TIPE 2016-2017

La génération algorithmique de nombres aléatoires

Utilisation à des fins de simulation

Louis Guichard

Introduction

Thème du TIPE: Optimalité : choix, contraintes, hasard.

Pourquoi générer des nombres aléatoires?

- *Hasard* est synonyme d'*imprévisible*
- Deux principales utilités :
  - Simuler des phénomènes imprévisibles
  - Protéger nos informations (cryptographie)

Comment s'y prendre ?

- Générateurs physiques
  - Lent et difficile à mettre en place
- Générateurs algorithmiques
  - Procédé déterministe $\Rightarrow$ paradoxe

Problématique: Comment un algorithme peut-il générer une suite de nombres qu'on pourrait qualifier d'aléatoire?

Table des matières

Introduction

I - Validité des algorithmes
- Définir le hasard
- Quelques idées...
- Test du Khi-Carré

II - Algorithmes
- Carré Médian de Von Neumann
- Générateurs congruentiels linéaires
- Algorithme de Mersenne-Twister

III - Conclusion
- Comparaison avec le BBS

Définir le hasard

3 idées équivalentes :

- **Martin-Löf** : une suite est aléatoire si elle ne possède aucune propriété exceptionnelle (1966).
- **Schnorr** : une suite aléatoire est imprévisible (1971).
- **Levin** et **Chaintin** : une suite aléatoire est incompressible (1974).

Paradoxe de Borel (antérieur aux définitions ci-dessus) :
Si on considère une suite aléatoire comme étant une suite n'ayant aucune propriété particulière, alors cela donne une caractéristique aux suites répondant à cette définition.

Si elles sont utilisées dans le domaine de la **simulation**, on retiendra qu'une suite aléatoire ne doit présenter **aucune propriété exceptionnelle**.

Quelques idées...

Objectif

Confondre les suites générées avec des suites *vraiment* aléatoires.

Première idée : Fréquence d'apparition des 0 et 1

Soit X une variable aléatoire définie par le nombre de zéros dans une série binaire de longueur n.
X suit une loi binomiale de paramètre n et $p = \frac{1}{2}$.
Si $n = 20$, on a donc $P(6 \leq X \leq 14) \simeq 0,96$.

Distance d'une loi de probabilité à une autre

Considérons un n-échantillon sur $\{1, ..., k\}$.
Soit $p = (p_1, ..., p_k)$ une loi de probabilité de référence sur cet ensemble et $q = (q_1, ..., q_k)$ une autre loi de probabilité.
La distance euclidienne $\mathcal{D}$ de la loi de probabilité p à la loi q est définie par : $\mathcal{D}(p, q) = \sum_{i=1}^{k} |p_i - q_i|$

Quelques idées...

$u_{20} = (0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1)$

Deuxième idée : Fréquence des paires de 0

Intuitivement, il semble peu probable de n'obtenir aucune paire de 0 consécutifs sur une série binaire de longueur 20.
On peut démontrer que la probabilité de n'obtenir aucune paire de 0 consécutifs dans une telle suite est inférieure à 2%

Distance entre deux 0 consécutifs

Soit X_i la variable aléatoire représentant la valeur du i^e élément d'une liste pour $i \in [|1, n|]$. On introduit la variable aléatoire D_i qui représente la distance entre le i^e zéro de la liste et le $i + 1^e$. Si les X_i suivent une loi uniforme sur $\{0, 1\}$, alors on a

$$\forall i \in [|1, p|], P(D_i = 0) = \frac{1}{2} \text{ et } P(D_i = k) = \frac{1}{2^{k+1}}$$

Test du Khi-Carré

Problème

En l'état, les deux tests précédents ne permettent que de comparer les résultats d'un algorithme à un autre. On ne peut pas dire qu'un algorithme est "bon", mais seulement qu'il est "meilleur qu'un autre".

Intérêt du test

Le test du Khi-Carré est un test d'adéquation, qui permet de vérifier si une série de données suit une certaine loi de probabilité.

Principe du test

Avec les mêmes notations que précédemment, on définit la distance du χ^2 par

$$\chi_n^2(p,q) = \sum_{i=1}^{k} \frac{(np_i - nq_i)^2}{np_i}$$

Test du Khi-Carré

Procédé

- Choisir un risque α de rejeter à tort l'hypothèse H_0 : "l'échantillon suit la loi p".
- A partir de la table du Khi-Carré, on détermine la borne b_α au-delà de laquelle le risque de rejeter à tort l'hypothèse H_0 est inférieur à celui choisi.
- On rejette l'hypothèse H_0 si la valeur de $\chi^2(p_{empirique}, p_{thorique})$ est supérieure à la borne b_α.

