Random Number Generators with Multiple Streams for Parallel Computing

Pierre L'Ecuyer

Université de Montréal, Canada
and
Inria–Rennes, France

Thanks to David Munger and Nabil Kemerchou

ADVCOMP 2015, Nice, July 2015

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):

$$01111?100110?1?10100110110010100011 1\ldots$$

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):

$$01111?100110?1?10100110110010100111\ldots$$

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

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

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

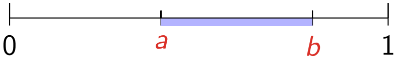
Want a probability of $1/8$ for each, independently of everything else.

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

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 $(0,1)$.

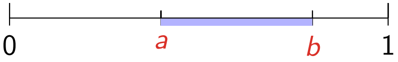
We want $\mathbb{P}[a \leq U_j \leq b] = b - a$.



Uniform distribution over $(0, 1)$

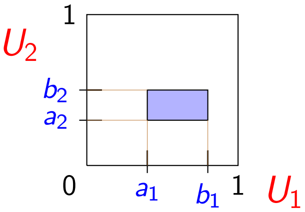
For simulation in general, we want (to imitate) a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniformly distributed over $(0, 1)$.

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



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

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



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.

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:

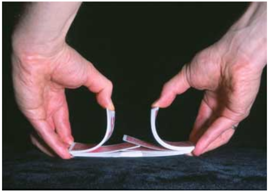
1 2 3 4 5 6 7	
1 2 3 4 6 7	5
1 3 4 6 7	5 2
3 4 6 7	5 2 1

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

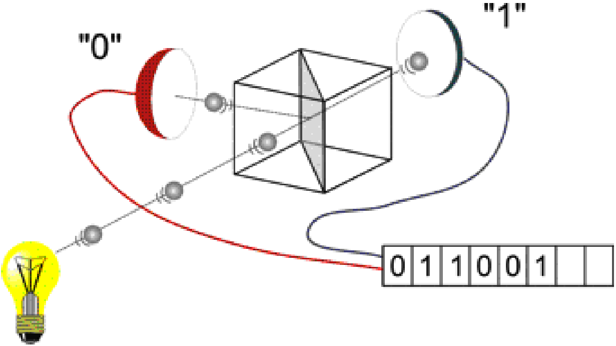
For n objets, choose an integer from 1 to n,
then an integer from 1 to $n-1$, then from 1 to $n-2$, ...
Each permutation should have the same probability.

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

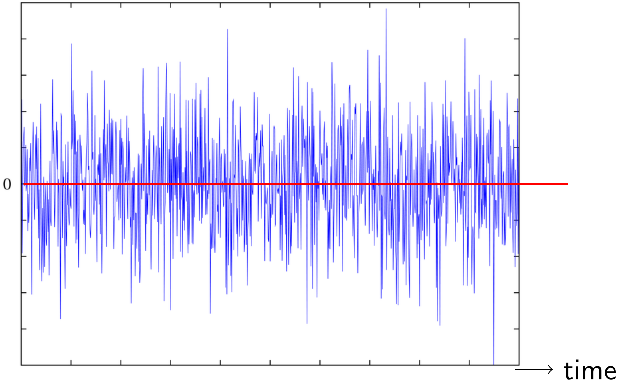


Physical devices for computers

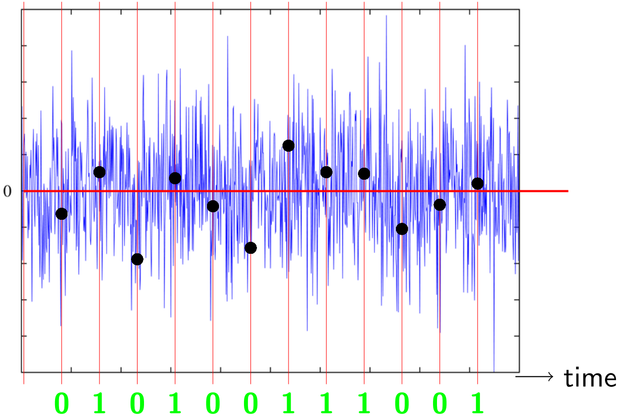
Photon trajectories (sold by **id-Quantique**):



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 has no (or little) mathematical analysis.

A much more important drawback: it is not reproducible.

Reproducibility *

Simulations are often required to be exactly replicable, and 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.

Algorithmic (pseudorandom) 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.

s_0

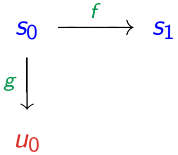
Algorithmic (pseudorandom) generator

$\mathcal{S}$, 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 (pseudorandom) generator

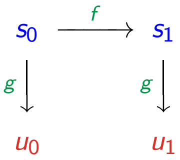
$\mathcal{S}$, finite state space; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$g : \mathcal{S} \to [0, 1]$, output function.



Algorithmic (pseudorandom) 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 (pseudorandom) 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.

$$
\begin{array}{ccccccccc}
s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
g\downarrow & & g\downarrow & & & & g\downarrow & & g\downarrow \\
u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}
$$

Algorithmic (pseudorandom) 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.

$$\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
 & & g\downarrow & & g\downarrow & & g\downarrow & & & & g\downarrow & & g\downarrow \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}$$

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

$$
\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
 & & \downarrow g & & \downarrow g & & \downarrow g & & & & \downarrow g & & \downarrow g \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}
$$

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

$$
\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
 & & g\downarrow & & g\downarrow & & g\downarrow & & & & g\downarrow & & g\downarrow \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}
$$

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.

$$
\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
 & & g\downarrow & & g\downarrow & & g\downarrow & & & & g\downarrow & & g\downarrow \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}
$$

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 $\mathrm{card}(\mathcal{S})$.

Lottery machines: modify the state s_n frequently.

$$
\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & s_{n+1} \\
 & & g\downarrow & & g\downarrow & & g\downarrow & & & & g\downarrow & & g\downarrow \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & u_{n+1}
\end{array}
$$

Goal: if we observe only $(u_0, u_1, \dots)$, 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 $\mathbf{u} = (u_0, \dots, u_{s-1})$.

