Uniform Random Number Generators

Pierre L'Ecuyer
Canada Research Chair in Stochastic Simulation and Optimization
DIRO, Université de Montréal, Canada

- Requirements, applications, multiple streams and substreams.
- Design principles and quality criteria.
- Examples: generators based on linear recurrences.
  Theoretical analysis and implementation issues.
- Combined and mixed linear/nonlinear generators.
- Empirical statistical testing.
  Empirical evaluation of some widely-used generators.
- Conclusion. RNGs for grid computing?

Articles and software: http://www.iro.umontreal.ca/~lecuyer

Random numbers? What do we want?

Produce randomly-looking sequences; e.g.:

— bit sequences: 011010100110110101001101100101000111...

— integers from 0 to 99 (say): 31, 83, 02, 72, 54, 26, ...

— real numbers between 0 and 1.

Random numbers? What do we want?

Produce randomly-looking sequences; e.g.:
— bit sequences: 01101010011011010100110110010101000111...
— integers from 0 to 99 (say): 31, 83, 02, 72, 54, 26, ...
— real numbers between 0 and 1.

Physical devices: dices, balls, roulette wheels, ...
thermal noise in resistances of electronic circuits,
devices that rely on quantum physics, and so on.

Random numbers? What do we want?

Produce randomly-looking sequences; e.g.:
— bit sequences: 01101010011011010100110110010101000111...
— integers from 0 to 99 (say): 31, 83, 02, 72, 54, 26, ...
— real numbers between 0 and 1.

Physical devices: dices, balls, roulette wheels, ...
thermal noise in resistances of electronic circuits,
devices that rely on quantum physics, and so on.

Contains true entropy, but not very convenient, not reproducible, not always reliable, and no (or little) mathematical analysis.

Can improve reliability by combining bits (XOR).

Many such devices on the market.

Random numbers? What do we want?

Produce randomly-looking sequences; e.g.:
— bit sequences: 01101010011011010100110110010101000111...
— integers from 0 to 99 (say): 31, 83, 02, 72, 54, 26, ...
— real numbers between 0 and 1.

Physical devices: dices, balls, roulette wheels, ...
thermal noise in resistances of electronic circuits,
devices that rely on quantum physics, and so on.

Contains true entropy, but not very convenient, not reproducible, not always reliable, and no (or little) mathematical analysis.

Can improve reliability by combining bits (XOR).

Many such devices on the market.

Algorithmic generators (or pseudo-random): Once the parameters and the seed are selected, everything else becomes deterministic. But much more convenient. No special hardware required.

1. Computer games: Good looking appearance may suffice.

1. Computer games: Good looking appearance may suffice.

2. Stochastic simulation (Monte Carlo): Simulate the behavior of complex systems. Want to reproduce the relevant statistical properties of the mathematical model.

Typical RNG: imitates a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniform over the interval (0,1).

1. Computer games: Good looking appearance may suffice.

2. Stochastic simulation (Monte Carlo): Simulate the behavior of complex systems. Want to reproduce the relevant statistical properties of the mathematical model.

Typical RNG: imitates a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniform over the interval (0,1).

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

1. Computer games: Good looking appearance may suffice.

2. Stochastic simulation (Monte Carlo): Simulate the behavior of complex systems. Want to reproduce the relevant statistical properties of the mathematical model.

Typical RNG: imitates a sequence $U_0, U_1, U_2, \ldots$ of independent random variables uniform over the interval (0,1).

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

4. Cryptology: Requirements are even stronger. Observing any part the output should not help guessing (with reasonable effort) any other part.

Requirements for stochastic simulation

We want to use an RNG to imitate a sequence $U_0, U_1, U_2, \ldots$ of independent random variables, uniformly distributed over $(0, 1)$.

Requirements for stochastic simulation

We want to use an RNG to imitate a sequence $U_0, U_1, U_2, \ldots$ of independent random variables, uniformly distributed over $(0, 1)$.

To generate variates from other distributions, we apply transformations to these U_j. For example, inversion: if $X_j = F^{-1}(U_j)$, then X_j imitates a random variable with distribution function F.

Requirements for stochastic simulation

We want to use an RNG to imitate a sequence $U_0, U_1, U_2, \ldots$ of independent random variables, uniformly distributed over $(0, 1)$.

To generate variates from other distributions, we apply transformations to these U_j. For example, inversion: if $X_j = F^{-1}(U_j)$, then X_j imitates a random variable with distribution function F.

Simulation on **multiple processors**: need multiple streams of random numbers.

Requirements for stochastic simulation

We want to use an RNG to imitate a sequence $U_0, U_1, U_2, \ldots$ of independent random variables, uniformly distributed over $(0, 1)$.

To generate variates from other distributions, we apply transformations to these U_j. For example, inversion: if $X_j = F^{-1}(U_j)$, then X_j imitates a random variable with distribution function F.

Simulation on **multiple processors**: need multiple streams of random numbers.

Also for **comparing** similar systems with common random numbers.

Suppose we simulate a communication network, or a telephone call center, or a supply chain or logistic system, or a factory, or the dynamic management of an investment portfolio, etc.

We want to compare two similar configurations (or control policies) for a given system.

The difference in performance will be due partly to the difference of configuration, and partly to stochastic noise. We want to minimize this second part.

Basic idea: simulate the two configurations with the same uniform random numbers U_j, used at exactly the same places.
There are several theoretical results concerning the efficiency improvement (variance reduction) that this brings.

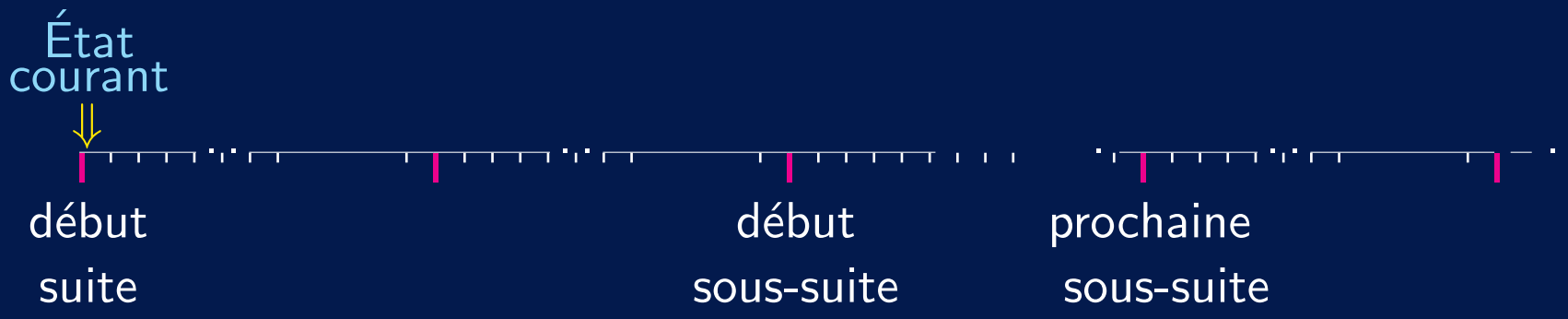
This requires good synchronization of the random numbers, and this can be complicated to implement and manage when the two configurations do not need the same number of U_j's (e.g., we may need one random variate in one case and not in the other case).

Basic idea: simulate the two configurations with the same uniform random numbers U_j, used at exactly the same places.
There are several theoretical results concerning the efficiency improvement (variance reduction) that this brings.

This requires good synchronization of the random numbers, and this can be complicated to implement and manage when the two configurations do not need the same number of U_j's (e.g., we may need one random variate in one case and not in the other case).

A solution: RNG with multiple streams and substreams.

Each stream can be seen as a virtual RNG.
It is also partitioned into substreams.
We should be able to create as many "independent' streams as we want.

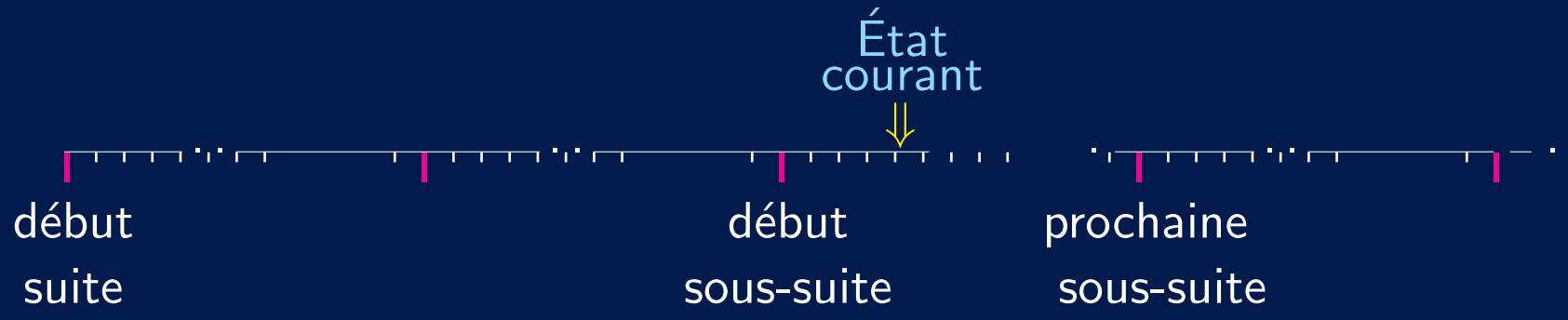


Basic idea: simulate the two configurations with the same uniform random numbers U_j, used at exactly the same places.
There are several theoretical results concerning the efficiency improvement (variance reduction) that this brings.

This requires good synchronization of the random numbers, and this can be complicated to implement and manage when the two configurations do not need the same number of U_j's (e.g., we may need one random variate in one case and not in the other case).

A solution: RNG with multiple streams and substreams.

Each stream can be seen as a virtual RNG.
It is also partitioned into substreams.
We should be able to create as many "independent' streams as we want.

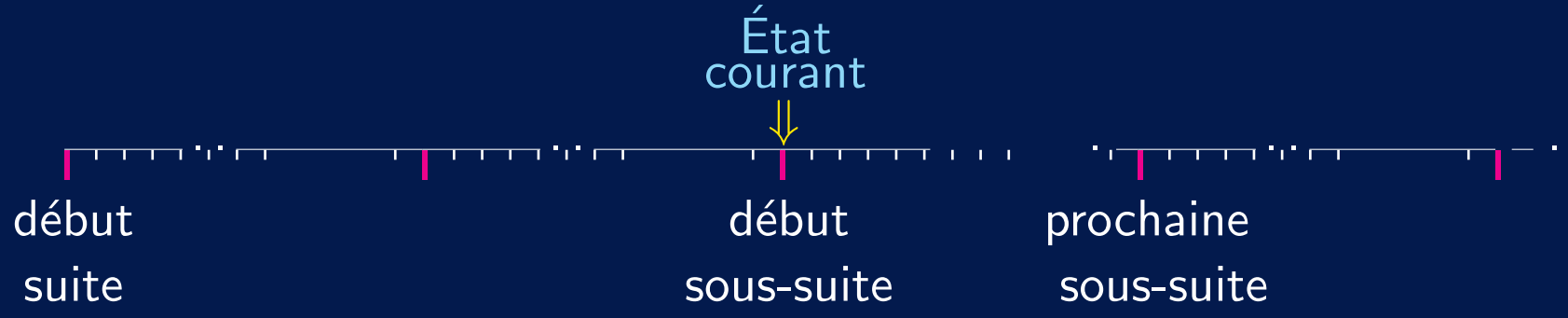


Basic idea: simulate the two configurations with the same uniform random numbers U_j, used at exactly the same places.
There are several theoretical results concerning the efficiency improvement (variance reduction) that this brings.

This requires good synchronization of the random numbers, and this can be complicated to implement and manage when the two configurations do not need the same number of U_j's (e.g., we may need one random variate in one case and not in the other case).

A solution: RNG with multiple streams and substreams.

Each stream can be seen as a virtual RNG.
It is also partitioned into substreams.
We should be able to create as many "independent' streams as we want.

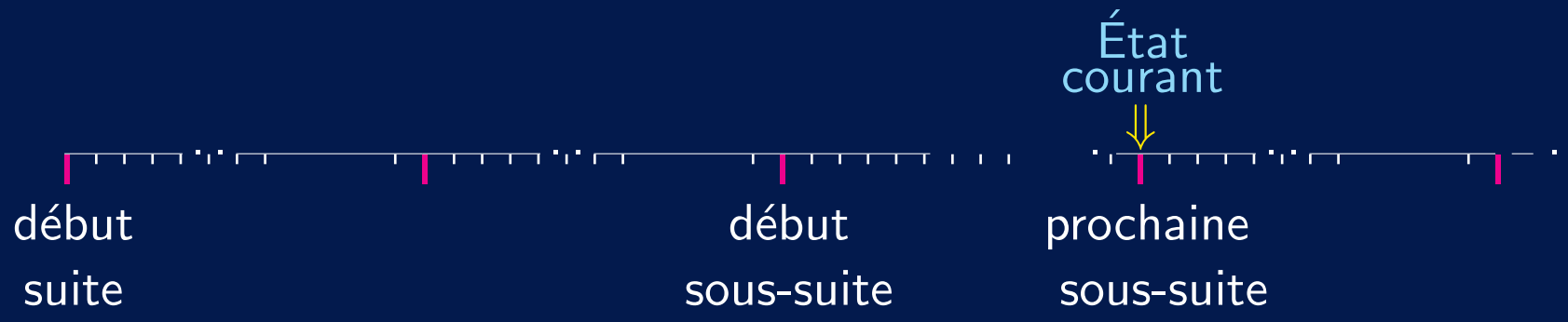


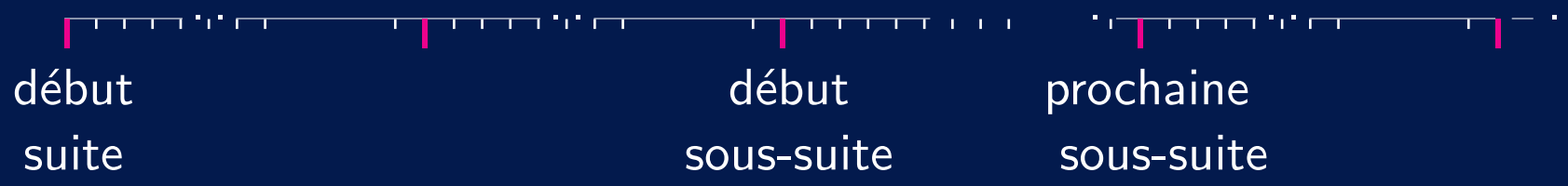
Basic idea: simulate the two configurations with the same uniform random numbers U_j, used at exactly the same places.
There are several theoretical results concerning the efficiency improvement (variance reduction) that this brings.

