1)^3$.

Ferrenberg et Landau (1991). "Critical behavior of the three-dimensional Ising model: A high-resolution Monte Carlo study."

Ferrenberg, Landau et Wong (1992). "Monte Carlo simulations: Hidden errors from "good" random number generators."

Tezuka, L'Ecuyer, and Couture (1993). "On the Add-with-Carry and Subtract-with-Borrow Random Number Generators."

Couture and L'Ecuyer (1994) "On the Lattice Structure of Certain Linear Congruential Sequences Related to AWC/SWB Generators."

Combined.

Two [or more] MRGs in parallel:

$$\begin{aligned} x_{1,n} &= (a_{1,1}x_{1,n-1} + \cdots + a_{1,k}x_{1,n-k}) \mod m_1, \\ x_{2,n} &= (a_{2,1}x_{2,n-1} + \cdots + a_{2,k}x_{2,n-k}) \mod m_2. \end{aligned}$$

One possible combinaison:

$$z_n := (x_{1,n} - x_{2,n}) \mod m_1; \qquad u_n := z_n/m_1;$$

Combined.

Two [or more] MRGs in parallel:

$$\begin{aligned} x_{1,n} &= (a_{1,1}x_{1,n-1} + \cdots + a_{1,k}x_{1,n-k}) \mod m_1, \\ x_{2,n} &= (a_{2,1}x_{2,n-1} + \cdots + a_{2,k}x_{2,n-k}) \mod m_2. \end{aligned}$$

One possible combinaison:

$$z_n := (x_{1,n} - x_{2,n}) \mod m_1; \qquad u_n := z_n/m_1;$$

L'Ecuyer (1996): the sequence $\{u_n,\ n \geq 0\}$ is also the output of an MRG of modulus $m = m_1 m_2$, with small added "noise". The period can reach $(m_1^k - 1)(m_2^k - 1)/2$.

Permits one to implement efficiently an MRG with large m and several large nonzero coefficients.

Parameters: L'Ecuyer (1999); L'Ecuyer et Touzin (2000).
Implementations with multiple streams.

A recommended generator: MRG32k3a

Choose 6 integers:
x_{-2}, x_{-1}, x_0 in $\{0, 1, \ldots, 4294967086\}$ (not all 0) and
y_{-2}, y_{-1}, y_0 in $\{0, 1, \ldots, 4294944442\}$ (not all 0). For $n = 1, 2, \ldots,$ let

$$
\begin{aligned}
x_n &= (1403580 x_{n-2} - 810728 x_{n-3}) \mod 4294967087, \\
y_n &= (527612 y_{n-1} - 1370589 y_{n-3}) \mod 4294944443, \\
u_n &= [(x_n - y_n) \mod 4294967087]/4294967087.
\end{aligned}
$$

A recommended generator: MRG32k3a

Choose 6 integers:
x_{-2}, x_{-1}, x_0 in $\{0, 1, \ldots, 4294967086\}$ (not all 0) and
y_{-2}, y_{-1}, y_0 in $\{0, 1, \ldots, 4294944442\}$ (not all 0). For $n = 1, 2, \ldots,$ let

$$\begin{aligned} x_n &= (1403580 x_{n-2} - 810728 x_{n-3}) \mod 4294967087, \\ y_n &= (527612 y_{n-1} - 1370589 y_{n-3}) \mod 4294944443, \\ u_n &= [(x_n - y_n) \mod 4294967087]/4294967087. \end{aligned}$$

(x_{n-2}, x_{n-1}, x_n) visits each of the $4294967087^3 - 1$ possible values.
(y_{n-2}, y_{n-1}, y_n) visits each of the $4294944443^3 - 1$ possible values.

The sequence $u_0, u_1, u_2, \ldots$ is periodic, with 2 cycles of period

$$\approx 2^{191} \approx 3.1 \times 10^{57}.$$

A recommended generator: MRG32k3a

Choose 6 integers:
x_{-2}, x_{-1}, x_0 in $\{0, 1, \ldots, 4294967086\}$ (not all 0) and
y_{-2}, y_{-1}, y_0 in $\{0, 1, \ldots, 4294944442\}$ (not all 0). For $n = 1, 2, \ldots,$ let

$$
\begin{aligned}
x_n &= (1403580 x_{n-2} - 810728 x_{n-3}) \mod 4294967087, \\
y_n &= (527612 y_{n-1} - 1370589 y_{n-3}) \mod 4294944443, \\
u_n &= [(x_n - y_n) \mod 4294967087]/4294967087.
\end{aligned}
$$

(x_{n-2}, x_{n-1}, x_n) visits each of the $4294967087^3 - 1$ possible values.
(y_{n-2}, y_{n-1}, y_n) visits each of the $4294944443^3 - 1$ possible values.

The sequence $u_0, u_1, u_2, \ldots$ is periodic, with 2 cycles of period

$$\approx 2^{191} \approx 3.1 \times 10^{57}.$$

Robust and reliable generator for simulation.
Used by SAS, R, MATLAB, Arena, Automod, Witness, Spielo gaming, ...

Faster RNG: operations on blocks of bits.

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

$x_{n-1} =$ 00010100101001101100110110100101

Faster RNG: operations on blocks of bits.

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

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

$x_{n-1} =$

```
      00010100101001101100110110100101
10010100101001101100110110100101
      00111101000101011010010011100101
```

Faster RNG: operations on blocks of bits.

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

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

$x_{n-1} =$ 00010100101001101100110110100101

100101 00101001101100110110100101

00111101000101011010010011100101

$B =$ 0011110100010101101 0010011100101

Faster RNG: operations on blocks of bits.

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
000101001010011011      00110110100100
```

Faster RNG: operations on blocks of bits.

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
000101001010011011                  00110110100100
x_n =                               00110110100100011110100010101101
```

Faster RNG: operations on blocks of bits.