Number of possibilities cannot exceed $\mathrm{card}(\mathcal{S})$.

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

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

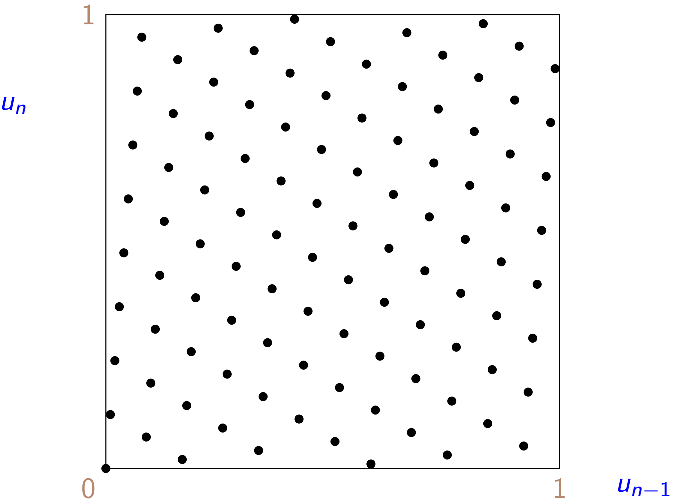
We want to approximate "$\mathbf{u}$ has the uniform distribution over $[0,1]^s$."

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

Design and analysis:

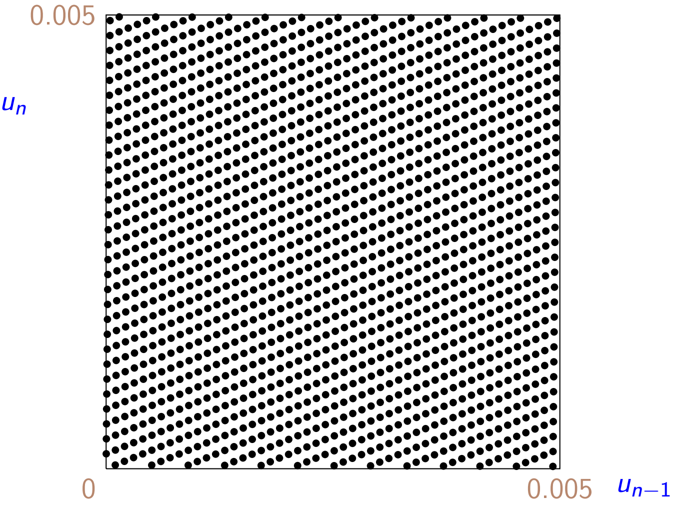
1. Define a uniformity measure for Ψ_s, computable without generating the points explicitly. Linear RNGs.
2. Choose a parameterized family (fast, long period, etc.) and search for parameters that "optimize" this measure.

Baby example:



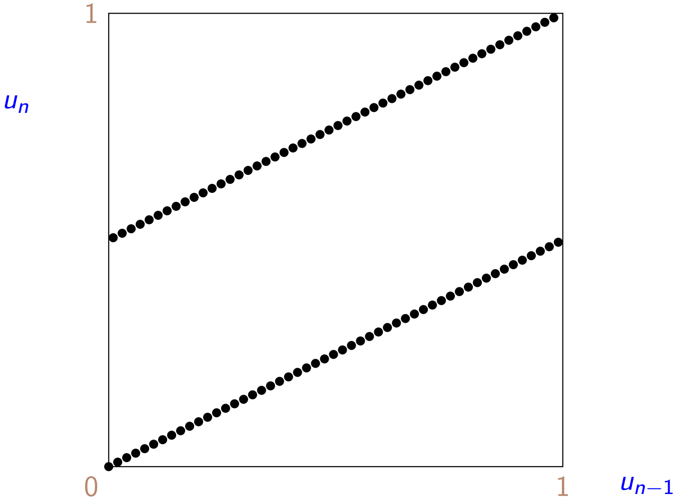
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$

Baby example:



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

Baby example:



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

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

Other examples: Subtract-with-borrow, lagged-Fibonacci, xorwow, etc.

A single RNG does not suffice.

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

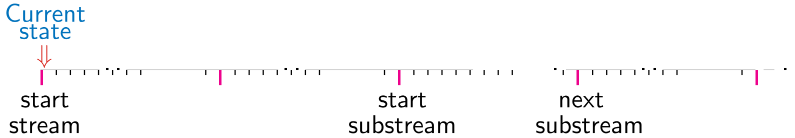
- Run a simulation on parallel processors.
- Compare systems with well synchronized common random numbers (CRNs). Can be complicated to implement and manage when different configurations do not need the same number of U_j's.

An existing 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}.

One stream:



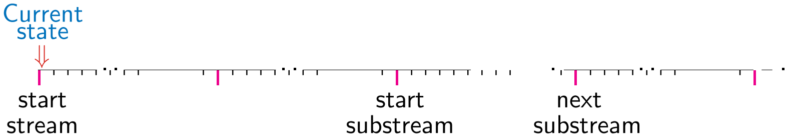
An existing solution: RNG with multiple streams and substreams.

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 mystream1 = createStream ();
double u = randomU01 (mystream1);
double z = normalDist.inverseF (randomU01(mystream1));
 ...
rewindSubstream (mystream1);
forwardToNextSubstream (mystream1);
rewindStream (mystream1);
```

One stream:



Comparing systems with CRNs: a simple inventory example

X_j = inventory level in morning of day j;
D_j = demand 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 for next morning with probability p.

Revenue for day j: sales − inventory costs − order costs
$= c \cdot \min(D_j, X_j) - h \cdot Y_j - (K + k \cdot (S - Yj)) \cdot \mathbb{I}[\text{an order arrives}]$.

Two streams of random numbers, one substream for each run.
Same streams and substreams for all policies (s, S).

Inventory example: C code to simulate m days

