La génération de nombres pseudo-aléatoires et ses applications

Pierre L'Ecuyer
Chercheur visiteur à l'IRISA
Chaire de Recherche du Canada en Simulation et Optimisation Stochastique
DIRO, Université de Montréal, Canada

- Les besoins, les applications.
- Bases théoriques et mesures de qualité.
- Exemples: générateurs basés sur des récurrences linéaires, pour la simulation.
- Tests statistiques empiriques.
  Évaluation de générateurs largement utilisés.
- Conclusion.

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

Qu'est-ce qu'on veut?

Produire des suites de nombres qui ont l'air d'être choisis complètement au hasard.
E.g., suites de bits: 011010100110110101001101100101000111...
suite d'entiers de 0 à 100: 31, 83, 02, 72, 54, 26, ...
suites d'entiers de 1 à n, 1 à $n-1$, 1 à $n-2$, ... (permutation aléatoire)
suite de nombres réels entre 0 et 1.

Qu'est-ce qu'on veut?

Produire des suites de nombres qui ont l'air d'être choisis complètement au hasard.
E.g., suites de bits: 0110101001101101010011011001010001111...
suite d'entiers de 0 à 100: 31, 83, 02, 72, 54, 26, ...
suites d'entiers de 1 à n, 1 à $n-1$, 1 à $n-2$, ... (permutation aléatoire)
suite de nombres réels entre 0 et 1.

Mécanismes physiques: lancer les dés, bouliers, roulettes, ...
bruit thermique dans les résistances de circuits electroniques,
capteurs de radiations, autres mécanismes basés sur la physique quantique,
microsecondes de l'horloge de l'ordinateur, ou d'un temps d'accès au disque, etc.

Qu'est-ce qu'on veut?

Produire des suites de nombres qui ont l'air d'être choisis complètement au hasard.
E.g., suites de bits: 011010100110110101001101100101000111...
suite d'entiers de 0 à 100: 31, 83, 02, 72, 54, 26, ...
suites d'entiers de 1 à n, 1 à $n-1$, 1 à $n-2$, ... (permutation aléatoire)
suite de nombres réels entre 0 et 1.

Mécanismes physiques: lancer les dés, bouliers, roulettes, ...
bruit thermique dans les résistances de circuits electroniques,
capteurs de radiations, autres mécanismes basés sur la physique quantique,
microsecondes de l'horloge de l'ordinateur, ou d'un temps d'accès au disque, etc.

Contient de la vraie entropie (incertitude), mais encombrant, pas facilement reproduisible, pas toujours fiable, peu ou pas d'analyse mathématique.

Certains de ces mécanismes sont brevetés.
Plusieurs sont disponibles commercialement.

On peut améliorer la fiabilité en combinant des blocs de bits (XOR).

Générateurs algorithmiques (ou pseudo-aléatoires) (GPA). Une fois les paramètres et l'état initial du GPA choisis, la suite produite est complètement déterministe.

Générateurs algorithmiques (ou pseudo-aléatoires) (GPA). Une fois les paramètres et l'état initial du GPA choisis, la suite produite est complètement déterministe.

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

Générateurs algorithmiques (ou pseudo-aléatoires) (GPA). Une fois les paramètres et l'état initial du GPA choisis, la suite produite est complètement déterministe.

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

Méthodes hybrides: par exemple, prendre des bits dans les mémoires caches des ordinateurs (Seznec et Sendrier, IRISA).

Générateurs algorithmiques (ou pseudo-aléatoires) (GPA). Une fois les paramètres et l'état initial du GPA choisis, la suite produite est complètement déterministe.

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

Méthodes hybrides: par exemple, prendre des bits dans les mémoires caches des ordinateurs (Seznec et Sendrier, IRISA).

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): Utilisé en sciences, gestion, etc.
On simule un modèle mathématique d'un système complexe, pour mieux comprendre le comportement du système, ou optimiser sa gestion, etc.
Souvent, on veut estimer une mesure de performance définie par une espérance mathématique (une intégrale).

Ici, on veut que les propriétés statistiques du modèle mathématique soient bien reproduites par le simulateur.

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

2. Simulation stochastique (Monte Carlo): Utilisé en sciences, gestion, etc.
On simule un modèle mathématique d'un système complexe, pour mieux comprendre le comportement du système, ou optimiser sa gestion, etc.
Souvent, on veut estimer une mesure de performance définie par une espérance mathématique (une intégrale).

Ici, on veut que les propriétés statistiques du modèle mathématique soient bien reproduites par le simulateur.

3. Loteries, machines de casinos, casinos sur Internet, ...
Française des Jeux, Unibet, Loto-Québec, etc.:
Il ne faut pas que quiconque puisse obtenir un avantage pour inférer les prochains numéros ou encore des combinaisons plus probables. Conditions plus exigeantes que pour la simulation.

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

2. Simulation stochastique (Monte Carlo): Utilisé en sciences, gestion, etc.
On simule un modèle mathématique d'un système complexe, pour mieux comprendre le comportement du système, ou optimiser sa gestion, etc.
Souvent, on veut estimer une mesure de performance définie par une espérance mathématique (une intégrale).

Ici, on veut que les propriétés statistiques du modèle mathématique soient bien reproduites par le simulateur.

3. Loteries, machines de casinos, casinos sur Internet, ...
Française des Jeux, Unibet, Loto-Québec, etc.:
Il ne faut pas que quiconque puisse obtenir un avantage pour inférer les prochains numéros ou encore des combinaisons plus probables. Conditions plus exigeantes que pour la simulation.

4. Cryptologie: Encore plus exigeant. L'observation d'une partie de l'output ne doit nous aider d'aucune manière à deviner quoi que ce soit dans le reste.

Besoins pour la Simulation Stochastique

On utilise habituellement un GPA qui imite une suite $U_0, U_1, U_2, \ldots$ de variables aléatoires indépendantes de loi uniforme sur l'intervalle $(0, 1)$.

Besoins pour la Simulation Stochastique

On utilise habituellement un GPA qui imite une suite $U_0, U_1, U_2, \ldots$ de variables aléatoires indépendantes de loi uniforme sur l'intervalle $(0, 1)$.

Pour générer des v.a. selon d'autres lois, on applique des transformations à ces U_j. Par exemple, l'inversion: si $X_j = F^{-1}(U_j)$, alors X_j imite une v.a. de fonction de répartition F.

Besoins pour la Simulation Stochastique

On utilise habituellement un GPA qui imite une suite $U_0, U_1, U_2, \ldots$ de variables aléatoires indépendantes de loi uniforme sur l'intervalle $(0, 1)$.

Pour générer des v.a. selon d'autres lois, on applique des transformations à ces U_j. Par exemple, l'inversion: si $X_j = F^{-1}(U_j)$, alors X_j imite une v.a. de fonction de répartition F.

Comparaison de systèmes semblables avec valeurs aléatoires communes.

On simule un réseau de communication, ou un centre d'appels téléphoniques, ou un réseau de distribution de biens, ou une usine, ou le trafic automobile dans une ville, ou la gestion dynamique d'un portefeuille d'investissements (finance), etc.

On veut comparer deux configurations (ou politiques de gestion) semblables du système.

Une partie de la différence de performance sera due à la différence de configuration, et une autre partie sera due au bruit stochastique. On veut minimiser cette seconde partie.

Idée de base: simuler les deux configurations avec les mêmes valeurs uniformes U_j, utilisées exactement aux mêmes endroits.
On dispose de nombreux résultats théoriques sur l'amélioration d'efficacité (réduction de variance) que cela apporte.

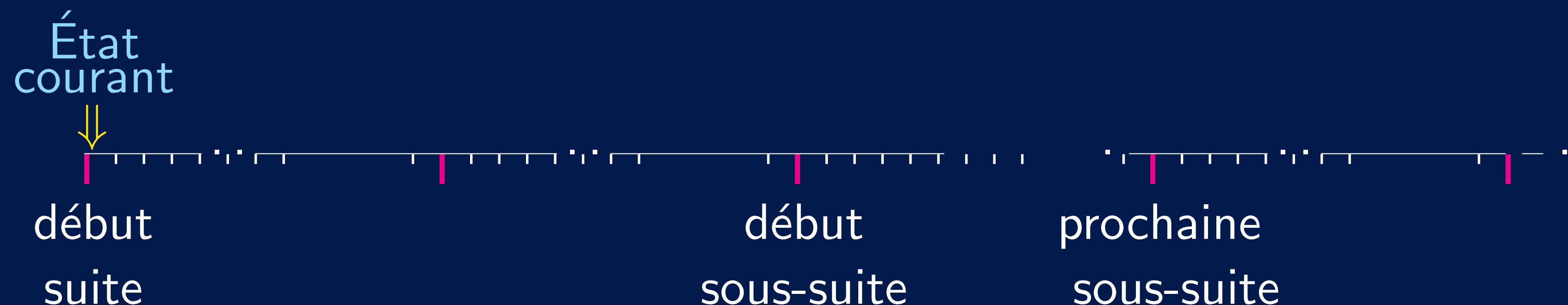
Mais l'implantation, avec synchronisation des v.a., peut être compliquée lorsque les deux configurations n'utilisent pas le même nombre de U_j (e.g., parfois on doit générer une v.a. dans un cas et pas dans l'autre).

Idée de base: simuler les deux configurations avec les mêmes valeurs uniformes U_j, utilisées exactement aux mêmes endroits.
On dispose de nombreux résultats théoriques sur l'amélioration d'efficacité (réduction de variance) que cela apporte.

Mais l'implantation, avec synchronisation des v.a., peut être compliquée lorsque les deux configurations n'utilisent pas le même nombre de U_j (e.g., parfois on doit générer une v.a. dans un cas et pas dans l'autre).

Une solution: GPA avec suites et sous-suites multiples.

Chaque suite peut être vue comme un GPA virtuel.
Elle est partitionnée en sous-suites.
On peut créer autant de suites (distinctes et "indépendantes") que l'on veut.

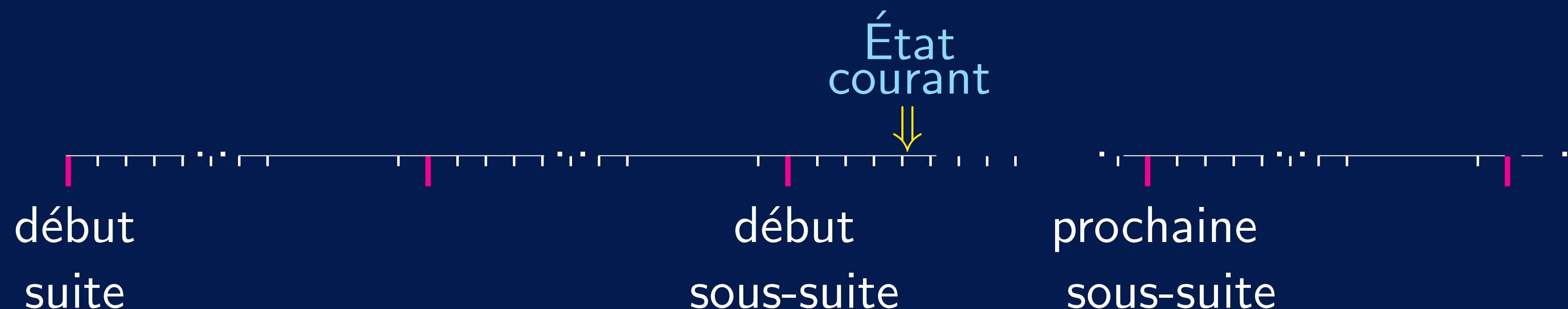


Idée de base: simuler les deux configurations avec les mêmes valeurs uniformes U_j, utilisées exactement aux mêmes endroits.
On dispose de nombreux résultats théoriques sur l'amélioration d'efficacité (réduction de variance) que cela apporte.

Mais l'implantation, avec synchronisation des v.a., peut être compliquée lorsque les deux configurations n'utilisent pas le même nombre de U_j (e.g., parfois on doit générer une v.a. dans un cas et pas dans l'autre).

Une solution: GPA avec suites et sous-suites multiples.

Chaque suite peut être vue comme un GPA virtuel.
Elle est partitionnée en sous-suites.
On peut créer autant de suites (distinctes et "indépendantes") que l'on veut.

