Le hasard artificiel

Pierre L'Ecuyer

DIRO, Université de Montréal

- Les besoins, les applications.
- Générateurs algorithmiques.
  Mesures de qualité.
- Exemples: récurrences linéaires.
- Tests statistiques.
  Évaluation de générateurs largement utilisés.

Qu'est-ce qu'on veut?

Des nombres qui ont l'air tirés au hasard.

Qu'est-ce qu'on veut?

Des nombres qui ont l'air tirés au hasard.

Exemple: Suites de bits (pile ou face):

$$011110100110110101001101100101000111?\ldots$$

Loi uniforme: chaque bit est 1 avec probabilité $1/2$.

Qu'est-ce qu'on veut?

Des nombres qui ont l'air tirés au hasard.

Exemple: Suites de bits (pile ou face):

$$01111\,?\,100110\,?\,1\,?\,1010011011001010001111\ldots$$

Loi uniforme: chaque bit est 1 avec probabilité $1/2$.

Uniformité et indépendance:
Exemple: on 8 possibilités pour les 3 bits **? ? ?**:

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

On veut une proba. de $1/8$ pour chacune, peu importe les autres bits.

Qu'est-ce qu'on veut?

Des nombres qui ont l'air tirés au hasard.

Exemple: Suites de bits (pile ou face):

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

Loi uniforme: chaque bit est 1 avec probabilité $1/2$.

Uniformité et indépendance:
Exemple: on 8 possibilités pour les 3 bits ? ? ?:

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

On veut une proba. de $1/8$ pour chacune, peu importe les autres bits.

Pour s bits, probabilité de $1/2^s$ pour chacune des 2^s possibilités.

Suite d'entiers de 1 à 6:

Suite d’entiers de 1 à 6:

Suite d’entiers de 1 à 100: 31, 83, 02, 72, 54, 26, ...

Permutation aléatoire:

1 2 3 4 5 6 7

Permutation aléatoire:

1 2 3 4 5 6 7

1 2 3 4 6 7 5

Permutation aléatoire:

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

Permutation aléatoire:

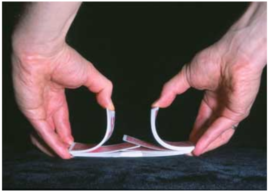
1 2 3 4 5 6 7	
1 2 3 4 6 7	5
1 3 4 6 7	5 2
3 4 6 7	5 2 1

Permutation aléatoire:

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

Pour n objets, on choisit un entier de 1 à n,
puis un autre entier de 1 à $n-1$, puis de 1 à $n-2$, ...
On veut que chaque permutation ait la même probabilité.

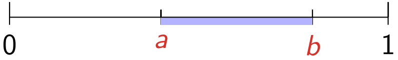
Ex.: pour permuter 52 cartes, il y a $52! \approx 2^{226}$ possibilités.



Loi uniforme sur $(0, 1)$

Pour la simulation en général, on voudrait une suite $U_0, U_1, U_2, \ldots$ de variables aléatoires indépendantes de loi uniforme sur l'intervalle $(0, 1)$.

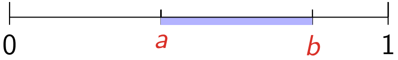
On veut $\mathbb{P}[a \leq U_j \leq b] = b - a$.



Loi uniforme sur $(0,1)$

Pour la simulation en général, on voudrait une suite $U_0, U_1, U_2, \ldots$ de variables aléatoires indépendantes de loi uniforme sur l'intervalle $(0,1)$.

On veut $\mathbb{P}[a \le U_j \le b] = b - a$.



Pour générer X telle que $\mathbb{P}[X \le x] = F(x)$:

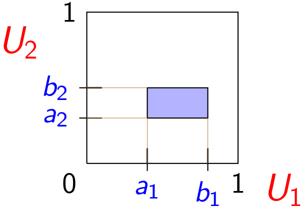
$$X = F^{-1}(U_j) = \inf\{x : F(x) \ge U_j\}.$$

Indépendance:

En s dimensions, on veut

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

On voudrait cela pour tout s et tout choix de la boîte rectangulaire.

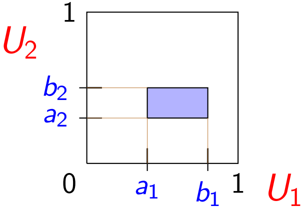


Indépendance:

En s dimensions, on veut

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

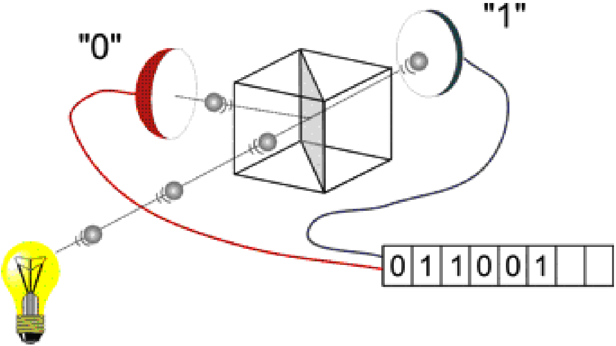
On voudrait cela pour tout s et tout choix de la boîte rectangulaire.



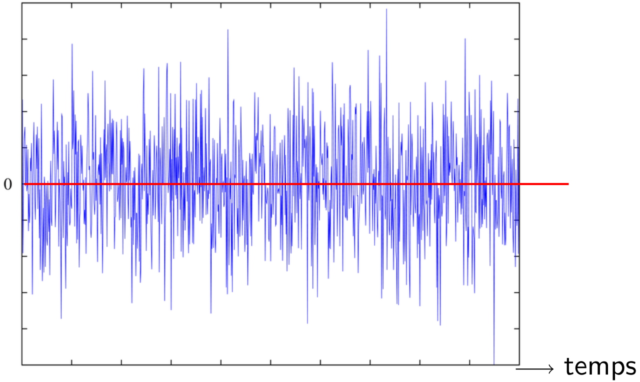
Cette notion de v.a. uniformes et indépendantes est une abstraction mathématique. N'existe peut-être pas dans la réalité!

Mécanismes physiques pour ordinateur

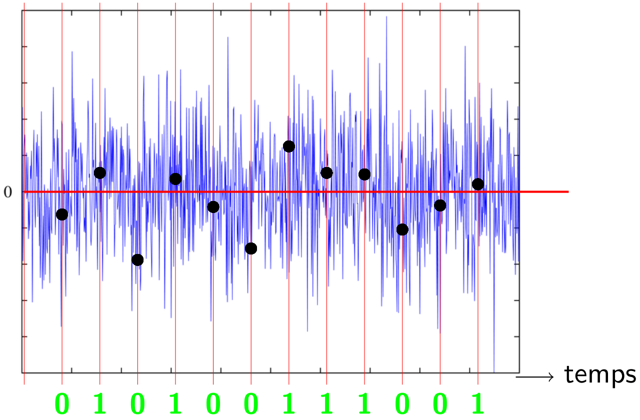
Trajectoires de photons (vendu par **id-Quantique**):



Bruit thermique dans les résistances de circuits électroniques

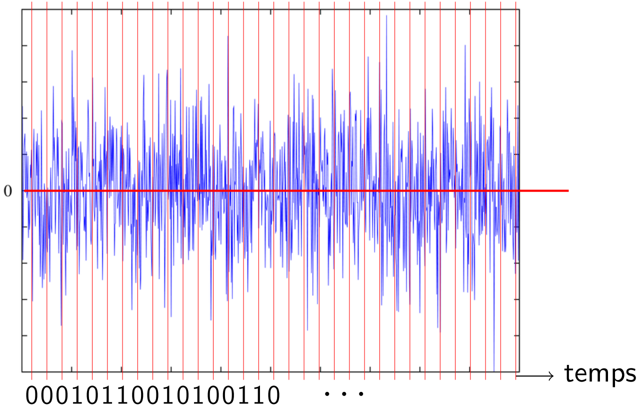


Bruit thermique dans les résistances de circuits électroniques



On échantillonne le signal périodiquement.

Bruit thermique dans les résistances de circuits électroniques



On échantillonne le signal périodiquement.

Plusieurs mécanismes sont brevetés et disponibles commercialement.

Aucun n'est parfait.

Plusieurs mécanismes sont brevetés et disponibles commercialement.

Aucun n'est parfait. On peut dimimuer le biais et/ou la dépendance en combinant des blocs de bits. Par exemple par un XOR:

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

Plusieurs mécanismes sont brevetés et disponibles commercialement.

Aucun n'est parfait. On peut dimimuer le biais et/ou la dépendance en combinant des blocs de bits. Par exemple par un XOR:

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

ou encore (élimine le biais):

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

Plusieurs mécanismes sont brevetés et disponibles commercialement.