```
double inventorySimulateOneRun (int m, int s, int S,
       clrngStream *stream_demand, clrngStream *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 - clrngRandomInteger (stream_demand, 0, L);
      if (Yj < 0) Yj = 0; // Lost demand.
      profit += c * (Xj - Yj) - h * Yj;
      if ((Yj < s) && (clrngRandomU01 (stream_order) < p)) {
         // We have a successful order.
         profit -= K + k * (S - Yj); // Pay for successful order.
         Xj = S;
      } else
         Xj = Yj;           // Order not received.
   }
   return profit / m;       // Return average profit per day.
}
```

Comparing p policies with CRNs

```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
clrngStream* stream_demand = clrngCreateStream();
clrngStream* stream_order  = clrngCreateStream();
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);
      clrngForwardToNextSubstream (stream_demand);
      clrngForwardToNextSubstream (stream_order);
   }
   clrngRewindStream (stream_demand);
   clrngRewindStream (stream_order);
}

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

Comparing p policies with CRNs

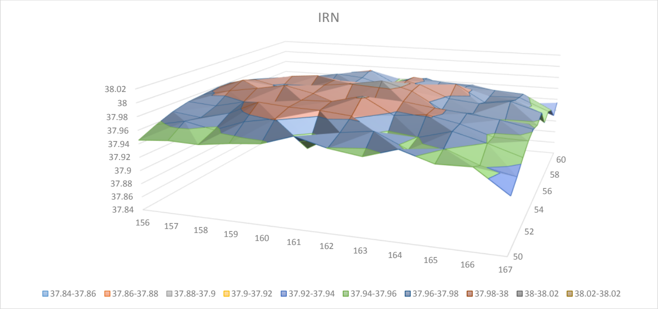
```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
clrngStream* stream_demand = clrngCreateStream();
clrngStream* stream_order  = clrngCreateStream();
for (int k = 0; k < p; k++) {   // for each policy
   for (int i = 0; i < n; i++) {    // perform n runs
      stat_profit[k, i] = inventorySimulateOneRun (m, s[k], S[k],
                                   stream_demand, stream_order);
      clrngForwardToNextSubstream (stream_demand);
      clrngForwardToNextSubstream (stream_order);
   }
   clrngRewindStream (stream_demand);
   clrngRewindStream (stream_order);
}

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

We would like to perform these pn simulations on thousands of parallel processors and obtain exactly the same results, using the same streams and substreams.

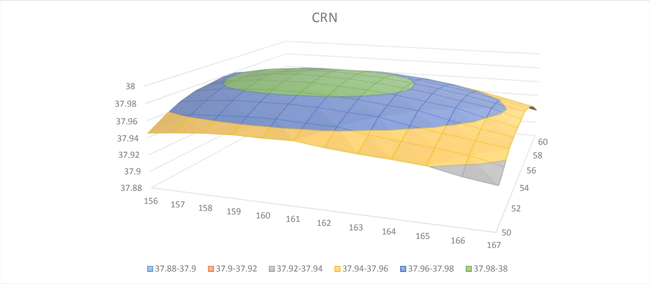
Comparison with independent random numbers

	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 CRNs

	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



Parallel computers

Processing elements (PEs) or “cores” are organized in a hierarchy. Many in a chip. SIMD or MIMD or mixture. Many chips per node, etc. Similar hierarchy for memory, usually more complicated and with many types of memory and access speeds.

Since about 10 years, clock speeds of processors no longer increase, but number of cores increases instead. Roughly doubles every 1.5 to 2 years.

Simulation algorithms (such as for RNGs) must adapt to this.

Some PEs, e.g., on GPUs, only have a small past-access (private) memory and have limited instruction sets.

Streams for parallel RNGs

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

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

Streams for parallel RNGs

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

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

Splitting: Single RNG with equally-spaced starting points for streams and for substreams. Recommended when possible. Requires fast computing of $s_{i+\nu} = f^{\nu}(s_i)$ for large ν, and single monitor to create all streams.

Streams for parallel RNGs

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

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

Splitting: Single RNG with equally-spaced starting points for streams and for substreams. Recommended when possible. Requires fast computing of $s_{i+\nu} = f^{\nu}(s_i)$ for large ν, and single monitor to create all streams.

Random starting points: acceptable if period ρ is huge.
For period ρ, and s streams of length ℓ,

$$\mathbb{P}[\text{overlap somewhere}] = P_o \approx s^2\ell/\rho.$$

Example: if $s = \ell = 2^{20}$, then $s^2\ell = 2^{60}$.
For $\rho = 2^{128}$, $P_o \approx 2^{-68}$. For $\rho = 2^{1024}$, $P_o \approx 2^{-964}$ (negligible).

How to use streams in parallel processing?

One can use several PEs to fill rapidly a large buffer of random numbers, and use them afterwards (e.g., on host processor). Many have proposed software tools to do that. But this is rarely what we want.

How to use streams in parallel processing?

One can use several PEs to fill rapidly a large buffer of random numbers, and use them afterwards (e.g., on host processor). Many have proposed software tools to do that. But this is rarely what we want.

Typically, we want independent streams produced and used by the threads. E.g., simulate the inventory model on each PE.

One stream per PE? One per thread? One per subtask? No.

For reproducibility and effective use of CRNs, streams must be assigned and used at a logical (hardware-independent) level, and it should be possible to have many distinct streams in a thread or PE at a time.

Single monitor to create all streams. Perhaps multiple creators of streams. To run on GPUs, the state should be small, say at most 256 bits. Some small robust RNGs such as LFSR113, MRG31k3p, and MRG32k3a are good for that. Also some counter-based RNGs.

Other scheme: streams that can split to create new children streams.

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.

Example of "poor" multiple streams:
good RNG but visible dependence between the streams.
Image synthesis on GPUs, with one stream per pixel.
(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: $\rho = m^k - 1$.

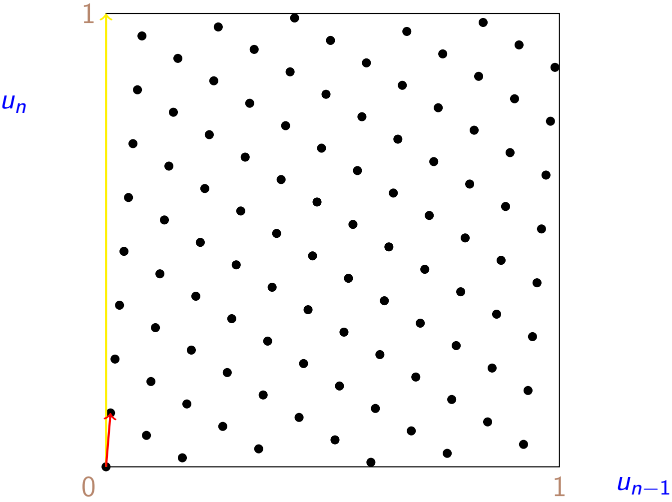
Numerous variants and implementations.

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.



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

Example of bad structure: lagged-Fibonacci

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

Very fast, but bad.

Example of bad structure: lagged-Fibonacci

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

Very fast, but bad. We always have $u_{n-k} + u_{n-r} - u_n = 0 \text{ mod } 1$.

This means: $u_{n-k} + u_{n-r} - u_n = q$ for some integer q.
All points (u_{n-k}, u_{n-r}, u_n) 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 MRGs.

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 recommendable generator: MRG32k3a

Choose six 32-bit 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 recommendable generator: MRG32k3a

Choose six 32-bit 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

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

A recommendable generator: MRG32k3a

Choose six 32-bit 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

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

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

A similar (faster) one: MRG31k3p

State is six 31-bit integers:
Two cycles of period $\rho \approx 2^{185}$.

Recurrence is implemented via shifts, masks, and additions.

Faster RNG: operations on blocks of bits.

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

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

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:

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

$x_{n-1} =$		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
  00010100101001101100110110100100
```