This requires good synchronization of the random numbers, and this can be complicated to implement and manage when the two configurations do not need the same number of U_j's (e.g., we may need one random variate in one case and not in the other case).

A solution: RNG with multiple streams and substreams.

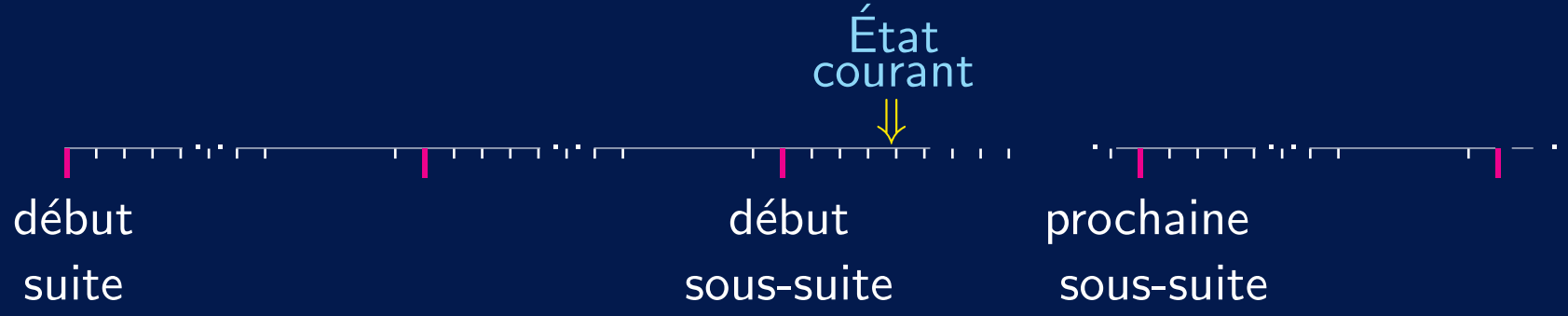
Each stream can be seen as a virtual RNG.
It is also partitioned into substreams.
We should be able to create as many "independent' streams as we want.





Suppose that for a sequence of customers arriving to a system, we must generate:
(1) the arrival time of each customer;
(2) a service time at server A for each customer;
(3) a service time at server B for groups of 10 to 20 customers.

To maintain synchonization, we can use one stream for each of these.



Suppose that for a sequence of customers arriving to a system, we must generate:
(1) the arrival time of each customer;
(2) a service time at server A for each customer;
(3) a service time at server B for groups of 10 to 20 customers.

To maintain synchonization, we can use one stream for each of these.

Example of a Java interface (in SSJ)

```
public interface RandomStream {
   public void  resetStartStream ();
```

Reinitializes the stream to its initial state.

```
   public void  resetStartSubstream ();
```

Reinitializes the stream to the beginning of its current substream.

```
   public void  resetNextSubstream ();
```

Reinitializes the stream to the beginning of its next substream.

```
   public double  nextDouble ();
```

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

```
   public int  nextInt (int i, int j);
```

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i + 1, \ldots, j\}$, using this stream.

```
}
```

```
public class MRG32k3a implements RandomStream {
```

An implementation based on a combined MRG with period length near 2^{181}, partitioned into streams whose initial states are 2^{127} steps apart. The substreams have length 2^{76}.

```
   public  MRG32k3a();
```

Constructs a new stream.

```
}
```

```
public class LFSR113 implements RandomStream {
```

An implementation based on a combined LFSR, with period length near 2^{113}.

```
   public  LFSR113();
```

Constructs a new stream.

```
}
```

This system (with MRG32k3a) has been adopted by leading simulation and statistical software such as SAS, Arena, Witness, Simul8, Automod, ns2, MATLAB, etc.

Non-uniform variate generators

Continuous or discrete. Default method: inversion.

```
public class RandomVariateGen {
```

`public RandomVariateGen (RandomStream s, Distribution dist)`

Creates a new generator from the distribution `dist`, using stream `s`.

`public double nextDouble()`

By default, generates a variate by inversion.

Non-uniform variate generators

Continuous or discrete. Default method: inversion.

```
public class RandomVariateGen {
   public RandomVariateGen (RandomStream s, Distribution dist)
```

Creates a new generator from the distribution `dist`, using stream `s`.

```
   public double nextDouble()
```

By default, generates a variate by inversion.

Also specialized (faster) generators for certain distributions.

```
public class Normal extends RandomVariateGen {
   public  Normal (RandomStream s, double mu, double sigma);
```

Constructs a new normal r.v. generator.

```
   public static double  nextDouble (RandomStream s, double mu,
double sigma);
```

Generates a new normal random variate using stream s.

Example: comparing two configurations:

```
          ⋮
RandomStream genArr   = new RandomStream(); // Inter-arriv.
RandomStream genServA = new RandomStream(); // Serveur A.
RandomStream genServB = new RandomStream(); // Serveur B.
          ⋮
```

Example: comparing two configurations:

```
        ⋮
RandomStream genArr   = new RandomStream(); // Inter-arriv.
RandomStream genServA = new RandomStream(); // Serveur A.
RandomStream genServB = new RandomStream(); // Serveur B.
        ⋮
    for (int rep = 0; rep < n; rep++) {
        genArr.resetNextSubstream();
        genServA.resetNextSubstream();
        genServB.resetNextSubstream();
           --- simuler configuration 1 ---

        genArr.resetStartSubstream();
        genServA.resetStartSubstream();
        genServB.resetStartSubstream();
           --- simuler configuration 2 ---
    }
```

Multiple streams are very useful (on a single processor), for example for:

- comparing similar systems
- sensitivity analysis, gradient estimation by finite differences
- optimization of sample function, etc.
- using a (simplified) similar system for an external control variate

Multiple streams are very useful (on a single processor), for example for:

- comparing similar systems
- sensitivity analysis, gradient estimation by finite differences
- optimization of sample function, etc.
- using a (simplified) similar system for an external control variate

Can also be used on multiple processors.

Multiple streams are very useful (on a single processor), for example for:

- comparing similar systems
- sensitivity analysis, gradient estimation by finite differences
- optimization of sample function, etc.
- using a (simplified) similar system for an external control variate

Can also be used on multiple processors.

In SSJ, a `RandomStream` can also be an iterator on a randomized quasi-Monte Carlo point set.

How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space;	s_0, seed (initial state);
$f : \mathcal{S} \to \mathcal{S}$, transition function;	$s_n = f(s_{n-1})$
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;	
$g : \mathcal{S} \to \mathcal{U}$, output function.	$u_n = g(s_n)$

How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space;	s_0, seed (initial state);
$f : \mathcal{S} \to \mathcal{S}$, transition function;	$s_n = f(s_{n-1})$
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;	
$g : \mathcal{S} \to \mathcal{U}$, output function.	$u_n = g(s_n)$

s_0

How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;
$g : \mathcal{S} \to \mathcal{U}$, output function.

s_0, seed (initial state);
$s_n = f(s_{n-1})$

$u_n = g(s_n)$

How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;
$g : \mathcal{S} \to \mathcal{U}$, output function.

s_0, seed (initial state);
$s_n = f(s_{n-1})$

$u_n = g(s_n)$

$$\begin{array}{ccc} s_0 & \xrightarrow{f} & s_1 \\ \downarrow{\scriptstyle g} & & \\ u_0 & & \end{array}$$

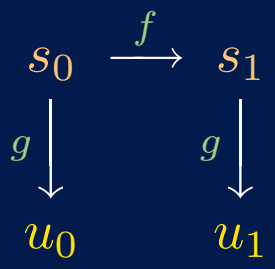
How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space;
$f : \mathcal{S} \to \mathcal{S}$, transition function;
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;
$g : \mathcal{S} \to \mathcal{U}$, output function.

s_0, seed (initial state);
$s_n = f(s_{n-1})$

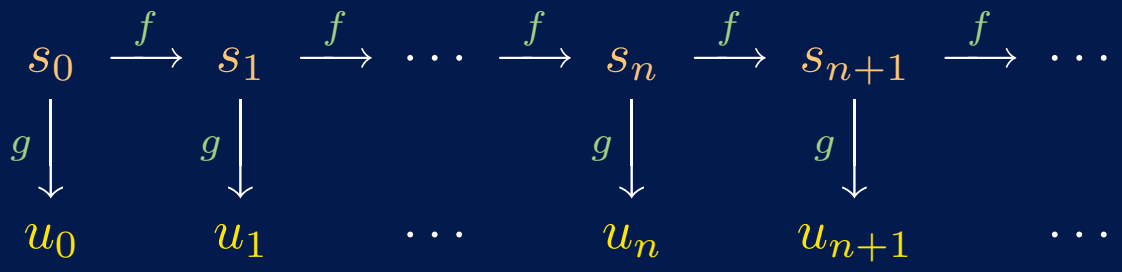
$u_n = g(s_n)$



How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space; s_0, seed (initial state);
$f : \mathcal{S} \to \mathcal{S}$, transition function; $s_n = f(s_{n-1})$
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;
$g : \mathcal{S} \to \mathcal{U}$, output function. $u_n = g(s_n)$

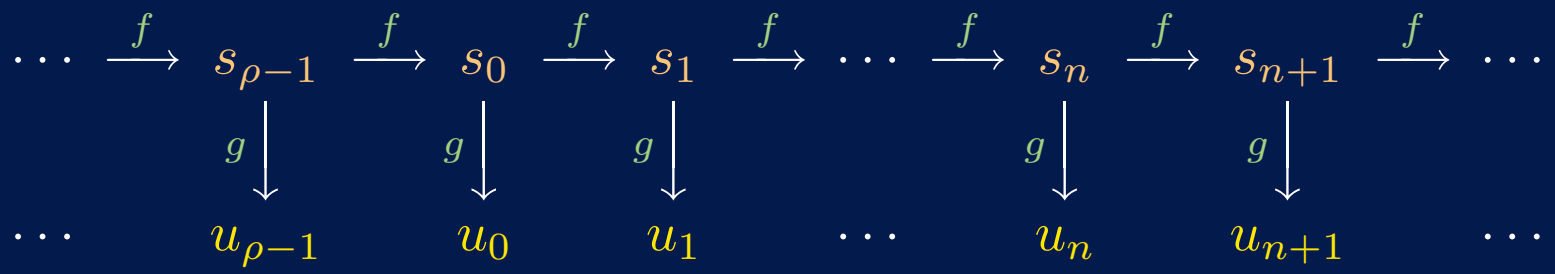


Period: $\rho \leq |\mathcal{S}|$. $s_{i+\rho} = s_i \quad \forall i \geq \tau$. Assume $\tau = 0$.

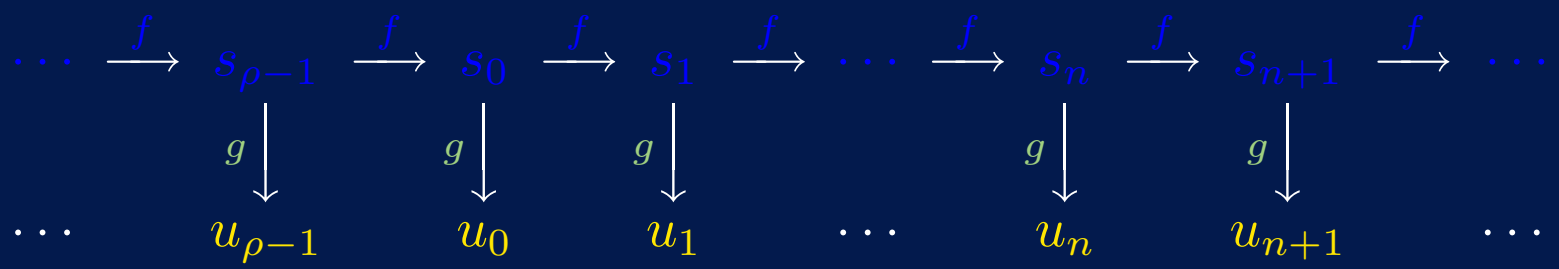
How to Define and Design an RNG?