Aucun n'est parfait. On peut dimimuer le biais et/ou la dépendance en combinant des blocs de bits. Par exemple par un XOR:

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

ou encore (élimine le biais):

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

Essentiel pour cryptologie, loteries, etc. Mais pas pour la simulation. Encombrant, pas reproduisible, pas toujours fiable, pas d'analyse mathématique de l'uniformité et de l'indépendance à long terme.

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

1. Choisir un nombre x_0 au hasard dans $\{1, \ldots, 100\}$.
2. Pour $n = 1, 2, 3, \ldots$, retourner $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

1. Choisir un nombre x_0 au hasard dans $\{1, \ldots, 100\}$.
2. Pour $n = 1, 2, 3, \ldots$, retourner $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Par exemple, si $x_0 = 1$:

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

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

1. Choisir un nombre x_0 au hasard dans $\{1, \ldots, 100\}$.
2. Pour $n = 1, 2, 3, \ldots$, retourner $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Par exemple, si $x_0 = 1$:

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

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

1. Choisir un nombre x_0 au hasard dans $\{1, \ldots, 100\}$.
2. Pour $n = 1, 2, 3, \ldots$, retourner $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Par exemple, si $x_0 = 1$:

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

Visite tous les nombres de 1 à 100 une fois chacun avant de revenir à x_0.

Générateurs algorithmiques (pseudo-aléatoires, GPA)

Mini-exemple: On veut imiter des nombres de 1 à 100 au hasard.

1. Choisir un nombre x_0 au hasard dans $\{1, \ldots, 100\}$.
2. Pour $n = 1, 2, 3, \ldots$, retourner $\boxed{x_n = 12\, x_{n-1} \bmod 101}$.

Par exemple, si $x_0 = 1$:

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

Visite tous les nombres de 1 à 100 une fois chacun avant de revenir à x_0.

Si on veut des nombres réels entre 0 et 1:

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

Exemple plus réaliste: MRG32k3a

On choisit 6 entiers:
x_0, x_1, x_2 dans $\{0, 1, \ldots, 4294967086\}$ (pas tous 0) et
y_0, y_1, y_2 dans $\{0, 1, \ldots, 4294944442\}$ (pas tous 0).

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

Exemple plus réaliste: MRG32k3a

On choisit 6 entiers:
x_0, x_1, x_2 dans $\{0, 1, \ldots, 4294967086\}$ (pas tous 0) et
y_0, y_1, y_2 dans $\{0, 1, \ldots, 4294944442\}$ (pas tous 0).

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

(x_{n-2}, x_{n-1}, x_n) visite chacune des $4294967087^3 - 1$ valeurs possibles.
(y_{n-2}, y_{n-1}, y_n) visite chacune des $4294944443^3 - 1$ valeurs possibles.

La suite $u_0, u_1, u_2, \ldots$ se répète avec une période proche de

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

Exemple plus réaliste: MRG32k3a

On choisit 6 entiers:
x_0, x_1, x_2 dans $\{0, 1, \ldots, 4294967086\}$ (pas tous 0) et
y_0, y_1, y_2 dans $\{0, 1, \ldots, 4294944442\}$ (pas tous 0).

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

(x_{n-2}, x_{n-1}, x_n) visite chacune des $4294967087^3 - 1$ valeurs possibles.
(y_{n-2}, y_{n-1}, y_n) visite chacune des $4294944443^3 - 1$ valeurs possibles.

La suite $u_0, u_1, u_2, \ldots$ se répète avec une période proche de

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

Excellent générateur, robuste et fiable!

Utilisé par SAS, R, MATLAB, Arena, Automod, Witness, ns-2, Spielo, ...

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Évolution:

$x_{n-1} =$ 00010100101001101100110110100101

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \dots, 2^{32} - 1\}$ (32 bits). Évolution:

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

$x_{n-1} =$

```
      00010100101001101100110110100101
10010100101001101100110110100101
      00111101000101011010010011100101
```

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Évolution:

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

$$
\begin{array}{lrl}
x_{n-1} = & & 00010100101001101100110110100101 \\
 & 100101 & 00101001101100110110100101 \\
 & & 00111101000101011010010011100101 \\
B = & & 0011110100010101101 \; 0010011100101
\end{array}
$$

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \dots, 2^{32} - 1\}$ (32 bits). Évolution:

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

$x_{n-1} =$ 00010100101001101100110110100101

100101 00101001101100110110100101

00111101000101011010010011100101

$B =$ 0011110100010101101 0010011100101

x_{n-1} 0001010010100110110011011010010 0

000101001010011011 0011011010010 0

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \ldots, 2^{32}-1\}$ (32 bits). Évolution:

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

$x_{n-1} =$		00010100101001101100110110100101	
	100101	00101001101100110110100101	
		00111101000101011010010011100101	
$B =$		0011110100010101101	0010011100101
x_{n-1}		00010100101001101100110110100100	
	000101001010011011	00110110100100	
$x_n =$		00110110100100011110100010101101	

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \ldots, 2^{32}-1\}$ (32 bits). Évolution:

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

$x_{n-1} =$		00010100101001101100110110100101	
	100101	00101001101100110110100101	
		00111101000101011010010011100101	
$B =$		0011110100010101101	0010011100101
x_{n-1}		00010100101001101100110110100100	
	000101001010011011	00110110100100	
$x_n =$		00110110100100011110100010101101	

Les 31 premiers bits de $x_1, x_2, x_3, \ldots,$ parcourent tous les entiers de 1 à 2147483647 ($= 2^{31}-1$) exactement une fois avant de revenir à x_0.

Plus rapide: opérations sur des blocs de bits.

Exemple: Choisir $x_0 \in \{2, \ldots, 2^{32} - 1\}$ (32 bits). Évolution:

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

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

Les 31 premiers bits de $x_1, x_2, x_3, \ldots,$ parcourent tous les entiers de 1 à 2147483647 ($= 2^{31} - 1$) exactement une fois avant de revenir à x_0.

Pour des nombres réels entre 0 et 1: $u_n = x_n / (2^{32} + 1)$.

Exemple plus réaliste: LFSR113

On prend 4 récurrences sur des blocs de 32 bits, en parallèle.
Les périodes sont $2^{31} - 1$, $2^{29} - 1$, $2^{28} - 1$, $2^{25} - 1$.

On additionne les 4 états par un XOR, puis on divise par $2^{32} + 1$.
La période de la sortie est environ $2^{113} \approx 10^{34}$.

Bon générateur, plus rapide que MRG32k3a, mais il y a des relations linéaires entre les bits à la sortie.

Exemple: subtract-with-borrow (SWB)

État $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ où $x_n \in \{0, \ldots, 2^{31} - 1\}$ et $c_n \in \{0, 1\}$:

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

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

Exemple: subtract-with-borrow (SWB)

État $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ où $x_n \in \{0, \ldots, 2^{31} - 1\}$ et $c_n \in \{0, 1\}$:

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

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

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

Super generateur?

Exemple: subtract-with-borrow (SWB)

État $(x_{n-48}, \ldots, x_{n-1}, c_{n-1})$ où $x_n \in \{0, \ldots, 2^{31} - 1\}$ et $c_n \in \{0, 1\}$:

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

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

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

Super generateur? Non pas du tout; très mauvais en fait...

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

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

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

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

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

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

Dépendance beaucoup trop évidente entre les valeurs successives.
Par exemple, les points $(u_n, u_{n+40}, u_{n+48})$ sont tous situés dans seulement deux plans parallèle dans le cube $[0, 1)^3$.

Générateurs algorithmiques

Une fois les paramètres et l'état initial x_0 du GPA choisis, la suite devient complètement déterministe.

Générateurs algorithmiques

Une fois les paramètres et l'état initial x_0 du GPA choisis, la suite devient complètement déterministe.

Avantages: pas de matériel à installer, un logiciel suffit; souvent plus rapide; on peut facilement répéter la même séquence.

Générateurs algorithmiques

Une fois les paramètres et l'état initial x_0 du GPA choisis, la suite devient complètement déterministe.

Avantages: pas de matériel à installer, un logiciel suffit; souvent plus rapide; on peut facilement répéter la même séquence.

Désavantage: ne peut pas créer de l'entropie!
Il y a nécessairement des dépendances entre les nombres en sortie.

Qualités requises: dépend des applications.

1. Jeux d'ordinateurs personnels: L'apparence suffit.

1. Jeux d'ordinateurs personnels: L'apparence suffit.

2. Simulation stochastique (Monte Carlo):
On simule un modèle mathématique d'un système pour comprendre son comportement, ou optimiser sa gestion, etc.
Exemples: hôpital, centre d'appels, logistique, transport, finance, etc.
On veut que les propriétés statistiques du modèle soient bien reproduites par le simulateur. Générateurs algorithmiques.

1. Jeux d'ordinateurs personnels: L'apparence suffit.

2. Simulation stochastique (Monte Carlo):
On simule un modèle mathématique d'un système pour comprendre son comportement, ou optimiser sa gestion, etc.
Exemples: hôpital, centre d'appels, logistique, transport, finance, etc.
On veut que les propriétés statistiques du modèle soient bien reproduites par le simulateur. Générateurs algorithmiques.