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	

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	
	00010100101001101 1	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|
                      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.

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 $\approx 2^{113} \approx 10^{34}$.

General linear recurrences modulo 2

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

General linear recurrences modulo 2

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

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

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

General linear recurrences modulo 2

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

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

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

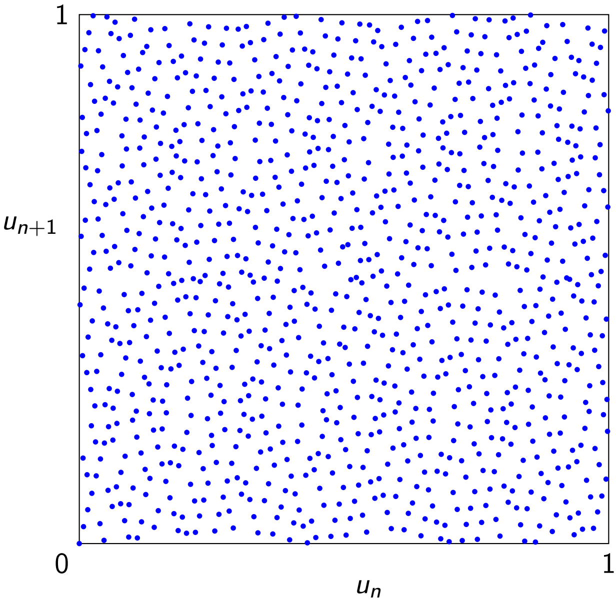
$$x_{n,j} = (\alpha_1 x_{n-1,j} + \cdots + \alpha_k x_{n-k,j}),$$

with characteristic polynomial

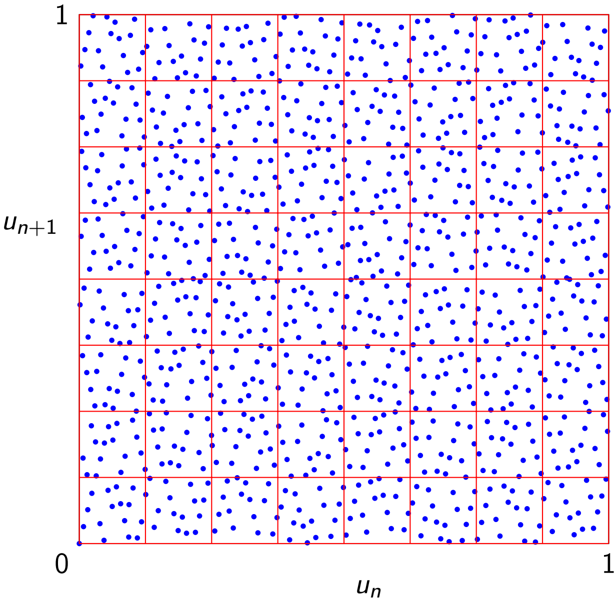
$$P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(\mathbf{A} - z\mathbf{I}).$$

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

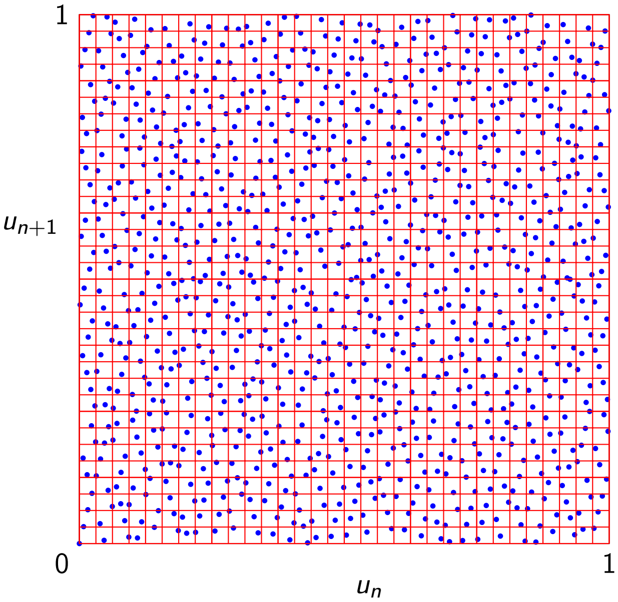
Uniformity measures. Example: $k = 10$, $2^{10} = 1024$ points



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 $s\ell \le k$, 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 $s\ell \le k$, the RNG is called maximally equidistributed.