A (Pseudo)random number generator (RNG):

$\mathcal{S}$, finite state space; s_0, seed (initial state);
$f : \mathcal{S} \to \mathcal{S}$, transition function; $s_n = f(s_{n-1})$
$\mathcal{U}$, output set, often $\mathcal{U} = (0, 1)$;
$g : \mathcal{S} \to \mathcal{U}$, output function. $u_n = g(s_n)$

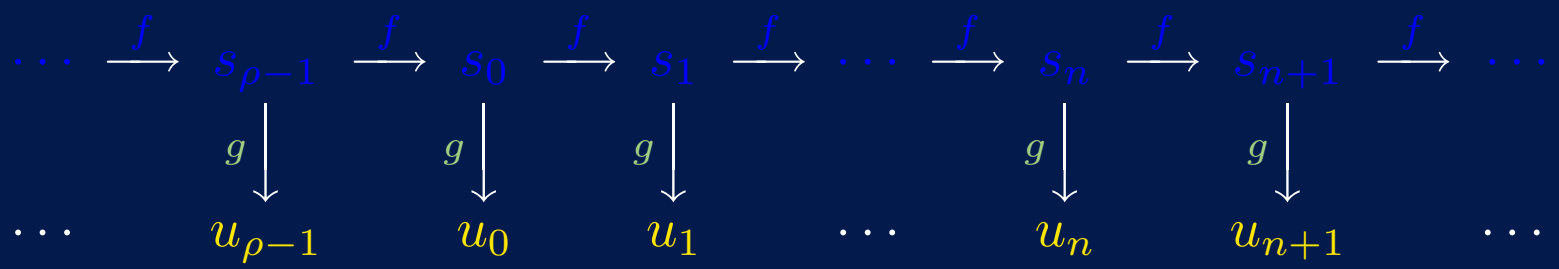


Period: $\rho \leq |\mathcal{S}|$. $s_{i+\rho} = s_i \quad \forall i \geq \tau$. Assume $\tau = 0$.



Aim: By observing only the output $(u_0, u_1, \ldots)$, it should be hard to distinguish it from the realizations of i.i.d. uniform random variables over $\mathcal{U}$.

Utopia: cannot distinguish better than by flipping a fair coin.
That is, passes all statistical tests.

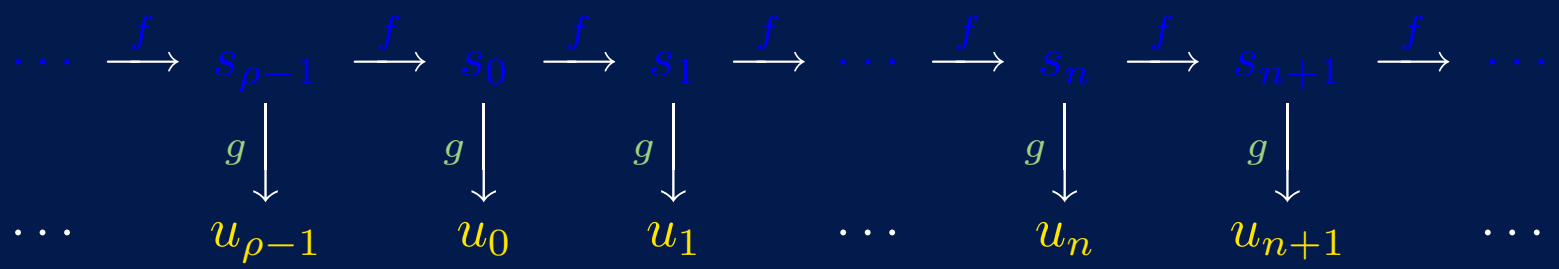


Aim: By observing only the output $(u_0, u_1, \ldots)$, it should be hard to distinguish it from the realizations of i.i.d. uniform random variables over $\mathcal{U}$.

Utopia: cannot distinguish better than by flipping a fair coin.
That is, passes all statistical tests.

Also want high speed, ease of implementation, and perfect reproducibility.

Compromise between speed / good statistical behavior / predictability.



Aim: By observing only the output $(u_0, u_1, \ldots)$, it should be hard to distinguish it from the realizations of i.i.d. uniform random variables over $\mathcal{U}$.

Utopia: cannot distinguish better than by flipping a fair coin.
That is, passes all statistical tests.

Also want high speed, ease of implementation, and perfect reproducibility.

Compromise between speed / good statistical behavior / predictability.

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

The uniform distribution over $[0,1]^t$:

The multiset $\Psi_t = \{(u_0, \ldots, u_{t-1}) = (g(s_0), \ldots, g(s_{t-1})),\ s_0 \in \mathcal{S}\}$ is viewed as the sample space, approximation of $[0,1]^t$.

The uniform distribution over $[0,1]^t$:
The multiset $\Psi_t = \{(u_0, \ldots, u_{t-1}) = (g(s_0), \ldots, g(s_{t-1})),\ s_0 \in \mathcal{S}\}$
is viewed as the sample space, approximation of $[0,1]^t$.

We suggest: Want Ψ_t uniformly spread over $[0,1]^t$ for all t up to some t_0.

The uniform distribution over $[0,1]^t$:
The multiset $\Psi_t = \{(u_0, \ldots, u_{t-1}) = (g(s_0), \ldots, g(s_{t-1})),\ s_0 \in \mathcal{S}\}$
is viewed as the sample space, approximation of $[0,1]^t$.

We suggest: Want Ψ_t uniformly spread over $[0,1]^t$ for all t up to some t_0.

Need a measure of uniformity of Ψ_t (or a measure of discrepancy from the uniform distribution). Several possible definitions.

Important: Must be efficiently computable.
For this, must understand the mathematical structure of Ψ_t.

This is why most good RNGs are based on linear recurrences.

The uniform distribution over $[0,1]^t$:
The multiset $\Psi_t = \{(u_0, \ldots, u_{t-1}) = (g(s_0), \ldots, g(s_{t-1})),\ s_0 \in \mathcal{S}\}$
is viewed as the sample space, approximation of $[0,1]^t$.

We suggest: Want Ψ_t uniformly spread over $[0,1]^t$ for all t up to some t_0.

Need a measure of uniformity of Ψ_t (or a measure of discrepancy from the uniform distribution). Several possible definitions.

Important: Must be efficiently computable.
For this, must understand the mathematical structure of Ψ_t.

This is why most good RNGs are based on linear recurrences.

Why not insist that Ψ_t behaves as a typical set of random points instead?
We need that behavior only for the tiny fraction of Ψ_t that we use.

The uniform distribution over $[0,1]^t$:
The multiset $\Psi_t = \{(u_0, \ldots, u_{t-1}) = (g(s_0), \ldots, g(s_{t-1})),\ s_0 \in \mathcal{S}\}$
is viewed as the sample space, approximation of $[0,1]^t$.

We suggest: Want Ψ_t uniformly spread over $[0,1]^t$ for all t up to some t_0.

Need a measure of uniformity of Ψ_t (or a measure of discrepancy from the uniform distribution). Several possible definitions.

Important: Must be efficiently computable.
For this, must understand the mathematical structure of Ψ_t.

This is why most good RNGs are based on linear recurrences.

Why not insist that Ψ_t behaves as a typical set of random points instead?
We need that behavior only for the tiny fraction of Ψ_t that we use.

Generalization: measure the uniformity of $\Psi_I = \{(u_{i_1}, \ldots, u_{i_t}) \mid s_0 \in \mathcal{S}\}$
for selected sets $I = \{i_1, i_2, \cdots, i_t\}$ of nonsuccessive indexes.
Make sure that Ψ_I is uniform for all $I \in \mathcal{J}$, for a given family $\mathcal{J}$.

Multiple Recursive Generator (MRG)

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

Max. period length: $\rho = m^k - 1$, for m prime.

Multiple Recursive Generator (MRG)

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

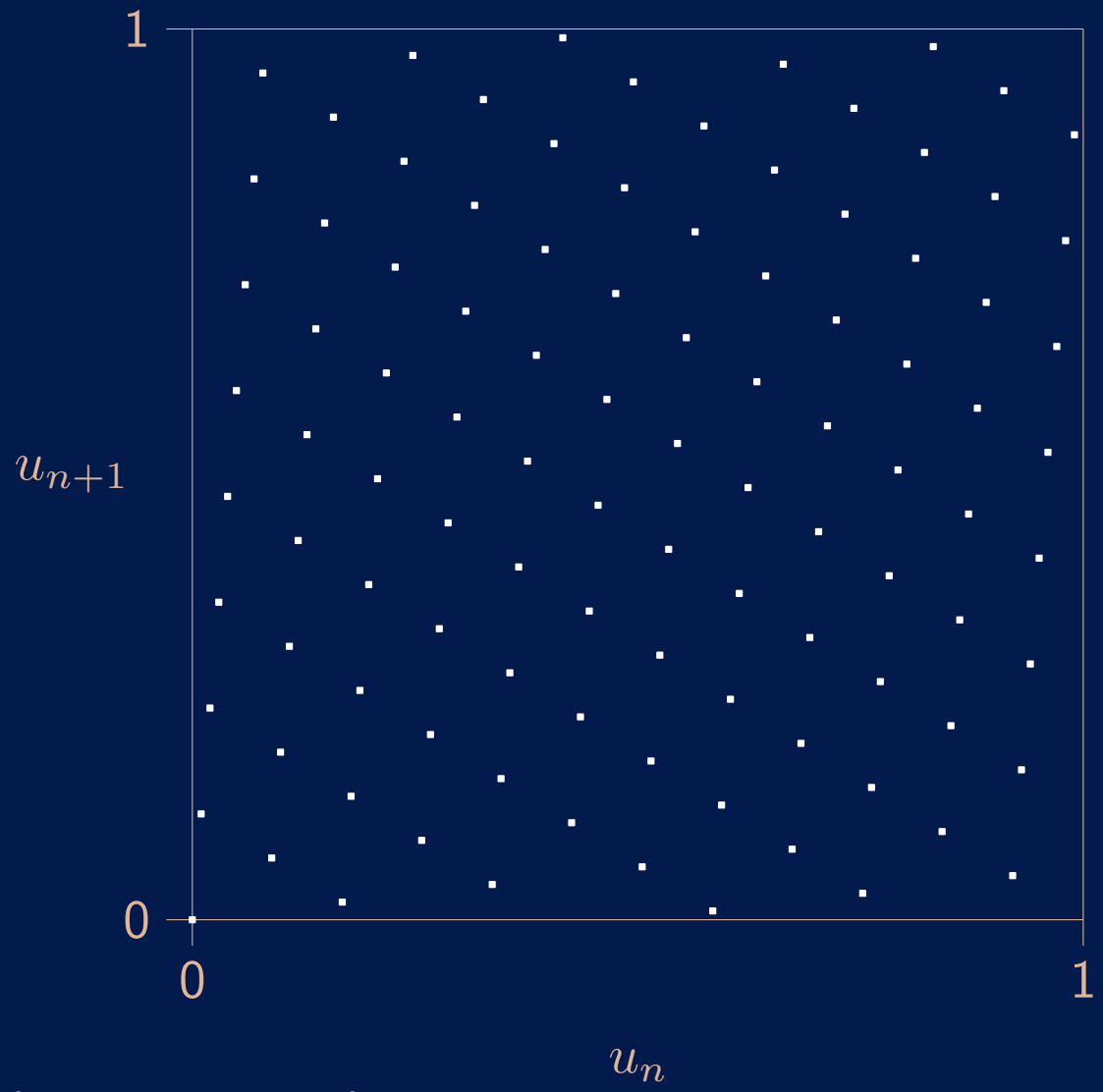
Max. period length: $\rho = m^k - 1$, for m prime.

Structure of Ψ_t: $(x_0, \ldots, x_{k-1})$ can take any value in $\{0, 1, \ldots, m-1\}^k$, then $x_k, x_{k+1}, \ldots$ are determined by the linear recurrence.
So, $(x_0, \ldots, x_{k-1}) \mapsto (x_0, \ldots, x_{k-1}, x_k, \ldots, x_{t-1})$ is a linear mapping.

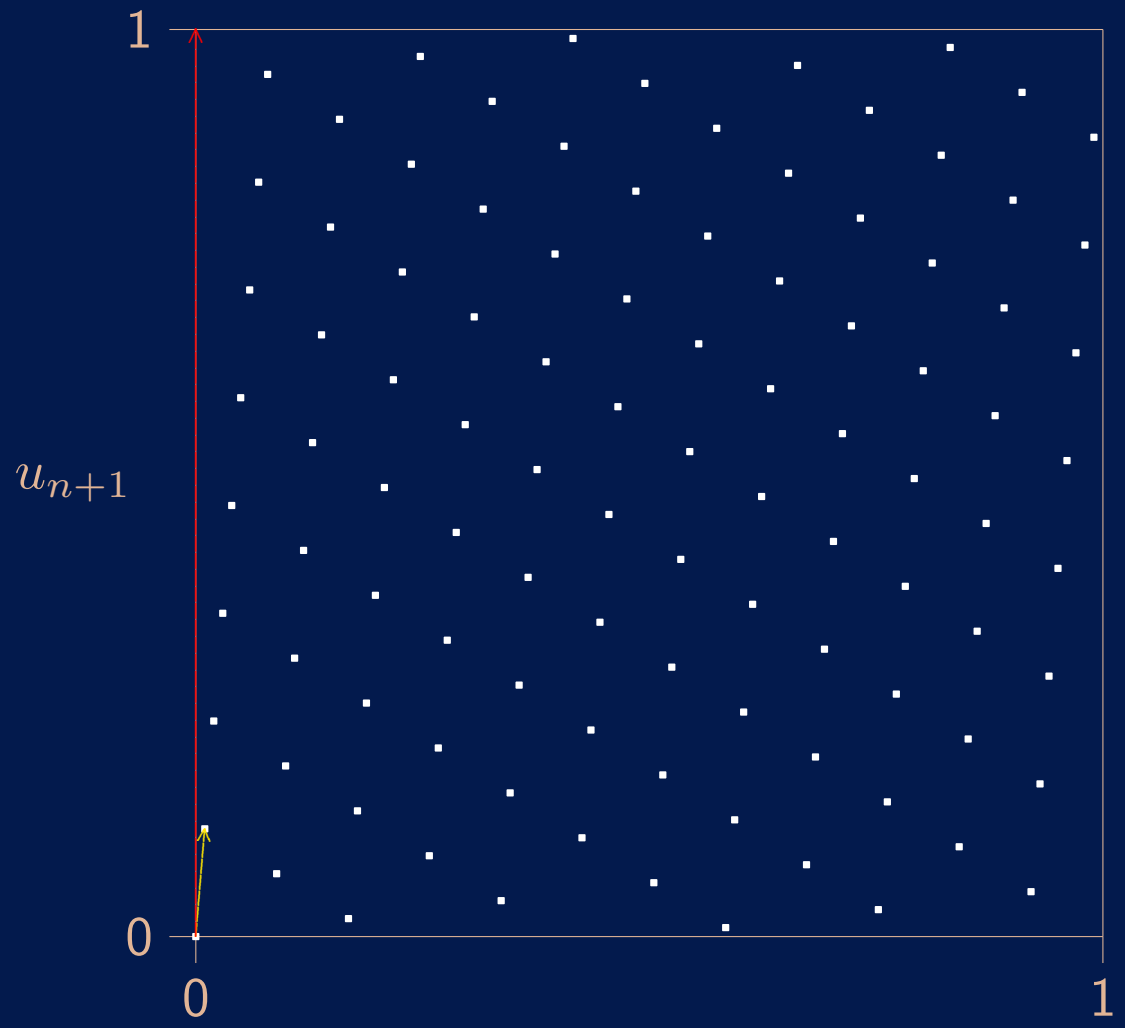
It follows (details omitted) that $\Psi_t = L_t \cap [0,1)^t$ where

$$L_t = \left\{ \mathbf{v} = \sum_{i=1}^{t} z_i \mathbf{v}_i \;\mid\; z_i \in \mathbb{Z} \right\}, \quad \text{a lattice in } \mathbb{R}^t, \text{ with}$$