α / ν	0,99	0,975	0,95	0,90	0,10	0,05	0,025	0,01	0,001
1	0,0002	0,001	0,004	0,016	2,71	**3,84**	5,02	**6,63**	10,83
2	0,02	0,05	0,10	0,21	4,61	**5,99**	7,38	**9,21**	13,82
3	0,11	0,22	0,35	0,58	6,25	**7,81**	9,35	**11,34**	16,27
4	0,30	0,48	0,71	1,06	7,78	**9,49**	11,14	**13,28**	18,47
5	0,55	0,83	1,15	1,61	9,24	**11,07**	12,83	**15,09**	20,51

Utilisation du test du χ^2

Idée 1 : Distance à la loi uniforme

Le test du χ^2 est un test d'adéquation. Il suffit donc de l'appliquer en calculant $\chi^2(p,q) = \sum_{i=0}^{k} \frac{(np_i - nq_i)^2}{nq_i}$ où p_i est la loi empirique et q_i la loi théorique (*ie*. la loi uniforme) pour vérifier l'hypothèse H_0 : "la suite est aléatoire".

Idée 2 : Indépendance des valeurs

La variable aléatoire D_i définie précédemment, et représentant la distance entre 2 zéros consécutifs dans la suite, suit la loi $\forall k \in \mathbb{N}$, $P(D_i = k) = \frac{1}{2^{k+1}}$. On va une nouvelle fois calculer la distance du χ^2 entre la loi empirique et la loi théorique.

En Python


```
    ###  TEST  DU  KHI-CARRE  ###

def KhiCarre(Empirique,Theorique):
    ''' Test du Khi-Carré appliqué aux
        listes Empiriques et Theoriques'''
    X = 0
    k = len(Empirique)
    for i in range(k):
        p = Empirique[i]
        q = Theorique [i]
        X += (p-q)**2 / q
    return X
```

Carré Médian de Von Neumann

Principe

Très simple : on prend un nombre, on l'élève au carré, et les digits centraux constituent la sortie. Et on recommence. Par exemple : $1234^2 = 1522756$ puis $2275^2 = 5175625$ puis $7562^2 = ...$

```
    ###  CARRE  MEDIAN  ###

def neumann(graine,n):
    ''' Génère une liste de n nombres
        aléatoires à partir de la graine '''
    G = graine
    L = []
    for i in range(n):
        G = G*G
        text = str(G)+'0000'
        G = int(text[1:5])
        L.append(G)
    return L
```

Carré Médian de Von Neumann

Problème

- Si le nombre "du milieu" est ’0000’, l’algorithme ne renvoie plus que des ’0000’ (’0000’ est un état absorbant).
- Dès qu’on retombe sur un nombre déjà utilisé, l’algorithme tourne en boucle

Cet algorithme n’est donc pas capable de produire de longues séries de nombres aléatoires : sa **période** est limitée.
De plus, la période dépend du premier nombre utilisé, appelé **graine**.
Certaines graines risquent de moins bien fonctionner que d’autres.

```
>>> testgraine(50)
[1, 2, 2, 2, 2, 2, 2, 4, 3, 3, 2, 107, 69, 133, 79, 2, 41, 87, 85,
 79, 2, 106, 42, 107, 84, 2, 84, 107, 42, 106, 2, 79, 85, 87, 48,
57, 48, 18, 80, 93, 2, 105, 5, 106, 68, 3, 91, 19, 44, 93]
```

Carré Médian de Von Neumann


```
def neumanngraine(k):
    ''' Teste les k premières graines '''
    R = []
    for i in range(k):
        G = i
        L = []
        compteur = 0
        while G not in L:
        # Test fin de période
            L.append(G)
            compteur += 1
            G = G*G
            text = str(G)+'0000'
            G = int(text[1:5])
        R.append(compteur)
    return R,max(R)
```

Résultat

En exécutant neumanngraine(10000), on obtient que la période la plus longue est de 167, et qu'elle est très inégale selon la graine choisie.