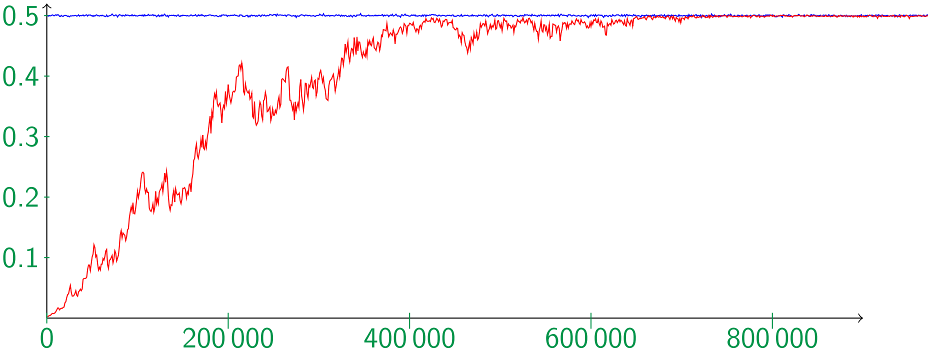
Can be generalized to rectangular boxes...

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.



Linear recurrence modulo m

State (vector) $\mathbf{x}_n$ evolves as

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

Jumping Ahead:

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

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

If output function $u_n = g(\mathbf{x}_n)$ is also linear, one can study the uniformity of each Ψ_s by studying the linear mapping. Many tools for this.

Combined linear/nonlinear generators

Linear generators fail statistical tests built to detect linearity.

Combined linear/nonlinear generators

Linear generators fail statistical tests built to detect linearity.

To escape linearity, we may

- use a nonlinear transition f;
- use a nonlinear output transformation g;
- do both;
- combine RNGs of different types.

There are various proposals in this direction. Many behave well empirically.

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

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.
It has a parameter c called the encoding key.
One can use a different key c for each stream.

Examples: MD5, TEA, SHA, AES, ChaCha, Threefish, etc.
The encoding is often simplified to make the RNG faster.
Threefry and Philox, for example. Very fast!
g_c : (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.
It has a parameter c called the encoding key.
One can use a different key c for each stream.

Examples: MD5, TEA, SHA, AES, ChaCha, Threefish, etc.
The encoding is often simplified to make the RNG faster.
Threefry and Philox, for example. Very fast!
g_c : (k-bit counter) $\mapsto$ (k-bit output), period $\rho = 2^k$.
E.g.: $k = 128$ or 256 or 512 or 1024.

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.

See Salmon, Moraes, Dror, Shaw (2011), for example.

An API for parallel RNGs in OpenCL

OpenCL is an emerging standard for programming GPUs and other similar devices. It extends (a subset of) the plain C language.
Limitations: On the device, no pointers to functions, no dynamic memory allocation, ... Low level.

clRNG is an API and library for RNGs in OpenCL, currently developed at Université de Montréal, in collaboration with Advanced Micro Devices (AMD).

Streams can be created only on the host, and can be used either on the host or on a device (such as by threads or work items on a GPU).

Must use a copy of the stream in private memory on the GPU device to generate random numbers.

Currently implements MRG32k3a, MRG31k3p, LFSR113, and Philox.

Also **clProbDist** and **clQMC**.

Host interface (subset)

Preprocessor replaces `clrng` by the name of desired base RNG.

On host computer, streams are created and manipulated as arrays of streams.

```
typedef struct ...    clrngStreamState;
```

State of a random stream. Definition depends on generator type.

```
typedef struct ...    clrngStream;
```

Current state of stream, its initial state, and initial state of current substream.

```
clrngStream* clrngAllocStreams(size_t count, size_t* bufSize,
        clrngStatus* err);
```

Reserve memory space for count stream objects.

```
clrngStream* clrngCreateStreams(clrngStreamCreator* creator,
        size_t count, size_t* bufSize, clrngStatus* err);
```

Reserve memory and create (and return) an array of count new streams.

```
clrngStatus clrngCreateOverStreams(clrngStreamCreator* creator,
        size_t count, clrngStream* streams);
```

Create new streams in preallocated buffer.

```
clrngStream* clrngCopyStreams(size_t count, const clrngStream* streams,
        clrngStatus* err);
```

Reserves memory and return in it a clone of array `streams`.

```
clrngStatus clrngCopyOverStreams(size_t count, clrngStream* destStreams,
        const clrngStream* srcStreams);
```

Copy (restore) `srcStreams` over `destStreams`, and all count stream inside.

```
clrngStatus clrngDestroyStreams(clrngStream* streams);
```

```
cl_double clrngRandomU01(clrngStream* stream);
cl_int clrngRandomInteger(clrngStream* stream, cl_int i, cl_int j);
clrngStatus clrngRandomU01Array(clrngStream* stream, size_t count,
        cl_double* buffer);
clrngStatus clrngRandomIntegerArray(clrngStream* stream,
        cl_int i, cl_int j, size_t count, cl_int* buffer);
```

```
clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
```

Reinitialize `streams` to their initial states.

```
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);
```

Reinitialize `streams` to the initial states of their current substreams.

```
clrngStatus clrngForwardToNextSubstreams(size_t count,
        clrngStream* streams);
```

```
clrngStatus clrngDeviceRandomU01Array(size_t streamCount,
        cl_mem streams, size_t numberCount, cl_mem outBuffer,
        cl_uint numQueuesAndEvents, cl_command_queue* commQueues,
        cl_uint numWaitEvents, const cl_event* waitEvents,
        cl_event* outEvents);
```

Fill buffer at `outBuffer` with `numberCount` uniform random numbers, using `streamCount` work items.