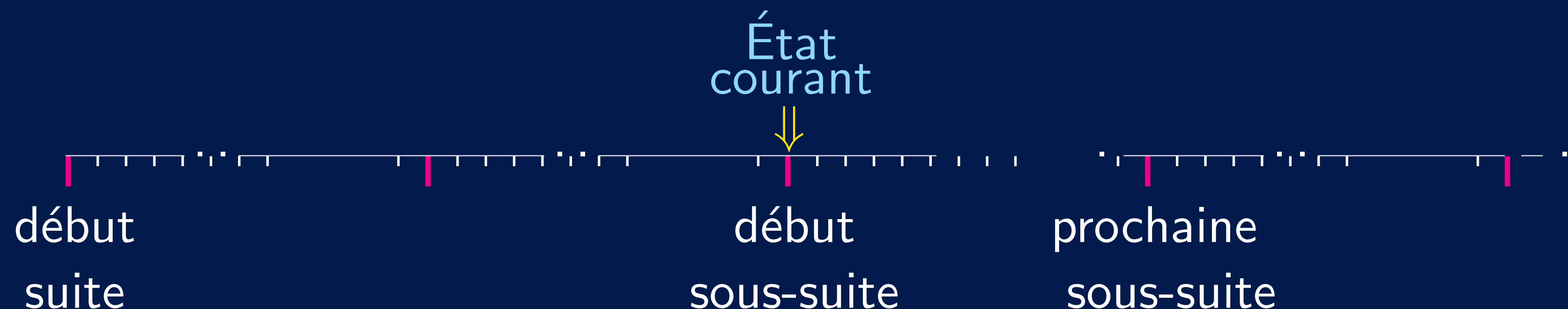


Idée de base: simuler les deux configurations avec les mêmes valeurs uniformes U_j, utilisées exactement aux mêmes endroits.
On dispose de nombreux résultats théoriques sur l'amélioration d'efficacité (réduction de variance) que cela apporte.

Mais l'implantation, avec synchronisation des v.a., peut être compliquée lorsque les deux configurations n'utilisent pas le même nombre de U_j (e.g., parfois on doit générer une v.a. dans un cas et pas dans l'autre).

Une solution: GPA avec suites et sous-suites multiples.

Chaque suite peut être vue comme un GPA virtuel.
Elle est partitionnée en sous-suites.
On peut créer autant de suites (distinctes et "indépendantes") que l'on veut.

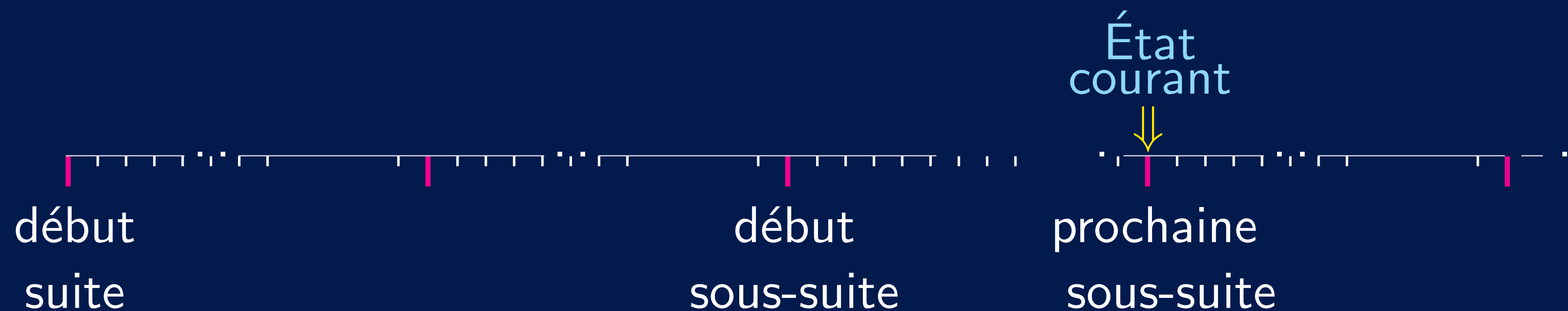


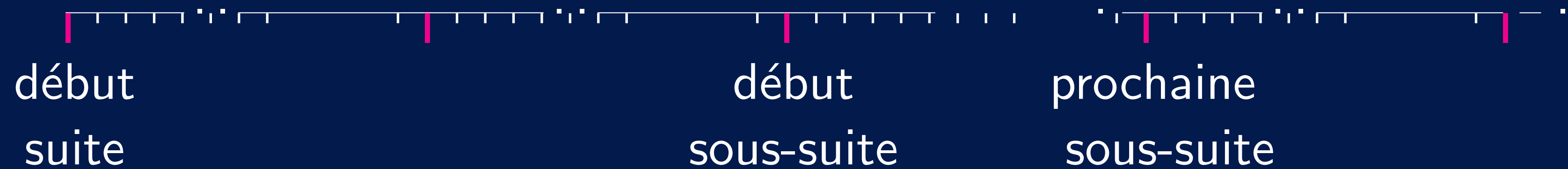
Idée de base: simuler les deux configurations avec les mêmes valeurs uniformes U_j, utilisées exactement aux mêmes endroits.
On dispose de nombreux résultats théoriques sur l'amélioration d'efficacité (réduction de variance) que cela apporte.

Mais l'implantation, avec synchronisation des v.a., peut être compliquée lorsque les deux configurations n'utilisent pas le même nombre de U_j (e.g., parfois on doit générer une v.a. dans un cas et pas dans l'autre).

Une solution: GPA avec suites et sous-suites multiples.

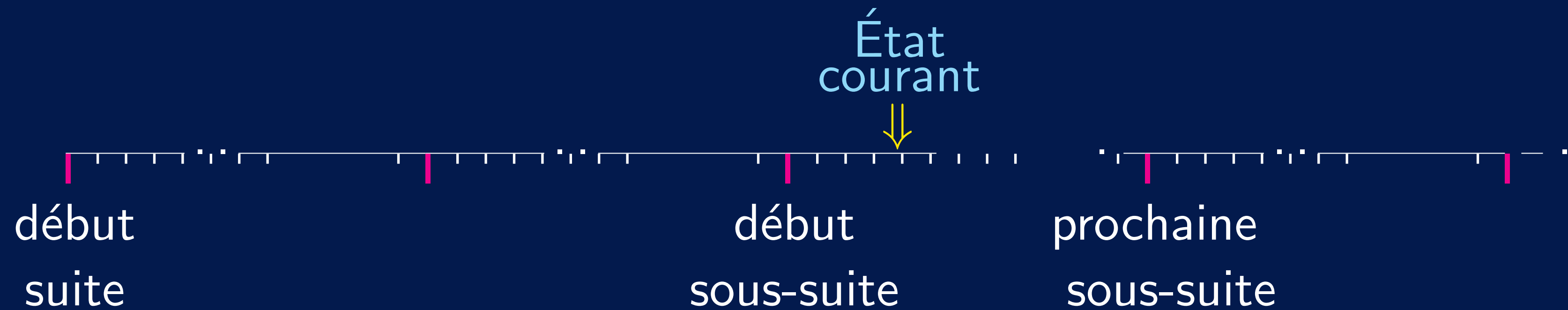
Chaque suite peut être vue comme un GPA virtuel.
Elle est partitionnée en sous-suites.
On peut créer autant de suites (distinctes et "indépendantes") que l'on veut.





Supposons que pour une suite de clients, on doit générer:
(1) un instant d'arrivée pour chaque client,
(2) une durée de service au serveur A pour chaque client
(3) une durée de service au serveur B pour des groupes de 10 à 20 clients.

Alors on peut utiliser une suite pour chacun de ces trois usages.



Supposons que pour une suite de clients, on doit générer:
(1) un instant d'arrivée pour chaque client,
(2) une durée de service au serveur A pour chaque client
(3) une durée de service au serveur B pour des groupes de 10 à 20 clients.

Alors on peut utiliser une suite pour chacun de ces trois usages.

Exemple d'interface Java (dans SSJ)

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

Réinitialise la suite à son état initial.

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

Réinitialise la suite au début de sa sous-suite courante. substream.

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

Réinitialise la suite au début de sa prochaine sous-suite. substream.

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

Retourne une v.a. $U(0,1)$ de cette suite et avance d'un pas.

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

Retourne une v.a. uniforme sur $\{i, i+1, \ldots, j\}$.

```
}
```

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

Une implantation particulière, basée sur un générateur de période $\approx 2^{181}$, partitionnée en suites disjointes de longueur 2^{127}, et sous-suites de longueur 2^{76}.

```
public MRG32k3a();
```

Construit une nouvelle suite.

```
}
```

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

Une implantation basée sur uns combainaison de “LFSR”, de période $\approx 2^{113}$.

```
public LFSR113();
```

Construit une nouvelle suite.

```
}
```

Ce système, basé sur MRG32k3a, a été récemment adopté dans plusieurs logiciels de simulation et de statistique, tels que SAS, Arena, Witness, Simul8, Automod, ns2, ...

Génération de valeurs non-uniformes

Lois continues ou discrètes. Méthode par défaut: inversion.

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

Crée un générateur pour la loi `dist`, avec la suite `s`.

```
   public double nextDouble()
```

Génère une nouvelle valeur.

Génération de valeurs non-uniformes

Lois continues ou discrètes. Méthode par défaut: inversion.

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

Crée un générateur pour la loi `dist`, avec la suite `s`.

```
   public double nextDouble()
```

Génère une nouvelle valeur.

Il y a aussi des générateurs spécialisés pour plusieurs distributions.

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

Crée un générateur de v.a. normales.

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

Génère une nouvelle v.a. normale, en utilisant la suite `s`.

Exemple de comparaison de deux configuration:

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

Exemple de comparaison de deux configuration:

```
        ⋮
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 ---
    }
```

Les suites multiples sont très utiles, par exemple pour:

- comparer des systèmes semblables
- analyse de sensibilité, estimation de dérivées (ou gradient) par différences finies
- optimisation d'une fonction empirique, ou optimisation en général
- utilisation d'un modèle similaire simplifié comme variable de contrôle externe

Les suites multiples sont très utiles, par exemple pour:

- comparer des systèmes semblables
- analyse de sensibilité, estimation de dérivées (ou gradient) par différences finies
- optimisation d'une fonction empirique, ou optimisation en général
- utilisation d'un modèle similaire simplifié comme variable de contrôle externe

Le `RandomStream` peut aussi représenter un ensemble de points à faible discrépance, pour quasi-Monte Carlo. C'est le cas dans SSJ.

Les suites multiples sont très utiles, par exemple pour:

- comparer des systèmes semblables
- analyse de sensibilité, estimation de dérivées (ou gradient) par différences finies
- optimisation d'une fonction empirique, ou optimisation en général
- utilisation d'un modèle similaire simplifié comme variable de contrôle externe

Le `RandomStream` peut aussi représenter un ensemble de points à faible discrépance, pour quasi-Monte Carlo. C'est le cas dans SSJ.

Questions?

Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini;
$f : \mathcal{S} \rightarrow \mathcal{S}$, fonction de transition;

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

Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);

$f : \mathcal{S} \to \mathcal{S}$, fonction de transition; $s_n = f(s_{n-1})$

$\mathcal{U}$, espace des valeurs de sortie;

$g : \mathcal{S} \to \mathcal{U}$, fonction de sortie. $u_n = g(s_n)$

Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini;
$f : \mathcal{S} \rightarrow \mathcal{S}$, fonction de transition;
$\mathcal{U}$, espace des valeurs de sortie;
$g : \mathcal{S} \rightarrow \mathcal{U}$, fonction de sortie.

