Random Number Generators
with Multiple Streams
for Parallel Computing

Pierre L’Ecuyer

DIRO, Université de Montréal

DMS, Université Laval, Octobre 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?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.
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 $\frac{1}{2^s}$ 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 $\mathbb{P}[a \leq U_j \leq b] = b - a$. 

\begin{center}
\begin{tikzpicture}
\draw (0,0) -- (4,0);
\draw[fill=blue!20] (1,0) rectangle (2,0.2);
\node at (0.5,0) {0}; \node at (1.5,0) {$a$}; \node at (2.5,0) {$b$}; \node at (3.5,0) {1};
\end{tikzpicture}
\end{center}
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 $\mathbb{P}[X \leq x] = F(x)$:

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

This is inversion.

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

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

to shuffle a deck of 52 cards: $52! \approx 2^{226}$ possibilities.
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 $n$ objects, 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.
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 $n$ objects, 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.
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 \( n \) objects, 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:

```
0 1 1 0 0 0 1 0 0 1 1 0 1 1 0 1 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.
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:

```
0 1 1 0 0 0 1 0 0 1 1 0 1 1 0 1 0 0
1 1 0 1 1 1 0 1 1 0
```

or (this eliminates the bias):

```
0 1 1 0 0 0 1 0 0 1 1 0 1 1 0 1 0 0
0 1 1 0 1 0 1 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:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

or (this eliminates the bias):

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

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

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

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

\[
\begin{array}{c}
S_0 \\
g \\
\downarrow \\
U_0
\end{array}
\]
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.

\[ s_0 \xrightarrow{f} s_1 \xrightarrow{g} u_0 \]

Period of \{\(s_n\), \(n \geq 0\)\}:

\[ \rho \leq \text{cardinality of } \mathcal{S} \]
Algorithmic (pseudorandom) generator

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

\[
\begin{array}{c}
s_0 \xrightarrow{f} s_1 \\
g \downarrow \quad g \\
u_0 \quad u_1
\end{array}
\]

Period of \( \{ s_n, n \geq 0 \} \):
\[ \rho \leq \text{cardinality of } S. \]
Algorithmic (pseudorandom) generator

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

\[ S_0 \xrightarrow{f} S_1 \xrightarrow{f} \cdots \xrightarrow{f} S_n \xrightarrow{f} S_{n+1} \]

\[ g \downarrow g \downarrow \cdots \downarrow g \downarrow \]

\[ u_0 \xrightarrow{g} u_1 \xrightarrow{g} \cdots \xrightarrow{g} u_n \xrightarrow{g} u_{n+1} \]
Algorithmic (pseudorandom) generator

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

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

$\cdots \xrightarrow{f} s_{\rho-1} \xrightarrow{f} s_0 \xrightarrow{f} s_1 \xrightarrow{f} \cdots \xrightarrow{f} s_n \xrightarrow{f} s_{n+1}$

$\cdots \xrightarrow{g} u_{\rho-1} \xrightarrow{g} u_0 \xrightarrow{g} u_1 \xrightarrow{g} \cdots \xrightarrow{g} u_n \xrightarrow{g} u_{n+1}$
Goal: if we observe only \( (u_0, u_1, \ldots) \), difficult to distinguish from a sequence of independent random variables over \((0, 1)\).
Goal: if we observe only \((u_0, u_1, \ldots)\), difficult to distinguish from a sequence of independant random variables over \((0, 1)\).

Utopia: passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.
\[ \cdots \xrightarrow{f} s_{\rho - 1} \xrightarrow{f} s_0 \xrightarrow{f} s_1 \xrightarrow{f} \cdots \xrightarrow{f} s_n \xrightarrow{f} s_{n+1} \]

\[ \cdots \xrightarrow{g} u_{\rho - 1} \xrightarrow{g} u_0 \xrightarrow{g} u_1 \xrightarrow{g} \cdots \xrightarrow{g} u_n \xrightarrow{g} u_{n+1} \]

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

**Utopia:** passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

With random seed \(s_0\), an RNG is a gigantic roulette wheel.

Selecting \(s_0\) at random and generating \(s\) random numbers means spinning the wheel and taking \(u = (u_0, \ldots, u_{s-1})\).

Number of possibilities cannot exceed \(\text{card}(S)\).

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 independent random variables over \((0, 1)\).

**Utopia:** passes all statistical tests. Impossible!

Compromise between speed / good statistical behavior / predictability.