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|
                    100101|00101001101100110110100101
                          |00111101000101011010010011100101|
B =                       |             0011110100010101101|0010011100101
x_{n-1}                   |00010100101001101100110110100100|
000101001010011011        |00110110100100
x_n =                     |00110110100100011110100010101101|
```

The first 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.

Faster RNG: operations on blocks of bits.

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
000101001010011011        001101101001000
x_n =                     00110110100100011110100010101101
```

The first 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.

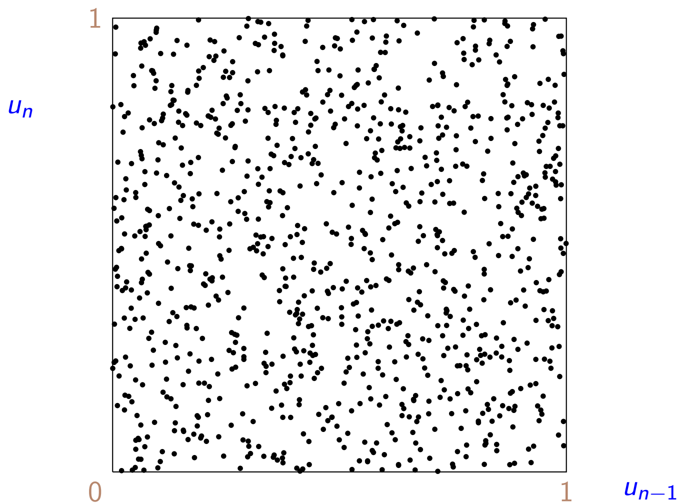
For real numbers in $(0,1)$: $u_n = x_n/(2^{32} + 1)$.

More realistic: LFSR113

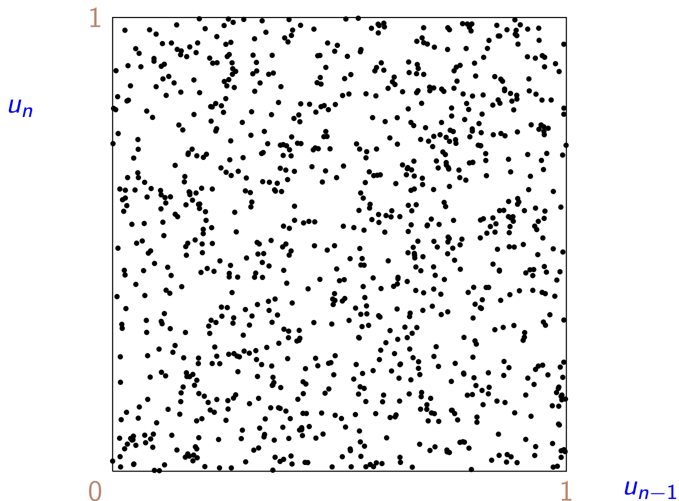
Take 4 recurrences on blocks of 32 bits, in parallel.
The periods are $2^{31} - 1$, $2^{29} - 1$, $2^{28} - 1$, $2^{25} - 1$.

We add these 4 states by a XOR, then we divide by $2^{32} + 1$.
The output has period $\approx 2^{113} \approx 10^{34}$.

Good generator, faster than MRG32k3a, although successive values of bit i of the output obey a linear relationship or order 113, for each i.



1000 points generated by LFSR113



1000 points generated by MRG32k3a + LFSR113 (add mod 1)

General linear recurrences modulo 2

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

General linear recurrences modulo 2

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

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

Special cases: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.

General 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.

Special cases: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.

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

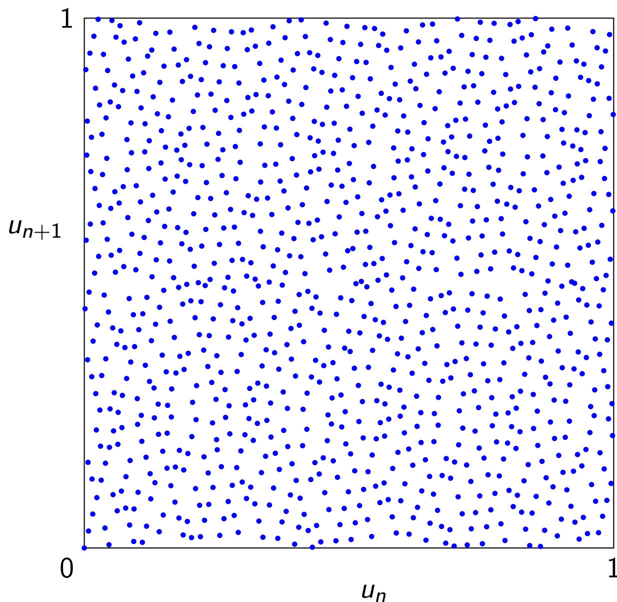
$$x_{n,j} = (\alpha_1 x_{n-1,j} + \cdots + \alpha_k x_{n-k,j}),$$

with characteristic polynomial

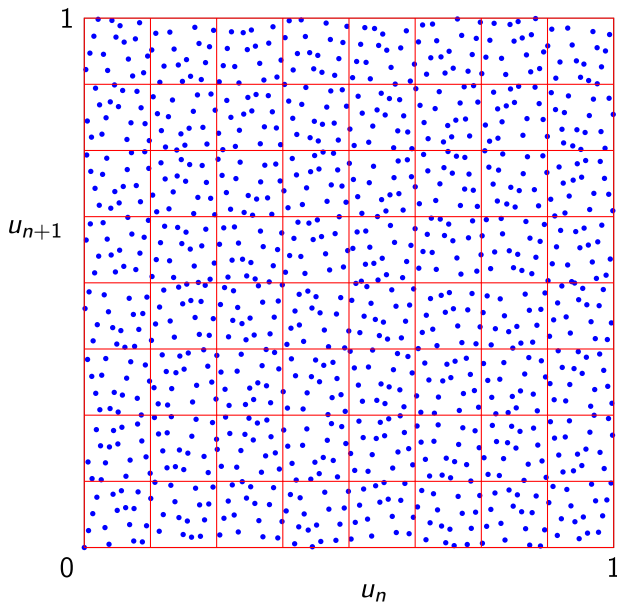
$$P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(\mathbf{A} - z\mathbf{I}).$$

