TIPE 2016-2017

La génération algorithmique de nombres aléatoires

Utilisation à des fins de cryptographie

Lucie Jobard

27 Mai 2017

Introduction

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

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 Blum Blum Shub

III - Conclusion
- Comparaison avec Mersenne-Twister

«Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin» - Von Neumann

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).

Si elles sont utilisées dans le domaine de la **cryptographie**, on retiendra qu'une suite est aléatoire si elle est imprédictible, *ie*. si prévoir les nombres générés revient à résoudre un problème difficile.

Quelques idées...

Objectif

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

Première idée : Répartition équitable des 0 et 1

Deuxième idée : Indépendance des nombres générés

Le $i+1^e$ nombre généré doit être indépendant du i^e.

```
def repartition(L):
    R=[0]*2
    n = len(L)
    for i in range(n):
        R[L[i]] +=1
    return R
```

```
def previsibilite(L,k):
    R=[0]*2
    n = len(L)
    for i in range(n-1):
        if L[i] == k :
            a = L[i+1]
            a = a%2
            R[a] +=1
    return R
```

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

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

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.

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 d'une 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
```

Générateurs congruentiels linéaires

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

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

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 parametré ainsi 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()
(0.258535502077103 6, 1.556543837357052 2)
```

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

Tests d'indépendance et de répartition

```
>>> L=gcl(1000,8)
>>> stat_independance(L,0)
(0.0, 1.0)
>>> stat_independance(L,1)
(1.0, 0.0)
>>> stat_repartition(L)
(0.5, 0.5)
```

Prévisibilité


```
def Euclide_etendu(L,M):
    #algorithme pour casser le GCL connaissant le modulo M
    x0,x1,x2=L[0],L[1],L[2]
    y1 = x1-x0
    y2=x2-x1
    m0=M
    b0=y1
    t0=0
    t=1
    q=int(m0/b0)
    r=m0-q*b0
    while r>0 :
        temp = t0 - q*t
        if temp >=0 : temp = temp%M
        else : temp = M - ((-temp)%M)
        t0=t
        t=temp
        m0=b0
        b0=r
        q=int(m0/b0)
        r=m0 - q*b0
    if b0!=1 : return -1
    else : return t
```

Prévisibilité


```
def casser_gcl(L,M):
    inv=euclide_etendu(L,M)
    if inv == -1 : return -1
    x0,x1,x2=L[0],L[1],L[2]
    y2=x2-x1
    a = (y2 * inv)%M
    b= (x1 - a*x0)%M
    return a,b,M
```

Par exemple:

```
>>> L = [97,188,235,293,604]
>>> casser_gcl(L,1023)
(23, 3, 1023)
```

Algorithme de Blum Blum Shub

Principe

On calcule la sortie en itérant la suite $x_{n+1} = x_n^2$ mod M où $M = p.q$ produit de deux grands nombres premiers congrus à 3 modulo 4.

```
def bbs(n,graine=0):
    ''' Applique l'algorithme de Blum Blum Shub '''
    L = []
    p = 347 # p et q congrus à 3 mod 4
    q = 191
    M = p*q # ici M < 100 000
    if graine != 0 : # Graine : en argument ou non
        x = graine
    else :
        x = int(input('premier nombre ? = graine : ')) # L'utilisateur définit
    for i in range (n):
        b=x%2
        x = (x**2)%M # Formule de BBS
        L.append(b)
    return L
```

Blum Blum Shub - Graines et période

Question

La période est-elle aussi limitée que celle du Carré Médian ? L'efficacité des graines est-elle aussi aléatoire ?

```
>>> bbs_graine(100)
([1, 1549, 1548, 1548, 1549, 1549, 173, 1549, 1548, 1548, 1549, 1548, 1548, 1549
, 1549, 1548, 1549, 1549, 1549, 1549, 1549, 1549, 1549, 1549, 1548, 1549, 1548,
1549, 1549, 1548, 1549, 1549, 1549, 1548, 1549, 1548, 1549, 1549, 172, 1548, 154
9, 1549, 1548, 1549, 1549, 1548, 1549, 1548, 172, 1549, 1549, 1548, 1549, 1549,
1549, 1549, 1549, 1549, 1548, 1549, 1549, 1549, 1549, 1548, 1549, 1549, 1548, 15
49, 1549, 1549, 1549, 1549, 1549, 1549, 1548, 1549, 1549, 1549, 1549, 1549, 1548
, 173, 1549, 1549, 1548, 1549, 1549, 1549, 1549, 1548, 1549, 1549, 1549, 1549, 1
549, 1549, 1549, 1549, 1549, 1548], 1549, 1)
>>>
```

Résultat

En testant toutes les graines de 1 à 10 000, on obtient que 94% d'entres elles ont une période de 1548 ou 1549. Cette période dépend bien sûr du nombre M (modulo) choisi dans l'algorithme. La période moyenne est de 1460.