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

$u_n = g(s_n)$

s_0

Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini;

$f : \mathcal{S} \to \mathcal{S}$, fonction de transition;

$\mathcal{U}$, espace des valeurs de sortie;

$g : \mathcal{S} \to \mathcal{U}$, fonction de sortie.

s_0, germe (état initial);

$s_n = f(s_{n-1})$

$u_n = g(s_n)$

$$s_0 \xrightarrow{g} u_0$$

Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini;

$f : \mathcal{S} \rightarrow \mathcal{S}$, fonction de transition;

$\mathcal{U}$, espace des valeurs de sortie;

$g : \mathcal{S} \rightarrow \mathcal{U}$, fonction de sortie.

s_0, germe (état initial);

$s_n = f(s_{n-1})$

$u_n = g(s_n)$

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

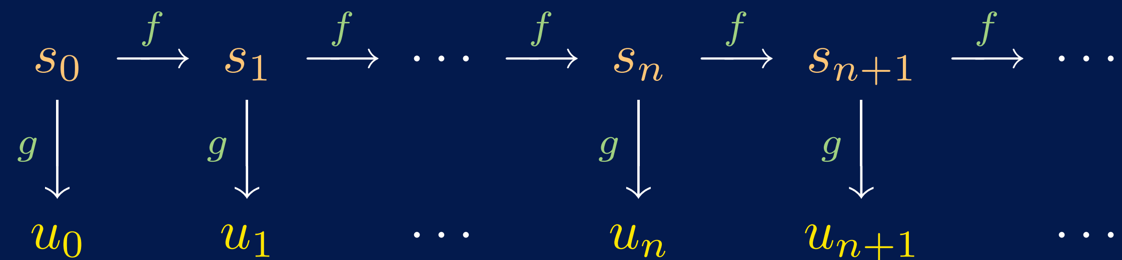
Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition; $s_n = f(s_{n-1})$
$\mathcal{U}$, espace des valeurs de sortie;
$g : \mathcal{S} \to \mathcal{U}$, fonction de sortie. $u_n = g(s_n)$

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

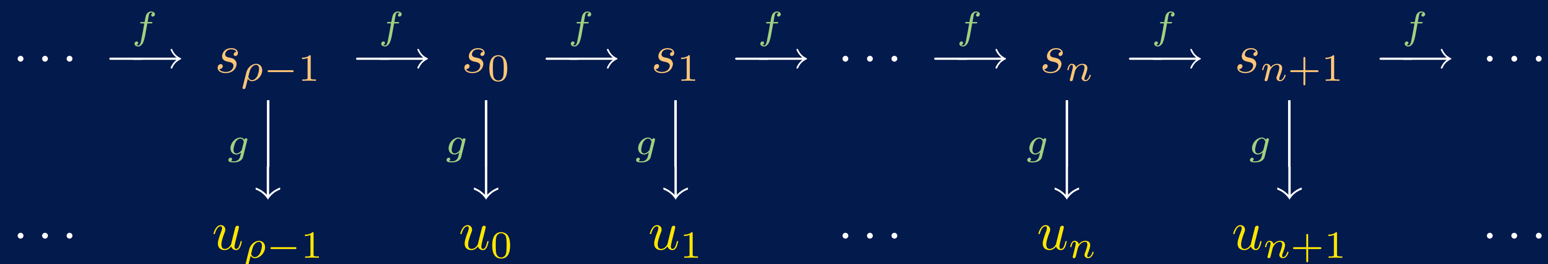
Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition; $s_n = f(s_{n-1})$
$\mathcal{U}$, espace des valeurs de sortie;
$g : \mathcal{S} \to \mathcal{U}$, fonction de sortie. $u_n = g(s_n)$



Définition et conception d'un GPA

$\mathcal{S}$, espace d'états fini; s_0, germe (état initial);
$f : \mathcal{S} \to \mathcal{S}$, fonction de transition; $s_n = f(s_{n-1})$
$\mathcal{U}$, espace des valeurs de sortie;
$g : \mathcal{S} \to \mathcal{U}$, fonction de sortie. $u_n = g(s_n)$

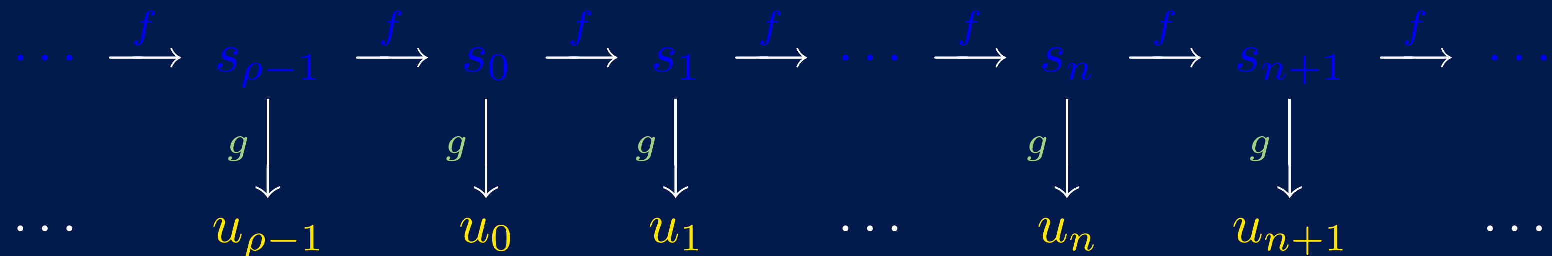


Période: $\rho \leq |\mathcal{S}|$. $s_{i+\rho} = s_i \quad \forall i \geq \tau$. On supposera $\tau = 0$ et $\mathcal{U} = [0, 1]$.

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

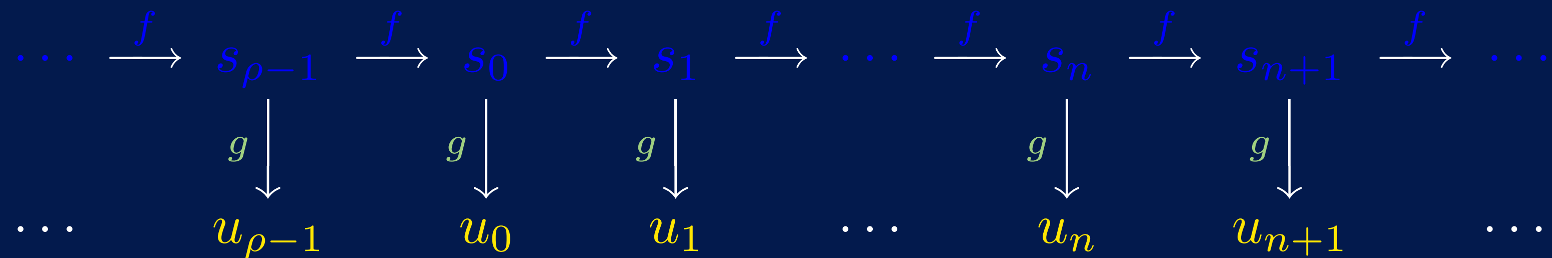
$$\begin{array}{cccccccc} & \downarrow g & \downarrow g & \downarrow g & & \downarrow g & \downarrow g & \\ \cdots & u_{\rho-1} & u_0 & u_1 & \cdots & u_n & u_{n+1} & \cdots \end{array}$$

Objectif: en observant seulement $(u_0, u_1, \ldots)$, il doit être difficile de distinguer cette suite de la réalisation d'une suite de v.a. i.i.d. uniformes sur $\mathcal{U}$.



Objectif: en observant seulement $(u_0, u_1, \ldots)$, il doit être difficile de distinguer cette suite de la réalisation d'une suite de v.a. i.i.d. uniformes sur $\mathcal{U}$.

Utopie: ne pas pouvoir distinguer mieux qu'en tirant à pile ou face.
Autrement dit, que la suite passe tous les tests statistiques imaginables.
Cela est impossible! On y reviendra plus loin.

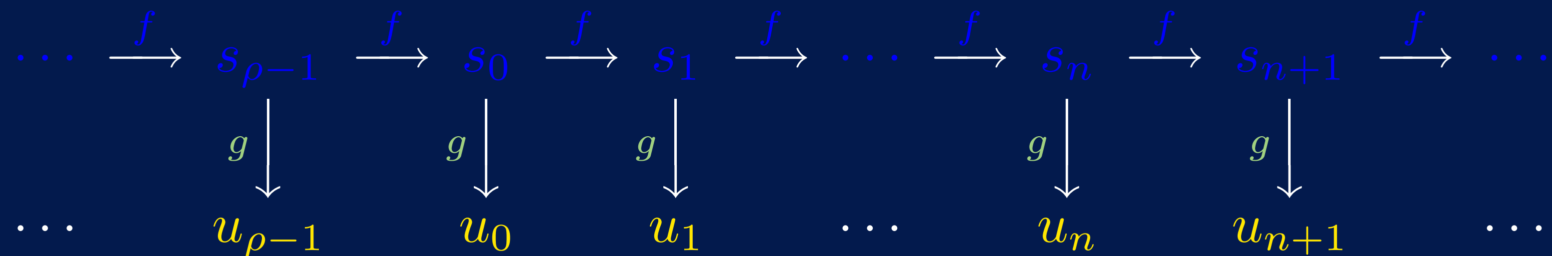


Objectif: en observant seulement $(u_0, u_1, \ldots)$, il doit être difficile de distinguer cette suite de la réalisation d'une suite de v.a. i.i.d. uniformes sur $\mathcal{U}$.

Utopie: ne pas pouvoir distinguer mieux qu'en tirant à pile ou face.
Autrement dit, que la suite passe tous les tests statistiques imaginables.
Cela est impossible! On y reviendra plus loin.

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

Compromis entre vitesse / bonnes propriétés statistiques / (im)prévisibilité.



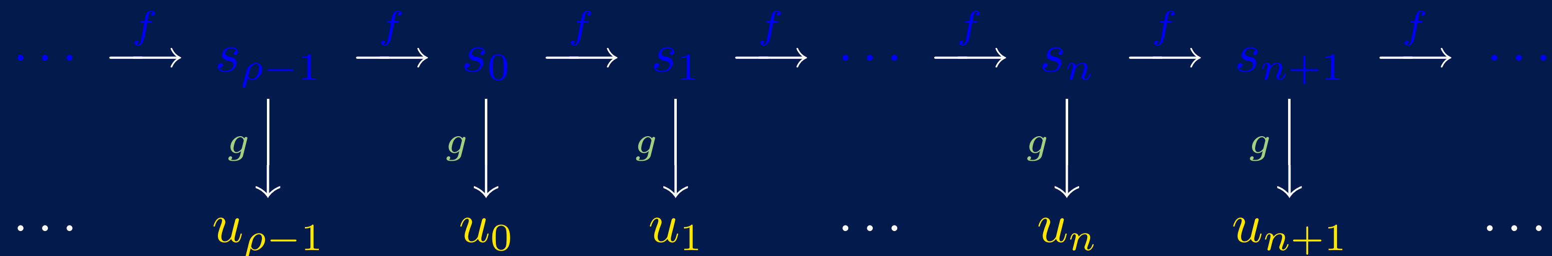
Objectif: en observant seulement $(u_0, u_1, \ldots)$, il doit être difficile de distinguer cette suite de la réalisation d'une suite de v.a. i.i.d. uniformes sur $\mathcal{U}$.

Utopie: ne pas pouvoir distinguer mieux qu'en tirant à pile ou face.
Autrement dit, que la suite passe tous les tests statistiques imaginables.
Cela est impossible! On y reviendra plus loin.

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

Compromis entre vitesse / bonnes propriétés statistiques / (im)prévisibilité.

Si l'état initial s_0 est choisi au hasard, le GPA est comme une roulette géante:
Pour générer t nombres aléatoires, on tourne la roulette pour choisir s_0, puis on retient $\mathbf{u} = (u_0, \ldots, u_{t-1})$.



Objectif: en observant seulement $(u_0, u_1, \ldots)$, il doit être difficile de distinguer cette suite de la réalisation d'une suite de v.a. i.i.d. uniformes sur $\mathcal{U}$.

Utopie: ne pas pouvoir distinguer mieux qu'en tirant à pile ou face.
Autrement dit, que la suite passe tous les tests statistiques imaginables.
Cela est impossible! On y reviendra plus loin.

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

Compromis entre vitesse / bonnes propriétés statistiques / (im)prévisibilité.

Si l'état initial s_0 est choisi au hasard, le GPA est comme une roulette géante:
Pour générer t nombres aléatoires, on tourne la roulette pour choisir s_0, puis on retient $\mathbf{u} = (u_0, \ldots, u_{t-1})$.

Machines de casinos et loteries: on réinitialise s_0 très souvent.

La loi uniforme sur $[0,1]^t$.
Choisir s_0 au hasard correspond à choisir un point au hasard dans l'espace échantillonnal

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

qui peut être interprété comme une approximation de $[0,1]^t$.

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

Choisir s_0 au hasard correspond à choisir un point au hasard dans l'espace échantillonnal

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

qui peut être interprété comme une approximation de $[0,1]^t$.

Critère: Ψ_t doit recouvrir $[0,1]^t$ très uniformément pour t jusqu'à (disons) t_0.

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

Choisir s_0 au hasard correspond à choisir un point au hasard dans l'espace échantillonnal

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

qui peut être interprété comme une approximation de $[0,1]^t$.

Critère: Ψ_t doit recouvrir $[0,1]^t$ très uniformément pour t jusqu'à (disons) t_0.

Il nous faut une mesure d'uniformité de Ψ_t (ou une mesure de discrépance entre la loi empirique de Ψ_t et la loi uniforme). Plusieurs définitions possibles.

Important: doit être facilement calculable sans générer les points.
Pour cela, il faut bien comprendre la structure mathématique de Ψ_t.
Pour cette raison, la plupart des GPA utilisés en simulation sont basés sur des récurrences linéaires.

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

Choisir s_0 au hasard correspond à choisir un point au hasard dans l'espace échantillonnal

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

qui peut être interprété comme une approximation de $[0,1]^t$.

Critère: Ψ_t doit recouvrir $[0,1]^t$ très uniformément pour t jusqu'à (disons) t_0.

Il nous faut une mesure d'uniformité de Ψ_t (ou une mesure de discrépance entre la loi empirique de Ψ_t et la loi uniforme). Plusieurs définitions possibles.

Important: doit être facilement calculable sans générer les points.
Pour cela, il faut bien comprendre la structure mathématique de Ψ_t.
Pour cette raison, la plupart des GPA utilisés en simulation sont basés sur des récurrences linéaires.

Pourquoi ne pas insister que Ψ_t lui-même ressemble à un ensemble de points choisis au hasard (e.g., ne soit pas trop uniforme)?
En fait, on veut cela seulement pour la fraction infime de Ψ_t que l'on utilise.