Interface on Devices

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

```
clrngStatus clrngCopyOverStreams(size_t count, clrngStream* destStreams,
        const clrngStream* srcStreams);
clrngStatus clrngCopyOverStreamsFromHost (size_t count,
        clrngStream* destStreams,
        __global const clrngHostStream* srcStreams);
clrngStatus clrngCopyOverStreamsToHost(size_t count,
        __global const clrngHostStream* destStreams,
        clrngStream* srcStreams);

cl_double clrngRandomU01(clrngStream* stream);
cl_int clrngRandomInteger(clrngStream* stream, cl_int i, cl_int j);
clrngStatus clrngRandomU01Array(clrngStream* stream, size_t count,
        cl_double* buffer);
clrngStatus clrngRandomIntegerArray(clrngStream* stream,
        cl_int i, cl_int j, size_t count, cl_int* buffer);

clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);
clrngStatus clrngForwardToNextSubstreams(size_t count,
        clrngStream* streams);
```

Inventory example

```
__kernel void inventorySimulPoliciesGPU (int m, int p,
            int *s, int *S, int n2,
            __global clrngStreams *streams_demand,
            __global clrngStreams *streams_order,
            __global double *stat_profit) {
   // Each of the n1*p work items simulates n2 runs.
   int gid = get_global_id(0);     // Id of this work item.
   int n1p = get_global_size(0);   // Total number of work items.
   int n1  = n1 / p;               // Number of streams.
   int k   = gid / n1;             // Policy index for this work item.
   int j   = gid % n1;             // Index of stream for this work item.

   // Make local copies of the stream states, in private memory.
   clrngStream stream_demand_d, stream_order_d;
   clrngCopyOverStreamsFromHost (1, &stream_demand_d, &streams_demand[j]);
   clrngCopyOverStreamsFromHost (1, &stream_order_d, &streams_order[j]);
   for (int i = 0; i < n2; i++) {
       stat_profit[i * n1p + gid] = inventorySimulateOneRun(m, s[k], S[k],
                                      &stream_demand_d, &stream_order_d);
       clrngForwardToNextSubstreams(1, &stream_demand_d);
       clrngForwardToNextSubstreams(1, &stream_order_d);
   }
}
```

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.

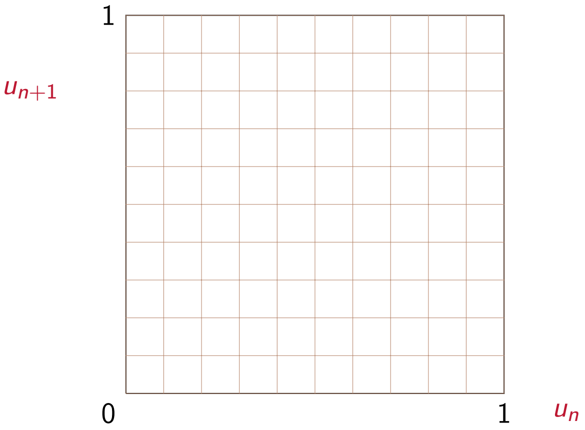
Different tests detect different deficiencies.

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

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

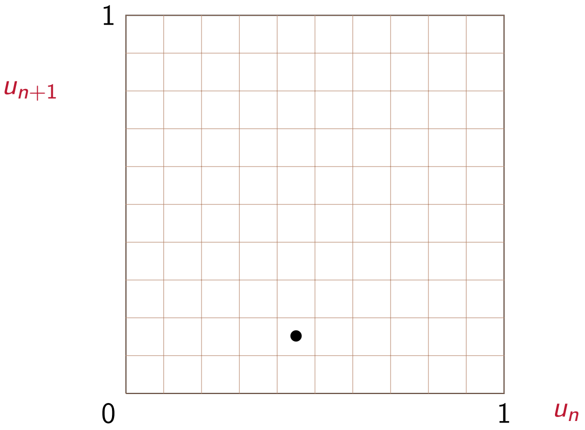
Compromise: Build an RNG that passes most reasonable tests.
Tests that fail are hard to find.
Formalization: computational complexity framework.

Example: A collision test



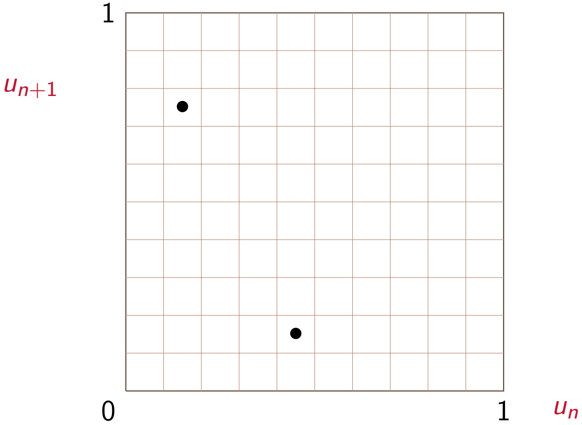
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



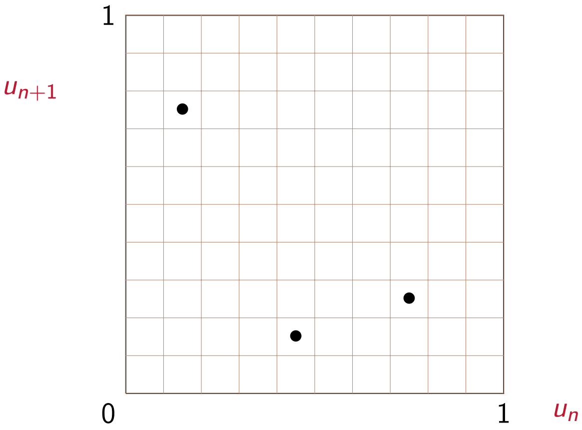
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



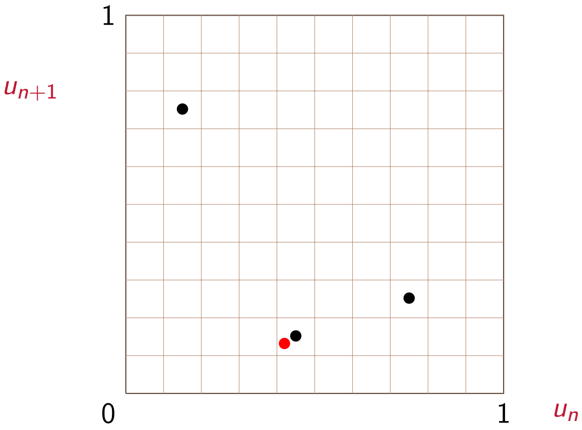
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