Application du test du Khi-Carré

Test du Khi-Carré

La période étant suffisamment longue, et les graines suffisamment sûres, on peut tester l'algorithme de Blum Blum Shub avec le test du Khi-Carré.

```
>>> Kbbs(50,55555)
8.830687830687830
>>> Kbbs(200,55555)
19.839378238341965
>>> Kbbs(500,55555)
11.500260552371026
>>> Kbbs(1000,55555)
9.895545715029956
>>> Kbbs(1500,55555)
8.038294922890312
>>> Kbbs(3000,55555)
17.998873385908656
>>> Kbbs(100000,55555)
583.2685044159003
>>>
```

Application du test du Khi-Carré

1 - Choix du risque

On peut choisir un risque $\alpha = 5\%$ de rejeter à tort l'algorithme.

1 - Borne associée

Dans la table de Khi-Deux, la borne associée à une expérience à 9 degrés de liberté et à un risque de 5% est $b_\alpha \simeq 16,92$

3 - Comparaison

La distance du χ^2 obtenue précédemment est de $8,04$
On a $\chi^2_n(p,q) \simeq 8,04 < 16,92 \simeq b_\alpha$
L'hypthèse est validée. On peut conclure que l'algorithme de Blum Blum Shub génère des nombres qui semblent aléatoires selon le test du Khi-Carré.

```
>>> L=bbs(1000,8)
>>> stat_repartition(L)
(0.507, 0.493)
>>> stat_independance(L,0)
(0.5335968379446641, 0.4664031620553336)
>>> stat_independance(L,1)
(0.4787018255578093, 0.5212981744421906)
```

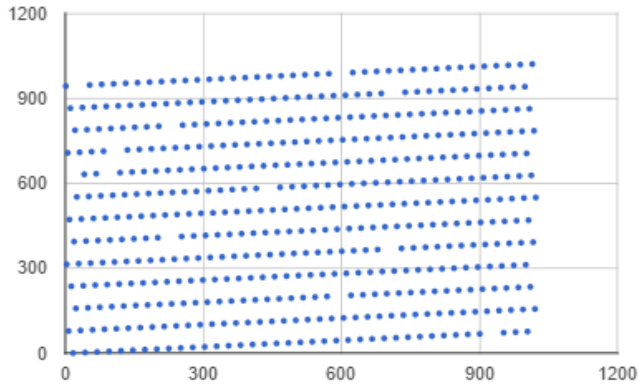
Bilan des générateurs

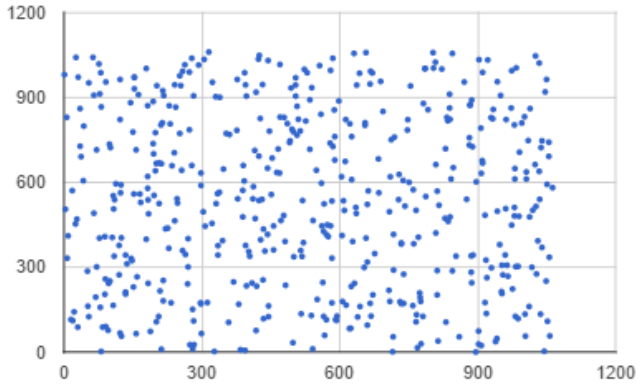
- L'algorithme du Carré Médian de Von Neumann a une période trop faible et trop dépendante de la graine pour être utilisé.
- Les générateurs congruentiels linéaires ont une bonne périodicité mais sont prédictibles.
- L'algorithme de Blum Blum Shub, en plus d'avoir de bonnes propriétés intrinsèques, a été démontré imprévisible.

Comparaison avec Mersenne-Twister

Algorithme de Mersenne-Twister

L'algorithme de Mersenne-Twister est le générateur de nombres pseudo-aléatoires utilisé dans Python. Il est très efficace en simulation. Il se divise en deux parties : la première est de type congruentiel linéaire comme étudié précédemment. La deuxième consiste en une série de décalages de bits.





Conclusion

Aléatoire au sens de Martin-Löf $\not\Leftrightarrow$ Cryptographiquement sûr.