3. Loteries, machines de casinos, casinos sur Internet, ...
On veut que personne ne puisse obtenir un avantage.
Plus exigeant que la simulation.
Générateurs algorithmiques + mécanismes physiques.

1. Jeux d'ordinateurs personnels: L'apparence suffit.

2. Simulation stochastique (Monte Carlo):
On simule un modèle mathématique d'un système pour comprendre son comportement, ou optimiser sa gestion, etc.
Exemples: hôpital, centre d'appels, logistique, transport, finance, etc.
On veut que les propriétés statistiques du modèle soient bien reproduites par le simulateur. Générateurs algorithmiques.

3. Loteries, machines de casinos, casinos sur Internet, ...
On veut que personne ne puisse obtenir un avantage.
Plus exigeant que la simulation.
Générateurs algorithmiques + mécanismes physiques.

4. Cryptologie: Plus exigeant. L'observation d'une partie de l'output ne doit pas nous aider à deviner une partie du reste.
Générateurs algorithmiques non-linéaires avec paramètres aléatoires.
Souvent: contraintes sur les ressources disponibles pour les calculs.

Générateur algorithmique

$\mathcal{S}$, espace d'états fini;
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0, 1]$, fonction de sortie.

s_0, germe (état initial);

s_0

Générateur algorithmique

$\mathcal{S}$, espace d'états fini;
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0,1]$, fonction de sortie.

s_0, germe (état initial);

$$\begin{array}{c} s_0 \\ \Big\downarrow{\scriptstyle g} \\ u_0 \end{array}$$

Générateur algorithmique

$\mathcal{S}$, espace d'états fini;
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0,1]$, fonction de sortie.

s_0, germe (état initial);

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

Générateur algorithmique

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0, 1]$, fonction de sortie.

$$\begin{array}{ccc} s_0 & \xrightarrow{f} & s_1 \\ g\downarrow & & g\downarrow \\ u_0 & & u_1 \end{array}$$

Générateur algorithmique

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0, 1]$, fonction de sortie.

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

Générateur algorithmique

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;
$g : \mathcal{S} \to [0,1]$, fonction de sortie.

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

Période de $\{s_n,\ n \geq 0\}$: $\rho \leq$ cardinalité de $\mathcal{S}$.

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

Objectif: en observant seulement $(u_0, u_1, \dots)$, difficile de distinguer d'une suite de v.a. indépendantes uniformes sur $(0, 1)$.

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

Objectif: en observant seulement $(u_0, u_1, \ldots)$, difficile de distinguer d'une suite de v.a. indépendantes uniformes sur $(0, 1)$.

Utopie: passe tous les tests statistiques imaginables.
Impossible! On doit se contenter d'une approximation.

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

Objectif: en observant seulement $(u_0, u_1, \ldots)$, difficile de distinguer d'une suite de v.a. indépendantes uniformes sur $(0, 1)$.

Utopie: passe tous les tests statistiques imaginables.
Impossible! On doit se contenter d'une approximation.

On veut aussi: vitesse, facilité d'implantation, suites reproduisibles.

Compromis entre vitesse / propriétés statistiques / imprévisibilité.

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

Objectif: en observant seulement $(u_0, u_1, \ldots)$, difficile de distinguer d'une suite de v.a. indépendantes uniformes sur $(0, 1)$.

Utopie: passe tous les tests statistiques imaginables.
Impossible! On doit se contenter d'une approximation.

On veut aussi: vitesse, facilité d'implantation, suites reproduisibles.

Compromis entre vitesse / propriétés statistiques / imprévisibilité.

Machines de casinos et loteries: on modifie l'état s_n régulièrement à l'aide de mécanismes physiques. **Exemples: Spielo, Casino de Montréal.**

La loi uniforme sur $[0,1]^s$.

Si on choisit s_0 au hasard dans $\mathcal{S}$ et on génère s nombres, cela correspond à choisir un point au hasard dans l'ensemble fini

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

On veut approximer: "$\mathbf{u}$ suit la loi uniforme sur $[0,1]^s$."

La loi uniforme sur $[0,1]^s$.

Si on choisit s_0 au hasard dans $\mathcal{S}$ et on génère s nombres, cela correspond à choisir un point au hasard dans l'ensemble fini

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

On veut approximer: "$\mathbf{u}$ suit la loi uniforme sur $[0,1]^s$."

Mesure de qualité: Ψ_s doit recouvrir $[0,1]^s$ très uniformément.

La loi uniforme sur $[0,1]^s$.

Si on choisit s_0 au hasard dans $\mathcal{S}$ et on génère s nombres, cela correspond à choisir un point au hasard dans l'ensemble fini

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

On veut approximer: "$\mathbf{u}$ suit la loi uniforme sur $[0,1]^s$."

Mesure de qualité: Ψ_s doit recouvrir $[0,1]^s$ très uniformément.

Conception et analyse théorique des générateurs:

1. Définir une mesure d'uniformité de Ψ_s, calculable sans générer les points explicitement. GPA linéaires.
2. Choisir un type de construction (rapide, longue période, etc.) et chercher des paramètres qui "optimisent" cette mesure.

Mythe 1. Après au moins 60 ans à étudier les GPA et des milliers d'articles publiés, ce problème est certainement réglé et les GPA disponibles dans les logiciels populaires sont certainement fiables.

Mythe 1. Après au moins 60 ans à étudier les GPA et des milliers d'articles publiés, ce problème est certainement réglé et les GPA disponibles dans les logiciels populaires sont certainement fiables.

Non.

Mythe 2. Dans votre logiciel favori, le générateur a une période supérieure à 2^{1000}. Il est donc certainement excellent!

Mythe 1. Après au moins 60 ans à étudier les GPA et des milliers d'articles publiés, ce problème est certainement réglé et les GPA disponibles dans les logiciels populaires sont certainement fiables.

Non.

Mythe 2. Dans votre logiciel favori, le générateur a une période supérieure à 2^{1000}. Il est donc certainement excellent!

Non.

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

Exemple 2. Subtract-with-borrow.

Un seul GPA (monolithique) ne suffit pas.

On a souvent besoin de plusieurs flux (ou suites, ou "streams") "indépendants" de nombres aléatoires. Exemples:

- exécuter une simulation sur plusieurs processeurs en parallèle,
- Comparaison de systèmes avec valeurs aléatoires communes (important pour analyse de sensibilité, estimation de dérivées, optimisation, ...).

Un seul GPA (monolithique) ne suffit pas.

On a souvent besoin de plusieurs flux (ou suites, ou “streams”) “indépendants” de nombres aléatoires. Exemples:

- exécuter une simulation sur plusieurs processeurs en parallèle,
- Comparaison de systèmes avec valeurs aléatoires communes (important pour analyse de sensibilité, estimation de dérivées, optimisation, ...).

Un logiciel développé au DIRO, fournit de tels `RandomStream` (objets). On peut en créer autant qu’on veut. Agisssent comme des GPA virtuels.

Intégré dans la librairie SSJ (“Stochastic Simulation in Java”), au DIRO. Adopté par MATLAB, SAS, Arena, Simul8, Automod, Witness, ns2, R, ...

Un seul GPA (monolithique) ne suffit pas.

On a souvent besoin de plusieurs flux (ou suites, ou “streams”) “indépendants” de nombres aléatoires. Exemples:

- exécuter une simulation sur plusieurs processeurs en parallèle,
- Comparaison de systèmes avec valeurs aléatoires communes (important pour analyse de sensibilité, estimation de dérivées, optimisation, ...).

Un logiciel développé au DIRO, fournit de tels `RandomStream` (objets). On peut en créer autant qu’on veut. Agisssent comme des GPA virtuels.

Intégré dans la librairie SSJ (“Stochastic Simulation in Java”), au DIRO. Adopté par MATLAB, SAS, Arena, Simul8, Automod, Witness, ns2, R, ...

Exemple: Synthèse d’image par Monte Carlo sur GPU.
(Merci à Steve Worley, de Worley laboratories).

Générateur linéaire récursif multiple (MRG)

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

État: $s_n = (x_{n-k+1}, \ldots, x_n)$. Période max. $\rho = m^k - 1$.

Générateur linéaire récursif multiple (MRG)

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

État: $s_n = (x_{n-k+1}, \ldots, x_n)$. Période max. $\rho = m^k - 1$.
Nombreuses variantes et implantations.

Si $k = 1$: générateur à congruence linéaire (GCL) classique.

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

Générateur linéaire récursif multiple (MRG)

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

État: $s_n = (x_{n-k+1}, \ldots, x_n)$. Période max. $\rho = m^k - 1$.
Nombreuses variantes et implantations.