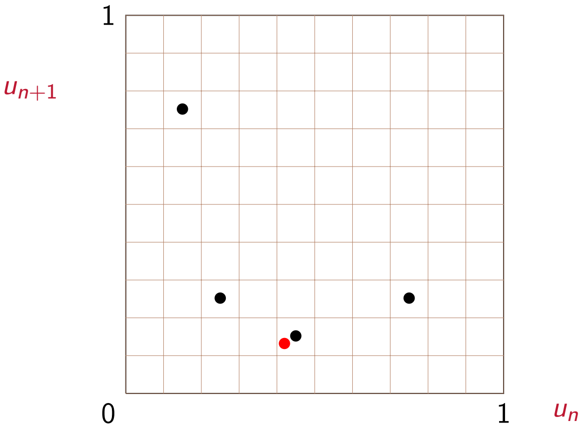
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



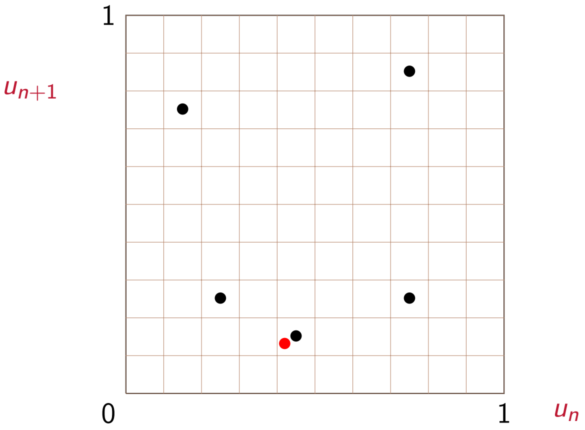
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



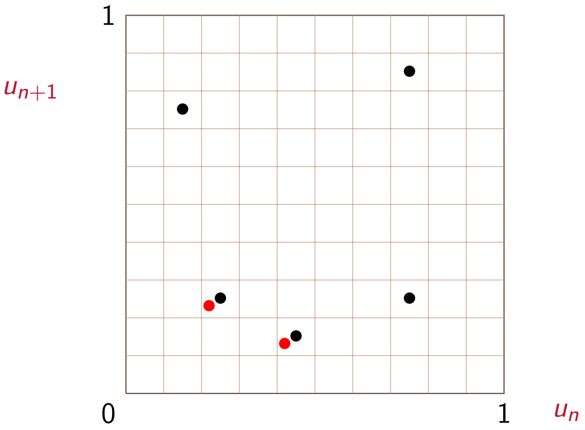
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



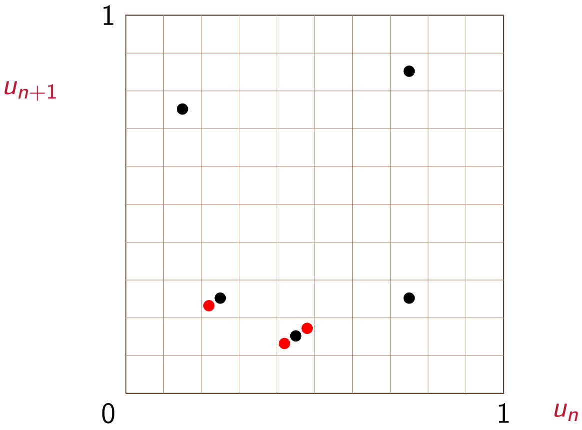
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



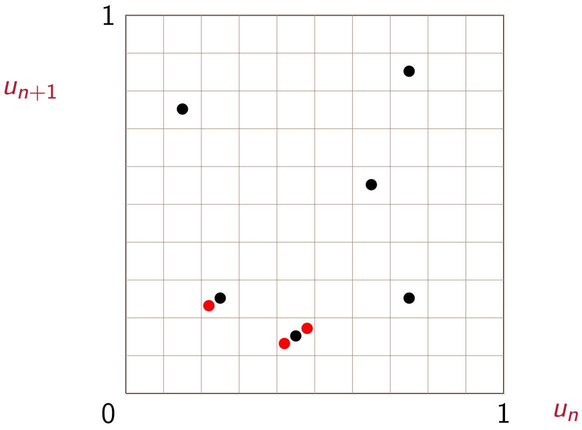
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



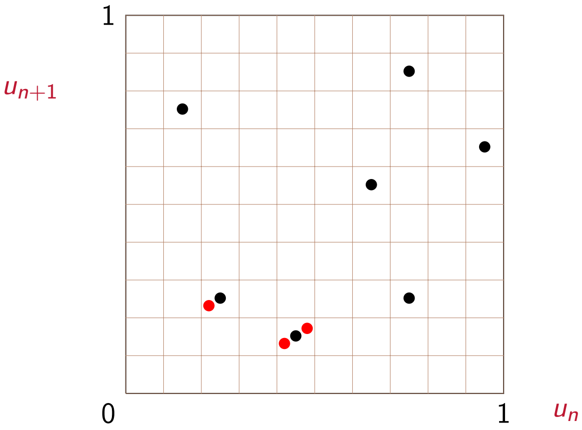
Throw $n = 10$ points in $k = 100$ boxes.

Example: A collision test



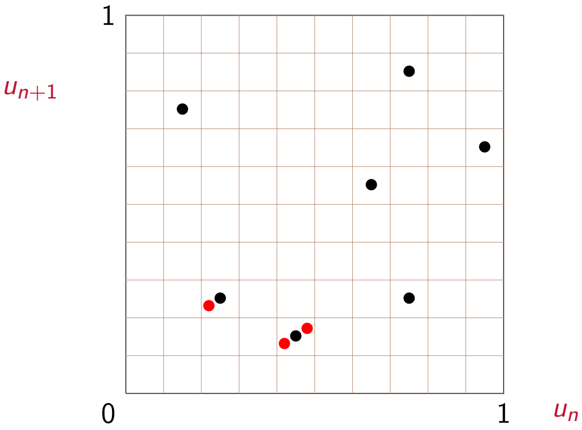
Throw $n = 10$ points in $k = 100$ boxes.

