Nombres aléatoires

Vincent Nozick

Pour quoi faire?

- Simulation physique.
- Étude de probabilité.
- Jeux.
- Algorithmes stochastiques* :
  - Quick sort.
  - Lancer de rayons.
- Autres ...

* comportement déterminé par une entrée + un nombre aléatoire.

Qualités requises

- Vitesse.
- Simplicité.
- Robustesse (ne se coince pas).
- Ne représente pas une suite logique apparente.
- Reproductible.

Qualités requises

“Ne présente pas de suite logique apparente” :

L’important ici n’est pas le nombre généré mais la suite de nombres générée.

Pour un bon générateur, on espère que la probabilité pour un nombre y de succéder à un nombre x est la même pour tout les y (sur un intervalle donné).

Familles de générateurs

Mécanismes physiques :

- Pile ou face.
- Loto.
- L'heure.
- Le temps écoulé entre 2 "bips" d'un compteur Geiger.
- ...

Introduction LCG Xorshift Mersenne Twister Tester un générateur Les distributions En C++

Familles de générateurs

Mécanismes physiques :

- Pile ou face.
- Loto.
- L'heure.
- Le temps écoulé entre 2 "bips" d'un compteur Geiger.
- ...

coûteux, ne peut pas reproduire la même suite.

Familles de générateurs

Générateurs pseudo-aléatoires :

- Fonctions déterministes :
  - soit x_i un état $x_{i+1} = f(x_i)$
  - x_0 = graine (seed)
- Une période p $(x_i = x_{i+p})$.
- Parfois : une phase de démarrage.

Générateurs à congruence linéaire

- Linear Congruencial Generator → LCG.
- Très simple.
- Très utilisé.
- Un des mieux connu.

Générateurs à congruence linéaire

Définition :

$$x_{n+1} = (a.x_n + b) \mod m$$

s'écrit LCG(m,a,b,x_0)

Exemple :

LCG(5,2,1,0) $\rightarrow x_{n+1} = (2x_n + 1) \mod 5$
0, 1, 3, 2, 0, ...

Générateurs à congruence linéaire

Propriété :

- Une graine x_0.
- Une période $p < m$ la plus grande possible.
- Une phase de démarrage la plus petite possible.
- Un LCG est dit :
  - mixte si $b > 0$
  - multiplicatif si $b = 0$

Générateurs à congruence linéaire

Période :

Si $p = m$, alors on peut choisir arbitrairement x_0 car toutes les valeurs entières entre 0 et $m - 1$ sont prises exactement une fois par cycle.

Générateurs à congruence linéaire

Théorème :
Un LCG est de période maximal $(p = m)$ ssi :

- m et b sont premiers entre eux.
- tout nombre premier qui divise m divise aussi $a - 1$.
- si 4 divise m, alors 4 divise $a - 1$.

Attention :
une longue période est nécessaire mais pas suffisante pour considérer qu'un générateur est bon.

Générateurs à congruence linéaire

Période de démarrage : Exemple :

LCG(24,2,1,0)

$$x_{n+1} = (2x_n + 1) \mod 24$$

$$\overbrace{0, 1, 3,}^{\text{démarrage}} \underbrace{7, 15,}_{\text{période}} 7, \ldots$$

LCG multiplicatifs

s'écrit MLCG(m,a,x_0)

$$x_{n+1} = ax_n \mod m$$

Ils sont très utilisés car ils présentent l'avantage d'économiser une addition par rapport aux LCG mixtes.

LCG multiplicatifs

Théorème :

Un MLCG est de période maximal $(p = m)$ ssi :

- m est premier.
- $m - 1$ est le plus petit entier k tel que $(a^k - 1) \mod m = 0$.

Exemple :

$x_{n+1} = (16807x_n) \mod (2^{31} - 1)$	Apple CarbonLib
$x_{n+1} = (214013x_n + 2531011) \mod 2^{32}$	Microsoft Visual C
$x_{n+1} = (134775813x_n + 1) \mod 2^{32}$	Delphi, virtual Pascal
$x_{n+1} = (1103515245x_n + 12345) \mod 2^{32}$	glibc (gcc)

LCG

Problème :

"Never use a generator principally based on a linear congruential generator (LCG) or a multiplicative linear congruential generator (MLCG)".

Numerical Recipes, 3rd edition

"Never use the built-in generators in the C and C++ languages, especially **rand** *and* **srand***. These have no standard implementation and are often badly flawed".*

Numerical Recipes, 3rd edition

Xorshift

- Composé de 3 XORs et 3 shifts $\rightarrow$ très rapide et très léger.
- Période maximale -1.
- Marsaglia, publié en 2003.

Xorshift

XOR ?

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

`unsigned int` x,y,z;
...
$z \leftarrow x \wedge y$;
$\rightarrow$ on applique l'opérateur XOR à chaque bit de x, y et z.