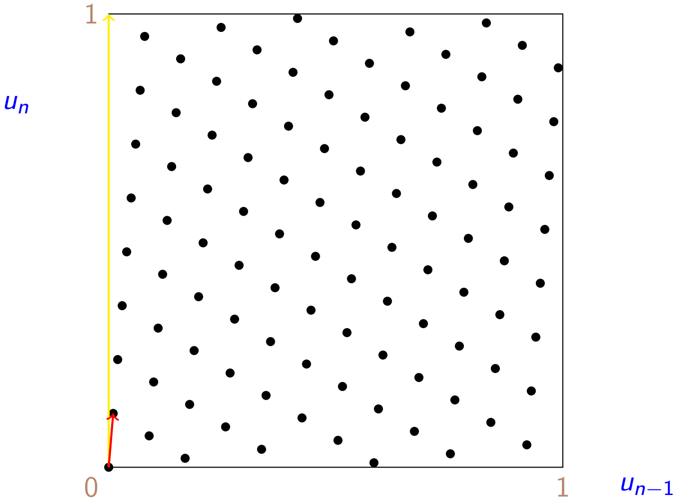
Si $k = 1$: générateur à congruence linéaire (GCL) classique.

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

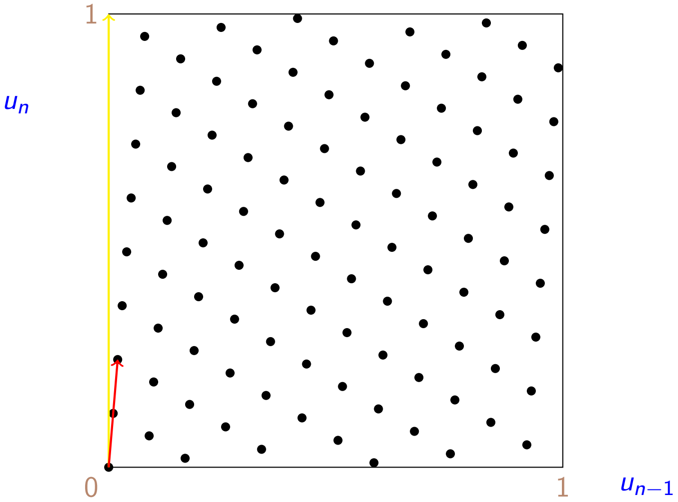
Structure des points Ψ_s:

$x_0, \ldots, x_{k-1}$ peuvent prendre n'importe quelle valeur de 0 à $m-1$, puis $x_k, x_{k+1}, \ldots$ sont déterminés par la récurrence linéaire. Ainsi, $(x_0, \ldots, x_{k-1}) \mapsto (x_0, \ldots, x_{k-1}, x_k, \ldots, x_{s-1})$ est une application linéaire.

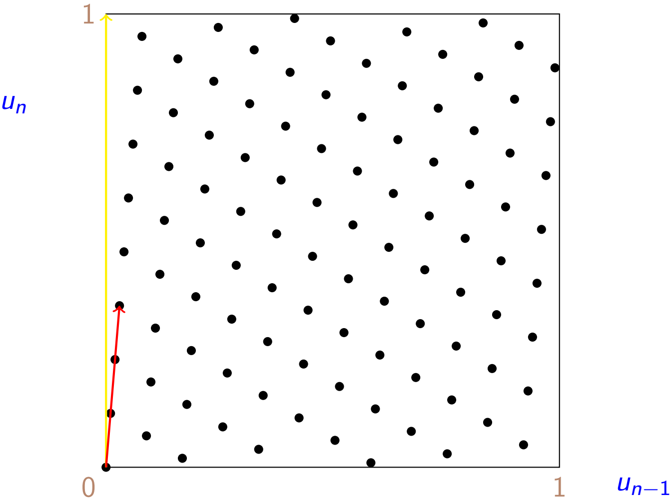
On peut en déduire que Ψ_s a une structure d'espace linéaire.



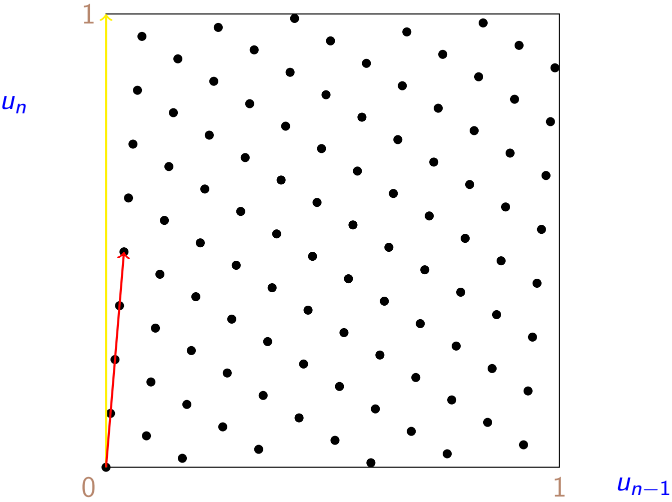
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



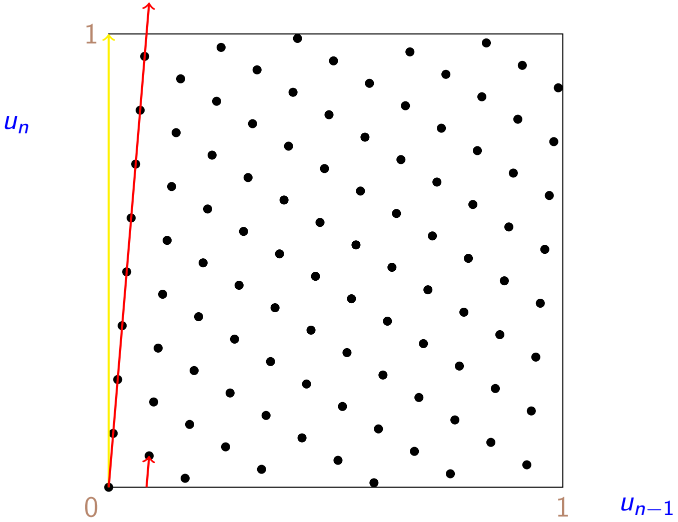
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



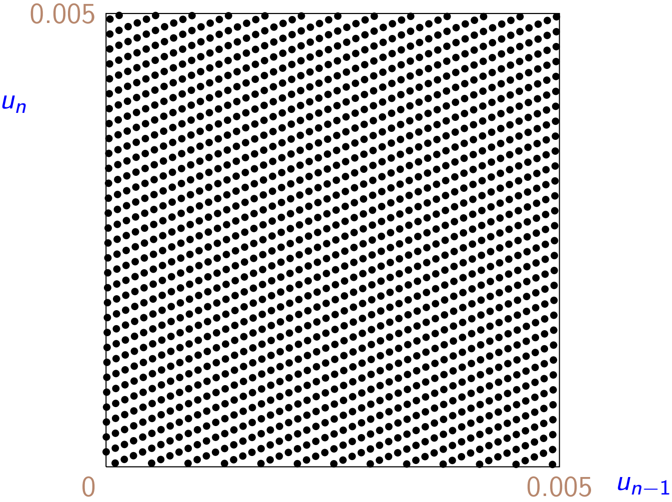
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



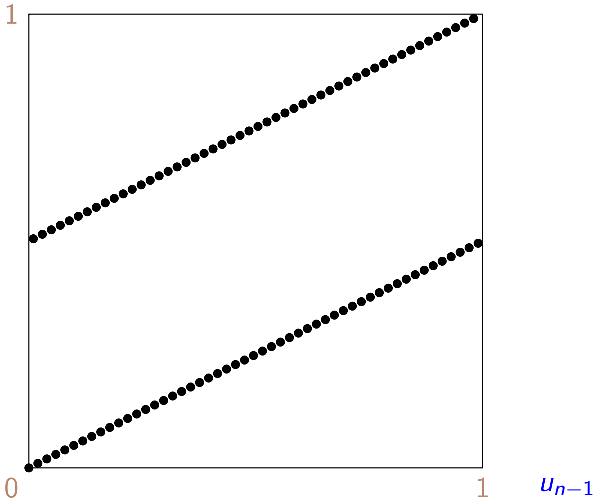
$$x_n = 12\,x_{n-1} \bmod 101; \qquad u_n = x_n/101$$