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

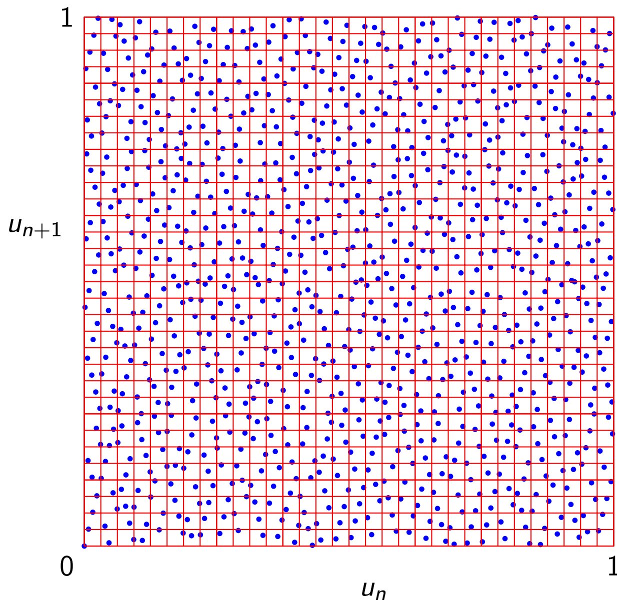
Uniformity measures. Example: $k = 10$, $2^{10} = 1024$ points



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



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



Uniformity measures based on equidistribution.

Example: 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 can be written as $\mathbf{M}\,\mathbf{x}_0$. Each box contains $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. We then say that those points are equidistributed for ℓ bits in s dimensions.

Uniformity measures based on equidistribution.

Example: 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 can be written as $\mathbf{M}\,\mathbf{x}_0$. Each box contains $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. 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.

Example: 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 can be written as $\mathbf{M}\,\mathbf{x}_0$. Each box contains $2^{k-s\ell}$ points of Ψ_s iff $\mathbf{M}$ has (full) rank $s\ell$. 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.

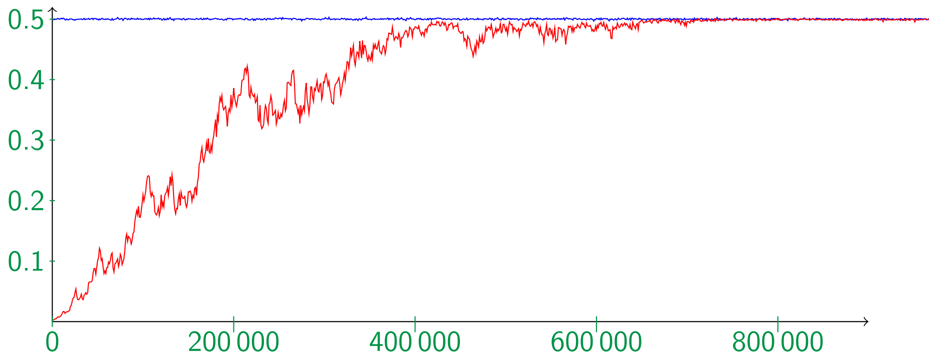
Can be generalized to rectangular boxes...

Examples: LFSR113, Mersenne twister (MT19937), the WELL family, ...

Impact of a matrix A that changes the state too slowly.

Experiment: take an initial state with a single bit at 1.
Try all k possibilities and take the average of the k values of u_n obtained for each n.

WELL19937 vs MT19937; moving average over 1000 iterations.



Jumping Ahead for Linear RNGs

State $\mathbf{x}_n$ evolves as

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

Then

$$\mathbf{x}_{n+\nu} = (\mathbf{A}^{\nu} \mod m) x_n \mod m.$$

The matrix $\mathbf{A}^{\nu} \mod m$ can be precomputed for selected values of ν.

Combined linear/nonlinear generators

All linear generators modulo 2 fail (of course) a statistical test that measures the (binary) linear complexity.

Combined linear/nonlinear generators

All linear generators modulo 2 fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

- to eliminate this linear structure;
- to keep some theoretical guarantees for the uniformity;
- a fast implantation.

Combined linear/nonlinear generators

All linear generators modulo 2 fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

- to eliminate this linear structure;
- to keep some theoretical guarantees for the uniformity;
- a fast implantation.

L'Ecuyer and Granger-Picher (2003): Large linear generator modulo 2 combined with a small nonlinear one, via XOR.

Combined linear/nonlinear generators

All linear generators modulo 2 fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

- to eliminate this linear structure;
- to keep some theoretical guarantees for the uniformity;
- a fast implantation.

L'Ecuyer and Granger-Picher (2003): Large linear generator modulo 2 combined with a small nonlinear one, via XOR.

Theorem: The combination has at least as much equidistribution as the linear component.

Combined linear/nonlinear generators

All linear generators modulo 2 fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

- to eliminate this linear structure;
- to keep some theoretical guarantees for the uniformity;
- a fast implantation.

L'Ecuyer and Granger-Picher (2003): Large linear generator modulo 2 combined with a small nonlinear one, via XOR.

Theorem: The combination has at least as much equidistribution as the linear component.