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

Number of possibilities cannot exceed \(\text{card}(S)\).
Lottery machines: modify the state \(s_n\) frequently.
Uniform distribution over $[0, 1]^s$.

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

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

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 $S$ and we generate $s$ numbers, this corresponds to choosing a random point in the finite set

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

We want to approximate “\(\mathbf{u}\) has the uniform distribution over \([0, 1]^s\).”

Measure of quality: $\Psi_s$ must cover $[0, 1]^s$ very evenly.
Uniform distribution over \([0, 1]^s\).

If we choose \(s_0\) randomly in \(S\) and we generate \(s\) numbers, this corresponds to choosing a random point in the finite set

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

We want to approximate “\(\mathbf{u}\) has the uniform distribution over \([0, 1]^s\).”

Measure of quality: \(\Psi_s\) must cover \([0, 1]^s\) very evenly.

Design and analysis:
1. Define a uniformity measure for \(\Psi_s\), 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:

\[ x_n = 12 \times x_{n-1} \mod 101; \quad u_n = x_n / 101 \]
Baby example:

\[ x_n = 4809922 \times x_{n-1} \mod 60466169 \text{ and } u_n = x_n/60466169 \]
Baby example:

\[ x_n = 51x_{n-1} \mod 101; \quad u_n = x_n/101. \]

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:** $u_n = (n/2^{1000}) \mod 1$ 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 $2^{76}$.

```java
RandomStream mystream1 = createStream ();
double u = randomU01 (mystream1);
double z = inverseCDF (normalDist, randomU01(mystream1));
...
rewindSubstream (mystream1);
forwardToNextSubstream (mystream1);
rewindStream (mystream1);
```

One stream:
Comparing systems with CRNs: a simple inventory example

\( X_j = \text{inventory level} \) in morning of day \( j \);

\( D_j = \text{demand} \) on day \( j \), uniform over \( \{0, 1, \ldots, L\} \);

\( \min(D_j, X_j) \) \text{ sales} on day \( j \);

\( Y_j = \max(0, X_j - D_j) \) \text{ 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

\[ = c \cdot \min(D_j, X_j) - h \cdot Y_j - (K + k \cdot (S - Y_j)) \cdot \mathbb{I}[\text{an order arrives}]. \]

Two streams of random numbers, one substream for each run.

Same streams and substreams for all policies \((s, S)\).

Number of calls to RNG for order arrivals is random!
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);
        // Realign starting points so they are the same for all policies
        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);

      // Realign starting points so they are the same for all policies
      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