Générateurs congruentiels linéaires

Générateur congruentiel linéaire

Un **générateur congruentiel linéaire** produit une séquence de nombres pseudo-aléatoires à partir de la relation de récurrence

$$u_{n+1} = au_n + b \bmod M$$

où a est appelé le *multiplicateur*, b l'*incrément* et M le *module*.

```
    ### GENERATEURS CONGRUENTIELS LINEAIRES ###

def lcg(graine,a,b,M,n):
    ''' Génère n nombres aléatoires avec la relation
        de récurrence X(n+1) = aX(n) + b mod M '''
    L = []
    x = graine
    for i in range(n):
        x = (a*x + b)%M
        L.append(x)
    return L
```

Prenons $a = 10$, $b = 5$, $M = 100$ et $X_0 = 10$.
La séquence générée par l'algorithme est $[5, 55, 55, 55, 55, ...]$

Avec $a = 25$, $b = 16$, $M = 256$ et $X_0 = 50$, la séquence générée par l'algorithme est $[50, 242, 178, 114, 50, 242, ...]$

Graine et autres paramètres

55 est un état absorbant pour cet algorithme, comme 0000 l'était pour celui de Von Neumann. L'avantage d'un générateur congruentiel linéaire est que l'on peut jouer sur les paramètres a, b et M afin d'obtenir des séquences de meilleures qualités.

Choix du module

Les ordinateurs calculant naturellement en base binaire, on utilisera systématiquement un module du type $M = 2^k$.

Générateurs congruentiels linéaires


```
def periode(graine,a,b,k):
    ''' Renvoie la longeur de la période
        du GCL(a,b,M) initialisé avec graine '''
    L = []
    x = graine
    while x not in L:
        L.append(x)
        x = (a*x + b)%(2**k)
    return len(L)

def periodemax(graine,k,n):
    ''' Test toutes les valeurs de a et b
        entre 1 et n '''
    L = []
    for a in range(n):
        for b in range(n):
            P = periode(graine,a,b,k)
            if P == 2**k :
                L.append([a,b])
    return L
```

Générateurs congruentiels linéaires

```
>>> periodemax(13,4,16)
[[1, 1], [1, 3], [1, 5], [1, 7], [1, 9], [1, 11], [1, 13],
[1, 15], [5, 1], [5, 3], [5, 5], [5, 7], [5, 9], [5, 11],
[5, 13], [5, 15], [9, 1], [9, 3], [9, 5], [9, 7], [9, 9],
[9, 11], [9, 13], [9, 15], [13, 1], [13, 3], [13, 5], [13,
7], [13, 9], [13, 11], [13, 13], [13, 15]]
```

Résultats

On peut conjecturer que

$$\begin{cases} a \equiv 1[4] \\ b \text{ est impair} \end{cases} \Leftrightarrow \lambda = 2^m$$

Démonstration

Théorème

Soit un générateur congruentiel linéaire de période λ suivant la relation $u_{n+1} = au_n + b \bmod M$. Alors :

$$\begin{cases} a \equiv 1[4] \\ b \text{ est impair} \Rightarrow \lambda = M \\ M = 2^m \end{cases}$$

Lemme:

Si p est premier, alors $\forall k \in \mathbb{N}$: $\begin{cases} x \equiv 1[p^k] \\ x \not\equiv 1[p^{k+1}] \end{cases} \Rightarrow \begin{cases} x^p \equiv 1[p^{k+1}] \\ x^p \not\equiv 1[p^{k+2}] \end{cases}$

Preuve de la lemme

Preuve de la lemme:

On peut écrire $x = 1 + qp^k$ où q est un entier premier avec p. Alors :
$x^p = (1+qp^k)^p = \sum_{i=0}^{p} \binom{p}{i} q^i p^{ik} = 1+\binom{p}{1} qp^k+...+\binom{p}{p-1} q^{(p-1)k}+q^p p^{kp}$
D'où
$x^p = 1 + qp^{k+1}(1 + \frac{1}{p}\binom{p}{2} qp^k + \frac{1}{p}\binom{p}{3} q^2 p^{2k} + ... + \frac{1}{p}\binom{p}{p} q^{p-1} p^{(p-1)k})$.
La quantité entre les parenthèses est un entier, et chaque terme entre les parenthèses à l'exception du premier est un multiple de p.
Aussi, $\forall n \in [|2, p-1|]$, p divise $\binom{p}{n}$. Ainsi, $\frac{1}{p}\binom{p}{n} q^{n-1} p^{(n-1)k}$ est divisible par $p^{(n-1)k}$.
De plus, le dernier terme $q^{p-1} p^{(p-1)k-1}$ est divisible par p puisque $(p-1)k \geq 2$.
Donc $x^p \equiv 1 + qp^{k+1}[p^{k+2}]$.