Xorshift

Initialisation :	`unsigned int` $x \neq 0$
Update :	$x \leftarrow x \wedge (x >> a_1)$
	$x \leftarrow x \wedge (x << a_2)$
	$x \leftarrow x \wedge (x >> a_3)$

- période : $2^{64} - 1$ pour les architectures 64 bits
- rapide (le plus rapide va seulement 2,5 fois plus vite)
- pour obtenir un nombre entre 1 et n : `rand()%(n-1)`

Xorshift

Exemple :

$$x = 0001\ 0100 = (20)_{10}$$

$x \wedge = x >> 3$
  $x >> 3 = 0000\ 0010$
  $0001\ 0100 \wedge 0000\ 0010 = 0001\ 0110$
$x \wedge = x << 2$
  $x << 2 = 0101\ 1000$
  $0001\ 0110 \wedge 0101\ 1000 = 0100\ 1110$
$x \wedge = x >> 4$
  $x >> 4 = 0000\ 0100$
  $0100\ 1110 \wedge 0000\ 0100 = 0100\ 1010 = (74)_{10}$

$\rightarrow$ suivant(20) = 74

Xorshift

Extension avec un LCG

```
unsigned long random64(const unsigned long u,v,w){
  u = u * 2862933555777941757LL + 7046029254386353087LL;

  v ^= v >> 17;
  v ^= v << 31;
  v ^= v >> 8;

  w = 4294957665U*(w & 0xffffffff) + (w >> 32);

  unsigned long x = u ^ (u << 21);
  x ^= x >> 35;
  x ^= x << 4;

  return (x + v) ^ w;
}
```

Recommandé par Numerical Recipes, 3rd edition.

Mersenne Twister

- Makoto Matsumoto et Takuji Nishimura, 1998.
  (actualisé récemment)
- Basé sur un **xorshift**.
- Très utilisé.
- Très rapide.
- Période $= 2^{19937} - 1$ (plus de 10^{6001}).
- On trouve une implémentation très facile d'emploi sur le net.

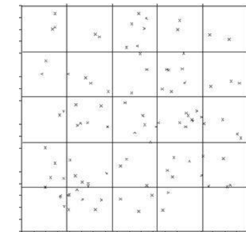
Tester un générateur

Pour savoir si un générateur est bon, on peut :

- Regarder sa période :
  - pas trop petite.
  - trop grande (10^{100}), ça ne sert à rien.
- Vérifier s'il ne satisfait pas une distribution particulière.

Test χ^2

$$\begin{array}{rcl} f : [\![0, m]\!] & \to & [\![0, m]\!] \\ f(x) & = & \text{suivant}(x) \end{array}$$

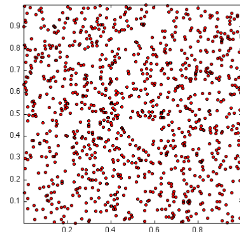


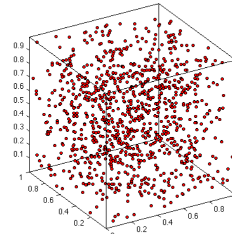
On divise le carré en k carrés égaux et on compte le nombre de points dans chaque carré.

$\to$ la répartition doit être homogène.

Test spectral

$$
\begin{aligned}
f : [\![0, m]\!] &\rightarrow [\![0, m]\!] \\
f(x) &= \text{suivant}(x)
\end{aligned}
$$



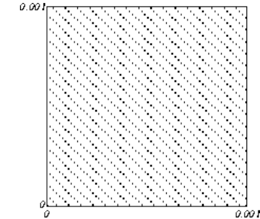


Détermine à quelle densité un k−tuple correspond à un hyperplan à k dimension.

Test spectral

Pour les LCG : 2D

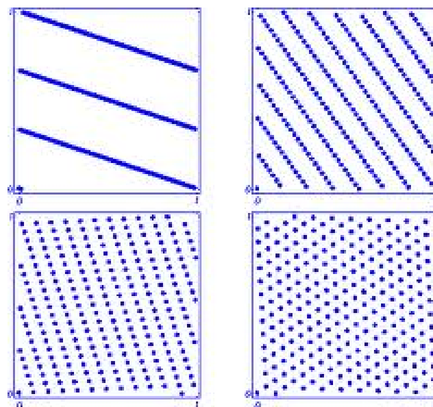
$$
\begin{aligned}
f : [\![0, m]\!] &\rightarrow [\![0, m]\!] \\
f(x) &= \text{suivant}(x)
\end{aligned}
$$



→ On voit des droites

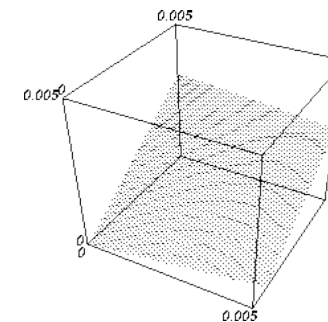
Test spectral

Pour les LCG : 2D



Test spectral

Pour les LCG : 3D



→ On voit des plans

Test Diehard

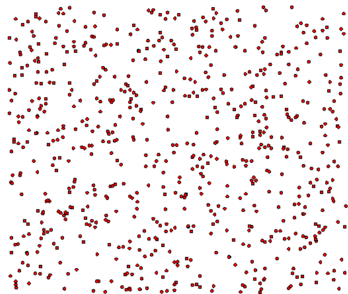
- **Birthday spacings :** sélectionne des points aléatoires sur un grand intervalle. L'espace entre les points doit satisfaire des propriétés statistiques.
- **Permutations :** génère 5 nombres aléatoires consécutifs. Leur ordonnancement (120 possibilités) doit être équiprobable.
- **Rang de matrice :** génère aléatoirement des matrices. Le rang des matrices doit satisfaire des propriétés statistiques.
- **Test du singe :** considère une séquence aléatoire comme des "mots" et compte les recouvrements de mots.