Généralisation: mesurer l'uniformité de $\Psi_I = \{(u_{i_1}, \ldots, u_{i_t}) \mid s_0 \in \mathcal{S}\}$ pour une classe choisie d'ensembles d'indices (non successifs) de forme $I = \{i_1, i_2, \cdots, i_t\}$.
On veut s'assurer que Ψ_I est bien uniforme pour tout $I \in \mathcal{J}$, pour une famille donnée $\mathcal{J}$.

Générateur linéaire récursif multiple (MRG)

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

En pratique, on prendra plutôt $u_n = (x_n + 1)/(m + 1)$,
ou encore $u_n = x_n/(m + 1)$ si $x_n > 0$ et $u_n = m/(m + 1)$ sinon.

Mais la structure reste essentiellement la même.

Générateur linéaire récursif multiple (MRG)

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

En pratique, on prendra plutôt $u_n = (x_n + 1)/(m + 1)$,
ou encore $u_n = x_n/(m + 1)$ si $x_n > 0$ et $u_n = m/(m + 1)$ sinon.

Mais la structure reste essentiellement la même.

Nombreuses variantes et implantations. Très répandu.

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

Générateur linéaire récursif multiple (MRG)

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

En pratique, on prendra plutôt $u_n = (x_n + 1)/(m + 1)$,
ou encore $u_n = x_n/(m + 1)$ si $x_n > 0$ et $u_n = m/(m + 1)$ sinon.

Mais la structure reste essentiellement la même.

Nombreuses variantes et implantations. Très répandu.

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

État à l'étape n: $s_n = \mathbf{x}_n = (x_{n-k+1}, \ldots, x_n)^{\mathsf{t}}$.
Espace d'états: $\mathbb{Z}_m^k$, de cardinalité m^k.

Générateur linéaire récursif multiple (MRG)

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

En pratique, on prendra plutôt $u_n = (x_n + 1)/(m + 1)$,
ou encore $u_n = x_n/(m + 1)$ si $x_n > 0$ et $u_n = m/(m + 1)$ sinon.

Mais la structure reste essentiellement la même.

Nombreuses variantes et implantations. Très répandu.

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

État à l'étape n: $s_n = \mathbf{x}_n = (x_{n-k+1}, \ldots, x_n)^{\mathsf{t}}$.
Espace d'états: $\mathbb{Z}_m^k$, de cardinalité m^k.

Période max. $\rho = m^k - 1$, pour m premier.

Polynôme caractéristique (avec $a_0 = -1$):

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k = -\sum_{j=0}^{k} a_j z^{k-j}.$$

Polynôme caractéristique (avec $a_0 = -1$):

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k = -\sum_{j=0}^{k} a_j z^{k-j}.$$

On peut représenter cette récurrence dans les trois espaces suivants:

(i) l'espace $\mathbb{Z}_m^k$ des vecteurs ayant k coordonnées dans $\mathbb{Z}_m$,

(ii) l'espace $\mathbb{Z}_m[z]/(P)$ des polynômes de degré $< k$ à coefficients dans $\mathbb{Z}_m$ (i.e., les polynômes réduits modulo $P(z)$),

(iii) l'espace $\mathcal{L}(P)$ des séries formelles de Laurent de la forme $\tilde{s}(z) = \sum_{j=1}^{\infty} c_j z^{-j}$, où les coefficients c_j sont dans $\mathbb{Z}_m$ et satisfont $c_j = (a_1 c_{j-1} + \cdots + a_k c_{j-k}) \bmod m$ pour tout $j > k$.

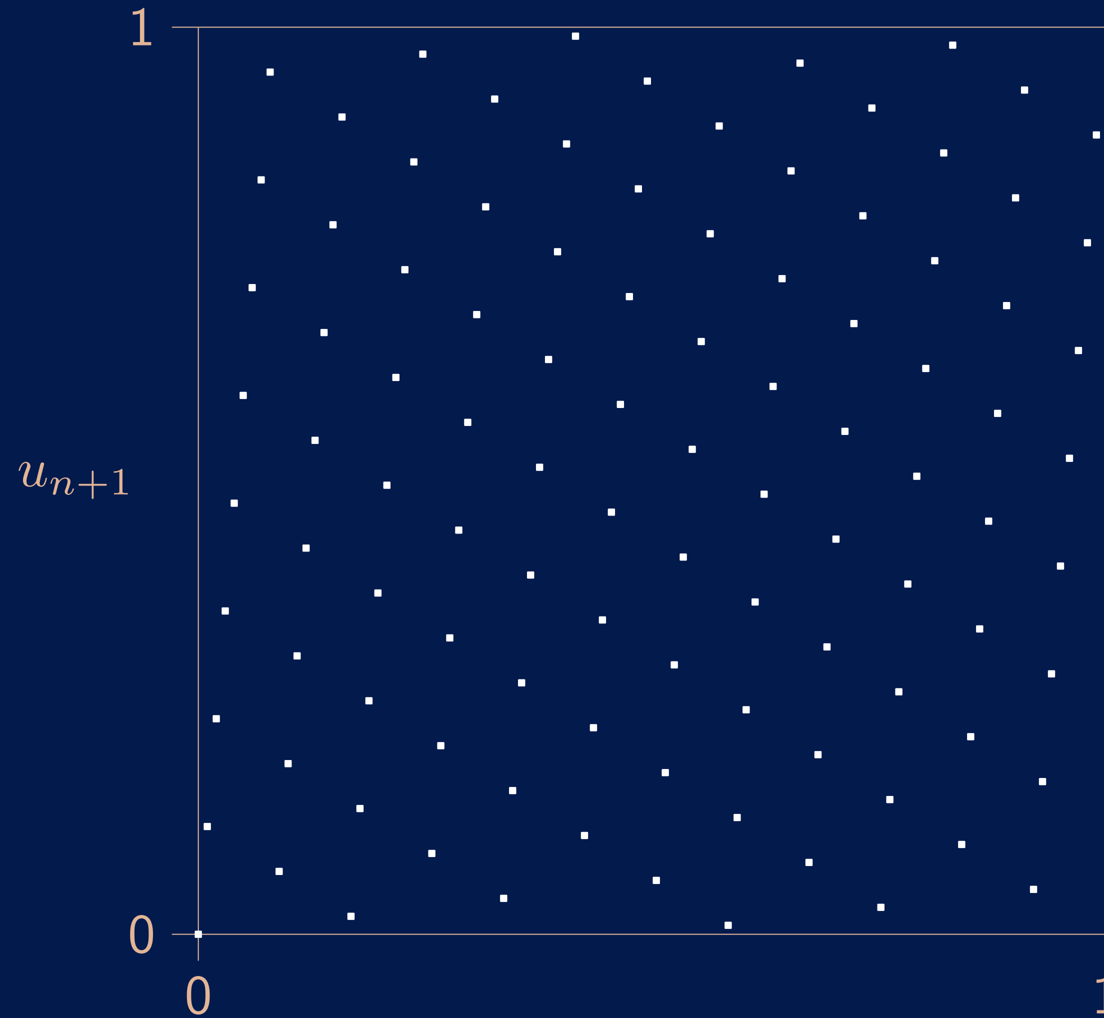
Il y a des bijections faciles à calculer entre les trois.
Utile pour étudier la théorie, e.g., vérifier la période.

Structure de Ψ_t: $(x_0, \ldots, x_{k-1})$ peut prendre n'importe quelle valeur dans $\{0, 1, \ldots, m-1\}^k$; ensuite $x_k, x_{k+1}, \ldots$ sont déterminés par la récurrence. Ainsi, $(x_0, \ldots, x_{k-1}) \mapsto (x_0, \ldots, x_{k-1}, x_k, \ldots, x_{t-1})$ est une application linéaire.

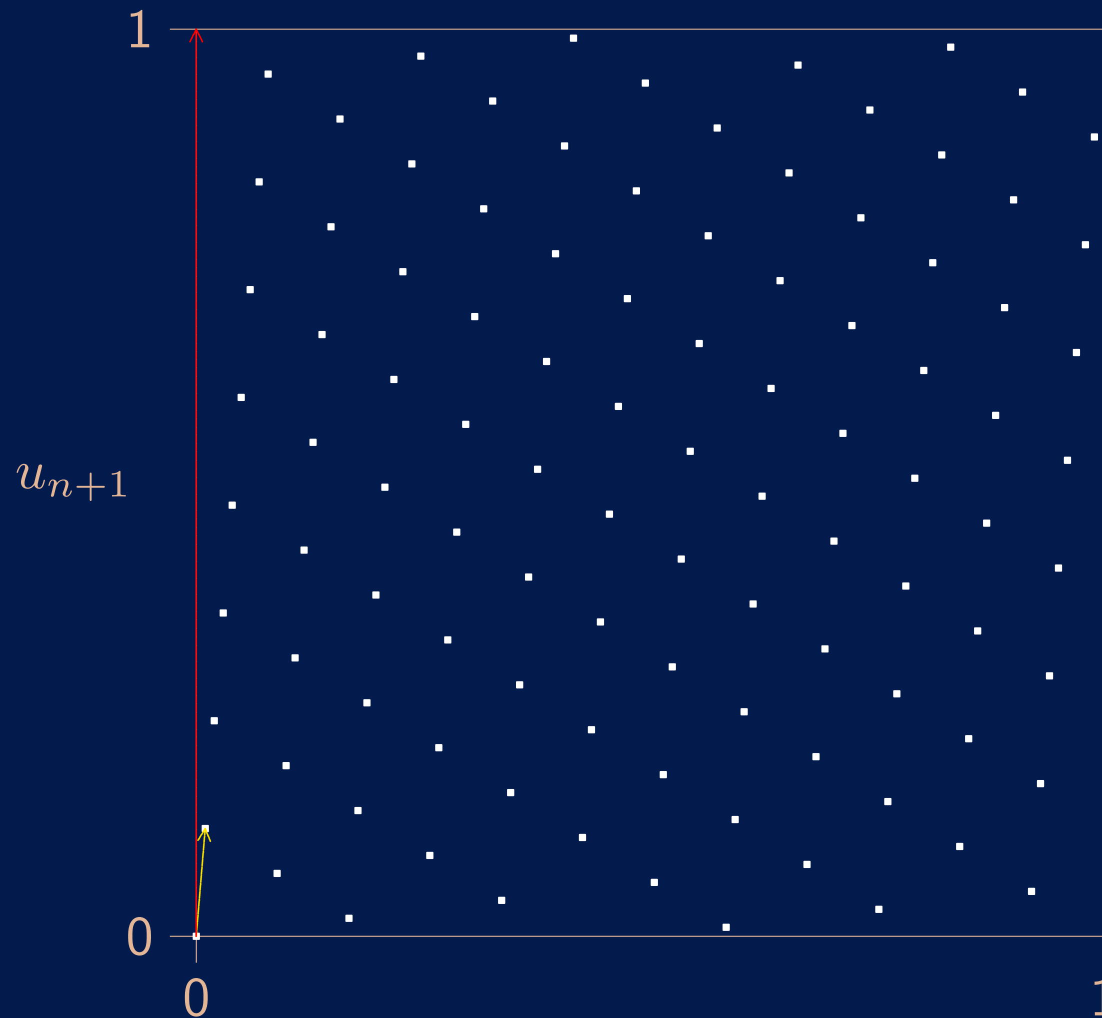
On peut en déduire que $\Psi_t = L_t \cap [0, 1)^t$ où

$$L_t = \left\{ \mathbf{v} = \sum_{i=1}^{t} z_i \mathbf{v}_i \;\middle|\; z_i \in \mathbb{Z} \right\},$$ un réseau (“lattice”) dans $\mathbb{R}^t$, avec