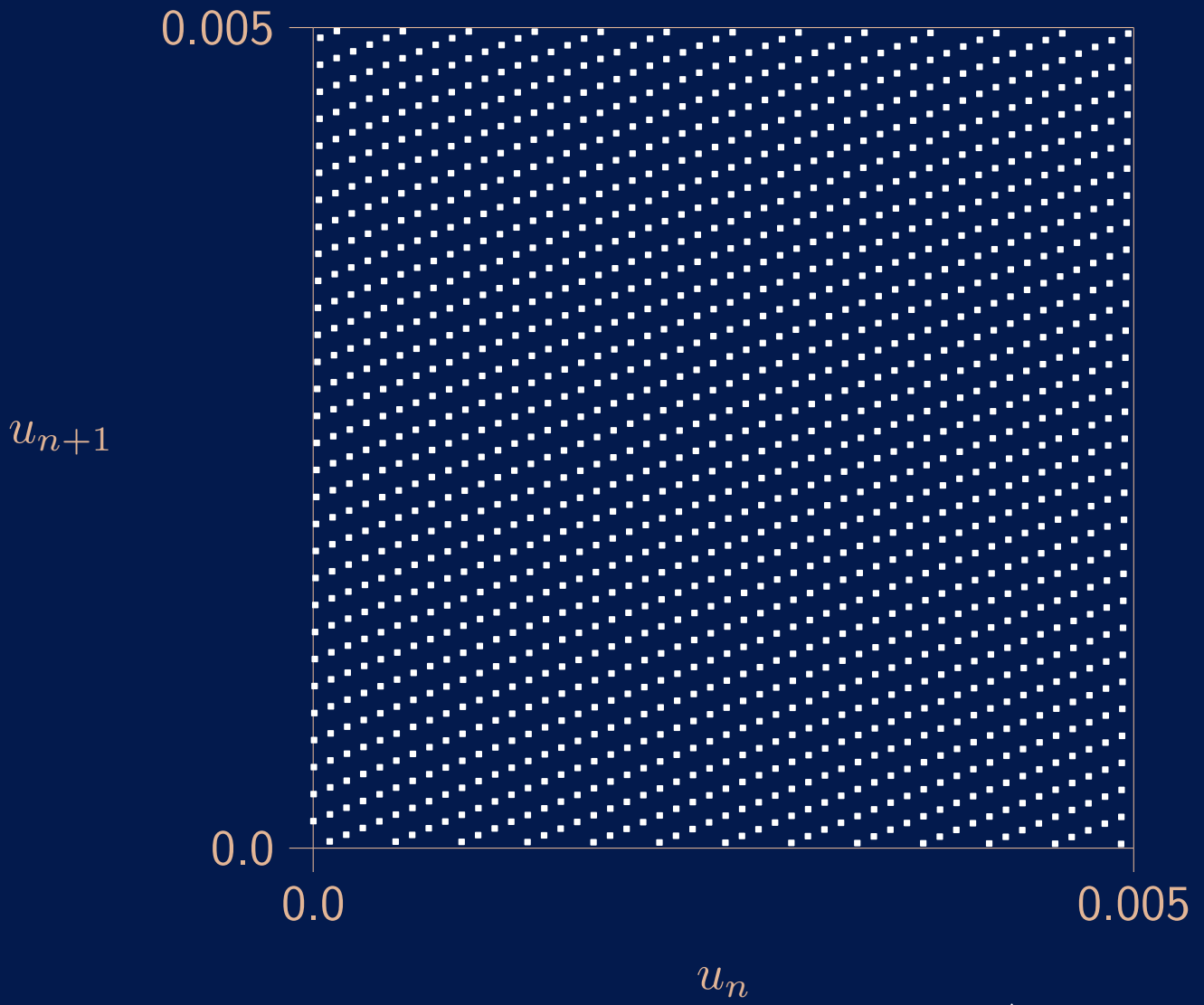
$$\begin{aligned}
\mathbf{v}_1 &= (1, 0, \ldots, 0, x_{1,k}, \ldots, x_{1,t-1})/m \\
\mathbf{v}_2 &= (0, 1, \ldots, 0, x_{2,k}, \ldots, x_{2,t-1})/m \\
&\vdots \qquad\qquad \vdots \\
\mathbf{v}_k &= (0, 0, \ldots, 1, x_{k,k}, \ldots, x_{k,t-1})/m \\
\mathbf{v}_{k+1} &= (0, 0, \ldots, 0, 1, \ldots, 0) \\
&\vdots \qquad\qquad \vdots \\
\mathbf{v}_t &= (0, 0, \ldots, 0, 0, \ldots, 1).
\end{aligned}$$



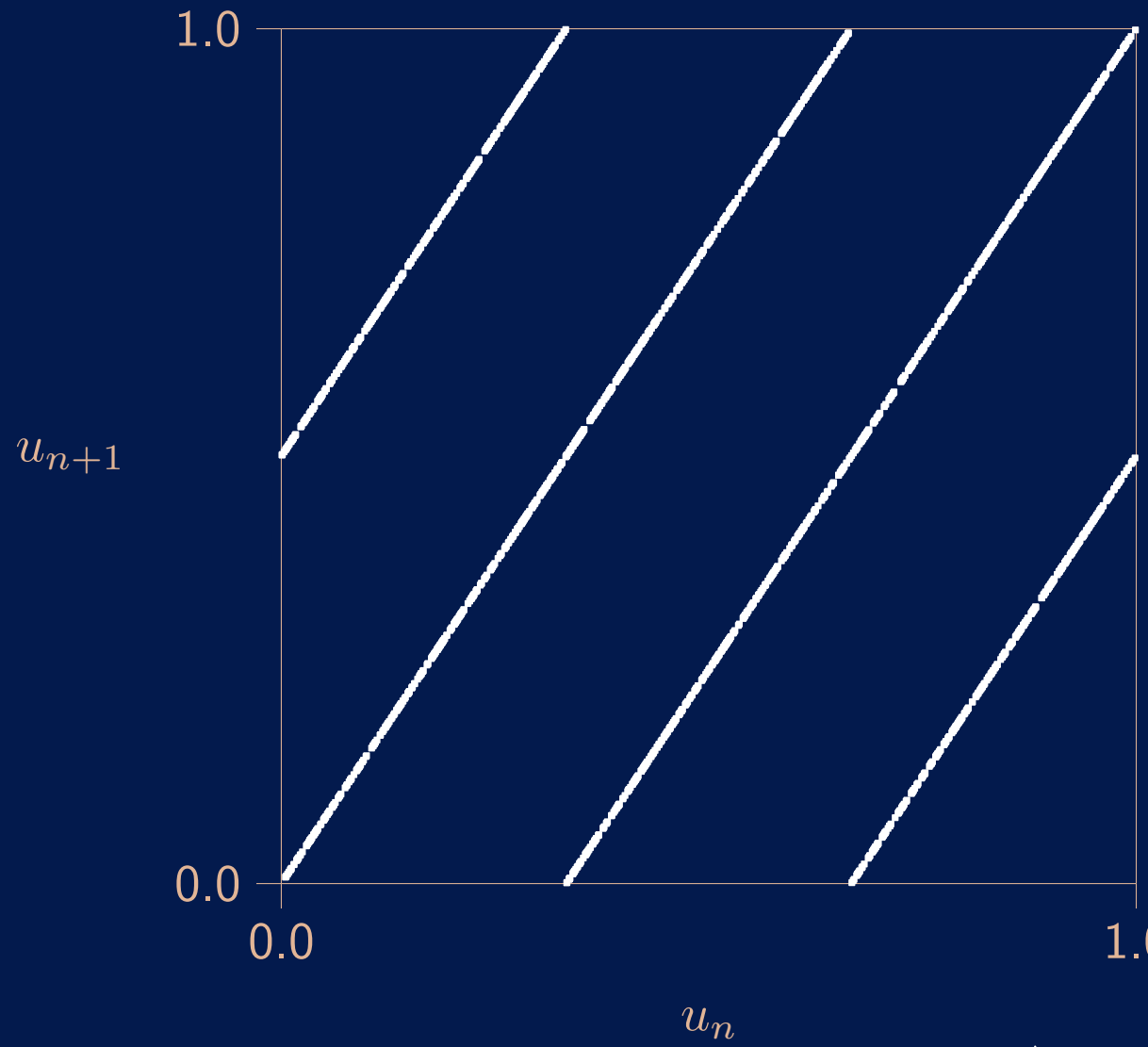
LCG with $m = 101$ and $a_1 = 12$;



LCG with $m = 101$ and $a_1 = 12$; $\mathbf{v}_1 = (1, 12)/m$, $\mathbf{v}_2 = (0, 1)$



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



$x_n = 30233086\, x_{n-1} \bmod 60466169$ and $u_n = x_n/60466169$

Uniformity can be measured in terms of this lattice structure in various dimensions.

Uniformity can be measured in terms of this lattice structure in various dimensions.

Can also take into account projections on subsets of coordinates $I = \{i_1, i_2, \cdots, i_t\}$:

$$\Psi_I \quad = \quad \{(u_{i_1}, \ldots, u_{i_t}) \mid s_0 = (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\} = L_I \cap [0,1)^t.$$

Can make computer searches for good parameters w.r.t. a criterion that account for the uniformity of a selected set of projections.

Uniformity can be measured in terms of this lattice structure in various dimensions.

Can also take into account projections on subsets of coordinates $I = \{i_1, i_2, \cdots, i_t\}$:

$$\Psi_I \quad = \quad \{(u_{i_1}, \ldots, u_{i_t}) \mid s_0 = (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\} = L_I \cap [0,1)^t.$$

Can make computer searches for good parameters w.r.t. a criterion that account for the uniformity of a selected set of projections.

Remark: it can be proved that if $1 + a_{i_2}^2 + \cdots + a_{i_t}^2$ is small, then the lattice structure of Ψ_I is bad.

Efficient Implementations

Requires efficient computation of $ax \bmod m$ for large m.

Efficient Implementations

Requires efficient computation of $ax \bmod m$ for large m.

Approximate factoring.
Valid if ($a^2 < m$) or ($a = \lfloor m/i \rfloor$ where $i^2 < m$). Uses integer arithmetic.
Precompute $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. Then,

$$y = \lfloor x/q \rfloor; \quad x = a(x - yq) - yr; \quad \text{if } x < 0 \text{ then } x = x + m.$$

Efficient Implementations

Requires efficient computation of $ax \bmod m$ for large m.

Approximate factoring.
Valid if ($a^2 < m$) or ($a = \lfloor m/i \rfloor$ where $i^2 < m$). Uses integer arithmetic.
Precompute $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. Then,

$$y = \lfloor x/q \rfloor; \quad x = a(x - yq) - yr; \quad \text{if } x < 0 \text{ then } x = x + m.$$

Floating point in double precision. Valid if $a\,m < 2^{53}$.

double $m, a, x, y;$ int $k;$

$$y = a * x; \quad k = \lfloor y/m \rfloor; \quad x = y - k * m;$$

Efficient Implementations

Requires efficient computation of $ax \bmod m$ for large m.

Approximate factoring.
Valid if ($a^2 < m$) or ($a = \lfloor m/i \rfloor$ where $i^2 < m$). Uses integer arithmetic.
Precompute $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. Then,

$$y = \lfloor x/q \rfloor; \quad x = a(x - yq) - yr; \quad \text{if } x < 0 \text{ then } x = x + m.$$

Floating point in double precision. Valid if $a\,m < 2^{53}$.

double $m, a, x, y;$ int $k;$

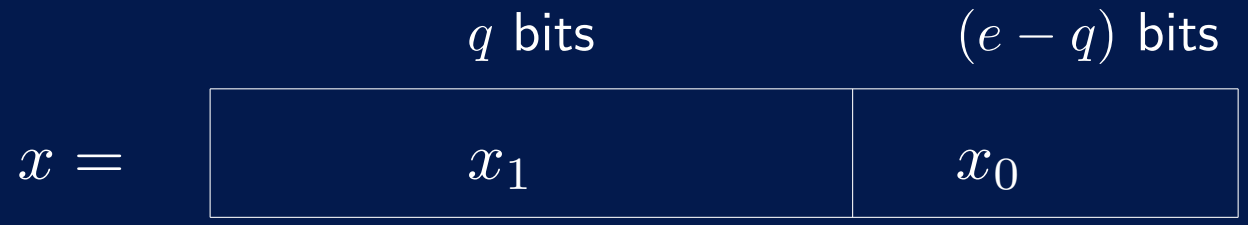
$$y = a * x; \quad k = \lfloor y/m \rfloor; \quad x = y - k * m;$$

Straightforward 64-bit integer arithmetic.
On 64-bit computers, just make sure that $am < 2^{63}$.

