IFT 6561
Simulation et modèles

Fabian Bastin
DIRO
Université de Montréal

Automne 2013

Générateur récursif multiple (MRG)

Nous pouvons généraliser la récurrence du GCL par

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

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 demeure essentiellement la même.

Si $k = 1$, nous retrouvons le générateur à congruence linéaire classique, avec $c = 0$.

L'état à l'étape n est $s_n = \boldsymbol{x}_n = (x_{n-k+1}, \ldots, x_n)^T$.
Espace d'états: $\mathcal{Z}_m^k$, de cardinalité m^k.
La période maximale est $\rho = m^k - 1$, pour m premier.

Polynôme caractéristique

On associe au MRG le polynôme caractéristique:

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

où $a_0 = -1$.

Pour $k > 1$, pour avoir une période maximale, il est possible de montrer qu'il suffit d'avoir au moins deux coefficients non nuls, dont a_k. Ainsi, la récurrence la plus économique a la forme:

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \mod m,$$

avec $0 < r < k$.

$m = 2^e$

Une erreur fréquente, commise en particulier par les informaticiens peu au fait des statistiques, est de considérer $m = 2^e$.

Utiliser une puissance de 2 pour m permet en effet de facilement calculer le produit $ax \mod m$, et est parfois décrit comme efficace, ce qui est vrai du point de la rapidité d'exécution.

Les effets sur la période sont pourtant dommageables, vu que

- pour $k = 1$ et $e \geq 4$, on a $\rho \leq 2^{e-2}$;
- pour $k > 1$, on a $\rho \leq (2^k - 1)2^{e-1}$.

$m = 2^e$: exemple

Si $k = 7$ et $m = 2^{31} - 1$, la période maximale est $(2^{31} - 1)^7 - 1 \approx 2^{217}$. Mais pour $m = 2^{31}$ on a $\rho \leq (2^7 - 1)2^{31-1} < 2^{37}$, i.e. 2^{180} fois plus petit!

Pire, si nous nous intéressons au i^{th} bit le moins significatif, pour $k = 1$, la période de $x_n \mod 2^i$ ne peut pas dépasser $\max(1, 2^{i-2})$. Pour $k > 1$, la période de $x_n \mod 2^i$ ne peut pas dépasser $(2^k - 1)2^{i-1}$.

$m = 2^e$: exemple

Récurrence $x_n = 10205 x_{n-1} \mod 2^{15}$:

$$
\begin{aligned}
x_0 &= 12345 = 011000000111001_2 \\
x_1 &= 20533 = 101000000110101_2 \\
x_2 &= 20673 = 101000011000001_2 \\
x_3 &= 7581 = 001110110011101_2 \\
x_4 &= 31625 = 111101110001001_2 \\
x_5 &= 1093 = 000010001000101_2 \\
x_6 &= 12945 = 011001010010001_2 \\
x_7 &= 15917 = 011111000101101_2.
\end{aligned}
$$

De tels générateurs restent populaires, mais sont à proscrire dans des simulations dignes de ce nom. Ainsi, la fonction `ran48` reste présente dans les librairies C standards BSD.

m	a	c	Source
2^{24}	1140671485	12820163	early MS VisualBasic
2^{31}	65539	0	RANDU (IBM)
2^{31}	134775813	1	early Turbo Pascal
2^{31}	1103515245	12345	`rand()` in BSD ANSI C
2^{32}	69069	1	VAX/VMS systems
2^{32}	2147001325	715136305	BCLP language
2^{35}	5^{15}	7261067085	Knuth (1998)
2^{48}	68909602460261	0	Fishman (1990)
2^{48}	25214903917	11	Unix's `rand48()`
2^{48}	44485709377909	0	CRAY system
2^{59}	13^{13}	0	NAG Fortran/C library

Variables aléatoires communes (VAC)

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.

Variables aléatoires communes

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

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

Générateurs à sous-suites multiples

Afin de pouvoir adéquatement représenter les différentes variables aléatoires, il peut être intéressants de pouvoir instancier des générateurs de variables aléatoires à volonté, et faire évoluer ceux-ci en parallèle, plutôt que d'utiliser un seul générateur et transformer les tirs dans les distributions voulues à la volée.

Nous voudrions pouvoir utiliser plusieurs fois un même générateur au sein d'un programme, mais en débutant avec des semences différentes afin de produire des suites aléatoires différentes.

