Nombres pseudo-aléatoires

Alessandro Torcini et Andreas Honecker

LPTM

Université de Cergy-Pontoise

Nombres Aléatoires

Entiers

Les nombres aléatoires entiers sont $x \in [0, 1]$ caractérisé par leur probabilités $P(x_i)$

$$\sum_{i=0}^{n} P(x_i) \equiv 1 \qquad x = 0, 1, 2, \ldots, n$$

nous allons nous focaliser sur les distributions uniformes sur l'intervalle

$$P(x_i) = P = \frac{1}{n+1} \ \forall \ x \in [0, n]$$

Flottantes

Les nombres aléatoires flottantes sont $x \in [0, n]$ caractérisé par leur distribution des probabilités $p(x)$

$$p(x)dx = \text{probabilite} \quad \text{que} \quad x \in (x, x + dx)$$

nous allons nous focaliser sur les distributions uniformes sur l'intervalle

$$p(x) = P \ \forall \ x \in [0, 1]$$

Nombres Aléatoires

Donc, nous cherchons des suites des nombres « aléatoires » x_i avec les propriétés suivantes :

1. La distribution des nombres aléatoires doit être uniforme sur un intervalle donné. Pour les flottantes on utilise souvent une distribution uniforme sur l'intervalle $[0, 1]$, pour les entiers l'intervalle $[0, n]$ avec un nombre n fixe.

   Ça suffit normalement comme on peut convertir les intervalles.

$$x \in [0, 1] \quad \rightarrow \quad y \in [a, b]$$

$$y = a + x \times (b - a)$$

2. Il n'y doit pas avoir des corrélations entre les nombres x_i, soit si on connait déjà $x_0, \ldots, x_{i-1}$, aucune prédiction sur le résultat x_i devrait être possible.

Prescription Déterministe

Des méthodes connues pour générer des nombres aléatoires utilisent les dés ou la roulette.
Du point du vue de la physique on peut penser à la désintégration radioactive ou le bruit thermique dans les circuits électriques.

Dans l'ordinateurs on utilise une prescription déterministe

$$x_i = f(x_{i-1}, x_{i-2}, \ldots) \tag{1}$$

pour construire une suite, qui n'est pas du tout aléatoire, on parle des nombres pseudo-aléatoires.

Naturellement, il faut rendre les prédictions le plus difficile possible.

La fonction f doit donc être suffisamment non-linéaire, même chaotique

Nous discuterons maintenant quelque générateurs pseudo-aléatoires qui utilisent une prescription déterministe de type (1). Tous ces générateur ont besoin de quelque forme de mémoire : au moins il faut connaître x_{i-1} pour calculer x_i

Le générateur de NumPy

D'abord plusieurs réalisations de générateurs sont déjà disponibles.
Le générateur de NumPy est accessible par la fonction

`random.random_integers(barriere_inferieure, barriere_superieure)`

du module `numpy`.

Cette fonction vous rend des entiers pseudo-aléatoires entre `barriere_inferieure` et `barriere_superieure`.

```
import numpy as np                  # importer le module comme "np"

# afficher quelques nombres aleatoirs
for i in range(0,10):
print np.random.random_integers(1,6)
```

Exercise : Réalisez un « dé » artificiel, soit des entiers aléatoires entre 1 et 6, avec `random.random_integers`. Regardez quelques nombres produit par votre « dé ». Vérifiez que la distribution est uniforme, c'est-à-dire que la probabilité de chaque nombre soit $1/6$.

IDLE

Le générateur congruentiel linéaire

Afin de comprendre mieux comment un générateur fonctionne et quelles problèmes sont possibles, nous réaliserons notre générateur propre.
Le générateur congruentiel linéaire. Ce générateur produit une suite dont chaque terme dépend du précédent, selon la formule suivante :

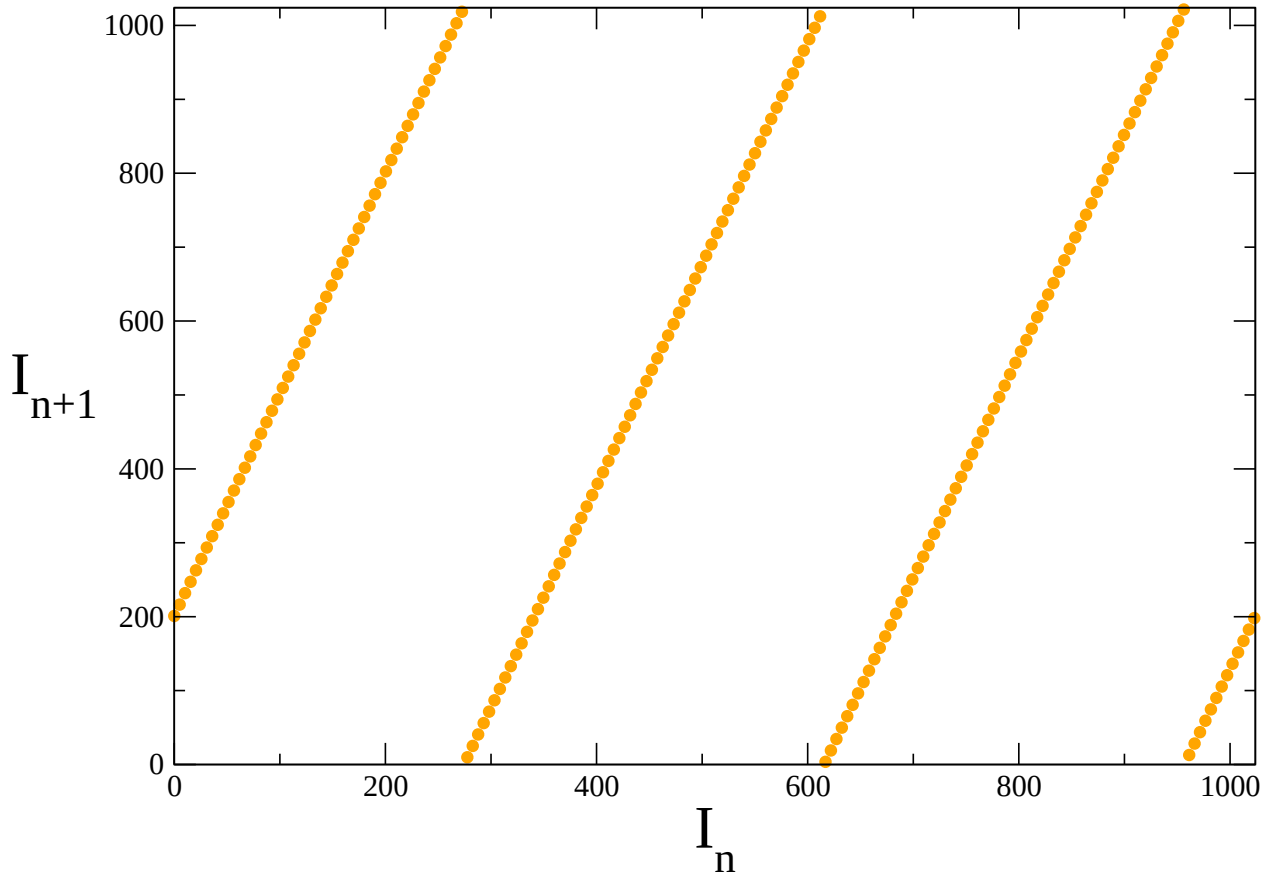
$$I_{n+1} = (a\, I_n + c) \quad (\text{modulo}\, m) \tag{2}$$