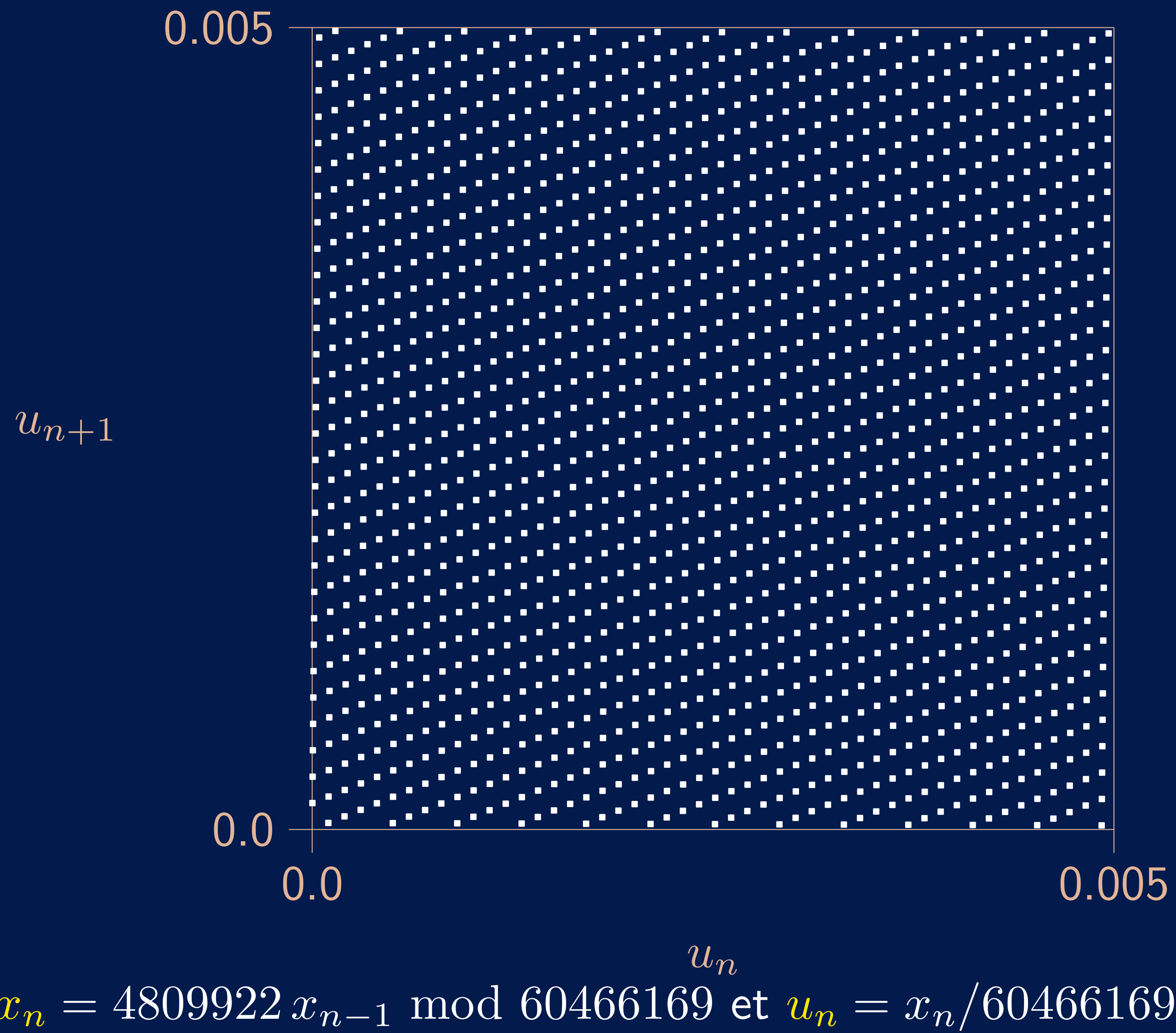
$$\begin{aligned}
\mathbf{v}_1 &= (1, 0, \ldots, 0, x_{1,k}, \ldots, x_{1,t-1})/m \\
\mathbf{v}_2 &= (0, 1, \ldots, 0, x_{2,k}, \ldots, x_{2,t-1})/m \\
\vdots \quad & \quad \vdots \\
\mathbf{v}_k &= (0, 0, \ldots, 1, x_{k,k}, \ldots, x_{k,t-1})/m \\
\mathbf{v}_{k+1} &= (0, 0, \ldots, 0, 1, \ldots, 0)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



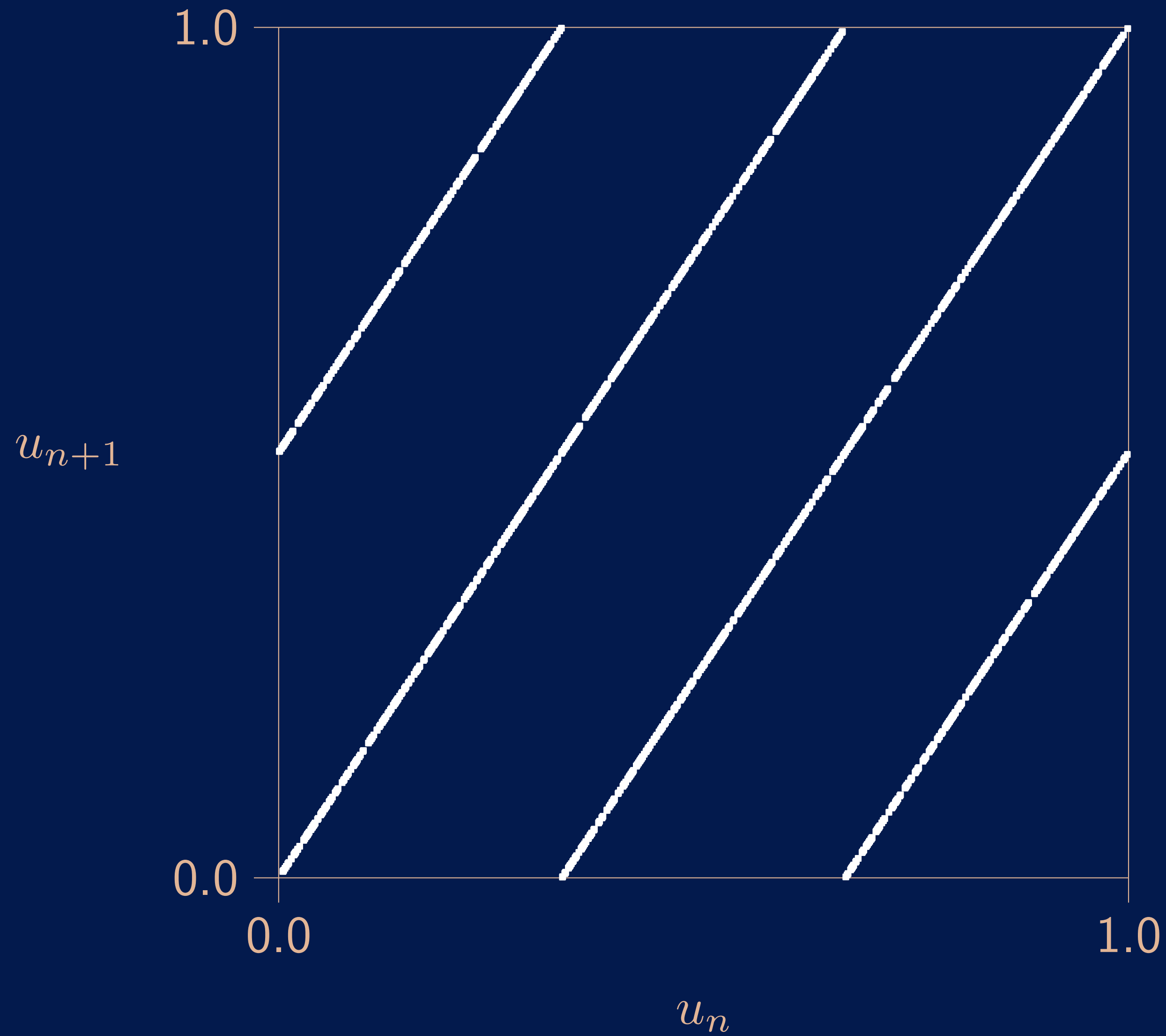
GCL ($k = 1$), $m = 101$, $a_1 = 12$, $t = 2$;



GCL ($k=1$), $m=101$, $a_1=12$, $t=2$; $\mathbf{v}_1=(1,12)/m$, $\mathbf{v}_2=(0,1)$



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



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

On peut mesurer l'uniformité en termes de cette structure de réseau, en plusieurs dimensions.

On peut mesurer l'uniformité en termes de cette structure de réseau, en plusieurs dimensions.

On peut aussi prendre en compte les projections sur des sous-ensembles de coordonnées $I = \{i_1, i_2, \cdots, i_t\}$:

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

On effectue des recherches par ordinateur pour trouver des bons paramètres par rapport à un critère qui prend compte de l'uniformité d'un ensemble choisi de projections.

On peut mesurer l'uniformité en termes de cette structure de réseau, en plusieurs dimensions.

On peut aussi prendre en compte les projections sur des sous-ensembles de coordonnées $I = \{i_1, i_2, \cdots, i_t\}$:

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

On effectue des recherches par ordinateur pour trouver des bons paramètres par rapport à un critère qui prend compte de l'uniformité d'un ensemble choisi de projections.

Remarque: on peut montrer que si $1 + a_{i_2}^2 + \cdots + a_{i_t}^2$ est petit, alors l'uniformité de Ψ_I est nécessairement mauvaise.

Implantations efficaces

Il faut calculer $ax \bmod m$ pour de grands m.

Implantations efficaces

Il faut calculer $ax \bmod m$ pour de grands m.

Factorisation approximative.
Valide si $(a^2 < m)$ ou $(a = \lfloor m/i \rfloor$ où $i^2 < m)$. Calculs sur des entiers.
Précalculer $q = \lfloor m/a \rfloor$ et $r = m \bmod a$. Ensuite,

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

Toutes les quantités intermédiaires demeurent entre $-m$ et m.

Implantations efficaces

Il faut calculer $ax \bmod m$ pour de grands m.

Factorisation approximative.
Valide si $(a^2 < m)$ ou $(a = \lfloor m/i \rfloor$ où $i^2 < m)$. Calculs sur des entiers.
Précalculer $q = \lfloor m/a \rfloor$ et $r = m \bmod a$. Ensuite,

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

Toutes les quantités intermédiaires demeurent entre $-m$ et m.

Calculs en point flottant, double précision.
Valide si $am < 2^{53}$.

$$\text{double } m, a, x, y; \qquad \text{int } k;$$

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

Décomposition en puissances de 2.

Supposons que $a = \pm 2^q \pm 2^r$ et $m = 2^e - h$ pour h petit.
(Wu 1997 pour $h = 1$; L'Ecuyer et Simard 1999 pour $h > 1$.)

Pour calculer $y = 2^q x \bmod m$, décomposer $x = x_0 + 2^{e-q} x_1$;

	q bits	$(e-q)$ bits
$x =$	x_1	x_0

Pour $h = 1$, on obtient y en permutant x_0 et x_1.

Décomposition en puissances de 2.

Supposons que $a = \pm 2^q \pm 2^r$ et $m = 2^e - h$ pour h petit.
(Wu 1997 pour $h = 1$; L'Ecuyer et Simard 1999 pour $h > 1$.)

Pour calculer $y = 2^q x \bmod m$, décomposer $x = x_0 + 2^{e-q} x_1$;

	q bits	$(e-q)$ bits
$x =$	x_1	x_0

Pour $h = 1$, on obtient y en permutant x_0 et x_1.

Pour $h > 1$,

$$y = 2^q (x_0 + 2^{e-q} x_1) \bmod (2^e - h) = (2^q x_0 + h x_1) \bmod (2^e - h).$$

Décomposition en puissances de 2.

Supposons que $a = \pm 2^q \pm 2^r$ et $m = 2^e - h$ pour h petit.
(Wu 1997 pour $h = 1$; L'Ecuyer et Simard 1999 pour $h > 1$.)

Pour calculer $y = 2^q x$ mod m, décomposer $x = x_0 + 2^{e-q} x_1$;

	q bits	$(e-q)$ bits
$x =$	x_1	x_0

Pour $h = 1$, on obtient y en permutant x_0 et x_1.

Pour $h > 1$,

$$y = 2^q(x_0 + 2^{e-q}x_1) \bmod (2^e - h) = (2^q x_0 + h x_1) \bmod (2^e - h).$$

Si $h < 2^q$ et $h(2^q - (h+1)2^{-e+q}) < m$, chaque terme est $< m$. Modulo: soustraire m si la somme est $\geq m$.

Coefficients égaux.

Par exemple, prendre tous les a_j non nuls égaux à a (Deng et Xu 2002).
Alors, $x_n = a(x_{n-i_1} + \cdots + x_{n-k}) \bmod m$. Une seule multiplication.

Coefficients égaux.
Par exemple, prendre tous les a_j non nuls égaux à a (Deng et Xu 2002).
Alors, $x_n = a(x_{n-i_1} + \cdots + x_{n-k}) \bmod m$. Une seule multiplication.

Lagged-Fibonacci (très utilisé, mais mauvaise idée):

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

Tous les vecteurs $(u_n, u_{n+k-r}, u_{n+k})$ sont dans seulemnt deux plans!

Coefficients égaux.
Par exemple, prendre tous les a_j non nuls égaux à a (Deng et Xu 2002).
Alors, $x_n = a(x_{n-i_1} + \cdots + x_{n-k}) \bmod m$. Une seule multiplication.

Lagged-Fibonacci (très utilisé, mais mauvaise idée):

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

Tous les vecteurs $(u_n, u_{n+k-r}, u_{n+k})$ sont dans seulemnt deux plans!

Même problème avec add-with-carry et subtract-with-borrow.

Erreur fréquente: croire qu'on est ok si la période est assez longue...