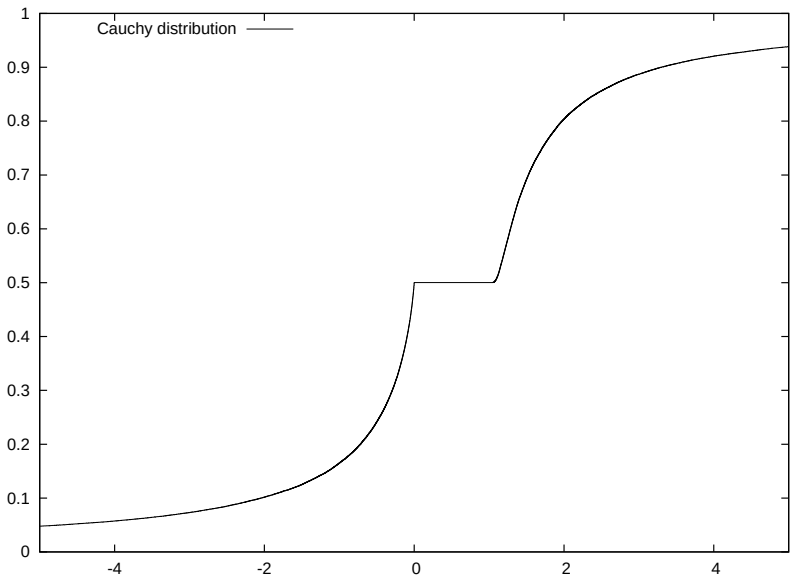
Générateurs à sous-suites multiples

Une première approche consiste à créer plusieurs générateurs, en spécifiant manuellement ces semences. Le danger majeur de cette approche est qu'il est difficile de prévoir la position des ces semences dans la séquence aléatoire, ce qui peut conduire à produire des séquences fortement corrélées. Le risque est d'autant plus élevé que la période du générateur est faible.

Exemple

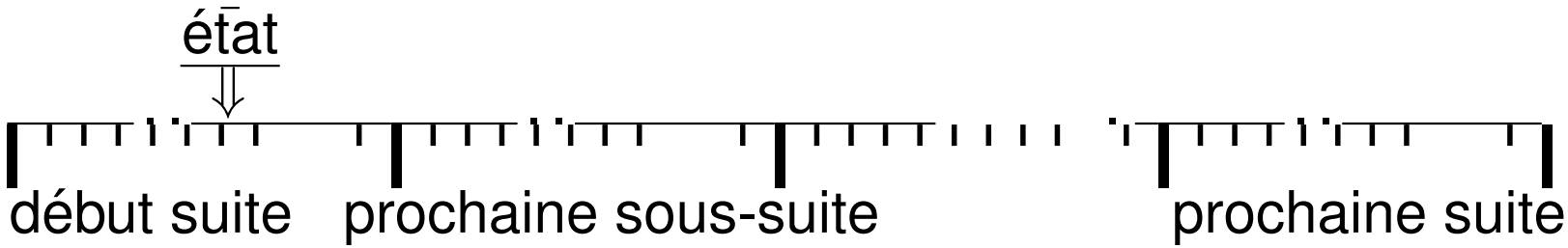
Soit X, Y, deux variables aléatoires normales $N(0,1)$ indépendantes. Il est possible de montrer que le rapport X/Y suit une distribution de Cauchy.

Générons ce rapport à l'aide du GCL Standard Minimal, avec 1 comme semence au numérateur, et 2 au dénominateur.



Générateurs à sous-suites multiples

Il est ainsi utile de pouvoir partitionner ces suites (ou “streams”) en sous-suites.



Sauts entre suites

Pour passer d'une suite à une autre, il est nécessaire de pouvoir calculer un point de la récurrence sans devoir générer tous les points intermédiaires. Or, nous pouvons écrire

$$\boldsymbol{x}_n = \boldsymbol{A}\boldsymbol{x}_{n-1} \mod m = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix} \boldsymbol{x}_{n-1} \mod m.$$

Ainsi

$$\boldsymbol{x}_{n+\nu} = \boldsymbol{A}^\nu \boldsymbol{x}_n \mod m = (\boldsymbol{A}^\nu \mod m)\boldsymbol{x}_n \mod m.$$