<table>
<thead>
<tr>
<th></th>
<th>156</th>
<th>157</th>
<th>158</th>
<th>159</th>
<th>160</th>
<th>161</th>
<th>162</th>
<th>163</th>
<th>164</th>
<th>165</th>
<th>166</th>
<th>167</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>37.94537</td>
<td>37.94888</td>
<td>37.95314</td>
<td>37.95718</td>
<td>37.97194</td>
<td>37.95955</td>
<td>37.95281</td>
<td>37.96711</td>
<td>37.95221</td>
<td>37.95325</td>
<td>37.92063</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>37.9574</td>
<td>37.9665</td>
<td>37.95732</td>
<td>37.97337</td>
<td>37.98137</td>
<td>37.94273</td>
<td>37.96965</td>
<td>37.97573</td>
<td>37.95425</td>
<td>37.96074</td>
<td>37.94185</td>
<td>37.93139</td>
</tr>
<tr>
<td>52</td>
<td>37.96725</td>
<td>37.96166</td>
<td>37.97192</td>
<td>37.99236</td>
<td>37.98856</td>
<td>37.98708</td>
<td>37.98266</td>
<td>37.94671</td>
<td>37.95961</td>
<td>37.97238</td>
<td>37.95982</td>
<td>37.94465</td>
</tr>
<tr>
<td>53</td>
<td>37.97356</td>
<td>37.96999</td>
<td>37.97977</td>
<td>37.97611</td>
<td>37.98929</td>
<td>37.99089</td>
<td>38.00219</td>
<td>37.97693</td>
<td>37.98191</td>
<td>37.97217</td>
<td>37.95713</td>
<td>37.95575</td>
</tr>
<tr>
<td>54</td>
<td>37.97593</td>
<td>37.9852</td>
<td>37.99233</td>
<td>38.00043</td>
<td>37.99056</td>
<td>37.9744</td>
<td>37.98008</td>
<td>37.9817</td>
<td>37.98168</td>
<td>37.97703</td>
<td>37.97145</td>
<td>37.96138</td>
</tr>
<tr>
<td>55</td>
<td>37.97865</td>
<td>37.9946</td>
<td>37.97297</td>
<td>37.98383</td>
<td>37.99527</td>
<td>38.00068</td>
<td>38.00826</td>
<td>37.97693</td>
<td>37.98191</td>
<td>37.97217</td>
<td>37.95713</td>
<td>37.95672</td>
</tr>
<tr>
<td>56</td>
<td>37.97871</td>
<td>37.9867</td>
<td>37.97672</td>
<td>37.9744</td>
<td>37.9955</td>
<td>37.9712</td>
<td>37.96967</td>
<td>37.99717</td>
<td>37.97736</td>
<td>37.97275</td>
<td>37.97968</td>
<td>37.96523</td>
</tr>
<tr>
<td>57</td>
<td>37.97414</td>
<td>37.97797</td>
<td>37.98816</td>
<td>37.99192</td>
<td>37.9678</td>
<td>37.98415</td>
<td>37.97774</td>
<td>37.97844</td>
<td>37.99203</td>
<td>37.96531</td>
<td>37.97226</td>
<td>37.93934</td>
</tr>
<tr>
<td>58</td>
<td>37.96869</td>
<td>37.97435</td>
<td>37.9625</td>
<td>37.96581</td>
<td>37.97331</td>
<td>37.95655</td>
<td>37.98382</td>
<td>37.97144</td>
<td>37.97409</td>
<td>37.96631</td>
<td>37.96764</td>
<td>37.94759</td>
</tr>
<tr>
<td>59</td>
<td>37.95772</td>
<td>37.94725</td>
<td>37.9711</td>
<td>37.97905</td>
<td>37.97504</td>
<td>37.96237</td>
<td>37.98182</td>
<td>37.97656</td>
<td>37.97212</td>
<td>37.96762</td>
<td>37.96429</td>
<td>37.93976</td>
</tr>
<tr>
<td>60</td>
<td>37.94434</td>
<td>37.95081</td>
<td>37.94275</td>
<td>37.95515</td>
<td>37.98134</td>
<td>37.95863</td>
<td>37.96581</td>
<td>37.95548</td>
<td>37.96573</td>
<td>37.93949</td>
<td>37.93839</td>
<td>37.9203</td>
</tr>
<tr>
<td>61</td>
<td>37.922</td>
<td>37.93006</td>
<td>37.92656</td>
<td>37.93281</td>
<td>37.94999</td>
<td>37.95799</td>
<td>37.96368</td>
<td>37.94849</td>
<td>37.954</td>
<td>37.92439</td>
<td>37.90535</td>
<td>37.93375</td>
</tr>
</tbody>
</table>

IRN

37.84-37.86 37.86-37.88 37.88-37.9 37.9-37.92 37.92-37.94 37.94-37.96 37.96-37.98 37.98-38 38-38.02 38.02-38.02
Comparison with CRNs

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>51</th>
<th>52</th>
<th>53</th>
<th>54</th>
<th>55</th>
<th>56</th>
<th>57</th>
<th>58</th>
<th>59</th>
<th>60</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>37.94537</td>
<td>37.9574</td>
<td>37.96725</td>
<td>37.97356</td>
<td>37.97593</td>
<td>37.97865</td>
<td>37.97871</td>
<td>37.97414</td>
<td>37.96869</td>
<td>37.95772</td>
<td>37.94434</td>
<td>37.922</td>
</tr>
<tr>
<td>156</td>
<td>37.94888</td>
<td>37.96169</td>
<td>37.97117</td>
<td>37.97852</td>
<td>37.98241</td>
<td>37.98235</td>
<td>37.98269</td>
<td>37.98035</td>
<td>37.98611</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>157</td>
<td>37.95166</td>
<td>37.96379</td>
<td>37.97402</td>
<td>37.98089</td>
<td>37.98589</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>158</td>
<td>37.95319</td>
<td>37.96524</td>
<td>37.97476</td>
<td>37.98243</td>
<td>37.98589</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>159</td>
<td>37.95274</td>
<td>37.96546</td>
<td>37.97492</td>
<td>37.98187</td>
<td>37.98703</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>160</td>
<td>37.95318</td>
<td>37.96379</td>
<td>37.97387</td>
<td>37.98079</td>
<td>37.98703</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>161</td>
<td>37.94887</td>
<td>37.96293</td>
<td>37.97171</td>
<td>37.97848</td>
<td>37.98703</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>162</td>
<td>37.94584</td>
<td>37.95726</td>
<td>37.96879</td>
<td>37.97436</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>163</td>
<td>37.94361</td>
<td>37.95295</td>
<td>37.96184</td>
<td>37.97088</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>164</td>
<td>37.94074</td>
<td>37.95589</td>
<td>37.96184</td>
<td>37.97088</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>165</td>
<td>37.93335</td>
<td>37.95589</td>
<td>37.96184</td>
<td>37.97088</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>166</td>
<td>37.92832</td>
<td>37.95589</td>
<td>37.96184</td>
<td>37.97088</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
<tr>
<td>167</td>
<td>37.9232</td>
<td>37.95589</td>
<td>37.96184</td>
<td>37.97088</td>
<td>37.97391</td>
<td>37.9874</td>
<td>37.98494</td>
<td>37.98035</td>
<td>37.98235</td>
<td>37.96302</td>
<td>37.94861</td>
<td>37.93169</td>
</tr>
</tbody>
</table>
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.