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 $[0, 1)^s$.

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

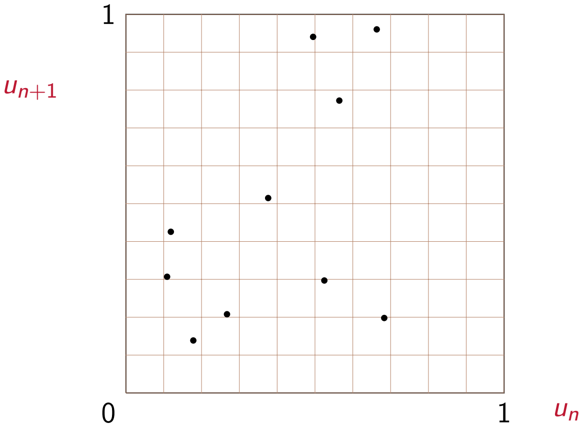
Under $\mathcal{H}_0$, $C \approx$ Poisson of mean $\lambda = n^2/(2k)$, if k is large and λ is small.

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

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

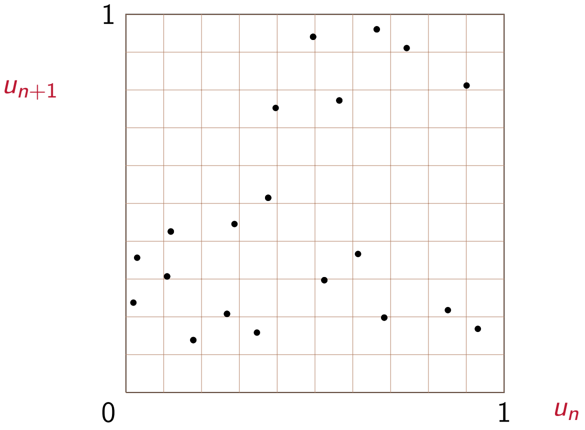
We reject $\mathcal{H}_0$ if $p^+(c)$ is too close to 0 (too many collisions)
or $p^-(c)$ is too close to 1 (too few collisions).

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



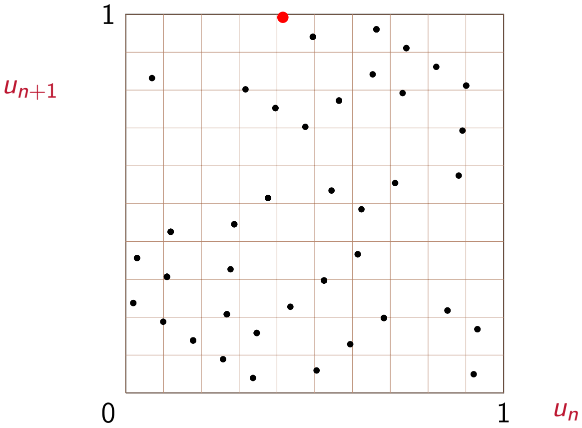
n	λ	C	$p^-(C)$
10	1/2	0	0.6281

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



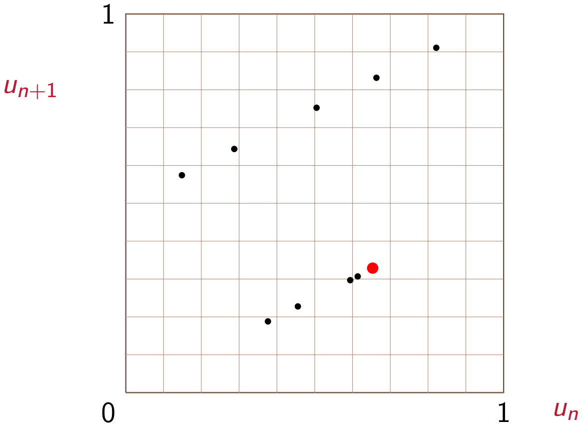
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304

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



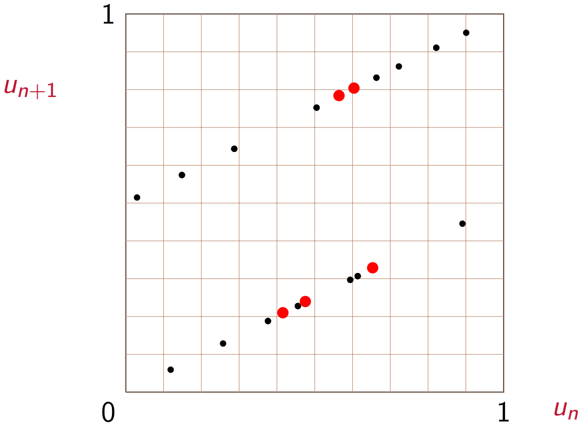
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304
40	8	1	0.0015

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



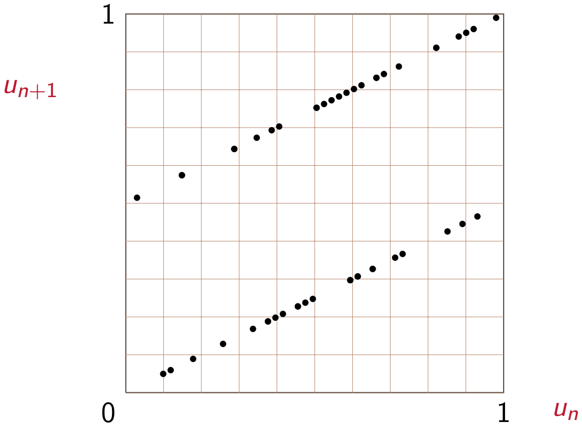
n	λ	C	$p^+(C)$
10	$1/2$	1	0.3718

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

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

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.

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++. Also OpenCL library, mostly for GPUs.
- Examples of recent proposals or work in progress:
  Fast nonlinear RNGs with provably good uniformity;
  RNGs based on multiplicative recurrences;
  Counter-based RNGs. RNGs with multiple streams for GPUs.