Test Diehard

- **Count the 1s :** converti une séquence aléatoire en "mots" et compte les occurrences du mot "words".
- **Place de parking :** génère aléatoirement des cercles unitaires dans un carré de 100×100. Si un nouveau cercle en recouvre un autre, recommencer. Après 12 000 essais, les tentatives réussies doivent satisfaire une distribution normale.
- **Distance min :** génère 8000 points aléatoires dans un carré 10.000^2. La distance min par paires doit satisfaire des propriétés statistiques.

Distributions

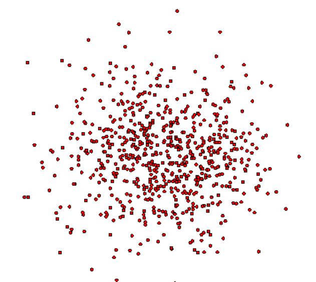
Distribution uniforme :



Résultat par défaut avec les méthodes précédentes.

Distributions

Distribution normale : (gaussienne)



$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

$\rightarrow$ générée à partir d'une distribution uniforme

Distributions

Conversion : distribution uniforme → normale

Algorithm 1: Forme cartésienne

input: moyenne μ et variance σ

repeat

$x =$ random() $\quad x \in [-1, 1]$

$y =$ random() $\quad y \in [-1, 1]$

$s = x^2 + y^2$

until $s \leq 1$

$$u = \mu + x\sigma\sqrt{\frac{-2\ln s}{s}} \quad \text{et} \quad v = \mu + y\sigma\sqrt{\frac{-2\ln s}{s}}$$

Introduction LCG Xorshift Mersenne Twister Tester un générateur Les distributions En C++

Distributions

Conversion : distribution uniforme → normale

Algorithm 2: Forme polaire

input: moyenne μ et variance σ

$x =$ random() $\quad x \in]0, 1]$

$y =$ random() $\quad y \in]0, 1]$

$r = \sigma\sqrt{-2\ln x}$

$\phi = 2\pi y$

$u = \mu + r\cos(\phi)$ et $v = \mu + r\sin(\phi)$

Introduction LCG Xorshift Mersenne Twister Tester un générateur Les distributions En C++

Distributions

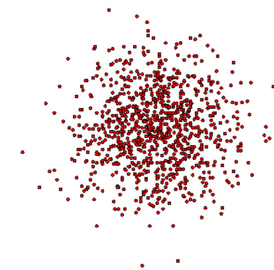
Remarque : Distribution uniforme → normale

La forme cartésienne n'utilise pas de fonctions trigonométriques (assez coûteuses) mais rejette environ $21,46\%$ des candidats issus de la distribution uniforme.

Introduction LCG Xorshift Mersenne Twister Tester un générateur Les distributions En C++

Distributions

Distribution de poisson :



$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!} \qquad \lambda \geq 0$$

Distributions

Conversion : distribution uniforme → poisson

Algorithm 3: Loi de poisson

input: λ

$s = \lambda$
$r = -1$
while $s \geq 0$ **do**
  $r = r + 1$
  x = random() $x \in]0,1]$
  $s = s + \log(x)$
end
return r

En C++ (11) : les générateurs

```
#include <random>

std::default_random_engine myGenerator();
int myValue = myGenerator();
```

- `std::default_random_engine`
  selection de la stl, usage basique
- `std::mt19937_64`
  Mersenne Twister 64 bits
- `std::ranlux24_base`
  subtract-with-carry sur 24-bit
- ...

Les distributions

```
#include <random>

std::mt19937_64 myGenerator();
std::uniform_real_distribution<float> myDistribution(1,6);
float myValue = myDistribution(myGenerator);
```

- `std::uniform_int_distribution<int>(min,max)`
  distribution uniforme entière entre min et max
- `std::uniform_real_distribution<float>(min,max)`
  distribution uniforme flottante entre min et max
- `std::normal_distribution<double>(mean,stdev)`
  distribution normale (gaussienne)
- `std::poisson_distribution<int>`
  distribution de Poisson
- ...

Graines et usages

Graines :

```
#include <random>

unsigned seed = ...;
std::mt19937_64 myGenerator(seed);
```

Distribution compacte :

```
std::mt19937_64 myGenerator();
std::uniform_int_distribution<int> uniformDistribution(1,6);
auto dice = std::bind(uniformDistribution, myGenerator);

int a = dice();
int b = dice();
...
```