Decomposition in powers of 2.

Suppose $a = \pm 2^q \pm 2^r$ and $m = 2^e - h$ for small h.
(Wu 1997 for $h = 1$; L'Ecuyer and Simard 1999 for $h > 1$.)

To compute $y = 2^q x \bmod m$, decompose $x = x_0 + 2^{e-q} x_1$;



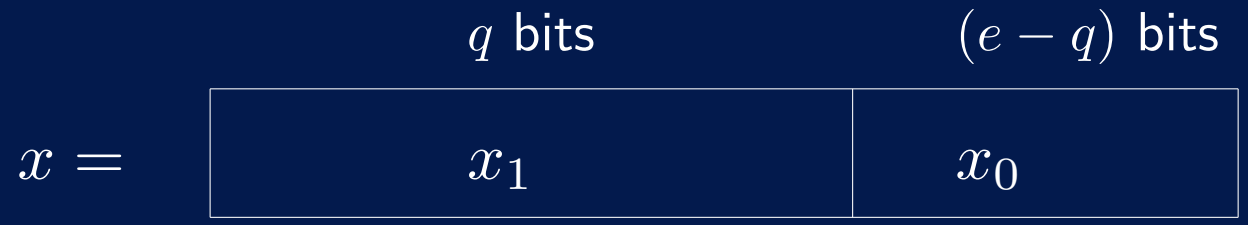
For $h = 1$, y is obtained by swapping x_0 and x_1.

For $h > 1$, requires a single multiplication of x_1 by h, plus a few shifts, masks, additions, and subtractions.

Decomposition in powers of 2.

Suppose $a = \pm 2^q \pm 2^r$ and $m = 2^e - h$ for small h.
(Wu 1997 for $h = 1$; L'Ecuyer and Simard 1999 for $h > 1$.)

To compute $y = 2^q x \bmod m$, decompose $x = x_0 + 2^{e-q} x_1$;



For $h = 1$, y is obtained by swapping x_0 and x_1.

For $h > 1$, requires a single multiplication of x_1 by h, plus a few shifts, masks, additions, and subtractions.

Lagged-Fibonacci (widely used, but bad idea):

$$x_n = (\pm x_{n-r} \pm x_{n-k}) \bmod m.$$

All vectors $(u_n, u_{n+k-r}, u_{n+k})$ lie in only two planes!

Lagged-Fibonacci (widely used, but bad idea):

$$x_n = (\pm x_{n-r} \pm x_{n-k}) \bmod m.$$

All vectors $(u_n, u_{n+k-r}, u_{n+k})$ lie in only two planes!

Same problem with add-with-carry and subtract-with-borrow RNGs.

Common mistake: if we cannot exhaust the period, then we are okay...

Lagged-Fibonacci (widely used, but bad idea):

$$x_n = (\pm x_{n-r} \pm x_{n-k}) \bmod m.$$

All vectors $(u_n, u_{n+k-r}, u_{n+k})$ lie in only two planes!

Same problem with add-with-carry and subtract-with-borrow RNGs.

Common mistake: if we cannot exhaust the period, then we are okay...

Variants that skip values are recommended by Luscher (1994) and Knuth (1997).

Lagged-Fibonacci (widely used, but bad idea):

$$x_n = (\pm x_{n-r} \pm x_{n-k}) \bmod m.$$

All vectors $(u_n, u_{n+k-r}, u_{n+k})$ lie in only two planes!

Same problem with add-with-carry and subtract-with-borrow RNGs.

Common mistake: if we cannot exhaust the period, then we are okay...

Variants that skip values are recommended by Luscher (1994) and Knuth (1997).

Safer AWC/SWB versions proposed by Couture and L'Ecuyer (1995), Klapper and Goresky (2003). But no significant advantage over MRGs.

Combined MRGs. Consider two MRGs (or more...) evolving in parallel:

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

Define the two combinations:

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \bmod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1. \end{aligned}$$

Combined MRGs. Consider two MRGs (or more...) evolving in parallel:

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

Define the two combinations:

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \bmod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1. \end{aligned}$$

One can show (L'Ecuyer 1996) that $\{w_n,\, n \geq 0\}$ is the output sequence of yet another MRG, with modulus $m = m_1 m_2$, and $\{u_n,\, n \geq 0\}$ is almost the same sequence if m_1 and m_2 are close. Can reach period length $(m_1^k - 1)(m_2^k - 1)/2$.

Combined MRGs. Consider two MRGs (or more...) evolving in parallel:

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

Define the two combinations:

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \bmod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1. \end{aligned}$$

One can show (L'Ecuyer 1996) that $\{w_n,\, n \geq 0\}$ is the output sequence of yet another MRG, with modulus $m = m_1 m_2$, and $\{u_n,\, n \geq 0\}$ is almost the same sequence if m_1 and m_2 are close. Can reach period length $(m_1^k - 1)(m_2^k - 1)/2$.

Permits efficient implementation of an MRG with large m and several large nonzero coefficients.

Combined MRGs. Consider two MRGs (or more...) evolving in parallel:

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

Define the two combinations:

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \bmod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1. \end{aligned}$$

One can show (L'Ecuyer 1996) that $\{w_n,\, n \geq 0\}$ is the output sequence of yet another MRG, with modulus $m = m_1 m_2$, and $\{u_n,\, n \geq 0\}$ is almost the same sequence if m_1 and m_2 are close. Can reach period length $(m_1^k - 1)(m_2^k - 1)/2$.

Permits efficient implementation of an MRG with large m and several large nonzero coefficients.

Tables of good parameters and codes in L'Ecuyer (1999) and L'Ecuyer and Touzin (2000).

Combined MRGs. Consider two MRGs (or more...) evolving in parallel:

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

Define the two combinations:

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \bmod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1. \end{aligned}$$

One can show (L'Ecuyer 1996) that $\{w_n,\, n \geq 0\}$ is the output sequence of yet another MRG, with modulus $m = m_1 m_2$, and $\{u_n,\, n \geq 0\}$ is almost the same sequence if m_1 and m_2 are close. Can reach period length $(m_1^k - 1)(m_2^k - 1)/2$.

Permits efficient implementation of an MRG with large m and several large nonzero coefficients.

Tables of good parameters and codes in L'Ecuyer (1999) and L'Ecuyer and Touzin (2000). Packages with multiple streams are available.

Jumping ahead (to make streams and substreams):

If $\mathbf{x}_n = (x_{n-k+1}, \ldots, x_n)^{\mathsf{t}}$ and

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix}$$

then

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

and therefore

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

Generators Based on Linear Recurrences in $\mathbb{F}_2$

Matrix linear recurrence over $\mathbb{F}_2$ ($\equiv \{0,1\}$, $\mod 2$):

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

Generators Based on Linear Recurrences in $\mathbb{F}_2$

Matrix linear recurrence over $\mathbb{F}_2$ ($\equiv \{0,1\}$, $\bmod 2$):

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

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

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

with characteristic polynomial

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

Max. period length $\rho = 2^k - 1$ is reached iff $P(z)$ is primitive over $\mathbb{F}_2$.

With clever choice of $\mathbf{A}$, implementation involves bit shifts, xor's, and's, etc.

Special cases: Tausworthe, linear feedback shift register (LFSR), GFSR, twisted GFSR, Mersenne twister, xorshift, polynomial LCG, etc.

With clever choice of $\mathbf{A}$, implementation involves bit shifts, xor's, and's, etc.

Special cases: Tausworthe, linear feedback shift register (LFSR), GFSR, twisted GFSR, Mersenne twister, xorshift, polynomial LCG, etc.

Would also like the number N_1 of nonzero α_j's to be roughly around $k/2$.

With clever choice of $\mathbf{A}$, implementation involves bit shifts, xor's, and's, etc.

Special cases: Tausworthe, linear feedback shift register (LFSR), GFSR, twisted GFSR, Mersenne twister, xorshift, polynomial LCG, etc.

Would also like the number N_1 of nonzero α_j's to be roughly around $k/2$.

To jump ahead, again:

$$\mathbf{x}_{n+\nu} = \underbrace{(X^\nu \bmod 2)}_{\text{precompute}} \mathbf{x}_n \bmod 2.$$

This is time-expensive when k is large. A more efficient algorithm proposed by Haramoto, L'Ecuyer, Matsumoto, Nishimura, Panneton (2006)

LCG in space of formal series

The generating function of coordinate j of the state is the formal series

$$s_j(z) = x_{0,j}z^{-1} + x_{1,j}z^{-2} + \cdots = \sum_{n=1}^{\infty} x_{n-1,j}z^{-n}.$$

We can easily show that

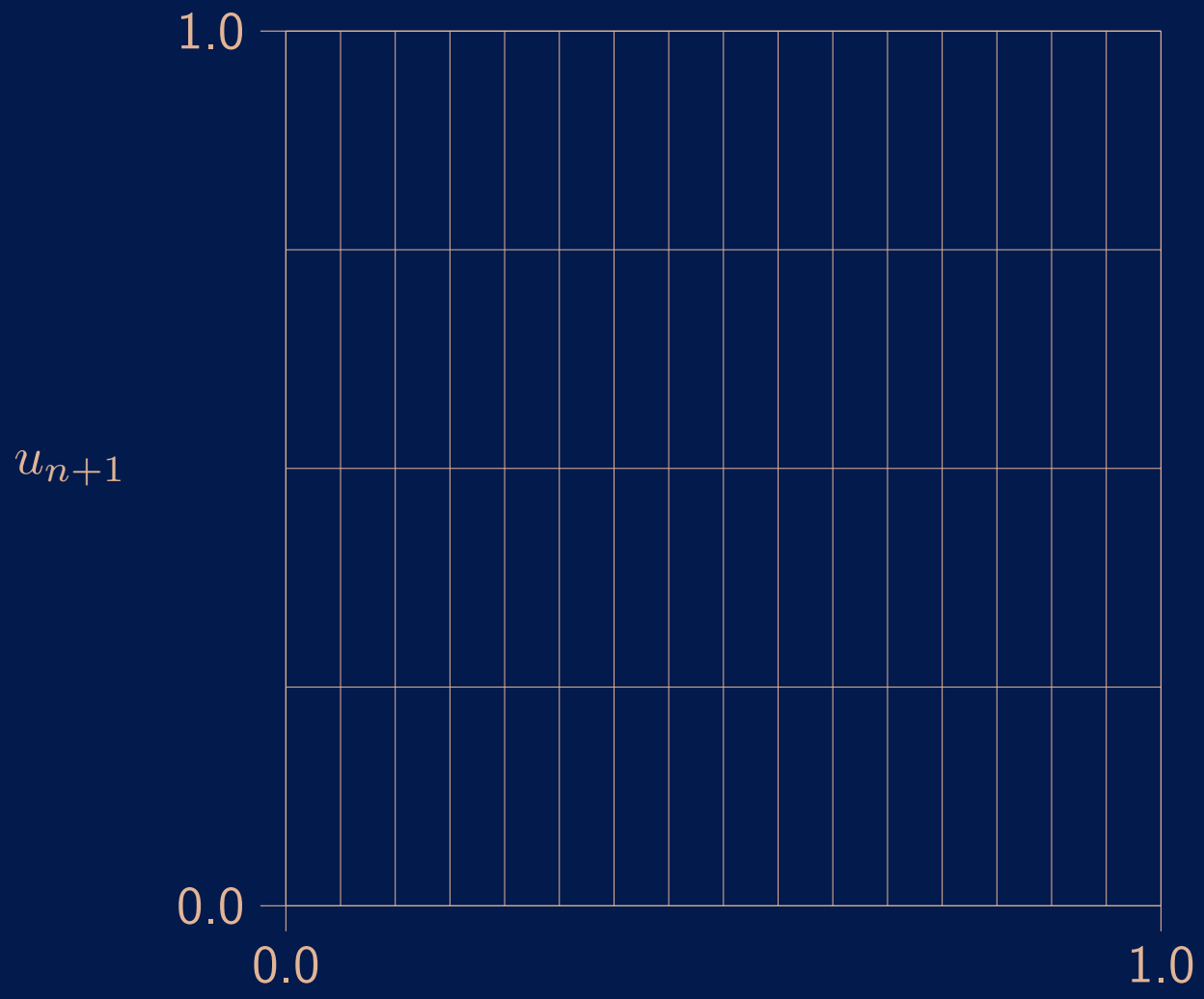
$$g_j(z) \stackrel{\text{def}}{=} s_j(z)P(z) = c_{j,1}z^{k-1} + \cdots + c_{j,k} \in \mathbb{F}_2[z],$$

where

$$\begin{pmatrix} c_{j,1} \\ c_{j,2} \\ \vdots \\ c_{j,k} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \alpha_1 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ \alpha_{k-1} & \dots & \alpha_1 & 1 \end{pmatrix} \begin{pmatrix} x_{0,j} \\ x_{1,j} \\ \vdots \\ x_{k-1,j} \end{pmatrix}.$$

Thus, there is a one-to-one correspondence between
(1) the states $(x_{0,j}, \dots, x_{k-1,j})$ of the recurrence;
(2) the polynomials $g_j(z)$ of degree $< k$ (i.e., in $\mathbb{F}_2[z]/P(z)$);
(3) the formal series of the form $s_j(z) = g_j(z)/P(z)$.