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

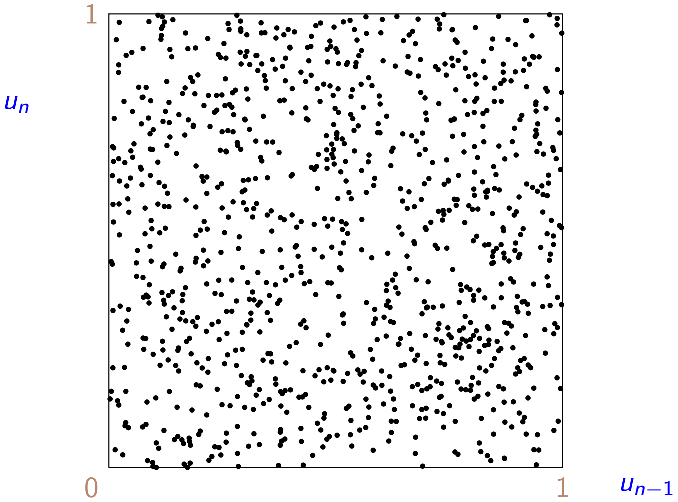


$$x_n = 4809922\, x_{n-1} \bmod 60466169 \text{ et } u_n = x_n/60466169$$

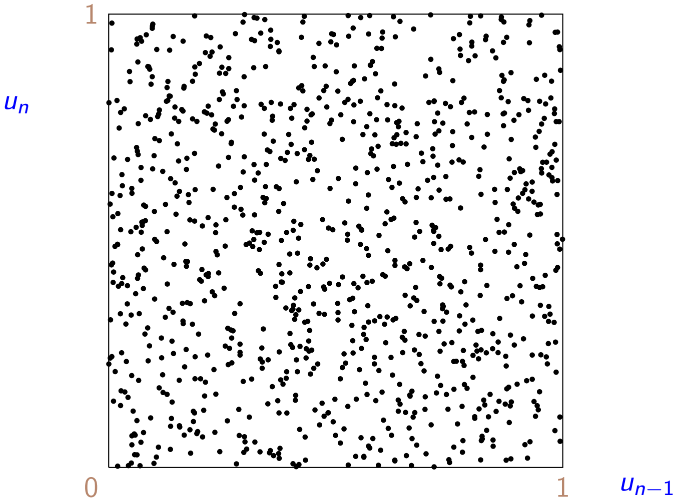


$$x_n = 51\, x_{n-1} \bmod 101; \qquad u_n = x_n/101.$$

Ici, on a une bonne uniformité en une dimension, mais pas en deux!



1000 points générés par MRG32k3a



1000 points générés par LFSR113

MRGs combinés.

Deux [ou plusieurs...] MRGs évoluant en parallèle:

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

Combinaison possible:

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

MRGs combinés.

Deux [ou plusieurs...] MRGs évoluant en parallèle:

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

Combinaison possible:

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

L'Ecuyer (1996): la suite $\{u_n,\ n \geq 0\}$ est la sortie d'un MRG de modulo $m = m_1 m_2$, avec un petit "bruit" ajouté. La période peut atteindre $(m_1^k - 1)(m_2^k - 1)/2$.

Permet d'implanter efficacement un MRG ayant un grand m et plusieurs grands coefficients non nuls.

Paramètres: L'Ecuyer (1999); L'Ecuyer et Touzin (2000). Implantations "multistreams" réalisées au DIRO.

Récurrences Linéaires Modulo 2

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

Récurrences Linéaires Modulo 2

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

Choix astucieux de **A**: transition via des décalages, XOR, AND, masques, etc., sur des blocs de bits. Très rapide.

Cas particuliers: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.

Récurrences Linéaires Modulo 2

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

Choix astucieux de **A**: transition via des décalages, XOR, AND, masques, etc., sur des blocs de bits. Très rapide.

Cas particuliers: Tausworthe, LFSR, GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, etc.

Chaque coordonnée de $\mathbf{x}_n$ et de $\mathbf{y}_n$ suit la récurrence

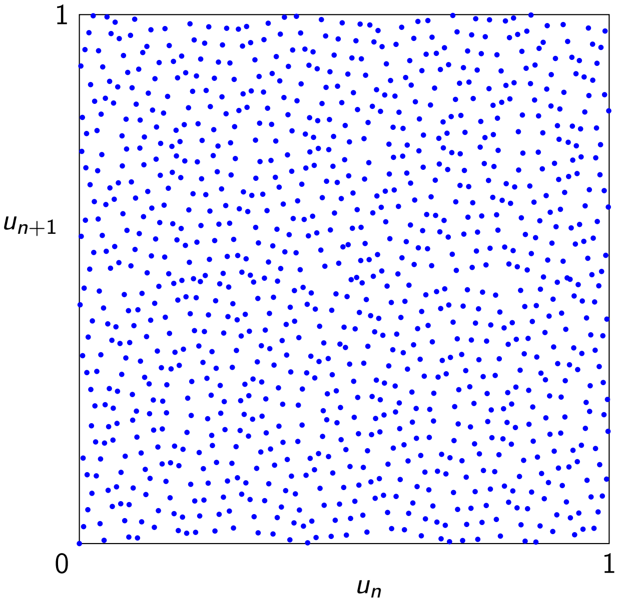
$$x_{n,j} = (\alpha_1 x_{n-1,j} + \cdots + \alpha_k x_{n-k,j}),$$

de polynôme caractéristique

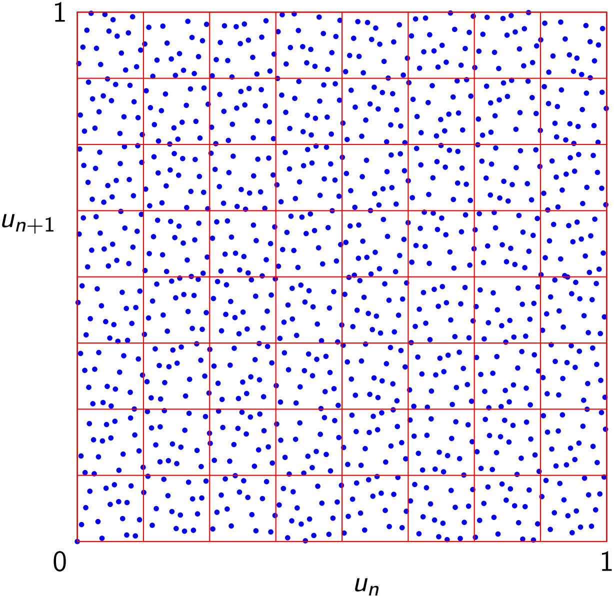
$$P(z) = z^k - \alpha_1 z^{k-1} - \cdots - \alpha_{k-1} z - \alpha_k = \det(\mathbf{A} - z\mathbf{I}).$$

La période max. $\rho = 2^k - 1$ est atteinte ssi $P(z)$ est primitif.

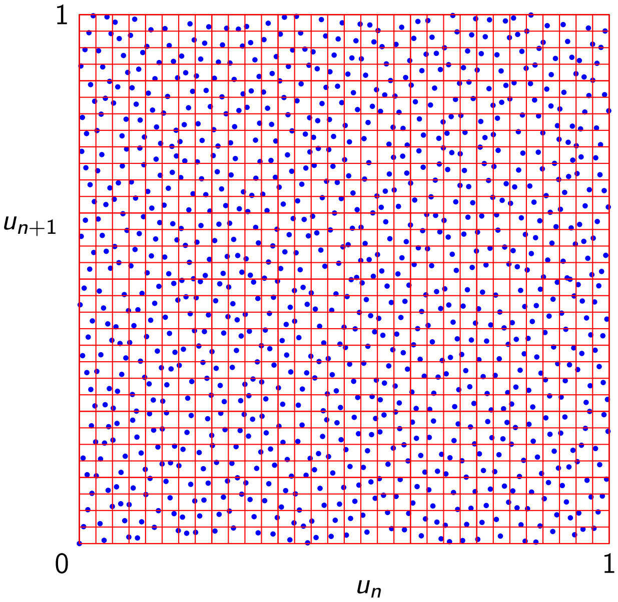
Mesures d'uniformité. Exemple: $k = 10$, $2^{10} = 1024$ points



Mesures d'uniformité. Exemple: $k = 10$, $2^{10} = 1024$ points



Mesures d'uniformité. Exemple: $k = 10$, $2^{10} = 1024$ points



Mesures d'uniformité basées sur l'équidistribution.

Exemple: on partitionne $[0,1)^s$ en 2^ℓ intervalles égaux.
Donne $2^{s\ell}$ boîtes cubiques.

Les points sont équidistribués pour ℓ bits en s dimensions si chaque boîte contient exactement $2^{k-s\ell}$ points de Ψ_s.

Mesures d'uniformité basées sur l'équidistribution.

Exemple: on partitionne $[0,1)^s$ en 2^ℓ intervalles égaux.
Donne $2^{s\ell}$ boîtes cubiques.

Les points sont équidistribués pour ℓ bits en s dimensions si chaque boîte contient exactement $2^{k-s\ell}$ points de Ψ_s.

Pour chaque s et ℓ, on peut écrire les $s\ell$ bits qui déterminent la boîte comme $M\,\mathbf{x}_0$ et on a l'équidistribution ssi la matrice M est de plein rang.

Mesures d'uniformité basées sur l'équidistribution.

Exemple: on partitionne $[0,1)^s$ en 2^ℓ intervalles égaux.
Donne $2^{s\ell}$ boîtes cubiques.