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 $\nu$, and single monitor to create all streams.

Random starting points: acceptable if period $\rho$ is huge.

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

Example: if $s = \ell = 2^{20}$, then $s^2 \ell = 2^{60}$.

For $\rho = 2^{128}$, $P_o \approx 2^{-68}$ (negligible). For $\rho = 2^{1024}$, $P_o \approx 2^{-964}$ (negligible).
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 \( \nu \), 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 $\nu$, and single monitor to create all streams.

Random starting points: acceptable if period $\rho$ is huge. For period $\rho$, and $s$ streams of length $\ell$,

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

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.


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)

\[ x_n = (a_1x_{n-1} + \cdots + a_kx_{n-k}) \mod m, \quad u_n = x_n/m. \]

State: \( s_n = (x_{n-k+1}, \ldots, x_n) \). Max. period: \( \rho = m^k - 1 \).
Linear multiple recursive generator (MRG)

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

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 \( \Psi_s \):**

\( x_0, \ldots, x_{k-1} \) 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 \( \Psi_s \) is a linear space; it is the intersection of a lattice with the unit cube.
\[ x_n = 12x_{n-1} \mod 101; \quad u_n = x_n/101 \]
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 \( u_{n-k} + u_{n-r} - u_n = 0 \mod 1. \)

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 \(c_n \in \{0, 1\}\):

\[
x_n = (x_{n-8} - x_{n-48} - c_{n-1}) \mod 2^{31},
\]
\[
c_n = 1 \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \quad c_n = 0 \text{ otherwise},
\]
\[
u_n = x_n/2^{31},
\]

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 \(c_n \in \{0, 1\}\):

\[
\begin{align*}
x_n &= (x_{n-8} - x_{n-48} - c_{n-1}) \mod 2^{31}, \\
c_n &= 1 \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \quad c_n = 0 \text{ otherwise,} \\
u_n &= x_n/2^{31},
\end{align*}
\]

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

In Mathematica versions \(\leq 5.2\): 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 \(c_n \in \{0, 1\}\):

\[
x_n = (x_{n-8} - x_{n-48} - c_{n-1}) \mod 2^{31},
\]
\[
c_n = 1 \text{ if } x_{n-8} - x_{n-48} - c_{n-1} < 0, \quad c_n = 0 \text{ otherwise,}
\]
\[
u_n = x_n / 2^{31},
\]

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

In Mathematica versions \(\leq 5.2\):
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 \([0, 1)^3\).


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


Combined MRGs.

Two [or more] MRGs in parallel:

\[
x_{1,n} = (a_{1,1}x_{1,n-1} + \cdots + a_{1,k}x_{1,n-k}) \mod m_1,
\]

\[
x_{2,n} = (a_{2,1}x_{2,n-1} + \cdots + a_{2,k}x_{2,n-k}) \mod m_2.
\]

One possible combinaison:

\[
z_n := (x_{1,n} - x_{2,n}) \mod m_1; \quad u_n := z_n/m_1;
\]

L’Ecuyer (1996): the sequence \( \{u_n, n \geq 0\} \) is also the output of an MRG of modulus \( m = m_1m_2 \), 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.