Empirical tests: excellent behavior, more robust than linear generators.

Counter-Based RNGs

State at step n is just n, so $f(n) = n + 1$, and $g(n)$ is more complicated.
Advantages: trivial to jump ahead, can generate a sequence in any order.

Typically, g is a bijective block cipher encryption algorithm.
Examples: MD5, TEA, SHA, AES, ChaCha, Threefish, etc.
The encoding is often simplified to make the RNG faster.
g : (k-bit counter) $\mapsto$ (k-bit output), period $\rho = 2^k$.
E.g.: $k = 128$ or 256 or 512 or 1024.

Counter-Based RNGs

State at step n is just n, so $f(n) = n + 1$, and $g(n)$ is more complicated.
Advantages: trivial to jump ahead, can generate a sequence in any order.

Typically, g is a bijective block cipher encryption algorithm.
Examples: MD5, TEA, SHA, AES, ChaCha, Threefish, etc.
The encoding is often simplified to make the RNG faster.
g : (k-bit counter) $\mapsto$ (k-bit output), period $\rho = 2^k$.
E.g.: $k = 128$ or 256 or 512 or 1024.

This g has a parameter called the encoding key.
One can use a new counter and a different key for each stream.

Changing one bit in n should change 50% of the output bits on average.

No theoretical analysis for the point sets Ψ_s.
But some of them perform very well in empirical statistical tests.

RNGs on Parallel Processors

Suppose we have several cores (or processing elements, PEs) that can compute in parallel. There could be several thousand PEs.

Each PE can execute one thread or work item (a program fragment) at a time.

In some settings, such as discrete GPU cards, each PE has only a small fast-access memory, and a limited set of instructions. Groups of threads are partitioned in warps or wavefronts of 32 or 64 threads each. All threads in a warp must perform the same instructions on each cycle (SIMT).

One can use several cores to produce a single stream of random numbers (e.g., to fill up a large buffer) at a faster rate. These random numbers can then be used at the host CPU level.

Or one can have different (independent) streams produced and used by the threads. Typically, we want each thread to have its own set of streams, at the software level.

Vectorized RNGs

Typical use: Fill a large array of random numbers.

Saito and Matsumoto (2008, 2013): SIMD version of the Mersenne twister MT19937. Block of successive numbers computed in parallel.

Brent (2007), Nadapalan et al. (2012), Thomas et al. (2009): Similar with xorshift+Weyl and xorshift+sum.

Bradley et al. (2011): CUDA library with multiple streams of flexible length, based on MRG32k3a and MT19937.

Barash and Shchur (2014): C library with several types of RNGs, with jump-ahead facilities.

Each thread running and using one or more streams

On a GPU, the state should be small, some say at most 128 bits.

Some authors suggest counter-based RNGs for this.

Popular RNGs such as LFSR113 and MRG32k3a are also good.

An API for parallel RNGs in OpenCL

In this setting, streams can be created only on the host, and can be used either on the host or on a device (such as a GPU).

Host interface for the MRG32k3a generator

This RNG was proposed by L'Ecuyer (1999). It has period length near 2^{191}, divided into disjoint streams of length $\nu = 2^{127}$. The state is a vector of six 32-bit integers.

An API for parallel RNGs in OpenCL

In this setting, streams can be created only on the host, and can be used either on the host or on a device (such as a GPU).

Host interface for the MRG32k3a generator

This RNG was proposed by L'Ecuyer (1999). It has period length near 2^{191}, divided into disjoint streams of length $\nu = 2^{127}$. The state is a vector of six 32-bit integers.

```
typedef struct {
    unsigned long long Cg[6]; // Current state
    unsigned long long Ig[6]; // Initial state
    } mrg32k3aRandomStream;
```

The status of a stream.

```
int mrg32k3aSetBaseSeed (unsigned long long seed[6]);
```

Optional. Sets initial state of first stream. Default seed is $(12345, 12345, 12345, 12345, 12345, 12345)$.

```
void mrg32k3aChangeStreamsSpacing (long e, long c);
```

Changes the spacing ν between the streams to $\nu = 2^e + c$.

```
mrg32k3aRandomStream* mrg32k3aCreateRandomStream(void);
```

Creates, initializes, and returns a new stream.

```
void mrg32k3aDeleteRandomStream (mrg32k3aRandomStream* stream);
```

Deletes `stream` and frees its memory.

```
mrg32k3aRandomStream* mrg32k3aCreateRandomStreamArray (int n);
```

Creates, initializes, and returns an array of n streams, spaced ν steps apart.

```
void mrg32k3aDeleteRandomStreamArray
    (mrg32k3aRandomStream* streamArray);
```

Deletes the n streams in `streamArray` and frees their memory.

```
void mrg32k3aResetStartStream (mrg32k3aRandomStream* stream);
```

Reinitializes `stream` to its initial state `Ig`.

```
void mrg32k3aResetStartStreamArray
    (mrg32k3aRandomStream* streamArray, int n);
```

Reinitializes all n streams in `streamArray` to their initial states.

```
void mrg32k3aAdvanceStreamState
    (mrg32k3aRandomStream* stream, long e, long c);
```

Advances the state of `stream` by $k = 2^e + c$ values. Avoid this!

```
void mrg32k3aCopyStream (mrg32k3aRandomStream* stream,
       mrg32k3aRandomStream* streamCopy);
```

Copies the status of `stream` into that of `streamCopy`.

```
void mrg32k3aCopyStreamArray (mrg32k3aRandomStream* streamArray,
       int n, mrg32k3aRandomStream* streamArrayCopy);
```

Copies the status of streams in `streamArray`, of size `n`, to those in `streamArrayCopy`.

```
double mrg32k3aRandomU01 (mrg32k3aRandomStream* stream);
```

Returns a $U(0,1)$ random number, using `stream`, after advancing the state by one step.

```
float mrg32k3aRandomU01Float (mrg32k3aRandomStream* stream);
```

Same, but single precision.

```
int mrg32k3aRandomInt (mrg32k3aRandomStream* stream,
       int i, int j);
```

Random integer from uniform dist. over $\{i, i+1, \dots, j\}$.