Coefficients égaux.
Par exemple, prendre tous les a_j non nuls égaux à a (Deng et Xu 2002).
Alors, $x_n = a(x_{n-i_1} + \cdots + x_{n-k}) \bmod m$. Une seule multiplication.

Lagged-Fibonacci (très utilisé, mais mauvaise idée):

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

Tous les vecteurs $(u_n, u_{n+k-r}, u_{n+k})$ sont dans seulemnt deux plans!

Même problème avec add-with-carry et subtract-with-borrow.

Erreur fréquente: croire qu'on est ok si la période est assez longue...

Des variantes qui sautent des valeurs sont recommandées par Luscher (1994) et Knuth (1997).

Mais pas très efficace...

MRGs combinés. Considérons deux [ou plusieurs...] MRGs évoluant en parallèle:

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

On définit les deux combinaisons:

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

MRGs combinés. Considérons deux [ou plusieurs...] MRGs évoluant en parallèle:

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

On définit les deux combinaisons:

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

La suite $\{w_n,\, n \geq 0\}$ est la sortie d'un autre MRG, de module $m = m_1 m_2$, et $\{u_n,\, n \geq 0\}$ est presque la même suite si m_1 et m_2 sont proches. Peut atteindre la période $(m_1^k - 1)(m_2^k - 1)/2$.

MRGs combinés. Considérons deux [ou plusieurs...] MRGs évoluant en parallèle:

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

On définit les deux combinaisons:

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

La suite $\{w_n,\ n \geq 0\}$ est la sortie d'un autre MRG, de module $m = m_1 m_2$, et $\{u_n,\ n \geq 0\}$ est presque la même suite si m_1 et m_2 sont proches. Peut atteindre la période $(m_1^k - 1)(m_2^k - 1)/2$.

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

MRGs combinés. Considérons deux [ou plusieurs...] MRGs évoluant en parallèle:

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

On définit les deux combinaisons:

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

La suite $\{w_n,\ n \geq 0\}$ est la sortie d'un autre MRG, de module $m = m_1 m_2$, et $\{u_n,\ n \geq 0\}$ est presque la même suite si m_1 et m_2 sont proches. Peut atteindre la période $(m_1^k - 1)(m_2^k - 1)/2$.

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

Tableaux de paramètres: L'Ecuyer (1999); L'Ecuyer et Touzin (2000).
Implantations "multisteams".

Sauter en avant (pour les suites et sous-suites):

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

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

alors

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

et donc

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

Récurrences Linéaires Modulo 2

Récurrence linéaire matricielles dans $\mathbb{F}_2$ $(\equiv \{0,1\}, \mod 2)$:

$$\mathbf{x}_n = A\mathbf{x}_{n-1}, \quad (k\text{-bit vecteur d'état})$$

$$\mathbf{y}_n = B\mathbf{x}_n, \quad (w\text{-bit vecteur de sortie})$$

$$u_n = \sum_{j=1}^{w} y_{n,j-1} 2^{-j} = .y_{n,0}\, y_{n,1}\, y_{n,2} \cdots, \quad (\text{sortie})$$

Récurrences Linéaires Modulo 2

Récurrence linéaire matricielles dans $\mathbb{F}_2$ ($\equiv \{0,1\}, \mod 2$):

$$\begin{aligned} \mathbf{x}_n &= A\mathbf{x}_{n-1}, && (k\text{-bit vecteur d'état}) \\ \mathbf{y}_n &= B\mathbf{x}_n, && (w\text{-bit vecteur de sortie}) \\ u_n &= \sum_{j=1}^{w} y_{n,j-1}2^{-j} = .y_{n,0}\ y_{n,1}\ y_{n,2}\ \cdots, && (\text{sortie}) \end{aligned}$$

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

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

La période max. $\rho = 2^k - 1$ est atteinte ssi $P(z)$ est primitif sur $\mathbb{F}_2$.

Avec un choix astucieux de $\mathbf{A}$, le calcul de transition se fait avec des décalages, xor, and, masques, etc., sur des blocs de bits. Très rapide.

Cas particuliers: Tausworthe, "linear feedback shift register" (LFSR), GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, polynomial LCG, etc.

Avec un choix astucieux de $\mathbf{A}$, le calcul de transition se fait avec des décalages, xor, and, masques, etc., sur des blocs de bits. Très rapide.

Cas particuliers: Tausworthe, "linear feedback shift register" (LFSR), GFSR, twisted GFSR, Mersenne twister, WELL, xorshift, polynomial LCG, etc.

Pour sauter en avant:

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

Haramoto, L'Ecuyer, Matsumoto, Nishimura, Panneton (2006) proposent une méthode plus efficace pour les grandes valeurs de k.

GCL dans un espace de séries formelles

La fonction génératrice de la coordonnée j de l'état est la série formelle

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

On peut montrer que

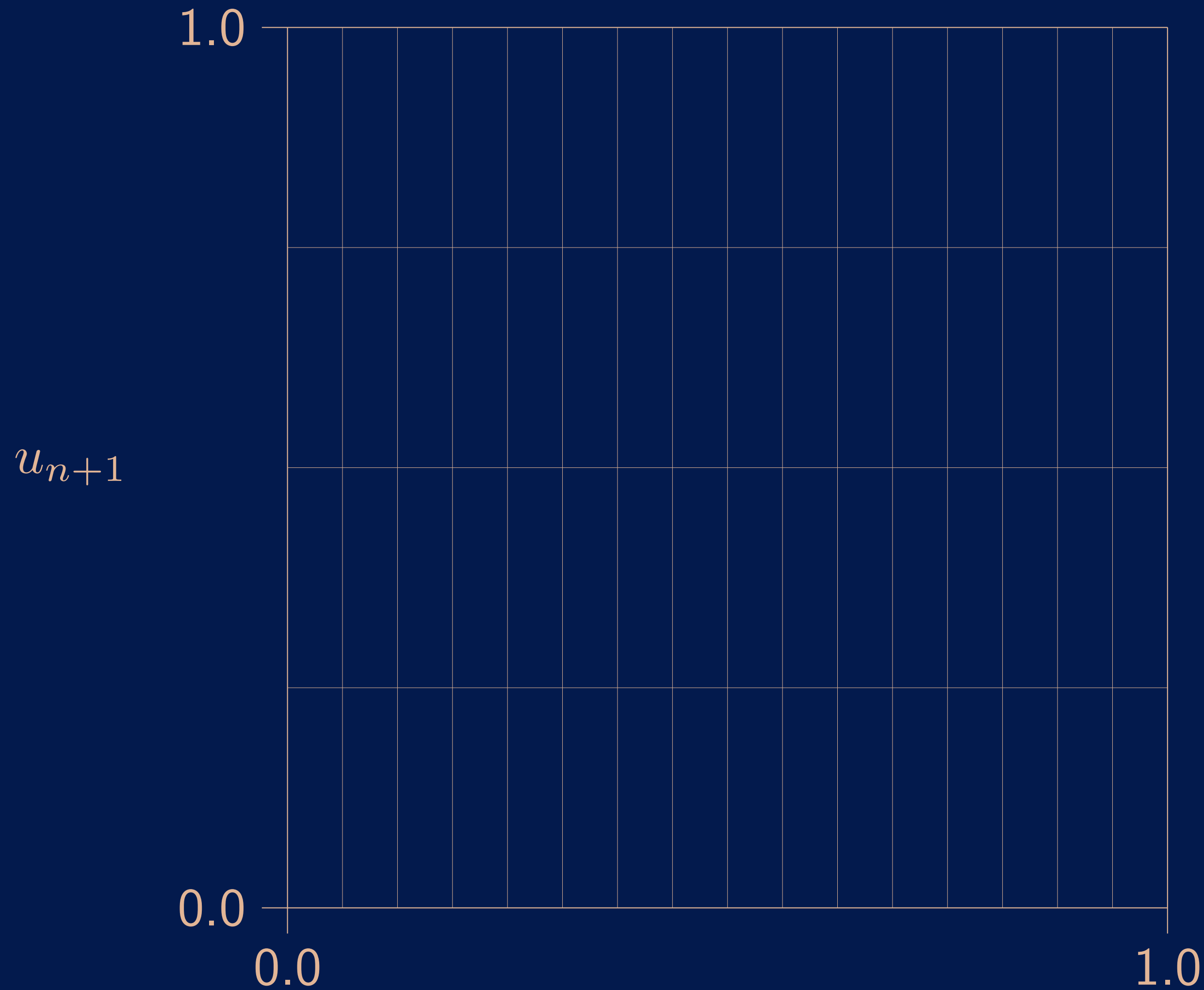
$$g_j(z) \stackrel{\text{def}}{=} s_j(z)P(z) = c_{j,1}z^{k-1} + \cdots + c_{j,k} \in \mathbb{F}_2[z],$$

où

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

Il y a une correspondance biunivoque entre
(1) les états $(x_{0,j}, \ldots, x_{k-1,j})$ de la récurrence;
(2) les polynômes $g_j(z)$ de degré $< k$ (i.e., dans $\mathbb{F}_2[z]/P(z)$);
(3) les séries formelles de la forme $s_j(z) = g_j(z)/P(z)$.

Équidistribution. Pour $j = 1, \ldots, t$, on partitionne le j-ième axe de $[0,1)^t$ en 2^{q_j} parties égales: cela donne 2^q boîtes, où $q = q_1 + \cdots + q_t$.



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

Équidistribution. Pour $j = 1, \ldots, t$, on partitionne le j-ième axe de $[0,1)^t$ en 2^{q_j} parties égales: cela donne 2^q boîtes, où $q = q_1 + \cdots + q_t$.

Si chaque boîte contient exactement 2^{k-q} points de Ψ_t, alors Ψ_t est dit $(q_1, \ldots, q_t)$-équidistribué en base 2.

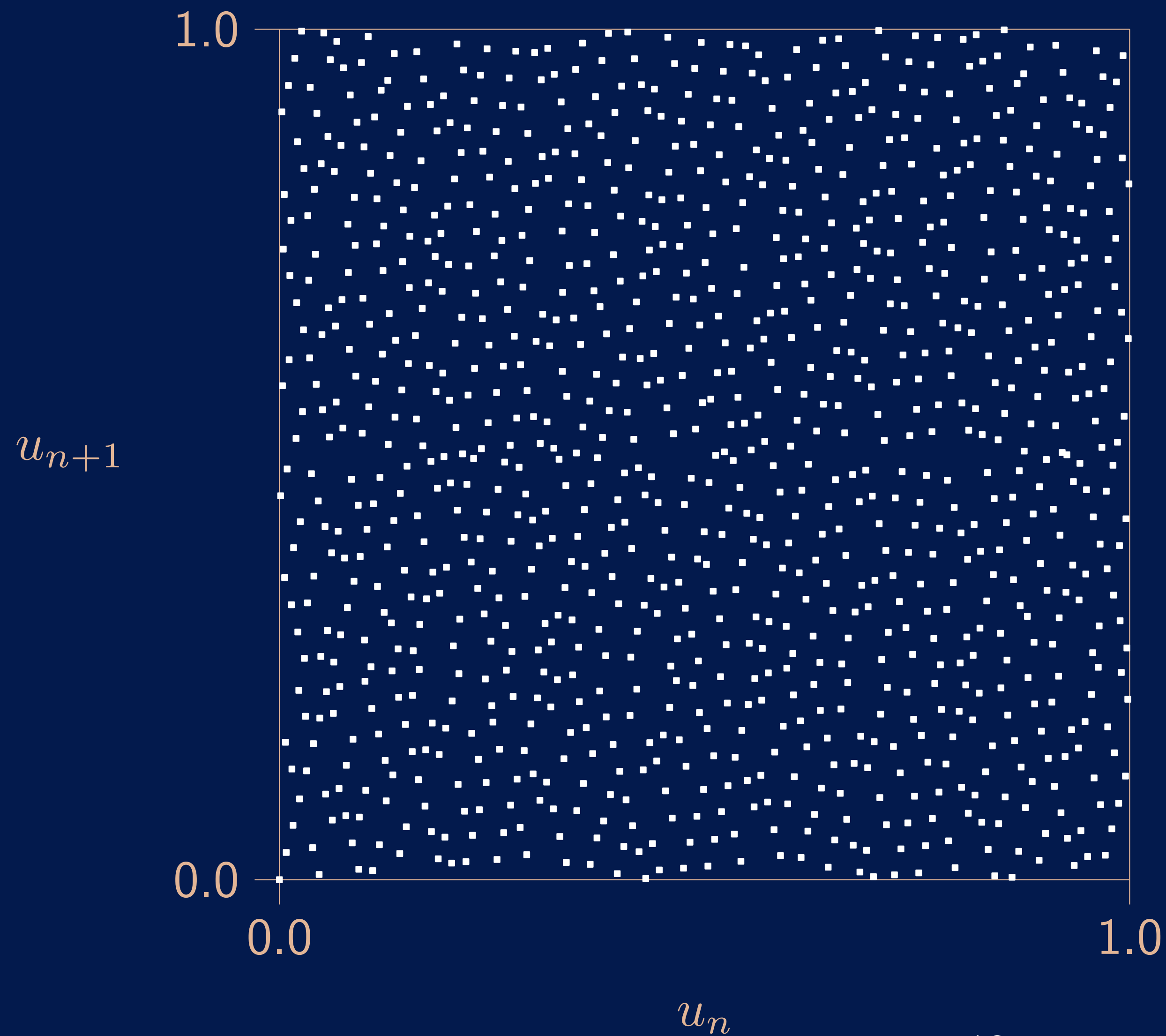
Cela veut dire que tous les vecteurs de dimension t, avec leur coordonnée j tronquée aux q_j premiers bits pour chaque j, apparait le même nombre de fois.