Sauts entre suites

Nous pouvons précalculer $\boldsymbol{A}^{\nu} \mod m$ au moyen de la procédure suivante:

$$\boldsymbol{A}^{\nu} \mod m = \begin{cases} (\boldsymbol{A}^{\nu/2} \mod m)(\boldsymbol{A}^{\nu/2} \mod m) \mod m & \text{si } \nu \text{ est pair;} \\ \boldsymbol{A}(\boldsymbol{A}^{\nu-1} \mod m) \mod m & \text{si } \nu \text{ est impair.} \end{cases}$$

Interface Java

```
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.

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

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

```
    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 $\{i, i+1, \ldots, j-1\}$.

```
}
```

Interface Java

```
public class RandMrg implements RandomStream {
```

Une implantation particulière: `MRG32k3a`.

```
    public RandMrg();
```

Construit une nouvelle suite de cette classe.

```
    public static void setPackageSeed (long seed[]);
```

Fixe l'état initial de la première suite. Les autres sont calculés selon un espacement prédéterminé.

```
}
```

MRGs combinés

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

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

On définit les deux combinaisons:

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

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

Pour accélérer la génération de point, il est possible de prendre tous les a_j non nuls égaux à a (Deng et Xu 2002). Alors, $x_n = a(x_{n-i_1} + \cdots + x_{n-k}) \mod m$. Une seule multiplication.

Les meilleurs générateurs ne jouissent cependant pas de cette propriété.
Tableaux de paramétres: L'Ecuyer (1999); L'Ecuyer et Touzin (2000).

MRG32K3a

$J = 2$, $k = 3$,
$m_1 = 2^{32} - 209$, $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$,
$m_2 = 2^{32} - 22853$, $a_{21} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$.
Combination: $z_n = (x_{1,n} - x_{2,n}) \bmod m_1$.

Le générateur correspond à un MRG caractérisé par $k = 3$,
$m = m_1 m_2 = 18446645023178547541$, et les paramétres
$a_1 = 181696684712528925 57$, $a_2 = 3186860506199273833$,
$a_3 = 873861326439822262 2$. Sa période ρ vaut
$(m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191}$.

MRG32K3a

```
#define norm 2.328306549295728e-10  /* 1/(m1+1) */
#define m1   4294967087.0
#define m2   4294944443.0
#define a12     1403580.0
#define a13n     810728.0
#define a21      527612.0
#define a23n    1370589.0

double  s10, s11, s12, s20, s21, s22;

double MRG32k3a ()
{
   long   k;
   double p1, p2;
```

```
    /* Component 1 */
    p1 = a12 * s11 - a13n * s10;
    k = p1 / m1;   p1 -= k * m1;   if (p1 < 0.0) p1 += m1;
    s10 = s11;   s11 = s12;   s12 = p1;

    /* Component 2 */
    p2 = a21 * s22 - a23n * s20;
    k  = p2 / m2;  p2 -= k * m2;   if (p2 < 0.0) p2 += m2;
    s20 = s21;   s21 = s22;   s22 = p2;

    /* Combination */
    if (p1 <= p2) return ((p1 - p2 + m1) * norm);
    else return ((p1 - p2) * norm);
}
```

Illustration avec SSJ

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.stat.*;

public class Collision {
   int k;              // Number of locations.
   int m;              // Number of items.
   double lambda;      // Theoretical expectation of C (as
   boolean[] used;     // Locations already used.

   public Collision (int k, int m) {
      this.k = k;
      this.m = m;
      lambda = (double) m * m / (2.0 * k);
      used = new boolean[k];
   }
```

Illustration avec SSJ

```
// Generates and returns the number of collisions.
public int generateC (RandomStream stream) {
   int C = 0;
   for (int i = 0; i < k; i++) used[i] = false;
   for (int j = 0; j < m; j++) {
      int loc = stream.nextInt (0, k-1);
      if (used[loc]) C++;
      else used[loc] = true;
   }
   return C;
}
```

Illustration avec SSJ

```
    // Performs n indep.
    public void simulateRuns (int n, RandomStream stream,
                              Tally statC) {
        statC.init();
        for (int i=0; i<n; i++)
            statC.add (generateC (stream));
        statC.setConfidenceIntervalStudent();
        System.out.println (statC.report (0.95, 3));
        System.out.println (" Theoretical mean: "
            + lambda);
    }

    public static void main (String[] args) {
        Tally statC = new Tally
            ("Statistics on collisions");
        Collision col = new Collision (10000, 500);
        col.simulateRuns (100000, new MRG32k3a(), statC);
    }
}
```