Les points sont équidistribués pour ℓ bits en s dimensions si chaque boîte contient exactement $2^{k-s\ell}$ points de Ψ_s.

Pour chaque s et ℓ, on peut écrire les $s\ell$ bits qui déterminent la boîte comme $M\,\mathbf{x}_0$ et on a l'équidistribution ssi la matrice M est de plein rang.

Si cette propriété tient pour tous s et ℓ tels que $s\ell \le k$, le générateur est dit équidistribué au maximum.

Mesures d'uniformité basées sur l'équidistribution.

Exemple: on partitionne $[0, 1)^s$ en 2^ℓ intervalles égaux.
Donne $2^{s\ell}$ boîtes cubiques.

Les points sont équidistribués pour ℓ bits en s dimensions si chaque boîte contient exactement $2^{k-s\ell}$ points de Ψ_s.

Pour chaque s et ℓ, on peut écrire les $s\ell$ bits qui déterminent la boîte comme $M\,\mathbf{x}_0$ et on a l'équidistribution ssi la matrice M est de plein rang.

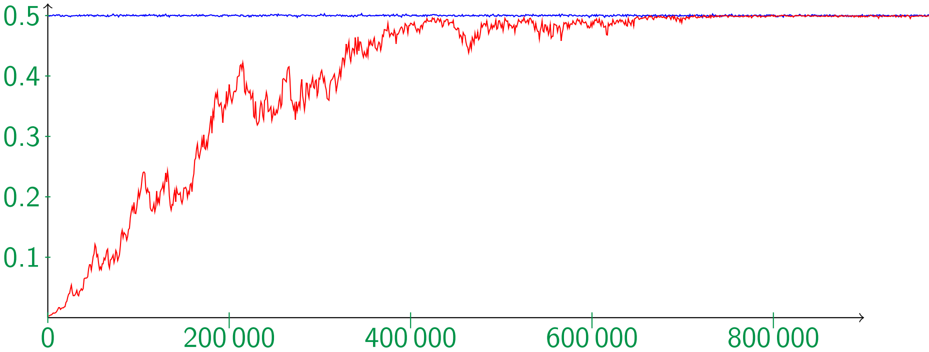
Si cette propriété tient pour tous s et ℓ tels que $s\ell \leq k$, le générateur est dit équidistribué au maximum.

Exemples: LFSR113, Mersenne twister (MT19937), famille WELL, ...

Impact d'une matrice A qui ne "modifie" pas assez l'état.

Expérience: choisir un état initial contenant un seul bit à 1.
Essayer toutes les k possibilités et faire la moyenne des k valeurs de u_n obtenues pour chaque n.

WELL19937 vs MT19937; moyenne mobile sur 1000 itérations.



Générateurs combinés linéaires/non-linéaires

Les générateurs linéaires modulo 2 échouent tous (bien sûr) des tests qui mesurent la complexité linéaire.

Générateurs combinés linéaires/non-linéaires

Les générateurs linéaires modulo 2 échouent tous (bien sûr) des tests qui mesurent la complexité linéaire.

On voudrait:

- éliminer la structure linéaire;
- des garanties théoriques sur l'uniformité;
- implantation rapide.

Générateurs combinés linéaires/non-linéaires

Les générateurs linéaires modulo 2 échouent tous (bien sûr) des tests qui mesurent la complexité linéaire.

On voudrait:

- éliminer la structure linéaire;
- des garanties théoriques sur l'uniformité;
- implantation rapide.

L'Ecuyer et Granger-Picher (2003): Gros générateur linéaire modulo 2 combiné avec un petit non-linéaire par un XOR.

Théorème: Si la composante linéaire est $(q_1, \ldots, q_t)$-équidistribuée, alors la combinaison l'est aussi.

Tests empiriques: excellent comportement, plus robuste que linéaire.

Vitesse de quelques générateurs dans SSJ

temps gen.: temps de CPU (sec) pour générer 10^9 nombres réels en 0 et 1.
temps saut: temps pour obtenir un nouveau flot (sauter en avant) 10^6 fois.

Java JDK 1.5, AMD 2.4 GHz 64-bit, `RngStream` dans SSJ

GPA	période	temps gen.	temps saut
LFSR113	2^{113}	31	0.1
LFSR258	2^{258}	35	0.2
WELL512	2^{512}	33	234
WELL1024	2^{1024}	34	917
MT19937	2^{19937}	36	—
MRG31k3p	2^{185}	51	0.9
MRG32k3a	2^{191}	70	1.1
RandRijndael	2^{130}	127	0.6

Tests statistiques empiriques

Hypothèse nulle $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \dots\}$ v.a. indép. $U(0,1)$".
On sait à l'avance que $\mathcal{H}_0$ est fausse, mais peut-on le détecter facilement?

Tests statistiques empiriques

Hypothèse nulle $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \ldots\}$ v.a. indép. $U(0,1)$".
On sait à l'avance que $\mathcal{H}_0$ est fausse, mais peut-on le détecter facilement?

Test:
— Choisir une v.a. T, fonction des u_i, de loi connue (approx.) sous $\mathcal{H}_0$.
— Rejeter $\mathcal{H}_0$ si T prend une valeur trop extrême p.r. à cette loi.
Si la valeur est "suspecte", on peut répéter le test.

Différents tests permettent de détecter différents types de défauts.

Tests statistiques empiriques

Hypothèse nulle $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \ldots\}$ v.a. indép. $U(0,1)$".
On sait à l'avance que $\mathcal{H}_0$ est fausse, mais peut-on le détecter facilement?

Test:
— Choisir une v.a. T, fonction des u_i, de loi connue (approx.) sous $\mathcal{H}_0$.
— Rejeter $\mathcal{H}_0$ si T prend une valeur trop extrême p.r. à cette loi.
Si la valeur est "suspecte", on peut répéter le test.

Différents tests permettent de détecter différents types de défauts.

Rêve: Construire un GPA qui passe tous les tests? Impossible.

Tests statistiques empiriques

Hypothèse nulle $\mathcal{H}_0$: "$\{u_0, u_1, u_2, \dots\}$ v.a. indép. $U(0,1)$".
On sait à l'avance que $\mathcal{H}_0$ est fausse, mais peut-on le détecter facilement?

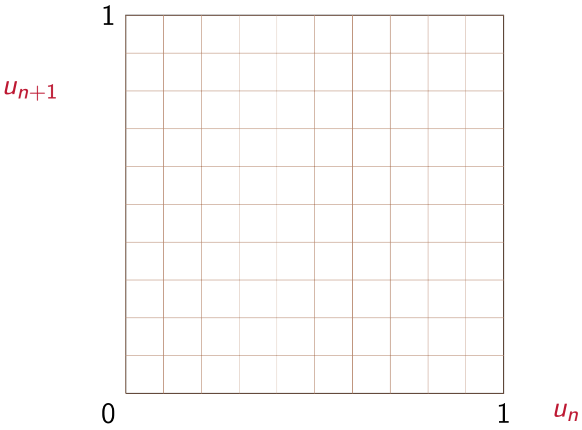
Test:
— Choisir une v.a. T, fonction des u_i, de loi connue (approx.) sous $\mathcal{H}_0$.
— Rejeter $\mathcal{H}_0$ si T prend une valeur trop extrême p.r. à cette loi.
Si la valeur est "suspecte", on peut répéter le test.

Différents tests permettent de détecter différents types de défauts.

Rêve: Construire un GPA qui passe tous les tests? Impossible.

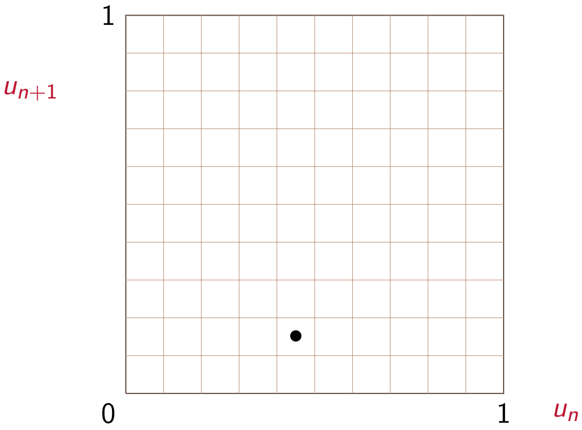
Compromis (heuristique): un GPA qui passe les tests raisonnables.
Les tests échoués doivent être très difficiles à trouver et exécuter.
Formalisation: complexité algorithmique, populaire en cryptologie.