A recommendable generator: MRG32k3a

Choose six 32-bit integers:

\[x_{-2}, x_{-1}, x_0 \text{ in } \{0, 1, \ldots, 4294967086\} \text{ (not all 0)} \] and
\[y_{-2}, y_{-1}, y_0 \text{ in } \{0, 1, \ldots, 4294944442\} \text{ (not all 0)}.\]

For \(n = 1, 2, \ldots\), let

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

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

\((x_{n-2}, x_{n-1}, x_n)\) 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:

\[ x_{-2}, x_{-1}, x_0 \text{ in } \{0, 1, \ldots, 4294967086\} \text{ (not all 0)} \] and
\[ y_{-2}, y_{-1}, y_0 \text{ in } \{0, 1, \ldots, 4294944442\} \text{ (not all 0)}. \] For \( n = 1, 2, \ldots \), let

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

\((x_{n-2}, x_{n-1}, x_n)\) 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:

$$x_{n-1} = \begin{array}{c}
0001010010101101100110110100101
\end{array}$$
Faster RNG: operations on blocks of bits.

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

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

\[
x_{n-1} = \begin{array}{c}
00010100101001101100110110100101 \\
100101 \\
00101001110110110110100101 \\
00111101000101011010010011100101 \\
00110110100100011110100010101101
\end{array}
\]

The first 31 bits of \( x_1, x_2, x_3, \ldots \), visit all integers from 1 to \( 2^{31} - 1 \) exactly once before returning to \( x_0 \). For real numbers in \((0, 1)\):

\[
u_n = \frac{x_n}{2^{32} + 1}
\]
Faster RNG: operations on blocks of bits.

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

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

$$x_{n-1} = \begin{array}{c}
00010100101001101100110110100101 \\
100101001001101100110110100101 \\
011110100101101101100101100101
\end{array}$$

$$B = \begin{array}{c}
001101001001101100110110100101 \\
001101001001101100110110100101 \\
00011101000101011010010011100101
\end{array}$$
Faster RNG: operations on blocks of bits.

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

\[
B = (((x_{n-1} \ll 6) \text{ XOR } x_{n-1}) \gg 13) \\
x_n = (((x_{n-1} \text{ with last bit at 0} \ll 18) \text{ XOR } B). \\
\]

\[
\begin{array}{l}
x_{n-1} = 00010100101001101100110110100101 \ 100101001001101100110110100101 \\
\quad 001111010010110100100110100100110100101 \\
B = 00111101000101011010010011100101 \\
x_{n-1} = 00010100101001101100110110100100 \ 000101001001101100110110100100100 \\
00010100101001101100110110100100 \ 00110110110100100110100100110100100 \\
\end{array}
\]
Faster RNG: operations on blocks of bits.

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

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

\[
\begin{array}{c|c}
\hline
x_{n-1} &= 00010100101001101100110110100101 \\
& \quad 10010100100110110011011010100101 \\
& \quad 0011110100101101001001101100101 \\
\hline
B &= 00111101000101011010011101000101 \\
\hline
x_{n-1} &= 00010100101001101100110110100100 \\
001010010100110110011011010100100 \\
001101101001101101010010011011011 \\
\hline
x_n &= 0011010010010011110110010001010110101101
\end{array}
\]
Faster RNG: operations on blocks of bits.

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

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

$$
x_n = (((x_{n-1} \text{ with last bit at 0} \ll 18) \text{ XOR } B).
$$

\begin{align*}
\begin{array}{c}
\text{\textbf{x}_{n-1} =} \\
\begin{array}{l}
00010100101001101100110110100101 \\
100101001001101100110110100101 \\
011110100010101101001011100101
\end{array}
\end{array}
\begin{array}{c}
\text{\textbf{B} =} \\
00111101000101011010110100101001100101
\end{array}
\begin{array}{c}
\text{\textbf{x}_n =} \\
00010100101001101100110110100100 \\
00110110100100011110100010101101
\end{array}
\end{align*}

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:

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

<table>
<thead>
<tr>
<th>$x_{n-1}$</th>
<th>00010100101001101100110110100101 100101 00101001101100110110100101</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>00111101000101011010110011000001</td>
</tr>
<tr>
<td>$x_n$</td>
<td>001101010011001100110110100100011110100010101101</td>
</tr>
</tbody>
</table>