Equidistribution. For $j = 1, \ldots, t$, partition the jth axis of $[0,1)^t$ into 2^{q_j} equal parts: this gives 2^q boxes, where $q = q_1 + \cdots + q_t$.



Example: $t = 2$, $q_1 = 4$, $q_2 = 2$.

Equidistribution. For $j = 1, \ldots, t$, partition the jth axis of $[0,1)^t$ into 2^{q_j} equal parts: this gives 2^q boxes, where $q = q_1 + \cdots + q_t$.

If each box contains exactly 2^{k-q} points from Ψ_t, then Ψ_t is called $(q_1, \ldots, q_t)$-equidistributed.

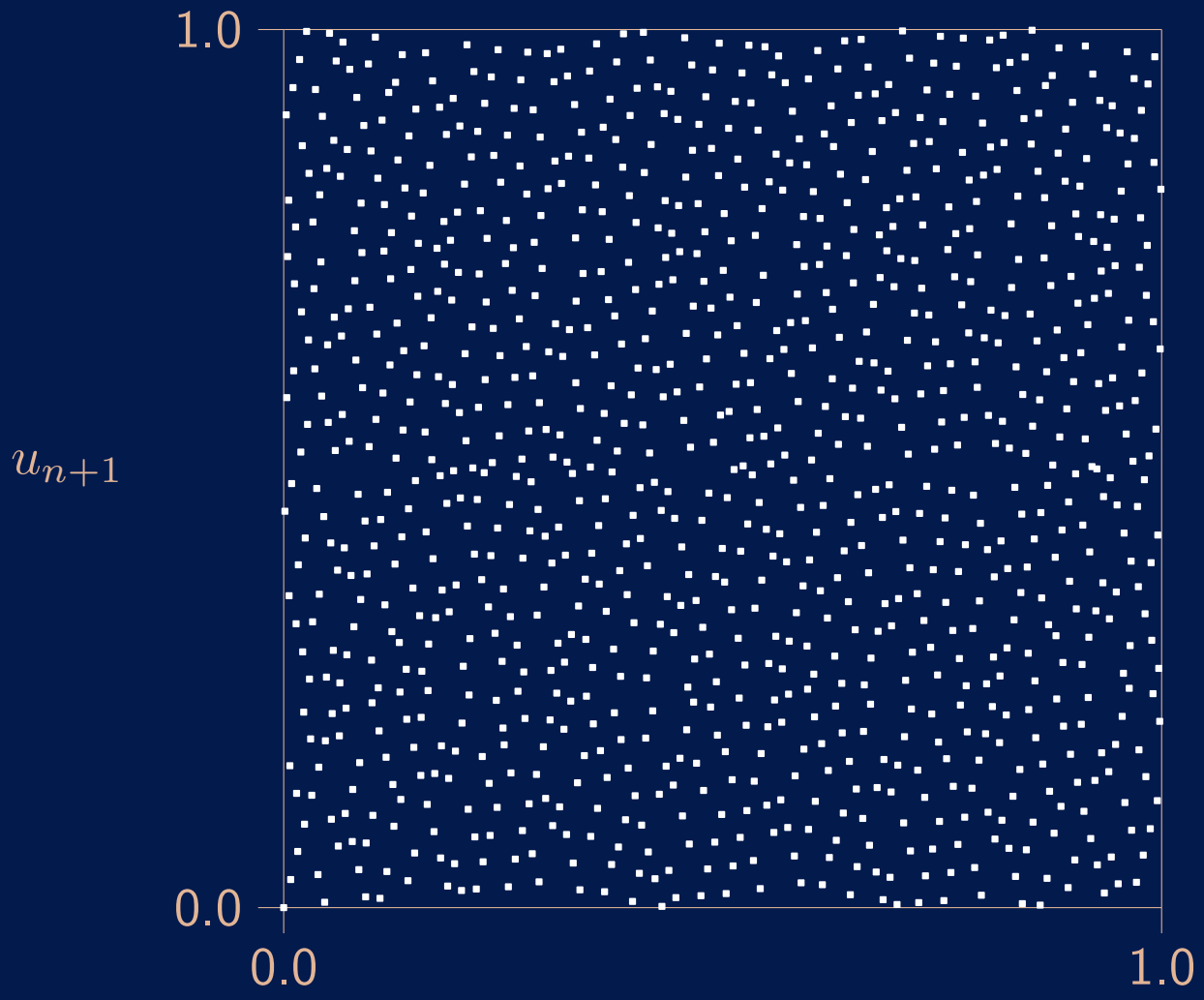
Means that all t-dim. vectors, up to q_j bits of resolution for each coordinate j, appear the same number of times.

Equidistribution. For $j = 1, \ldots, t$, partition the jth axis of $[0,1)^t$ into 2^{q_j} equal parts: this gives 2^q boxes, where $q = q_1 + \cdots + q_t$.

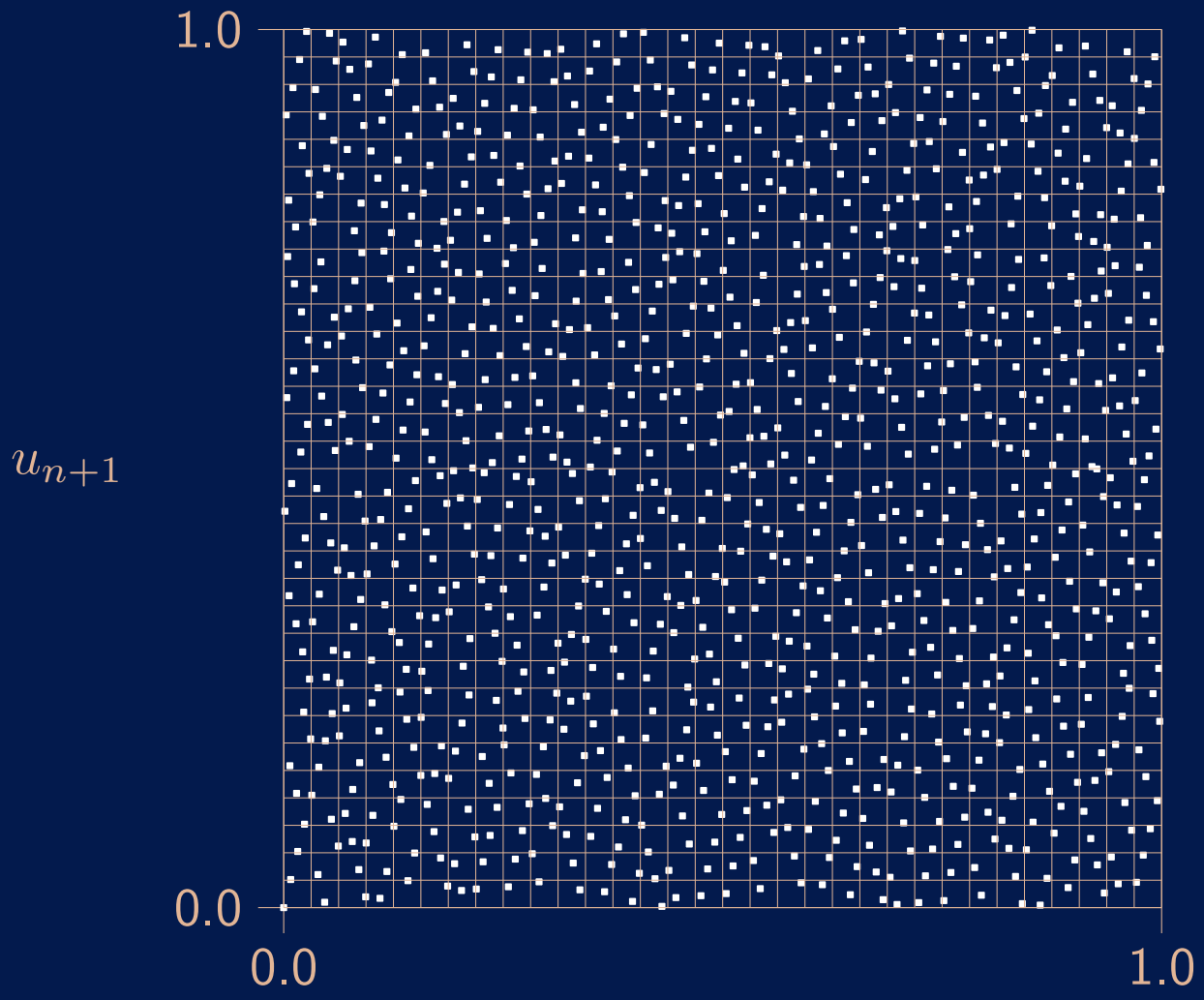
If each box contains exactly 2^{k-q} points from Ψ_t, then Ψ_t is called $(q_1, \ldots, q_t)$-equidistributed.

Means that all t-dim. vectors, up to q_j bits of resolution for each coordinate j, appear the same number of times.

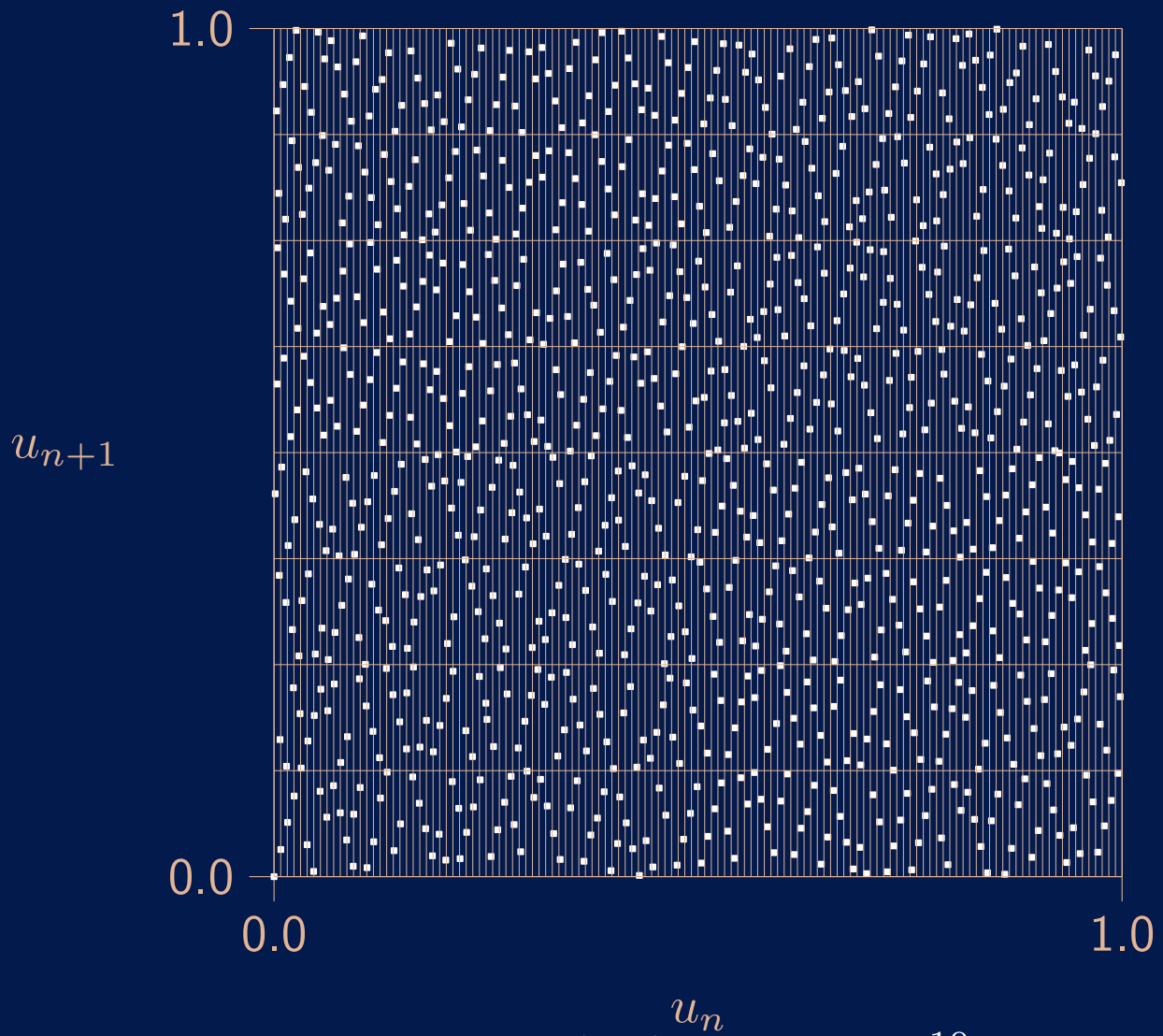
Can express the q bits of interest as $M\mathbf{x}_0$ for some matrix M.
One has $(q_1, \ldots, q_t)$-equidistribution iff M has full rank.