Interface on Devices

Methods that can be called on a device (such as a GPU):

```
void mrg32k3aResetStartStream (mrg32k3aRandomStream* stream);

void mrg32k3aCopyStreamFromGlobal
        (__global mrg32k3aRandomStream* stream,
        mrg32k3aRandomStream* streamCopy);

void mrg32k3aCopyStreamToGlobal (mrg32k3aRandomStream* stream,
        __global mrg32k3aRandomStream* streamSaved);

double mrg32k3aRandomU01 (mrg32k3aRandomStream* stream);

float mrg32k3aRandomU01float (mrg32k3aRandomStream* stream);

int mrg32k3aRandomInt (mrg32k3aRandomStream* stream,
        int i, int j);
```

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 $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \dots\}$ 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.

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 RNG that passes all the tests? Formally impossible.

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.

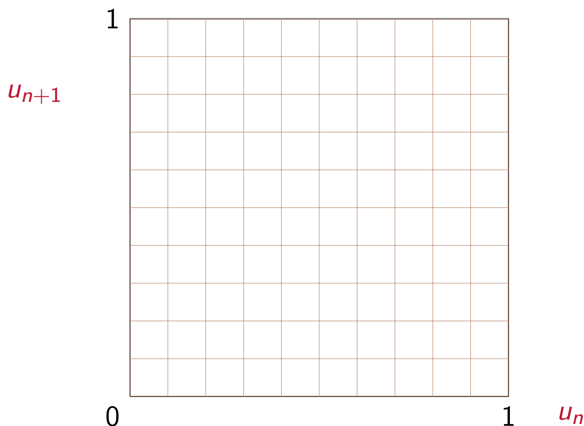
Different tests detect different deficiencies.

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

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

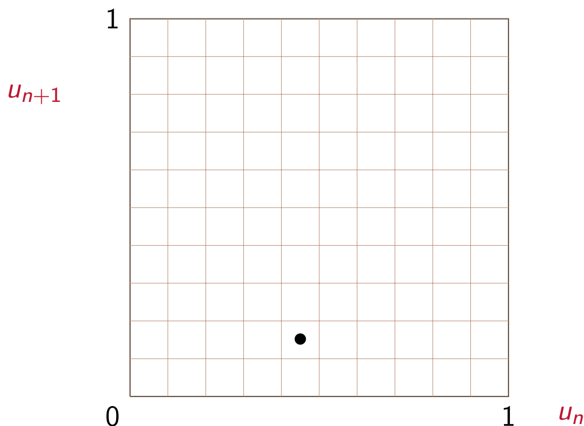
Compromise: Build an RNG that passes most reasonable tests.
Tests that fail are hard to find.
Formalization: computational complexity framework.

Example: A collision test



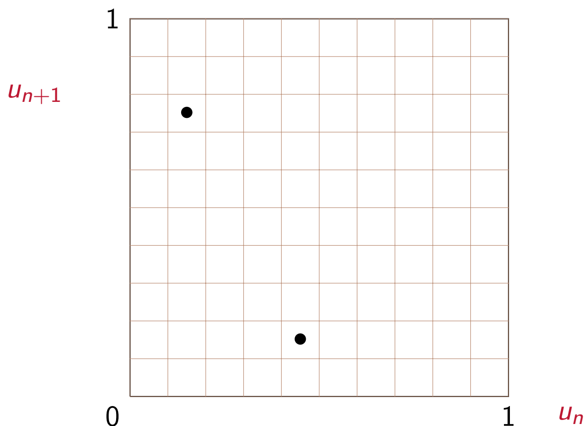
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



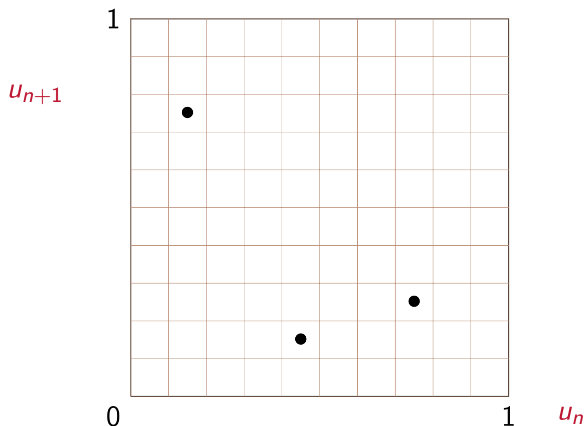
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



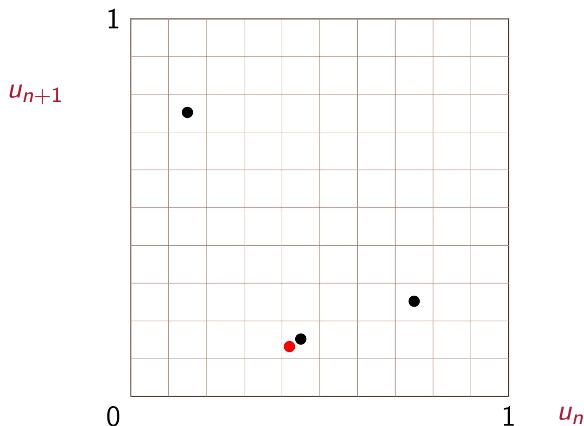
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