Exemple: Un test de collisions



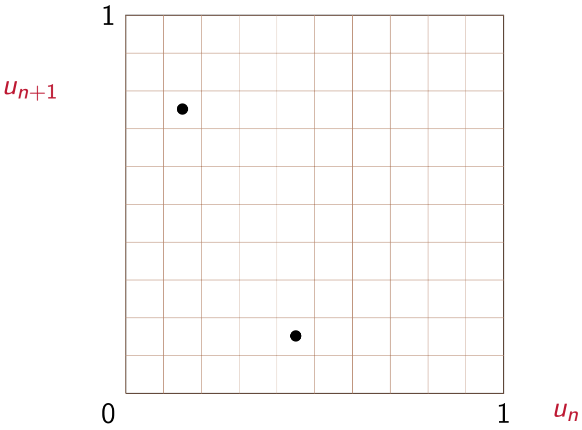
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



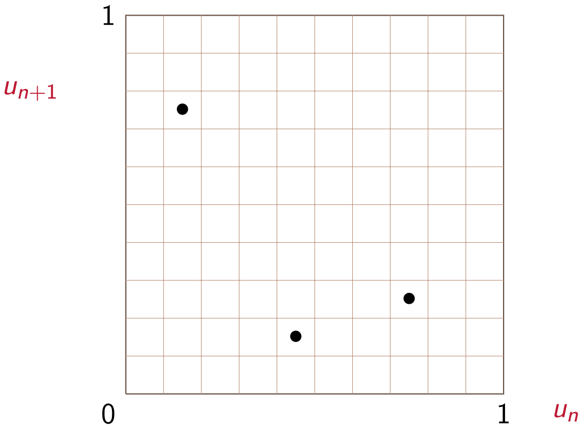
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



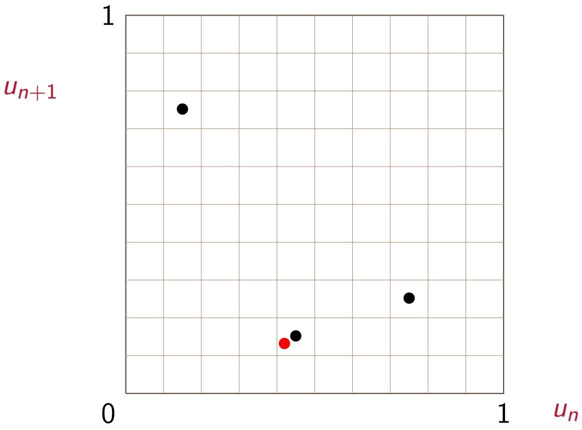
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



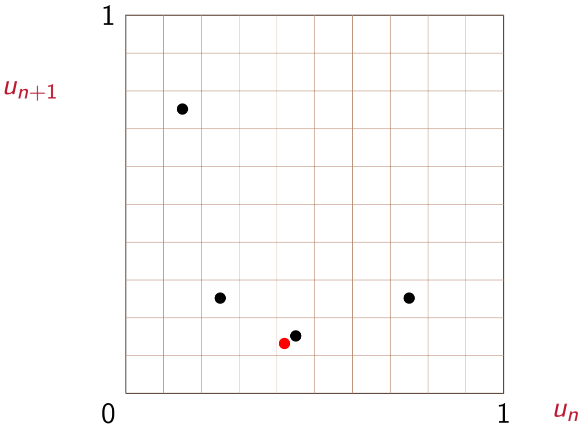
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



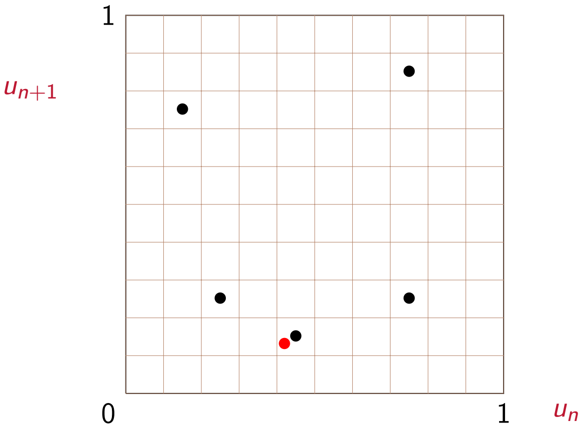
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



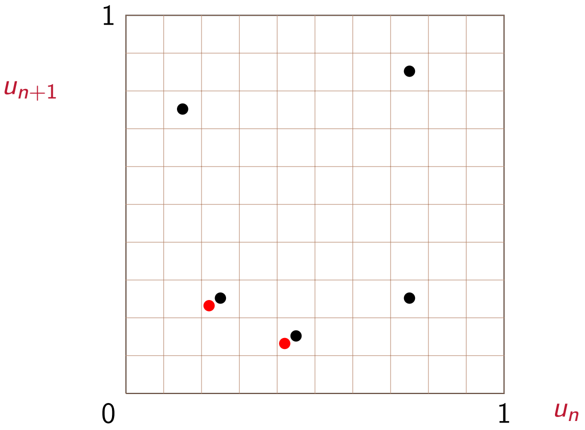
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



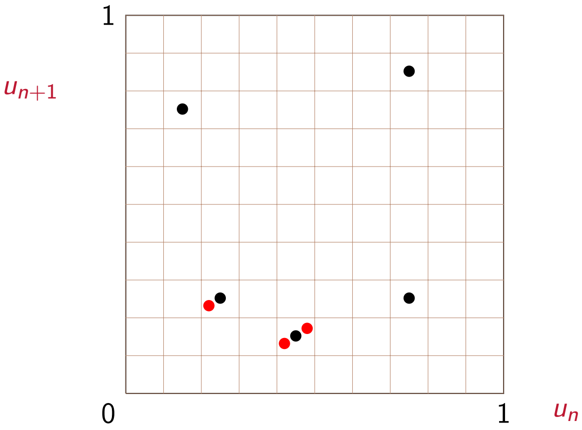
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



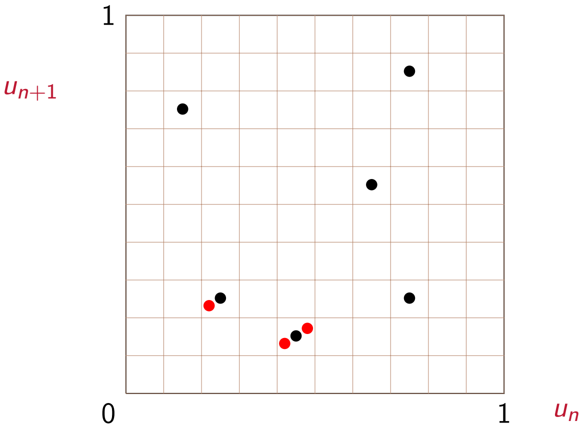
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



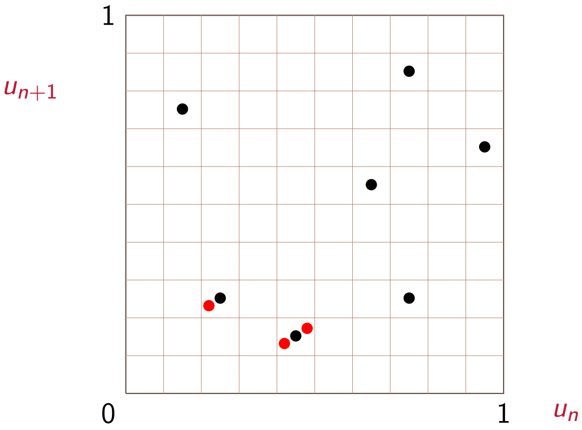
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



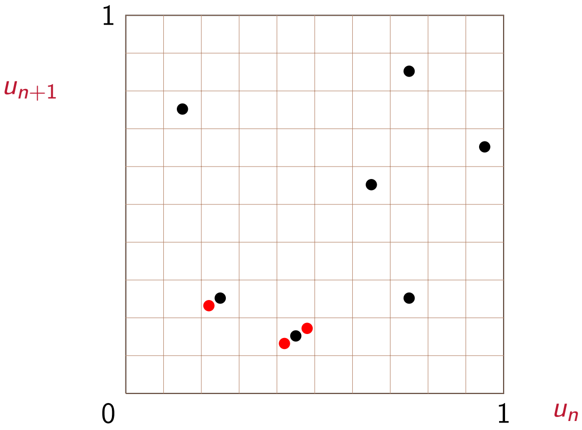
On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



On lance $n = 10$ points dans $k = 100$ cases.

Exemple: Un test de collisions



On lance $n = 10$ points dans $k = 100$ cases.