Démonstration

On suppose que b est impair, que $a \equiv 1[4]$ et que $M = 2^m$. Notons λ la période du générateur. Les paramètres a, b, M sont positifs.

$a \equiv 1\ [4] \Leftrightarrow \exists q \in \mathbb{N}\ /\ a = 4q + 1$
Ainsi, $\exists q'$ impair, $k \geq 2\ /\ a = 2^k q' + 1$ ($q' = 1$ si q pair, $q' = q$ sinon)
donc $a \equiv 1\ [2^k]$ et $a \not\equiv 1\ [2^{k+1}]$ avec $k \geq 2$.

En utilisant la lemme, on obtient $\begin{cases} a^2 \equiv 1[2^{k+1}] \\ a^2 \not\equiv 1[2^{k+2}] \end{cases}$

Par récurrence, on a $\forall k' \geq k$: $\begin{cases} a^{2^{k'}} \equiv 1[2^{k+k'}] \\ a^{2^{k'}} \not\equiv 1[2^{k+k'+1}] \end{cases}$

De plus, $\frac{a^{2^{k'}}-1}{a-1} = \frac{q2^{k+k'}}{q'2^k} = \frac{q}{q'}2^{k'} \equiv 0[2^{k'}]$.

En particulier, $m > k$ donc $\frac{a^{2^m}-1}{a-1} \equiv 0[2^m]$.

Démonstration

Le générateur a pour formule de récurrence $x_{n+1} = ax_n + b \bmod M$.
Par récurrence, il vient $x_{n+k} = a^k x_n + \frac{a^k-1}{a-1} b \bmod M$.
La période est de λ, ce qui se traduit par
$\forall n > M$, $x_n \equiv x_{n+\lambda} \equiv a^\lambda x_n + \frac{a^\lambda-1}{a-1} b \equiv \frac{a^k-1}{a-1}(x_n(a-1)+b)\ [M]$.
Or, b est impair et $a-1$ est pair donc $x_n(a-1)+b$ est impair.
Ainsi, $M = 2^m$ divise $\frac{a^\lambda-1}{a-1}$ et donc $\frac{a^\lambda-1}{a-1} \equiv 0[2^m]$

D'où, $\frac{a^{2^m}-1}{a-1} \equiv \frac{a^\lambda-1}{a-1}[2^m]$ et $a^{2^m} \equiv a^\lambda[2^m]$ et λ divise 2^m.

Ainsi, $\exists j \leq m \mid \lambda = 2^j$. Si $j < m$, $\frac{a^{2^j}-1}{a-1} \not\equiv 0[2^{j+1}]$.

A fortiori, $\frac{a^{2^j}-1}{a-1} \not\equiv 0 \bmod 2^m$ donc $\lambda \neq 2^j$.
Donc $\lambda = 2^m$.

Générateurs congruentiels linéaires

Prenons, $a = 3 \times 4 + 1 = 13$, $b = 17$ et $M = 2^{10} = 1024$.
La formule de récurrence est alors $u_{n+1} = 13u_n + 17 \ mod \ 1024$.

La période du générateur congruentiel linéaire ainsi parametré est 1024, **quelque soit la graine**.

Tests statistiques

Il est maintenant temps d'utiliser les deux tests élaborés en première partie.

```
def convbin(L):
    ''' Convertie une suite aléatoire d'entier
        en une suite aléatoire binaires '''
    K = []
    for i in L :
        B = bin(i)[3:]
        for b in B : K.append(b)
    return K
```

Générateurs congruentiels linéaires


```
def testlcg(graine,a=13,b=17,M=1024,n=100):
    C = convbin(lcg(graine,a,b,M,n))
    return KhiCarre(frequence(C),uniforme(2,len(C)))

def testtout(a=13,b=17,M=1024,n=100):
    m = 0
    S = 0
    for i in range(M):
        K = testlcg(i,a,b,M,n)
        m = max(m,K)
        S += K
    return S/M,m
```

```
>>> testtout()
(0.2585355020771036, 1.5565438373570522)
```

α / ν	0,99	0,975	0,95	0,90	0,10	0,05	0,025	0,01	0,001
1	0,0002	0,001	0,004	0,016	2,71	**3,84**	5,02	**6,63**	10,83
2	0,02	0,05	0,10	0,21	4,61	**5,99**	7,38	**9,21**	13,82