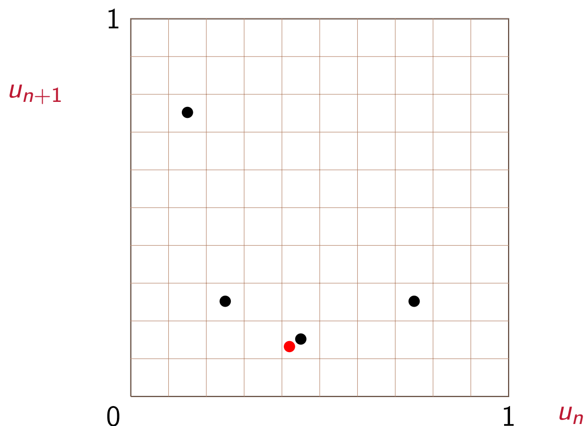
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



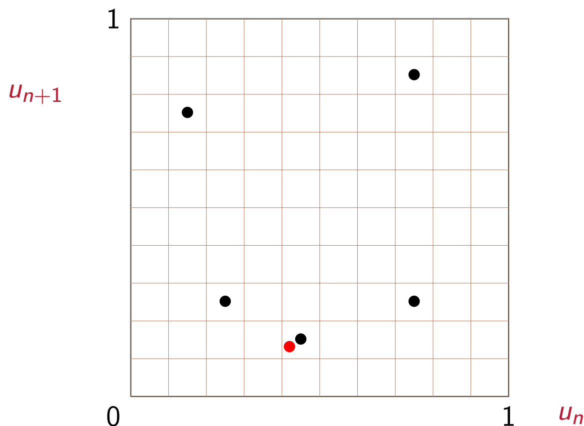
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



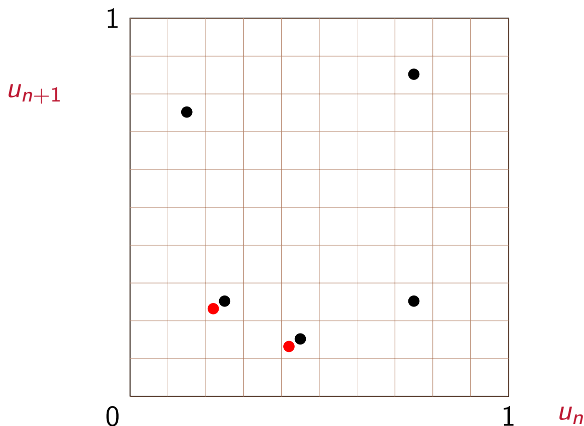
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



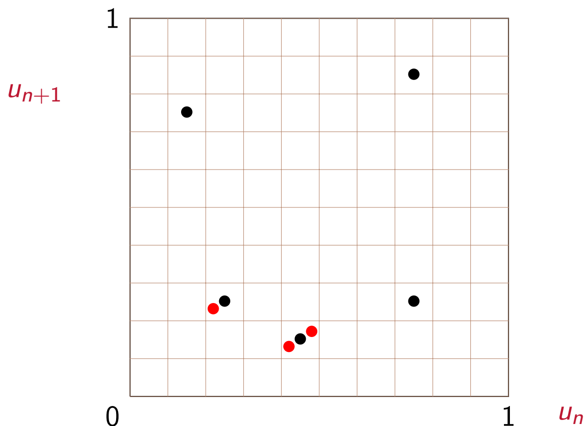
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



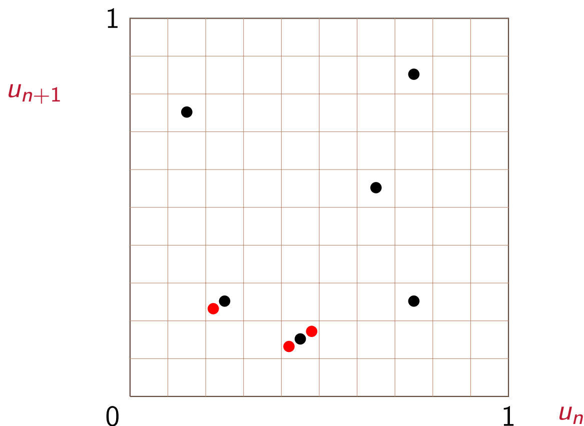
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



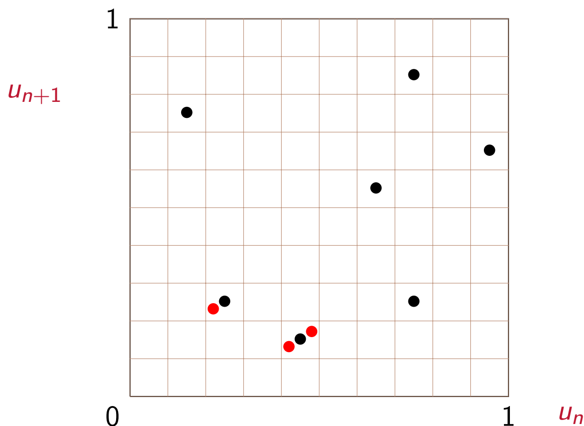
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



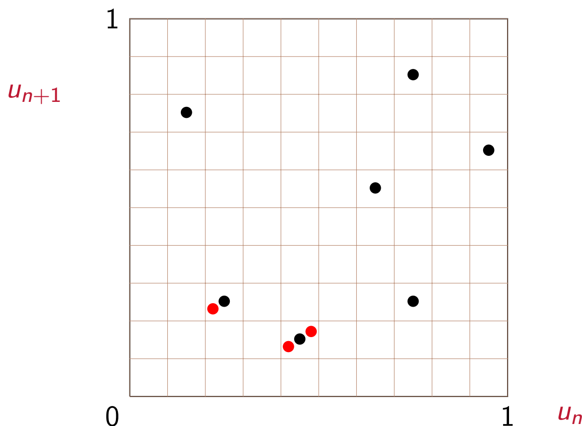
Throw $n = 10$ points in $k = 100$ boxes.