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

\[ x_n = A x_{n-1} \pmod{2} = (x_{n,0}, \ldots, x_{n,k-1})^t, \quad \text{(state, } k \text{ bits)} \]
\[ y_n = B x_n \pmod{2} = (y_{n,0}, \ldots, y_{n,w-1})^t, \quad \text{(} w \text{ bits)} \]
\[ u_n = \sum_{j=1}^w y_{n,j-1} 2^{-j} = y_{n,0} y_{n,1} y_{n,2} \cdots, \quad \text{(output)} \]
General linear recurrences modulo 2

\[ x_n = A x_{n-1} \mod 2 = (x_{n,0}, \ldots, x_{n,k-1})^t, \quad \text{(state, k bits)} \]
\[ y_n = B x_n \mod 2 = (y_{n,0}, \ldots, y_{n,w-1})^t, \quad \text{(w bits)} \]
\[ u_n = \sum_{j=1}^w y_{n,j-1} 2^{-j} = \cdot y_{n,0} y_{n,1} y_{n,2} \ldots, \quad \text{(output)} \]

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

\[ x_n = A x_{n-1} \mod 2 = (x_{n,0}, \ldots, x_{n,k-1})^t, \quad (\text{state, } k \text{ bits}) \]

\[ y_n = B x_n \mod 2 = (y_{n,0}, \ldots, y_{n,w-1})^t, \quad (w \text{ bits}) \]

\[ u_n = \sum_{j=1}^{w} y_{n,j-1} 2^{-j} = \cdot y_{n,0} \ y_{n,1} \ y_{n,2} \cdot \cdot \cdot, \quad (\text{output}) \]

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 \( x_n \) and of \( y_n \) follows the recurrence

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

with characteristic polynomial

\[ P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(A - zI). \]

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 $2^\ell$ equal intervals.
Gives $2^{s\ell}$ cubic boxes.

For each $s$ and $\ell$, the $s\ell$ bits that determine the box can be written as $Mx_0$. Each box contains $2^{k-s\ell}$ points of $\Psi_s$ iff $M$ has (full) rank $s\ell$. We then say that those points are equidistributed for $\ell$ bits in $s$ dimensions.
Uniformity measures based on equidistribution.

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

For each $s$ and $\ell$, the $s\ell$ bits that determine the box can be written as $Mx_0$. Each box contains $2^{k-s\ell}$ points of $\Psi_s$ iff $M$ has (full) rank $s\ell$. We then say that those points are equidistributed for $\ell$ bits in $s$ dimensions.

If this holds for all $s$ and $\ell$ such that $s\ell \leq k$, the RNG is called maximally equidistributed.
Uniformity measures based on equidistribution.

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

For each $s$ and $\ell$, the $s\ell$ bits that determine the box can be written as $M x_0$. Each box contains $2^{k-s\ell}$ points of $\Psi_s$ iff $M$ has (full) rank $s\ell$. We then say that those points are equidistributed for $\ell$ bits in $s$ dimensions.

If this holds for all $s$ and $\ell$ 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.
Linear recurrence modulo $m$

State (vector) $x_n$ evolves as

$$x_n = A x_{n-1} \mod m.$$ 

Jumping Ahead:

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

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

If output function $u_n = g(x_n)$ is also linear, one can study the uniformity of each $\Psi_s$ 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!

$g_c : (k\text{-bit counter}) \mapsto (k\text{-bit output})$, 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!

$g_c : (k\text{-bit counter}) \mapsto (k\text{-bit output})$, 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 $\Psi_s$.
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, 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):

```c
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);
```
___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);
    }
}
Hypothesis $H_0$: “$\{u_0, u_1, u_2, \ldots\}$ are i.i.d. $U(0, 1)$ r.v.’s”.
We know that $H_0$ is false, but can we detect it?
Empirical statistical Tests

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

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

Different tests detect different deficiencies.
Empirical statistical Tests

Hypothesis $\mathcal{H}_0$: “$\{u_0, u_1, u_2, \ldots\}$ are i.i.d. $U(0, 1)$ r.v.’s”.

We know that $\mathcal{H}_0$ is false, but can we detect it?

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

Different tests detect different deficiencies.