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

Test de collisions

On partitionne $[0,1)^s$ en $k = d^s$ boîtes cubiques de même taille.
On génère n points $(u_{is}, \ldots, u_{is+s-1})$ dans $[0,1)^s$.

$$C = \text{nombre de collisions.}$$

Test de collisions

On partitionne $[0,1)^s$ en $k = d^s$ boîtes cubiques de même taille.
On génère n points $(u_{is}, \ldots, u_{is+s-1})$ dans $[0,1)^s$.

$$C = \text{nombre de collisions.}$$

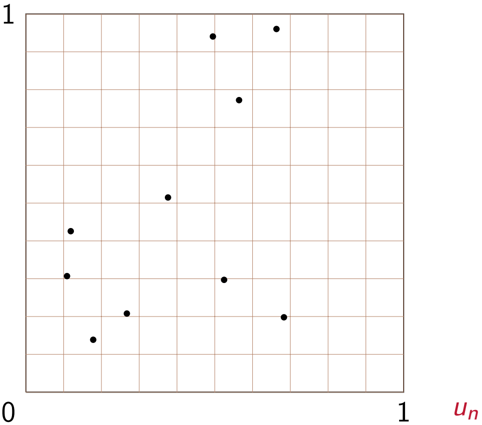
Sous $\mathcal{H}_0$, $C \approx$ Poisson de moyenne $\lambda = n^2/(2k)$, si k est grand et λ petit.

Si on observe c collisions, on calcule les p-valeurs:

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

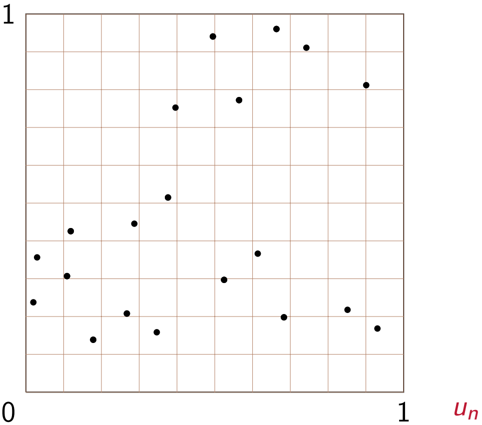
On rejette $\mathcal{H}_0$ si $p^+(c)$ est trop proche de 0 (trop de collisions)
ou $p^-(c)$ est trop proche de 1 (pas assez de collisions).

GCL avec $m = 101$ et $a = 12$:



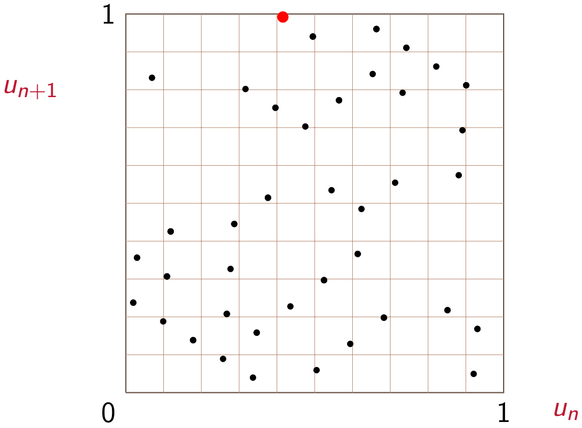
n	λ	C	$p^-(C)$
10	1/2	0	0.6281

GCL avec $m = 101$ et $a = 12$:



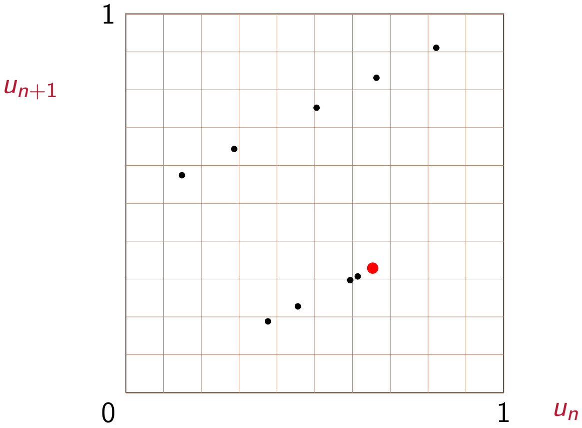
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304

GCL avec $m = 101$ et $a = 12$:



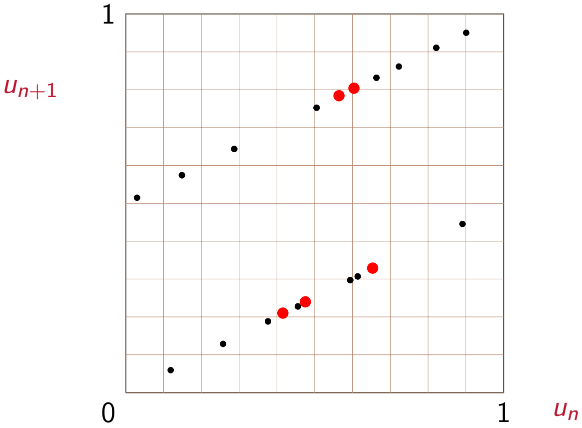
n	λ	C	$p^-(C)$
10	1/2	0	0.6281
20	2	0	0.1304
40	8	1	0.0015

GCL avec $m = 101$ et $a = 51$:



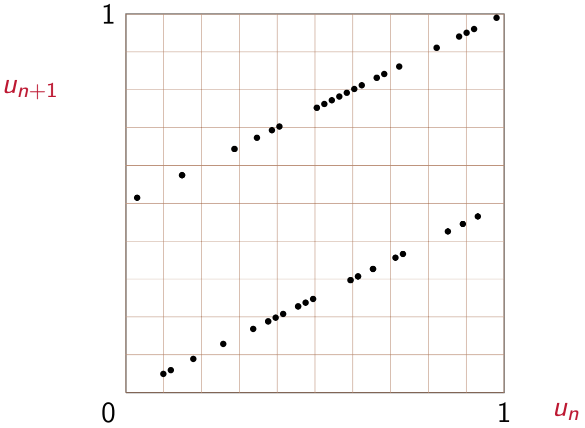
n	λ	C	$p^+(C)$
10	$1/2$	1	0.3718

GCL avec $m = 101$ et $a = 51$:



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

GCL avec $m = 101$ et $a = 51$:



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

SWB de Mathematica (Devoir 1 de IFT-6561, A-2009).

Dans le cube à 3 dimensions, on divise chaque axe en $d = 100$ intervalles: donne $k = 100^3 = 1$ million de cases.

On génére $n = 10\,000$ vecteurs en 25 dimensions: $(U_0, \ldots, U_{24})$.
Pour chacun, on regarde la case où tombe (U_0, U_{20}, U_{24}).
Ici, $\lambda = 50$.

SWB de Mathematica (Devoir 1 de IFT-6561, A-2009).

Dans le cube à 3 dimensions, on divise chaque axe en $d = 100$ intervalles: donne $k = 100^3 = 1$ million de cases.

On génére $n = 10\,000$ vecteurs en 25 dimensions: $(U_0, \dots, U_{24})$.
Pour chacun, on regarde la case où tombe (U_0, U_{20}, U_{24}).
Ici, $\lambda = 50$.

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

SWB de Mathematica (Devoir 1 de IFT-6561, A-2009).

Dans le cube à 3 dimensions, on divise chaque axe en $d = 100$ intervalles: donne $k = 100^3 = 1$ million de cases.

On génére $n = 10\,000$ vecteurs en 25 dimensions: $(U_0, \ldots, U_{24})$.
Pour chacun, on regarde la case où tombe (U_0, U_{20}, U_{24}).
Ici, $\lambda = 50$.

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

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

Autres exemples de tests

Paires de points les plus proches $[0, 1)^s$.

Trier des jeux de cartes (poker, etc.).

Rang d'une matrice binaire aléatoire.

Complexité linéaire d'une suite binaire.

Mesures d'entropie.

Mesures de complexité basées sur la facilité de compression de la suite.

Etc.

Le Logiciel TestU01

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

- Grande variété de tests statistiques.
  Pour générateurs algorithmiques ou physiques.
  Très largement utilisé. Disponible sur ma page web.
- Quelques batteries de tests prédéfinies:
  SmallCrush: vérification rapide, 15 secondes;
  Crush: 96 tests statistiques, 1 heure;
  BigCrush: 144 tests statistiques, 6 heures;
  Rabbit: pour les suites de bits.
- Plusieurs générateurs couramment utilisés échouent ces batteries.

Quelques résultats. $\rho =$ période du GPA;
t-32 et t-64 donnent le temps de CPU pour générer 10^8 nombres réels.

Résultats de batteries de tests pour des GPA bien connus.

Nombre de tests échoués (p-valeur $< 10^{-10}$ ou $> 1 - 10^{-10}$).

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

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

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

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

Conclusion

- Une foule d'applications informatiques reposent sur les GPAs.
  Un mauvais générateur peut fausser complètement les résultats d'une simulation, ou permettre de tricher dans les loteries ou déjouer les machines de jeux, ou mettre en danger la sécurité d'informations importantes.
- Ne jamais se fier aveuglément aux GPAs fournis dans les logiciels commerciaux ou autres, même les plus connus, surtout s'ils utilisent des algorithmes secrets!
- Des GPAs avec suites et sous-suites multiples sont disponibles via ma page web, en Java, C, et C++.

http://www.iro.umontreal.ca/~lecuyer