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

Collision test

Partition $[0,1)^s$ in $k = d^s$ cubic boxes of equal size.
Generate n points $(u_{is}, \ldots, u_{is+s-1})$ in $[0,1)^s$.

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

Collision test

Partition $[0,1)^s$ in $k = d^s$ cubic boxes of equal size.
Generate n points $(u_{is}, \ldots, u_{is+s-1})$ in $[0,1)^s$.

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

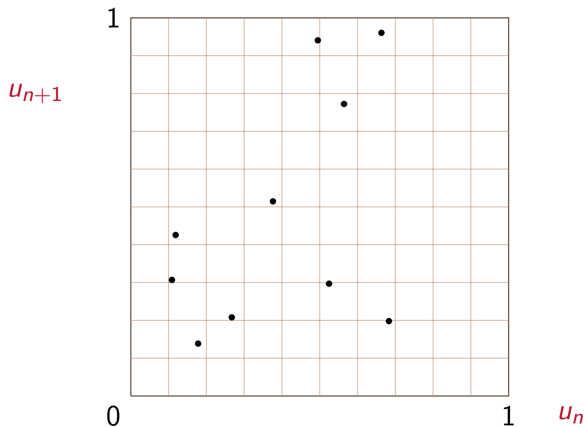
Under $\mathcal{H}_0$, $C \approx$ Poisson of mean $\lambda = n^2/(2k)$, if k is large and λ is small.

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

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

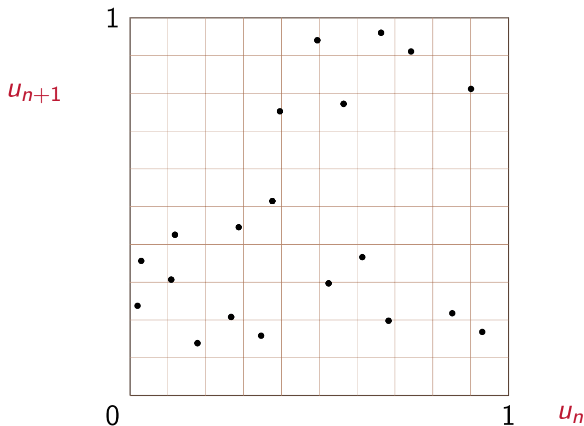
We reject $\mathcal{H}_0$ if $p^+(c)$ is too close to 0 (too many collisions)
or $p^-(c)$ is too close to 1 (too few collisions).

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



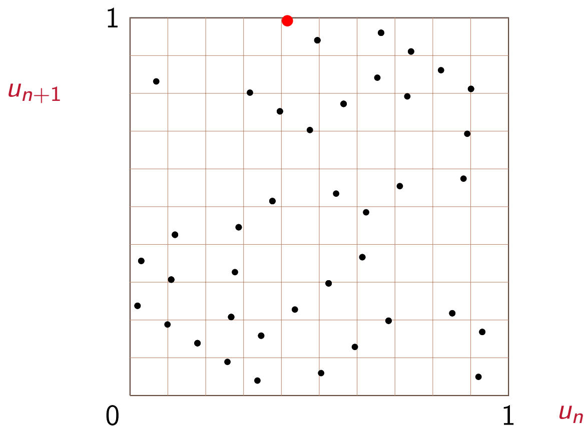
n	λ	C	$p^-(C)$
10	1/2	0	0.6281

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



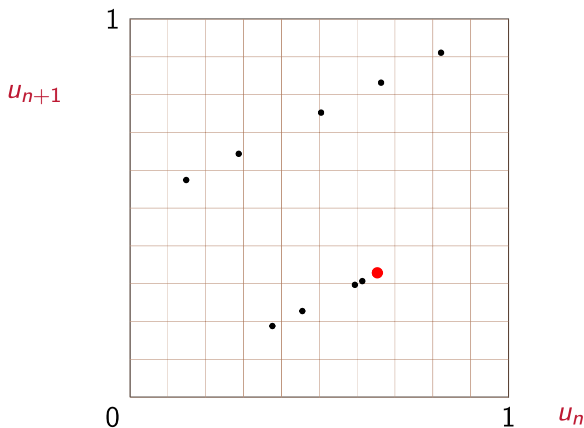
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304

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



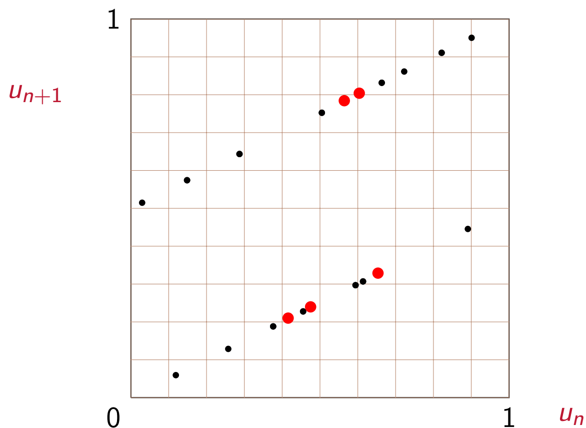
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304
40	8	1	0.0015

LCG with $m = 101$ and $a = 51$:



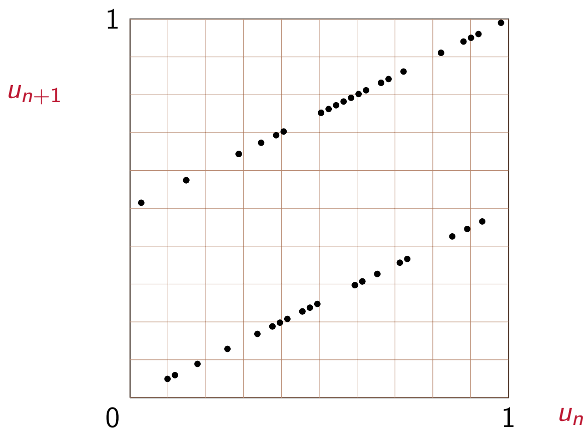
n	λ	C	$p^+(C)$
10	1/2	1	0.3718

LCG with $m = 101$ and $a = 51$:



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

LCG with $m = 101$ and $a = 51$:



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

SWB in Mathematica

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, $\lambda = 50$.