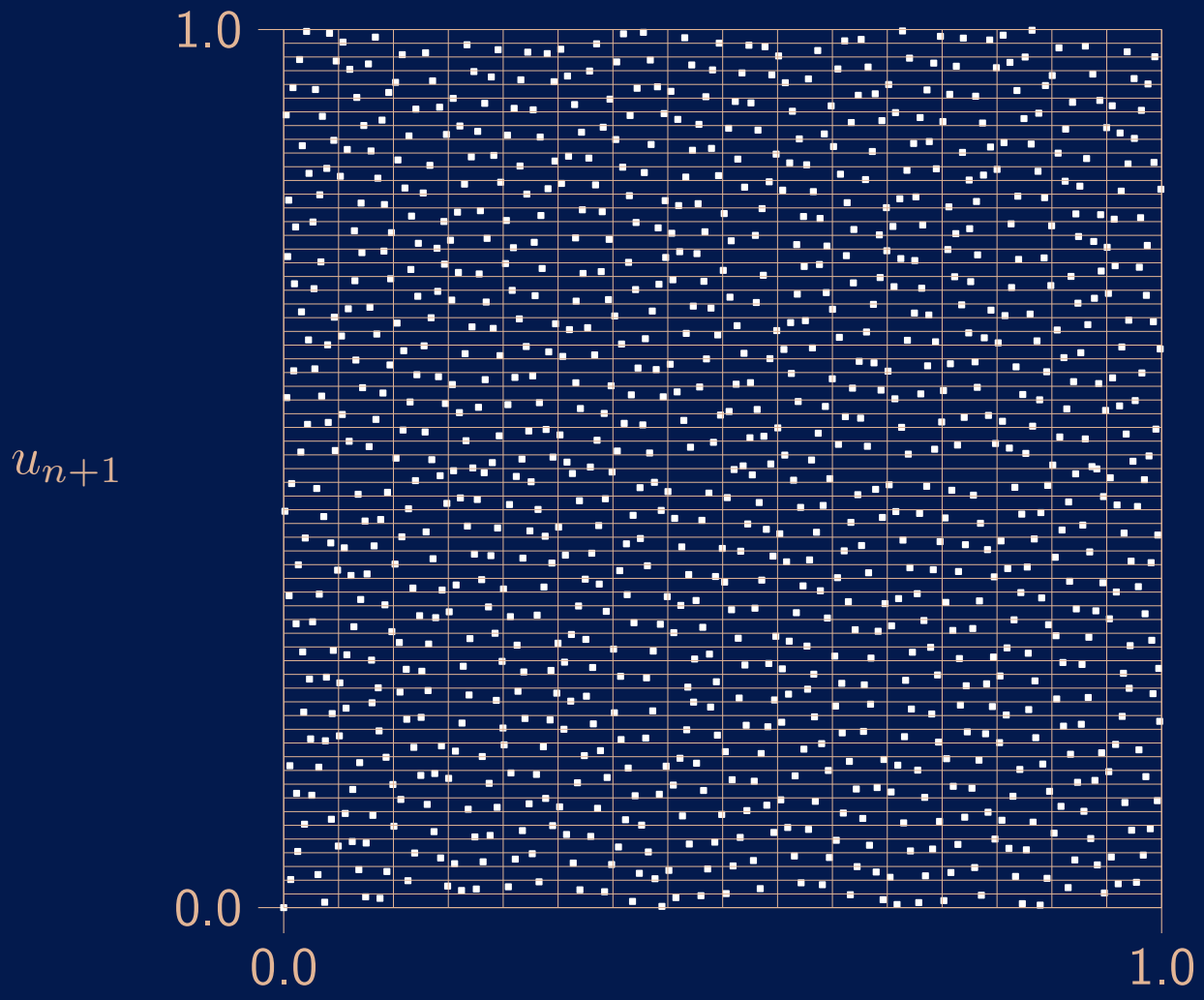
Toy example: LFSR generator with $|\Psi_t| = 1024 = 2^{10}$.



Toy example: LFSR generator with $|\Psi_t| = 1024 = 2^{10}$.



Toy example: LFSR generator with $|\Psi_t| = 1024 = 2^{10}$. 128×8



Toy example: LFSR generator with $|\Psi_t| = 1024 = 2^{10}$. 16×64

For general set of indices $I = \{i_1, i_2, \ldots, i_t\}$, resolution gap:

$$\delta_I = \min(\lfloor k/t \rfloor, w) - \max\{\ell : \Psi_I \text{ is } (\ell, \ldots, \ell)\text{-equidist.}\}.$$

Potential figures of merit:

$$\Delta_{\mathcal{J}} = \max_{I \in \mathcal{J}} \delta_I \quad \text{or} \quad V_{\mathcal{J}} = \sum_{I \in \mathcal{J}} \delta_I$$

where $\mathcal{J}$ is a selected class of sets I.

Choice of $\mathcal{J}$ is a question of compromise.

Tausworthe or LFSR generator (Tausworthe 1965):

$$
\begin{aligned}
x_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod 2, \\
u_n &= \sum_{j=1}^{\infty} x_{n\nu+j-1} 2^{-j},
\end{aligned}
$$

$$
\mathbf{A} = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix}^{\nu}.
$$

Often, only two nonzero a_j's: bad!

Polynomial LCG (Couture, L'Ecuyer, Panneton 2000):

$$\begin{aligned} Q(z) &= z^k - a_1 z^{k-1} - \cdots - a_{k-1} z - a_k \quad \text{primitive and pgcd}(\nu, 2^k - 1) = 1; \\ p_n(z) &= z^\nu p_{n-1}(z) \bmod Q(z) = c_{n,1} z^{k-1} + \cdots + c_{n,k}, \\ u_n &= \sum_{j=1}^{k} c_{n,j} 2^{-j}. \end{aligned}$$

If we define the output as

$$\begin{aligned} s_n(z) &= p_n(z)/Q(z) = \sum_{j=1}^{\infty} x_{n,j-1} z^{-j}; \\ u_n &= \sum_{j=1}^{\infty} x_{n,j-1} 2^{-j}, \end{aligned}$$

we just get an alternative definition of the LFSR (Tezuka and L'Ecuyer 1991).

Generalized feedback shift register (GFSR) (Lewis and Payne 1973):

$$\begin{aligned}
\mathbf{v}_n &= (a_1\mathbf{v}_{n-1} + \cdots + a_r\mathbf{v}_{n-r}) \bmod 2 = (v_{n,0}, \ldots, v_{n,w-1})^{\mathsf{t}}, \\
\mathbf{y}_n &= \mathbf{v}_n, \\
u_n &= \sum_{j=1}^{w} v_{n,j-1} 2^{-j}.
\end{aligned}$$

State has rw bits, but maximal period length is $2^r - 1$.

Usually, only two nonzero a_j's:

$$\mathbf{v}_n = (a_q\mathbf{v}_{n-q} + a_r\mathbf{v}_{n-r}) \bmod 2.$$

$\Rightarrow$ characteristic polynomial is too "lean."

Twisted GFSR (Matsumoto and Kurita 1992, 1994):

$$\begin{aligned} \mathbf{v}_n &= (\mathbf{v}_{n+m-r} + A\mathbf{v}_{n-r}) \bmod 2 \\ \mathbf{y}_n &= \mathbf{v}_n \quad \text{ou} \quad \mathbf{y}_n = T\mathbf{v}_n. \end{aligned}$$

Maximal period length is $2^{rw} - 1$.

Flag carrier: `TT800`, with period length $2^{800} - 1$.

Mersenne Twister (Matsumoto and Nishimura 1998):

$$\begin{aligned} \mathbf{v}_n &= (\mathbf{v}_{n+m-r} + A(\mathbf{v}_{n-r}^{(w-p)} | \mathbf{v}_{n-r+1}^{(p)})) \bmod 2 \\ \mathbf{y}_n &= T\mathbf{v}_n. \end{aligned}$$

Maximal period length is $2^{rw-p} - 1$.

Flag carrier: `MT19937`, with period length $2^{19937} - 1$.

Linear cellular automata (e.g., Wolfram 1983):

In one dimension, we have

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & & & & & \\ a_{1,-1} & a_{1,0} & a_{1,1} & & & & \\ & a_{2,-1} & a_{2,0} & a_{2,1} & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & & & \\ & & & & a_{k-2,-1} & a_{k-2,0} & a_{k-2,1} \\ & & & & & a_{k-1,-1} & a_{k-1,0} \end{pmatrix}.$$

Maximal period length: $2^k - 1$.

Rule 150: $a_{j,-1} = a_{j,0} = a_{j,1} = 1$.

Rule 90: $a_{j,0} = 0$ et $a_{j,-1} = a_{j,1} = 1$.

Xorshift generators (Marsaglia 2003).

Xorshift operation: $\mathbf{x} = \mathbf{x} \oplus (\mathbf{x} \ll a)$ or $\mathbf{x} = \mathbf{x} \oplus (\mathbf{x} \gg b)$.

Marsaglia proposes specific parameters for generators with:
— maximum of 3 xorshifts per transition;
— maximal period.

These generators are extremely fast.

Unfortunately, they have bad equidistribution and fail miserably several simple statistical tests (Panneton and L'Ecuyer 2004).

Linear output tempering.

Take (carefully selected) $B \neq I$ to improve the uniformity of the output.

Example: Matsumoto and Kurita's tempering for the TGFSR:

$$\begin{aligned}
\overline{\mathbf{x}}_n &= \mathsf{trunc}_w(\mathbf{x}_n) \\
\mathbf{s}_n &= \overline{\mathbf{x}}_n \oplus (\overline{\mathbf{x}}_n \gg u) \\
\mathbf{r}_n &= \mathbf{s}_n \oplus ((\mathbf{s}_n \ll s_1) \,\&\, \mathbf{b}) \\
\mathbf{t}_n &= \mathbf{r}_n \oplus ((\mathbf{r}_n \ll s_2) \,\&\, \mathbf{c}) \\
\mathbf{y}_n &= \mathbf{t}_n \oplus (\mathbf{t}_n \gg l)
\end{aligned}$$

where $w = 32$, $s_1, \ldots, s_4$ are integers, and $\mathbf{b}$ and $\mathbf{c}$ are bit vectors.

The WELL RNGs (Well-Equidistributed Long-period Linear)
(Panneton, L'Ecuyer, et Matsumoto 2004).

Idea: Build the transition matrix $\mathbf{A}$ by placing simple operations on 32-bit blocs (exclusive-or's, shifts, bit masks, etc.) at strategic places.

We optimise the figure of merit

$$\Delta_1 = \sum_{\ell=1}^{w} (\min(\lfloor k/\ell \rfloor, w) - \max\{t : \Psi_t \text{ is } (\ell, \ldots, \ell)\text{-équidist.}\}),$$

under the constraints: (1) maximal period and (2) comparable speed with MT19937.

Specific generators for period lengths ranging from $2^{521} - 1$ to $2^{44497} - 1$.

Better equidistribution than MT19937 for same speed and period length.
Also larger number N_1 of nonzero coefficients in $P(z)$.

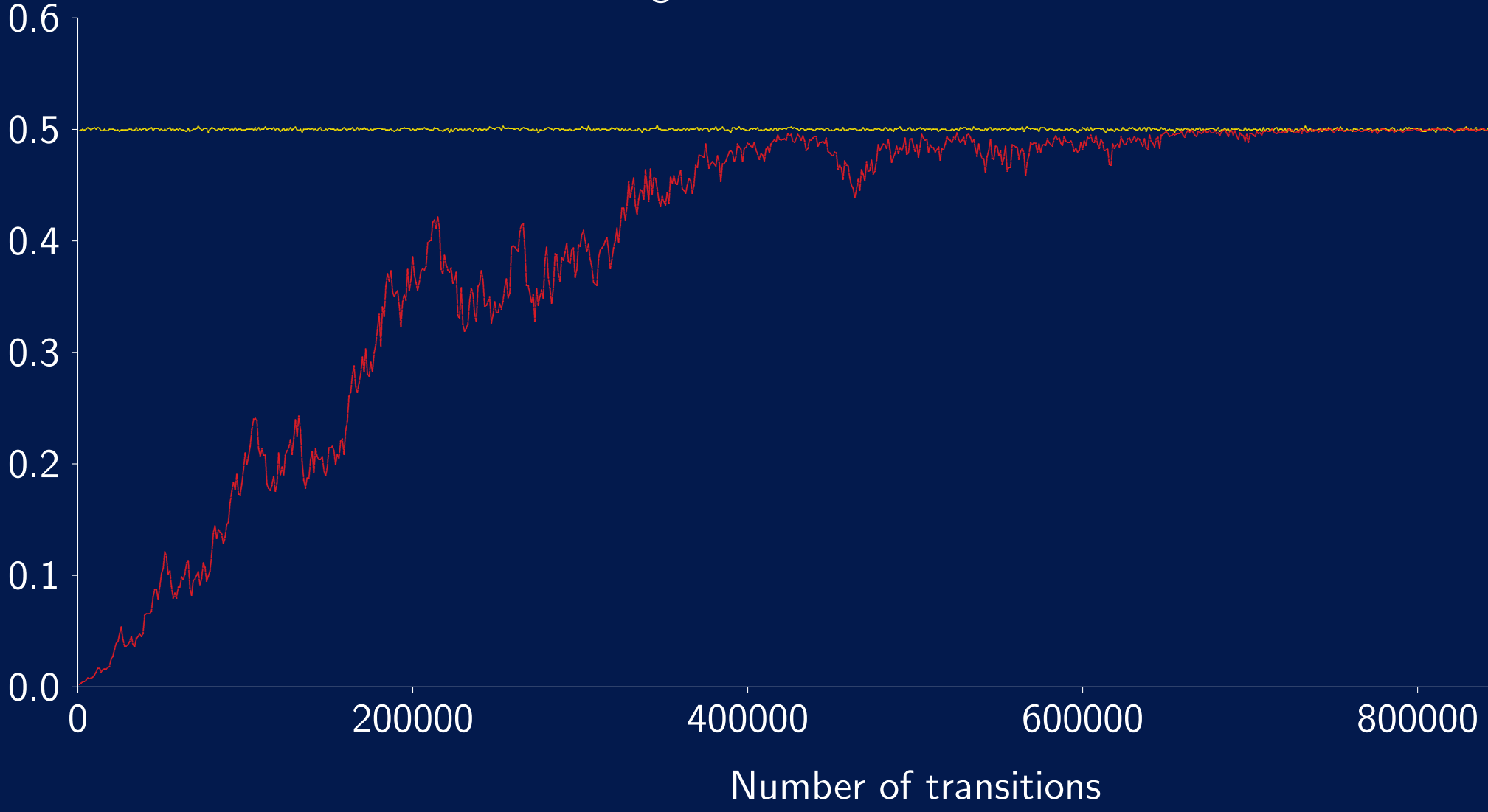
Examples of WELL generators, vs MT, for $w = 32$ bits:

Name	k	r	N_1	Δ_1
WELL19937a	19937	624	8585	4
WELL19937c	19937	624	8585	0
WELL44497a	44497	1391	16883	7
WELL44497b	44497	1391	16883	0
MT19937	19937	624	135	6750

Impact of N_1 too small and/or Δ_1 too large.