Équidistribution. Pour $j = 1, \ldots, t$, on partitionne le j-ième axe de $[0,1)^t$ en 2^{q_j} parties égales: cela donne 2^q boîtes, où $q = q_1 + \cdots + q_t$.

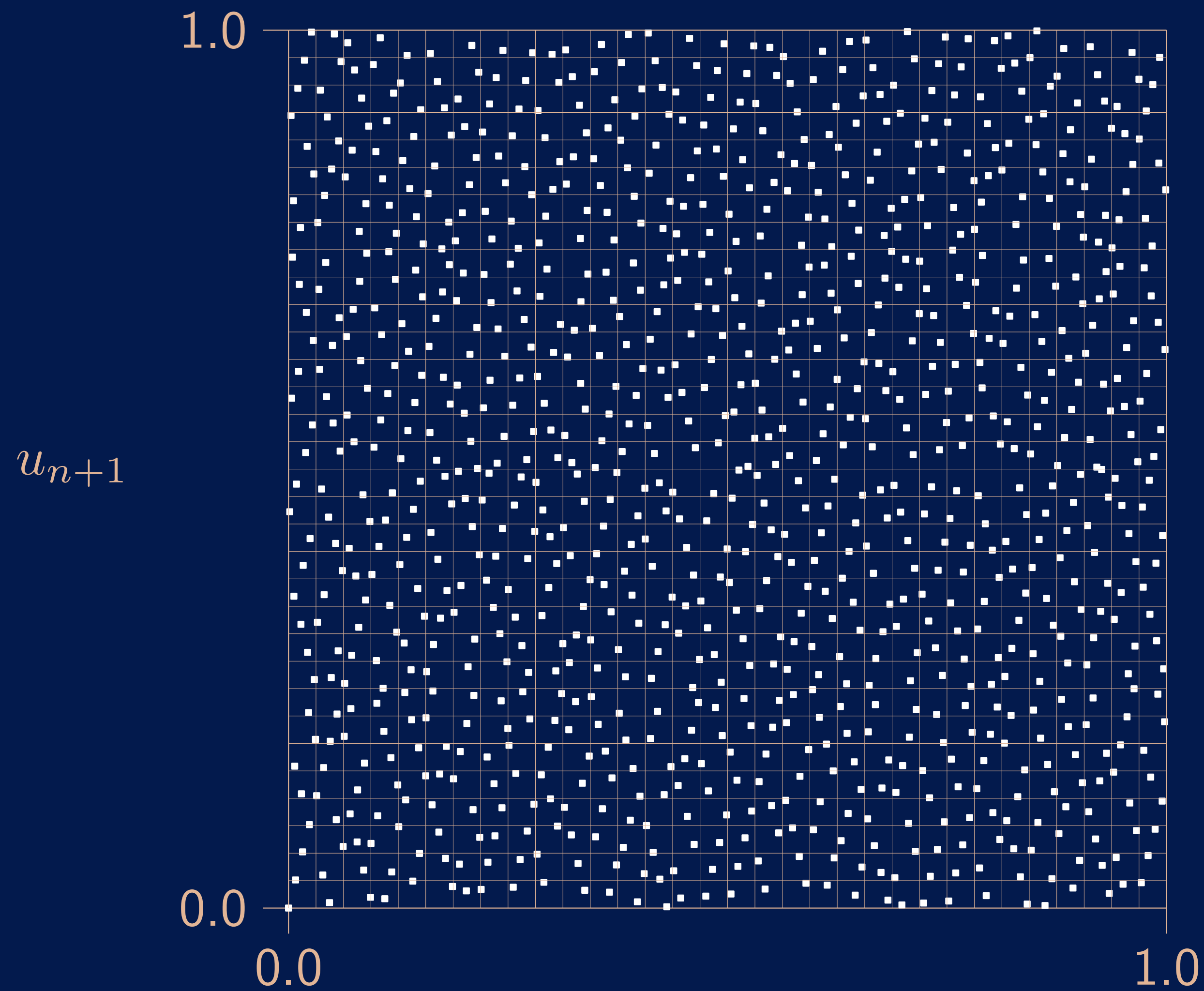
Si chaque boîte contient exactement 2^{k-q} points de Ψ_t, alors Ψ_t est dit $(q_1, \ldots, q_t)$-équidistribué en base 2.

Cela veut dire que tous les vecteurs de dimension t, avec leur coordonnée j tronquée aux q_j premiers bits pour chaque j, apparait le même nombre de fois.

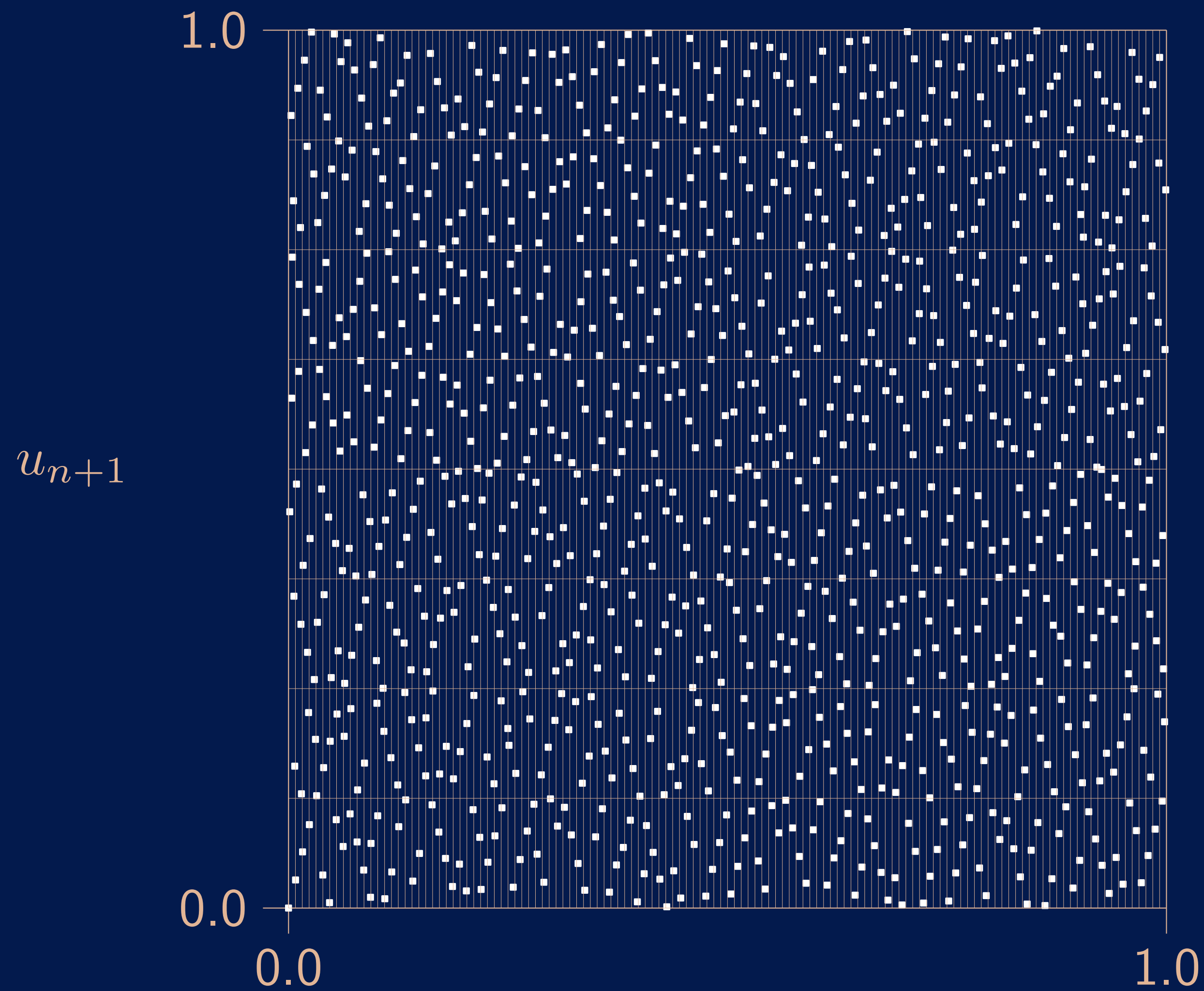
On peut écrire les q bits qui déterminent dans quelle boîte on tombe comme $M\mathbf{x}_0$ pour une certaine matrice M.
Ψ_t est $(q_1, \ldots, q_t)$-équidistribué ssi M est de plein rang.



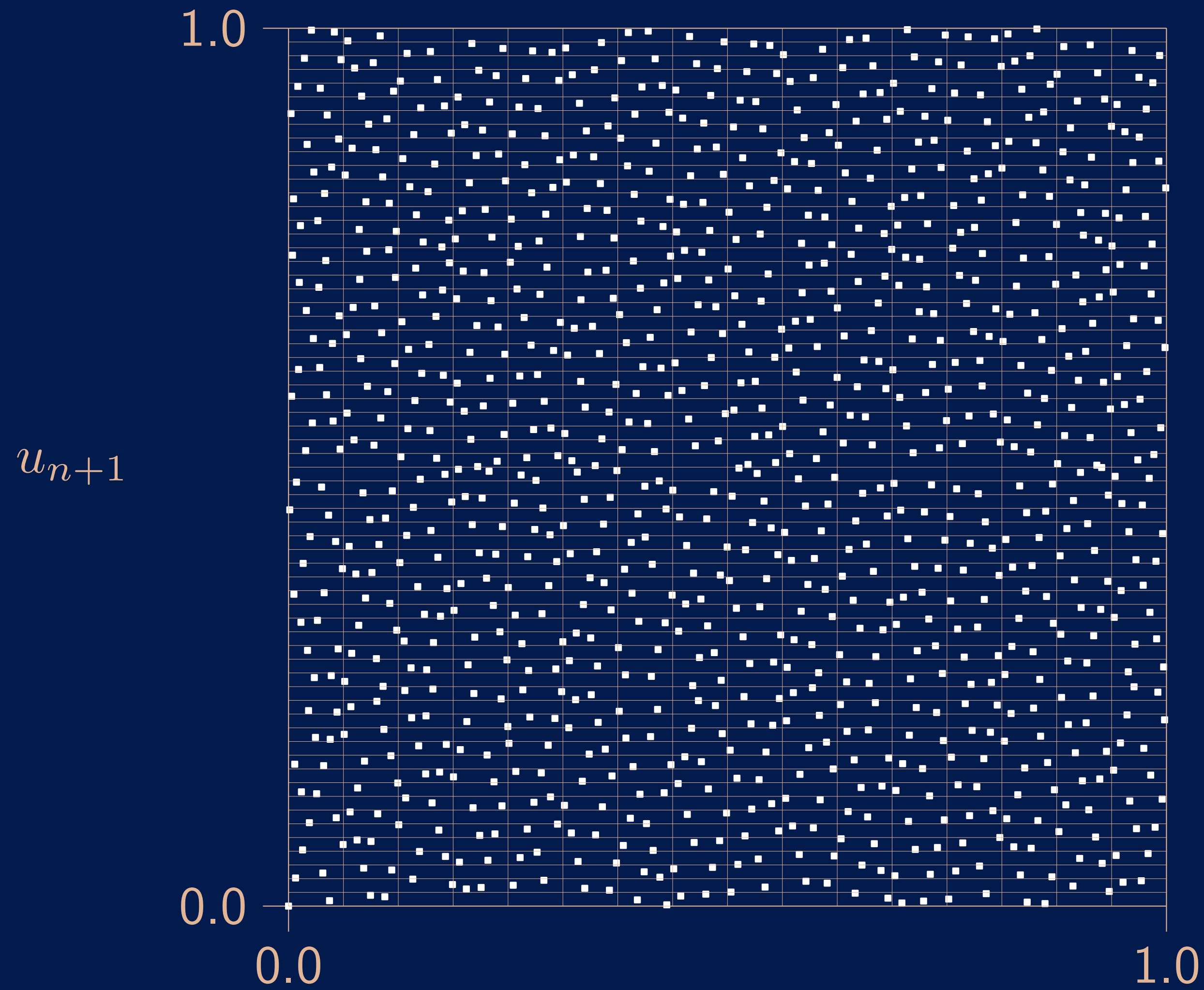
Exemple jouet: générateur LFSR avec $|\Psi_t| = 1024 = 2^{10}$.