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

Ultimate dream: Build an 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 \(\mathcal{H}_0\), \(C \approx \text{Poisson of mean } \lambda = n^2/(2k)\), if \(k\) is large and \(\lambda\) is small. If we observe \(c\) collisions, we compute the \(p\)-values:

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

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 $a = 12$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\lambda$</th>
<th>$C$</th>
<th>$p^{-}(C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1/2</td>
<td>0</td>
<td>0.6281</td>
</tr>
</tbody>
</table>

Graph showing $u_{n+1}$ vs $u_n$.
Example: LCG with $m = 101$ and $a = 12$:

\[
u_{n+1} = \frac{n \lambda C}{p^{-}(C)}
\]

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\lambda$</th>
<th>$C$</th>
<th>$p^{-}(C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1/2</td>
<td>0</td>
<td>0.6281</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0</td>
<td>0.1304</td>
</tr>
</tbody>
</table>
Example: LCG with $m = 101$ and $a = 12$:

<table>
<thead>
<tr>
<th>n</th>
<th>$\lambda$</th>
<th>C</th>
<th>$p^{-}(C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1/2</td>
<td>0</td>
<td>0.6281</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0</td>
<td>0.1304</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>1</td>
<td>0.0015</td>
</tr>
</tbody>
</table>
LCG with $m = 101$ and $a = 51$:

\[
u_{n+1}
\]

\[
\begin{array}{cccc}
  n & \lambda & C & p^+(C) \\
  10 & \frac{1}{2} & 1 & 0.3718
\end{array}
\]
LCG with \( m = 101 \) and \( a = 51 \):

\[
\begin{array}{cccc}
\begin{array}{c|c|c|c}
 n & \lambda & C & p^+(C) \\
- & - & - & - \\
10 & 1/2 & 1 & 0.3718 \\
20 & 2 & 5 & 0.0177 \\
\end{array}
\end{array}
\]
LCG with $m = 101$ and $a = 51$:

\[ u_{n+1} \]

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\lambda$</th>
<th>$C$</th>
<th>$p^+(C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$1/2$</td>
<td>1</td>
<td>0.3718</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>5</td>
<td>0.0177</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>20</td>
<td>$2.2 \times 10^{-9}$</td>
</tr>
</tbody>
</table>
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 $n = 10000$ 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


- 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

$p = \text{period length};$

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.