Générateurs congruentiels linéaires


```
def distance(L,k=100):
    ''' k est la distance max observée '''
    R = [0 for i in range(k)]
    compteur = 0
    for i in L :
        if i == 0 :
            if compteur < k :
                R[compteur] += 1
            else : R[k-1] += 1
            compteur = 0
        else :
            compteur +=1
    return R
```

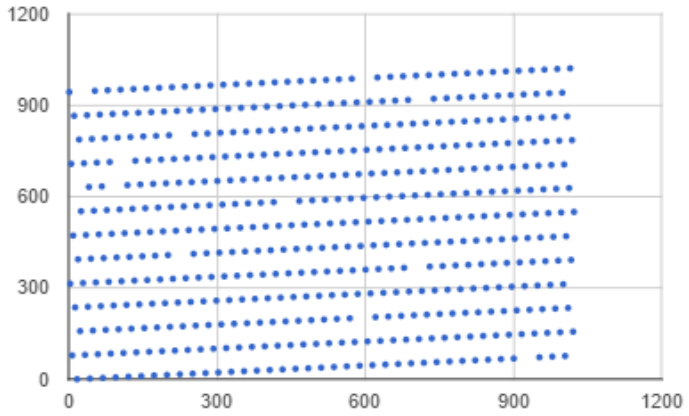
```
>>> testtout2()
(7.161688007075893 5, 16.661944808274544)
```

ν \ α	0,10	0,05	0,025	0,01	0,001
1	2,71	3,84	5,02	6,63	10,83
20	28,41	31,41	34,17	37,57	45,31

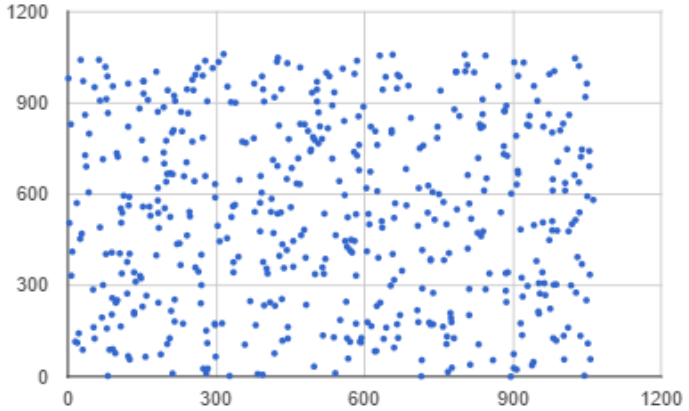
Algorithme de Mersenne-Twister


```
def decalage(vect, d=0, g=0):
    ''' vect vecteur de bits et d et g décalage de bits dr/ga '''
    L = vect
    L = d*[0] + L[:(bits-d)]
    L = L[g:] + g*[0]
    return L

def tempering(vect):
    ''' Mélange les bits de x '''
    Y = vect
    Y = addition(Y,decalage(Y,u))
    Y = addition(Y,multiplication(decalage(Y,0,s),b))
    Y = addition(Y,multiplication(decalage(Y,0,t),c))
    Y = addition(Y,decalage(Y,v))
    return Y

def mersenne(n, graine):
    R = []
    L = graine
    for i in range(n):
        x = recurrence(L)
        L = L[1:]+[x]
        R.append(sortie(tempering(x)))
    return R
```

Analyse spectral



Générateur congruentiel linéaire ou algorithme Blum Blum Shub



Algorithme de Mersenne-Twister

Comparaison avec le BBS

Algorithme de Blum Blum Shub

L'algorithme de Blum Blum Shub est un générateur de nombres pseudo-aléatoires **cryptographiquement sûr**. Il génère une suite d'entiers selon la relation de récurrence $u_{n+1} = u_n^2 \ mod \ M$.

Qualité du générateur

L'algorithme de Blum Blum Shub a le même défaut que l'ensemble des générateurs congruentiels linéaires : il échoue au test spectral à des dimensions pourtant faibles (en général 2 ou 3).

Conclusion

Un générateur cryptographiquement sûr (ie. imprévisible en temps polynomial) peut ne pas réussir la plupart des tests statistiques que réussissent les bons générateurs.
Cryptographiquement sûr $\not\Rightarrow$ Aléatoire au sens de Martin-Löf
Mon binôme a constaté que la réciproque était fausse aussi.

Merci de votre attention !