SWB in Mathematica

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, $\lambda = 50$.

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

SWB in Mathematica

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: $(U_0, \dots, U_{24})$.
For each, note the box where (U_0, U_{20}, U_{24}) falls.
Here, $\lambda = 50$.

Results: $C = 2070,\ 2137,\ 2100,\ 2104,\ 2127,\ \dots$.

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

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.

The TestU01 software

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

- 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.

Results of test batteries applied to some well-known RNGs

ρ = period length;
t-32 and t-64 gives the CPU time to generate 10^8 random numbers.

Number of failed tests (p-value $< 10^{-10}$ or $> 1 - 10^{-10}$) in each battery.

Generator	$\log_2 \rho$	t-32	t-64	S-Crush	Crush	B-Crush
LCG in Microsoft VisualBasic	24	3.9	0.66	14	—	—
LCG(2^{32}, 69069, 1), VAX	32	3.2	0.67	11	106	—
LCG(2^{32}, 1099087573, 0) Fishman	30	3.2	0.66	13	110	—
LCG(2^{48}, 25214903917, 11), Unix	48	4.1	0.65	4	21	—
Java.util.Random	47	6.3	0.76	1	9	21
LCG(2^{48}, 44485709377909, 0), Cray	46	4.1	0.65	5	24	—
LCG(2^{59}, 13^{13}, 0), NAG	57	4.2	0.76	1	10	17
LCG(2^{31}–1, 16807, 0), Wide use	31	3.8	3.6	3	42	—
LCG(2^{31}–1, 397204094, 0), SAS	31	19.0	4.0	2	38	—
LCG(2^{31}–1, 950706376, 0), IMSL	31	20.0	4.0	2	42	—
LCG(10^{12}–11, ..., 0), Maple	39.9	87.0	25.0	1	22	34

Generator	$\log_2 \rho$	t-32	t-64	S-Crush	Crush	B-Crush
Wichmann-Hill, MS-Excel	42.7	10.0	11.2	1	12	22
CombLec88, boost	61	7.0	1.2		1	
Knuth(38)	56	7.9	7.4		1	2
ran2, in Numerical Recipes	61	7.5	2.5			
CombMRG96	185	9.4	2.0			
MRG31k3p	185	7.3	2.0			
MRG32k3a SSJ + others	191	10.0	2.1			
MRG63k3a	377	—	4.3			
LFib(2^{31}, 55, 24, +), Knuth	85	3.8	1.1	2	9	14
LFib(2^{31}, 55, 24, −), Matpack	85	3.9	1.5	2	11	19
ran3, in Numerical Recipes		2.2	0.9		11	17
LFib(2^{48}, 607, 273, +), boost	638	2.4	1.4		2	2
Unix-random-32	37	4.7	1.6	5	101	—
Unix-random-64	45	4.7	1.5	4	57	—
Unix-random-128	61	4.7	1.5	2	13	19

Generator	$\log_2 \rho$	t-32	t-64	S-Crush	Crush	B-Crush
Knuth-ran_array2	129	5.0	2.6		3	4
Knuth-ranf_array2	129	11.0	4.5			
SWB(2^{24}, 10, 24)	567	9.4	3.4	2	30	46
SWB($2^{32}-5$, 22, 43)	1376	3.9	1.5		8	17
Mathematica-SWB	1479	—	—	1	15	—
GFSR(250, 103)	250	3.6	0.9	1	8	14
TT800	800	4.0	1.1		12	14
MT19937, widely used	19937	4.3	1.6		2	2
WELL19937a	19937	4.3	1.3		2	2
LFSR113	113	4.0	1.0		6	6
LFSR258	258	6.0	1.2		6	6
Marsaglia-xorshift	32	3.2	0.7	5	59	—

Generator	$\log_2 \rho$	t-32	t-64	S-Crush	Crush	B-Crush
Matlab-rand, (until 2008)	1492	27.0	8.4		5	8
Matlab in randn (normal)	64	3.7	0.8		3	5
SuperDuper-73, in S-Plus	62	3.3	0.8	1	25	—
R-MultiCarry, (changed)	60	3.9	0.8	2	40	—
KISS93	95	3.8	0.9		1	1
KISS99	123	4.0	1.1			
AES (OFB)		10.8	5.8			
AES (CTR)	130	10.3	5.4			
AES (KTR)	130	10.2	5.2			
SHA-1 (OFB)		65.9	22.4			
SHA-1 (CTR)	442	30.9	10.0			

Conclusion

- A flurry of computer applications require RNGs.
  A poor generator can severely bias simulation results, or permit one to cheat in computer lotteries or games, or cause important security flaws.
- Don't trust blindly the RNGs of commercial or other widely-used software, especially if they hide the algorithm (proprietary software...).
- Some software products have good RNGs; check what it is.
- RNGs with multiple streams are available from my web page in Java, C, and C++. Just Google "pierre lecuyer."
- Examples of work in progress:
  Fast nonlinear RNGs with provably good uniformity;
  RNGs based on multiplicative recurrences;
  RNGs with multiple streams for GPUs.