Experiment: start from an initial state with a single bit at 1.
Try all k possibilities and average the outputs after each transition.

WELL19937 vs MT19937; moving averate over 1000 transitions.



Combined F_2-Linear Generators

Run two or more smaller generators with relatively prime period lengths.
Combine outputs by bitwise xor.

Equivalent to a single larger $\mathbb{F}_2$-linear generator.

Advantages: easier to implement and jump ahead with smaller components.

Nonlinear Generators

Cubic Congruential:

$$\begin{aligned} x_n &= (ax_{n-1}^3 + 1) \bmod m, \\ u_n &= x_n/m. \end{aligned}$$

Explicit Inversive:

$$\begin{aligned} x_n &= (an + c) \bmod m, \\ u_n &= (x_n^{-1} \bmod m)/m. \end{aligned}$$

Arbitrary permutation implemented as a table.
Can combine several tables: efficient.

AES, SHA-1, etc. Used in cryptology.
Chaotic dynamical systems? No.

Combined linear/nonlinear generators

All the $\mathbb{F}_2$-linear generators discussed here fail (of course) a statistical test that measures the (binary) linear complexity.

Combined linear/nonlinear generators

All the $\mathbb{F}_2$-linear generators discussed here fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

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

Combined linear/nonlinear generators

All the $\mathbb{F}_2$-linear generators discussed here fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

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

L'Ecuyer and Granger-Picher (2003): Large $\mathbb{F}_2$-linear generator combined with a small nonlinear one, via XOR.

Combined linear/nonlinear generators

All the $\mathbb{F}_2$-linear generators discussed here fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

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

L'Ecuyer and Granger-Picher (2003): Large $\mathbb{F}_2$-linear generator combined with a small nonlinear one, via XOR.

Theorem: If the linear component is $(q_1, \ldots, q_t)$-equidistributed, then this holds also for the combination.

Combined linear/nonlinear generators

All the $\mathbb{F}_2$-linear generators discussed here fail (of course) a statistical test that measures the (binary) linear complexity.

We would like:

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

L'Ecuyer and Granger-Picher (2003): Large $\mathbb{F}_2$-linear generator combined with a small nonlinear one, via XOR.

Theorem: If the linear component is $(q_1, \ldots, q_t)$-equidistributed, then this holds also for the combination.

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

Some generators used in SSJ

gen. time: CPU time to generate 10^9 uniform random numbers.
jump time: time to get a new stream (jump ahead) 10^6 times.

Some generators used in SSJ

gen. time: CPU time to generate 10^9 uniform random numbers.
jump time: time to get a new stream (jump ahead) 10^6 times.

Java JDK 1.5, 2.4 GHz 64-bit computer

RNG	period	gen. time	jump time
LFSR113	2^{113}	31	0.1
WELL512	2^{512}	33	234
WELL1024	2^{1024}	34	917
LFSR258	2^{258}	35	0.2
MT19937	2^{19937}	36	46 *
MRG31k3p	2^{185}	51	0.9
MRG32k3a	2^{191}	70	1.1
RandRijndael	2^{130}	127	0.6

Empirical statistical Tests

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

Empirical statistical Tests

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

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

Power and efficiency of the test strongly depend on the class of alternatives.
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.

Power and efficiency of the test strongly depend on the class of alternatives.
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.

Power and efficiency of the test strongly depend on the class of alternatives.
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

Partition the box $[0,1)^t$ into $k = d^t$ cubic boxes of equal sizes.
Generate n points $(u_i, \ldots, u_{i+t-1})$ in $[0,1)^t$.
Let X_j = number of points in box j.
Number of collisions:

$$C = \sum_{j=0}^{k-1} \max(0,\, X_j - 1).$$

Under $\mathcal{H}_0$, $C \approx$ Poisson with mean $\lambda = n^2/k$, for large k, small λ.

If we observe c collisions, we compute the right and left p-values as

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

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

Example: Birthday spacings

Partition $[0,1)^t$ into $k = d^t$ cubic boxes and generate n points as before.
Let $I_1 \leq I_2 \leq \cdots \leq I_n$ the box numbers where the points fall.
Compute the spacings $S_j = I_{j+1} - I_j$, $1 \leq j \leq n-1$.
Let $S_{(1)}, \ldots, S_{(n-1)}$ be the sorted spacings.
Number of collisions between the spacings:

$$Y = \sum_{j=1}^{n-1} I[S_{(j+1)} = S_{(j)}].$$

Example: Birthday spacings

Partition $[0,1)^t$ into $k = d^t$ cubic boxes and generate n points as before.
Let $I_1 \leq I_2 \leq \cdots \leq I_n$ the box numbers where the points fall.
Compute the spacings $S_j = I_{j+1} - I_j$, $1 \leq j \leq n-1$.
Let $S_{(1)}, \ldots, S_{(n-1)}$ be the sorted spacings.
Number of collisions between the spacings:

$$Y = \sum_{j=1}^{n-1} I[S_{(j+1)} = S_{(j)}].$$

For large k, Y is approx. Poisson with mean $\lambda = n^3/(4k)$.

Example: Birthday spacings

Partition $[0,1)^t$ into $k = d^t$ cubic boxes and generate n points as before.
Let $I_1 \leq I_2 \leq \cdots \leq I_n$ the box numbers where the points fall.
Compute the spacings $S_j = I_{j+1} - I_j$, $1 \leq j \leq n-1$.
Let $S_{(1)}, \ldots, S_{(n-1)}$ be the sorted spacings.
Number of collisions between the spacings:

$$Y = \sum_{j=1}^{n-1} I[S_{(j+1)} = S_{(j)}].$$

For large k, Y is approx. Poisson with mean $\lambda = n^3/(4k)$.

If Y takes the value y, the right p-value is

$$p^+(y) = P[X \geq y \mid X \sim \text{ Poisson}(\lambda)].$$

Other examples of tests

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

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 and Simard, ACM Trans. on Math. Software, 2007].

- Implements a large variety of statistical tests and RNGs (hardware or software), written in C, freely available.

- Also contains 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.

Some test results.
$\rho =$ 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.

Results of test batteries applied to some well-known RNGs

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
LCG in Microsoft VisualBasic	24	3.9	0.66	14	—	—
LCG(2^{31}, 65539, 0)	29	3.3	0.65	14	125 (6)	—
LCG(2^{32}, 69069, 1)	32	3.2	0.67	11 (2)	106 (2)	—
LCG(2^{32}, 1099087573, 0)	30	3.2	0.66	13	110 (4)	—
LCG(2^{46}, 5^{13}, 0)	44	4.2	0.75	5	38 (2)	—
LCG(2^{48}, 25214903917, 11), Unix	48	4.1	0.65	4	21 (1)	—
Java.util.Random	47	6.3	0.76	1	9 (3)	21 (1)
LCG(2^{48}, 5^{19}, 0)	46	4.1	0.65	4	21 (2)	—
LCG(2^{48}, 33952834046453, 0)	46	4.1	0.66	5	24 (5)	—
LCG(2^{48}, 44485709377909, 0)	46	4.1	0.65	5	24 (5)	—
LCG(2^{59}, 13^{13}, 0)	57	4.2	0.76	1	10 (1)	17 (5)
LCG(2^{63}, 5^{19}, 1)	63	4.2	0.75		5	8
LCG(2^{31}–1, 16807, 0), Wide use	31	3.8	3.6	3	42 (9)	—
LCG(2^{31}–1, $2^{15} - 2^{10}$, 0)	31	3.8	1.7	8	59 (7)	—
LCG(2^{31}–1, 397204094, 0), (SAS)	31	19.0	4.0	2	38 (4)	—
LCG(2^{31}–1, 742938285, 0)	31	19.0	4.0	2	42 (5)	—
LCG(2^{31}–1, 950706376, 0)	31	20.0	4.0	2	42 (4)	—
LCG(10^{12}–11, ..., 0), in Maple	39.9	87.0	25.0	1	22 (2)	34 (1)
LCG(2^{61}–1, $2^{30} - 2^{19}$, 0)	61	71.0	4.2		1 (4)	3 (1)

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
Wichmann-Hill, in Excel	42.7	10.0	11.2	1	12 (3)	22 (8)
CombLec88	61	7.0	1.2		1	
Knuth(38)	56	7.9	7.4		1 (1)	2
ran2, in Numerical Recipes	61	7.5	2.5			
CLCG4	121	12.0	5.0			
Knuth(39)	62	81.0	43.3		(1)	3 (2)
MRGk5-93	155	6.5	2.0			
DengLin (2^{31}–1, 2, 46338)	62	6.7	15.3	(1)	11 (1)	19 (2)
DengLin (2^{31}–1, 4, 22093)	124	6.7	14.6	(1)	2	4 (2)
DX-47-3	1457	—	1.4			
DX-1597-2-7	49507	—	1.4			
Marsa-LFIB4	287	3.4	0.8			
CombMRG96	185	9.4	2.0			
MRG31k3p	185	7.3	2.0		(1)	
MRG32k3a SSJ + others	191	10.0	2.1			
MRG63k3a	377	—	4.3			
LFib(2^{31}, 55, 24, +)	85	3.8	1.1	2	9	14 (5)
LFib(2^{31}, 55, 24, −)	85	3.9	1.5	2	11	19
ran3, in Numerical Recipes		2.2	0.9	(1)	11 (1)	17 (2)
LFib(2^{48}, 607, 273, +)	638	2.4	1.4		2	2
Unix-random-32	37	4.7	1.6	5 (2)	101 (3)	—
Unix-random-64	45	4.7	1.5	4 (1)	57 (6)	—
Unix-random-128	61	4.7	1.5	2	13	19 (3)
Unix-random-256	93	4.7	1.5	1 (1)	8	11 (1)

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
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 (2)
SWB(2^{24}, 10, 24)[24, 48]	566	18.0	7.0		6 (1)	16 (1)
SWB(2^{24}, 10, 24)[24, 97]	565	32.0	12.0			
SWB(2^{24}, 10, 24)[24, 389]	567	117.0	43.0			
SWB(2^{32}–5, 22, 43)	1376	3.9	1.5	(1)	8	17
SWB(2^{31}, 8, 48)	1480	4.4	1.5	(2)	8 (2)	11
Mathematica-SWB	1479	—	—	1 (2)	15 (3)	—
SWB(2^{32}, 222, 237)	7578	3.7	0.9		2	5 (2)
GFSR(250, 103)	250	3.6	0.9	1	8	14 (4)
GFSR(521, 32)	521	3.2	0.8		7	8
GFSR(607, 273)	607	4.0	1.0		8	8
Ziff98	9689	3.2	0.8		6	6
T800	800	3.9	1.1	1	25 (4)	—
TT800	800	4.0	1.1		12 (4)	14 (3)
MT19937, widely used	19937	4.3	1.6		2	2
WELL1024a	1024	4.0	1.1		4	4
WELL19937a	19937	4.3	1.3		2 (1)	2
LFSR113	113	4.0	1.0		6	6
LFSR258	258	6.0	1.2		6	6
Marsa-xor32 (13, 17, 5)	32	3.2	0.7	5	59 (10)	—
Marsa-xor64 (13, 7, 17)	64	4.0	0.8	1	8 (1)	7

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
Matlab-rand	1492	27.0	8.4		5	8 (1)
Matlab-LCG-Xor (normal)	64	3.7	0.8		3	5 (1)
SuperDuper-73, in S-Plus	62	3.3	0.8	1 (1)	25 (3)	—
SuperDuper64	128	5.9	1.0			
R-MultiCarry	60	3.9	0.8	2 (1)	40 (4)	—
KISS93	95	3.8	0.9		1	1
KISS99	123	4.0	1.1			
Brent-xor4096s	131072	3.9	1.1			
ICG(2^{31}–1, 22211, 11926380)	31	74.0	69.0		5	10 (8)
EICG(2^{31}–1, 1288490188, 1)	31	55.0	64.0		6	14 (6)
SNWeyl	32	12.0	4.2	1	56 (12)	—
Coveyou-32	30	3.5	0.7	12	89 (5)	—
Coveyou-64	62	—	0.8		1	2
LFib(2^{64}, 17, 5, $*$)	78	—	1.1			
LFib(2^{64}, 55, 24, $*$)	116	—	1.0			
LFib(2^{64}, 607, 273, $*$)	668	—	0.9			
LFib(2^{64}, 1279, 861, $*$)	1340	—	0.9			
ISAAC		3.7	1.3			
AES (OFB)		10.8	5.8			
AES (CTR)	130	10.3	5.4			(1)
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 wisely-used software, especially if they hide the algorithm (proprietary software...).

- Some software products have good RNGs; check what it is.

- RNGs with multiple streams are available from my web page in Java, C, and C++. Just type Pierre L'Ecuyer in Google.

- What about grids?