Exemple jouet: générateur LFSR avec $|\Psi_t| = 1024 = 2^{10}$.



Exemple jouet: générateur LFSR avec $|\Psi_t| = 1024 = 2^{10}$. 128×8



Exemple jouet: générateur LFSR avec $|\Psi_t| = 1024 = 2^{10}$. 16×64

Pour un ensemble d'indice $I = \{i_1, i_2, \ldots, i_t\}$, on défini l'écart de résolution:

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

Figure de mérite possible:

$$V_{\mathcal{J}} = \sum_{I \in \mathcal{J}} \delta_I$$

où $\mathcal{J}$ est une classe choisie d'ensembles I.

Le choix de $\mathcal{J}$ est une affaire de compromis.

Pour un ensemble d'indice $I = \{i_1, i_2, \ldots, i_t\}$, on défini l'écart de résolution:

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

Figure de mérite possible:

$$V_{\mathcal{J}} = \sum_{I \in \mathcal{J}} \delta_I$$

où $\mathcal{J}$ est une classe choisie d'ensembles I.

Le choix de $\mathcal{J}$ est une affaire de compromis.

On veut aussi que le nombre N_1 de coefficients non nuls α_j's soit proche de $k/2$.

Générateur de Tausworthe ou LFSR (Tausworthe 1965):

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

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

Souvent, seulement deux a_j's non nuls: mauvais!

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

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

L'état a rw mais la période maximale n'est que de $2^r - 1$.

Habituellement, seulement deux a_j's non nuls:

$$\mathbf{v}_n = (\mathbf{v}_{n-q} \oplus \mathbf{v}_{n-r}).$$

$\Rightarrow$ pas assez de mélange des bits.

Twisted GFSR (Matsumoto and Kurita 1992, 1994):

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

Période maximale: $2^{rw} - 1$.

Représentant vedette: TT800, de période $2^{800} - 1$.

Mersenne Twister (Matsumoto and Nishimura 1998):

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

Période maximale: $2^{rw-p} - 1$.

On ajoute une transformation linéaire à la sortie.

Représentant vedette: `MT19937`, de période $2^{19937} - 1$.

MT19937 est devenu très populaire récemment.

Générateurs **Xorshift** (Marsaglia 2003).

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

Marsaglia propose des générateurs spécifiques avec:
— 3 xorshifts par transition;
— période maximale.

Ces générateurs sont très rapides.

Malheureusement, ils ont une très mauvaise équidistribution et échouent de nombreux tests statistiques (Panneton and L'Ecuyer 2004).

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

Idée: Construire la matrice $\mathbf{A}$ en plaçant des opérations simples sur des blocs de 32 bits (ou-exclusifs, décalages, masques binaires, etc.) à des endroits stratégiques.

On optimise la mesure d'uniformité

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

sous les contraintes: (1) période maximale et (2) vitesse comparable à MT19937.

Générateurs spécifiques de périodes allant de $2^{521} - 1$ à $2^{44497} - 1$.

Meilleure équidistribution que MT19937 pour vitesse et période comparables.
Un plus grand nombre N_1 de coefficients non nuls dans $P(z)$.

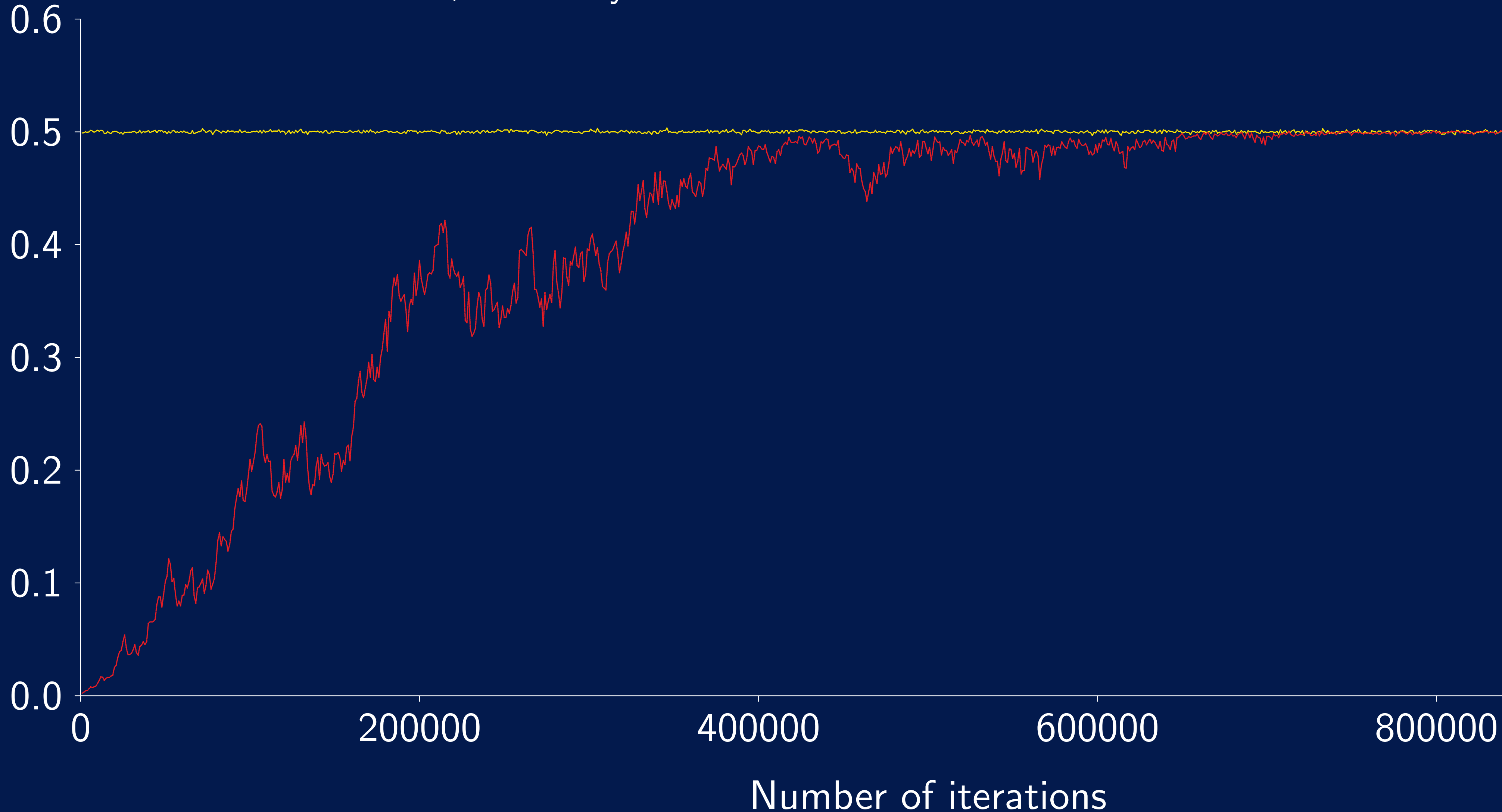
Examples de générateurs WELL, vs MT, pour $w = 32$ bits:

Nom	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 d'un N_1 trop petit et/ou Δ_1 trop grand.

Expérience: choisir un état initial contenant un seul bit à 1.
Essayer toutes les k possibilités et faire la moyenne des k valeurs obtenues à la sortie après chaque transition.

WELL19937 vs MT19937; moyenne mobile sur 1000 itérations.



Générateurs non-linéaires

Congruentiel cubique:

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

Inversif explicite:

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

Une permutation arbitraire de $\{0, \ldots, \rho - 1\}$ stockée dans un tableau.
Peut combiner plusieurs tableaux avec des ρ_j relativement premiers (rapide).
AES, SHA-1, etc. Utilisés en cryptologie.
Systèmes dynamiques chaotiques? Non.

Générateurs combinés linéaires/non-linéaires

Les générateurs $\mathbb{F}_2$-linéaires discutés ici échouent bien sûr un test de complexité linéaire.

Générateurs combinés linéaires/non-linéaires

Les générateurs $\mathbb{F}_2$-linéaires discutés ici échouent bien sûr un test de complexité linéaire.

On voudrait:

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

Générateurs combinés linéaires/non-linéaires

Les générateurs $\mathbb{F}_2$-linéaires discutés ici échouent bien sûr un test de complexité linéaire.

On voudrait:

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

L'Ecuyer et Granger-Picher (2003): Gros générateur $\mathbb{F}_2$-linéaire combiné avec un petit non-linéaire par un XOR.

Générateurs combinés linéaires/non-linéaires

Les générateurs $\mathbb{F}_2$-linéaires discutés ici échouent bien sûr un test de complexité linéaire.

On voudrait:

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

L'Ecuyer et Granger-Picher (2003): Gros générateur $\mathbb{F}_2$-linéaire 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.

Générateurs combinés linéaires/non-linéaires

Les générateurs $\mathbb{F}_2$-linéaires discutés ici échouent bien sûr un test de complexité linéaire.

On voudrait:

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

L'Ecuyer et Granger-Picher (2003): Gros générateur $\mathbb{F}_2$-linéaire 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.

Tests statistiques empiriques

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

Tests statistiques empiriques

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

Test:
— On définit une v.a. T, fonction des u_i, dont la loi sous $\mathcal{H}_0$ est connue (approx.).
— On rejette $\mathcal{H}_0$ si T prend une valeur trop extrême par rapport à cette loi.
Si la valeur est "suspecte", on peut répéter le test plusiques fois.

Quels sont les meilleurs tests? Pas de réponse à cela.
Différents tests permettent de détecter différents types de défauts.

Tests statistiques empiriques

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

Test:
— On définit une v.a. T, fonction des u_i, dont la loi sous $\mathcal{H}_0$ est connue (approx.).
— On rejette $\mathcal{H}_0$ si T prend une valeur trop extrême par rapport à cette loi.
Si la valeur est "suspecte", on peut répéter le test plusiques fois.

Quels sont les meilleurs tests? Pas de réponse à cela.
Différents tests permettent de détecter différents types de défauts.

Idéal: le comportement de T ressemble à celui des v.a. qui nous intéressent dans nos simulations. Mais pas pratique...

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

Tests statistiques empiriques

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

Test:
— On définit une v.a. T, fonction des u_i, dont la loi sous $\mathcal{H}_0$ est connue (approx.).
— On rejette $\mathcal{H}_0$ si T prend une valeur trop extrême par rapport à cette loi.
Si la valeur est "suspecte", on peut répéter le test plusiques fois.

Quels sont les meilleurs tests? Pas de réponse à cela.
Différents tests permettent de détecter différents types de défauts.

Idéal: le comportement de T ressemble à celui des v.a. qui nous intéressent dans nos simulations. Mais pas pratique...

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

Compromis (heuristique): Se satisfaire d'un GPA qui passe les tests raisonnables.
Les tests échoués sont très difficiles à trouver et exécuter.
Formalisation: cadre de complexité algorithmique, populaire en cryptologie.

Exemple: Un test de collisions

On partitionne la boîte $[0,1)^t$ en $k = d^t$ boîtes cubiques de même taille.
On génère n points $(u_{ti}, \ldots, u_{ti+t-1})$ dans $[0,1)^t$, $i = 0, \ldots, n-1$.
Soit X_j le nombre de points dans la boîte j.
Nombre de collisions:

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

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

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

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

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

Exemple: espacement des anniversaires

On partitionne encore $[0,1)^t$ en $k = d^t$ cubes et on génère n points.
Soient $I_1 \le I_2 \le \cdots \le I_n$ les numéros des boîtes où tombent les points.
On calcule les espacements $S_j = I_{j+1} - I_j$, $1 \le j \le n-1$.
Soient $S_{(1)}, \ldots, S_{(n-1)}$ les espacements triés.
Nombre de collisions entre les espacements:

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

Si k est grand, sous $\mathcal{H}_0$, Y est approx. Poisson de moyenne $\lambda = n^3/(4k)$.

Si Y prend la valeur y, la p-valeur à droite est

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

Autres exemples

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

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].

- Implantation d'une grande variété de tests statistiques pour des générateurs quelconques (logiciels ou matériels). Écrit en C. Disponible sur ma page web.

- Contient aussi des 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.

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

Résultats de batteries de tests appliqués à des GPA bien connus

Générateur	$\log_2 \rho$	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)

Générateur	$\log_2 \rho$	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)

Générateur	$\log_2 \rho$	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

Générateur	$\log_2 \rho$	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

- 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 en vue, surtout s'ils utilisent des algorithmes secrets!

- Certains logiciels ont d'excellents GPAs; vérifier.

- Des GPAs avec suites et sous-suites multiples sont disponibles via ma page web, en Java, C, et C++. Tapez Pierre L'Ecuyer dans Google et cliquez sur Software.

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