<table>
<thead>
<tr>
<th>Generator</th>
<th>log$_2 \rho$</th>
<th>t-32</th>
<th>t-64</th>
<th>S-Crush</th>
<th>Crush</th>
<th>B-Crush</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCG in Microsoft VisualBasic</td>
<td>24</td>
<td>3.9</td>
<td>0.66</td>
<td>14</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{32}$, 69069, 1), VAX</td>
<td>32</td>
<td>3.2</td>
<td>0.67</td>
<td>11</td>
<td>106</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{32}$, 1099087573, 0) Fishman</td>
<td>30</td>
<td>3.2</td>
<td>0.66</td>
<td>13</td>
<td>110</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{48}$, 25214903917, 11), Unix</td>
<td>48</td>
<td>4.1</td>
<td>0.65</td>
<td>4</td>
<td>21</td>
<td>—</td>
</tr>
<tr>
<td>Java.util.Random</td>
<td>47</td>
<td>6.3</td>
<td>0.76</td>
<td>1</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>LCG($2^{48}$, 44485709377909, 0), Cray</td>
<td>46</td>
<td>4.1</td>
<td>0.65</td>
<td>5</td>
<td>24</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{59}$, 13$^{13}$, 0), NAG</td>
<td>57</td>
<td>4.2</td>
<td>0.76</td>
<td>1</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>LCG($2^{31}-1$, 16807, 0), Wide use</td>
<td>31</td>
<td>3.8</td>
<td>3.6</td>
<td>3</td>
<td>42</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{31}-1$, 397204094, 0), SAS</td>
<td>31</td>
<td>19.0</td>
<td>4.0</td>
<td>2</td>
<td>38</td>
<td>—</td>
</tr>
<tr>
<td>LCG($2^{31}-1$, 950706376, 0), IMSL</td>
<td>31</td>
<td>20.0</td>
<td>4.0</td>
<td>2</td>
<td>42</td>
<td>—</td>
</tr>
<tr>
<td>LCG($10^{12}-11$, ..., 0), Maple</td>
<td>39.9</td>
<td>87.0</td>
<td>25.0</td>
<td>1</td>
<td>22</td>
<td>34</td>
</tr>
<tr>
<td>Generator</td>
<td>( \log_2 \rho )</td>
<td>t-32</td>
<td>t-64</td>
<td>S-Crush</td>
<td>Crush</td>
<td>B-Crush</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------</td>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>Wichmann-Hill, MS-Excel</td>
<td>42.7</td>
<td>10.0</td>
<td>11.2</td>
<td>1</td>
<td>12</td>
<td>22</td>
</tr>
<tr>
<td>CombLec88, boost</td>
<td>61</td>
<td>7.0</td>
<td>1.2</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Knuth(38)</td>
<td>56</td>
<td>7.9</td>
<td>7.4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ran2, in Numerical Recipes</td>
<td>61</td>
<td>7.5</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CombMRG96</td>
<td>185</td>
<td>9.4</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRG31k3p</td>
<td>185</td>
<td>7.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRG32k3a SSJ + others</td>
<td>191</td>
<td>10.0</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRG63k3a</td>
<td>377</td>
<td>—</td>
<td>4.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LFib((2^{31}, 55, 24, +), Knuth)</td>
<td>85</td>
<td>3.8</td>
<td>1.1</td>
<td>2</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>LFib((2^{31}, 55, 24, -), Matpack)</td>
<td>85</td>
<td>3.9</td>
<td>1.5</td>
<td>2</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>ran3, in Numerical Recipes</td>
<td>2.2</td>
<td>0.9</td>
<td></td>
<td>11</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>LFib((2^{48}, 607, 273, +), boost)</td>
<td>638</td>
<td>2.4</td>
<td>1.4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Unix-random-32</td>
<td>37</td>
<td>4.7</td>
<td>1.6</td>
<td>5</td>
<td>101</td>
<td>—</td>
</tr>
<tr>
<td>Unix-random-64</td>
<td>45</td>
<td>4.7</td>
<td>1.5</td>
<td>4</td>
<td>57</td>
<td>—</td>
</tr>
<tr>
<td>Unix-random-128</td>
<td>61</td>
<td>4.7</td>
<td>1.5</td>
<td>2</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>Generator</td>
<td>log₂ ρ</td>
<td>t-32</td>
<td>t-64</td>
<td>S-Crush</td>
<td>Crush</td>
<td>B-Crush</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>Knuth-ran_array2</td>
<td>129</td>
<td>5.0</td>
<td>2.6</td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Knuth-ranf_array2</td>
<td>129</td>
<td>11.0</td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWB(2^{24}, 10, 24)</td>
<td>567</td>
<td>9.4</td>
<td>3.4</td>
<td>2</td>
<td>30</td>
<td>46</td>
</tr>
<tr>
<td>SWB(2^{32} − 5, 22, 43)</td>
<td>1376</td>
<td>3.9</td>
<td>1.5</td>
<td></td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>Mathematica-SWB</td>
<td>1479</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>15</td>
<td>—</td>
</tr>
<tr>
<td>GFSR(250, 103)</td>
<td>250</td>
<td>3.6</td>
<td>0.9</td>
<td>1</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>TT800</td>
<td>800</td>
<td>4.0</td>
<td>1.1</td>
<td></td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>MT19937, widely used</td>
<td>19937</td>
<td>4.3</td>
<td>1.6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>WELL19937a</td>
<td>19937</td>
<td>4.3</td>
<td>1.3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LFSR113</td>
<td>113</td>
<td>4.0</td>
<td>1.0</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LFSR258</td>
<td>258</td>
<td>6.0</td>
<td>1.2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Marsaglia-xorshift</td>
<td>32</td>
<td>3.2</td>
<td>0.7</td>
<td>5</td>
<td>59</td>
<td>—</td>
</tr>
<tr>
<td>Generator</td>
<td>$\log_2 \rho$</td>
<td>t-32</td>
<td>t-64</td>
<td>S-Crush</td>
<td>Crush</td>
<td>B-Crush</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------</td>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>Matlab-rand, (until 2008)</td>
<td>1492</td>
<td>27.0</td>
<td>8.4</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Matlab in randn (normal)</td>
<td>64</td>
<td>3.7</td>
<td>0.8</td>
<td>1</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>SuperDuper-73, in S-Plus</td>
<td>62</td>
<td>3.3</td>
<td>0.8</td>
<td>1</td>
<td>25</td>
<td>—</td>
</tr>
<tr>
<td>R-MultiCarry, (changed)</td>
<td>60</td>
<td>3.9</td>
<td>0.8</td>
<td>2</td>
<td>40</td>
<td>—</td>
</tr>
<tr>
<td>KISS93</td>
<td>95</td>
<td>3.8</td>
<td>0.9</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>KISS99</td>
<td>123</td>
<td>4.0</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES (OFB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES (CTR)</td>
<td>130</td>
<td>10.3</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES (KTR)</td>
<td>130</td>
<td>10.2</td>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA-1 (OFB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA-1 (CTR)</td>
<td>442</td>
<td>30.9</td>
<td>10.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.