1. a, c et m sont des constantes entiers qui doivent être bien choisi.
2. Ce type de générateur est périodique, soit $I_{n+k} = I_n$ pour tous n avec une période k.
3. Comme il y aux maximum m valeurs possibles de I_n et I_n définit tous les termes futurs, la période et limité par m, soit $k \leq m$
4. Donc, il faudra mieux choisir un m grand.
5. En fonction de la choix des paramètres ce générateur peut être bon ou mauvais.

L'application de récurrence

$I_{n+1}=a*I_n+c$ (modulo m)

a=3 c=201 m=1024



Un mauvais exemple

Choisir les valeurs a, c et m « au hasard » n'est pas une bonne idée.
Prenons un exemple, soit le générateur congruentiel linéaire employant les valeurs $a = 3$, $c = 6$, $m = 5$, nous obtenons :

```
a, c, m = 3,6,5 # Valeurs des parametres
def random():   # Le generateur
  random.RNG = (a*random.RNG+c) % m  # la prescription
  return random.RNG

random.RNG = 11 #Si on veut garder une valeur a la exterieur d'une fonction
                #le nom de la fonction au debut du nom de la variable
                #et utiliser la variable APRES la fonction

for i in range(0,10):    # Afficher quelques valeurs
  print random()
```

Il est clair que ces suites ne peuvent être considérées comme aléatoires. On voit donc bien que l'on doit choisir avec précaution les paramètres du générateur si l'on espère obtenir des nombres qui s'approchent de l'aléa parfait. Il faut alors se demander comment choisir a, c et m convenablement.

Listes en Python

La liste est une structure de donnée beaucoup utilisée en Python.
Il s'agit d'un vecteur, dans lequelle on peut mettre plusieurs variables.
Une liste se note entre crochets « `[, ]` » avec la virgule comme séparateur, soit les listes ont le forme

```
[l0, l1, ..., ln]
```

On crée donc une liste vide par

```
liste = []
```

et après on peut ajouter des éléments à la fin de la liste avec

```
liste.append(element)
```

ou

```
liste = liste + [element]
```

Listes en Python

1. `liste[i]` rend le l'élément `i+1` de la liste ($0 \to 1, 2 \to 3$)

```
>>> liste = [1, 10, 100, 250, 500]
>>> liste[0]
1
>>> liste[-1] # Cherche la dernière occurence
500
>>> liste[-4:] # Affiche les 4 dernières occurrences
[500, 250, 100, 10]
>>> liste[:] # Affiche toutes les occurences
[1, 10, 100, 250, 500]
>>> liste[2:4] = [69, 70]
[1, 10, 69, 70, 500]
>>> liste[:] = [] # vide la liste
[]
```

2. `liste[i:k]` rend les élément entre i et $k-1$ Si on supprime le `i`, la partie commence avec le début de la `liste`, si on supprime `k`, la partie se termine à la fin de la `liste`.

Listes en Python

1. On peut joindre deux listes avec le commande

```
combinaison = liste1 + liste2
```

2. Supprimer une entrée avec un index fonction del

```
>>> liste = ["a", "b", "c"]
>>> del liste[1]
>>> liste
['a', 'c']
```

3. Supprimer une entrée avec sa valeur (avec liste.remove)

```
>>> liste = ["a", "b", "c"]
>>> liste.remove("a")
>>> liste
['b', 'c']
```

Absence des corrélations

Jusqu'à maintenant nous avons seulement vérifié que la distribution est uniforme, soit la propriété (i).

La propriété (ii), soit absence des corrélations, reste encore à vérifier.

Le test suivant graphique accomplit tous les deux buts : on dessine des points sur le plan avec les coordonnées I_{n-1}, I_n, soit on prend un nombre aléatoire pour l'abcisse et le suivant pour l'ordonnée.

Si la distribution est uniforme et les deux termes sont indépendants, la distribution des points sur le plan doit être uniforme.

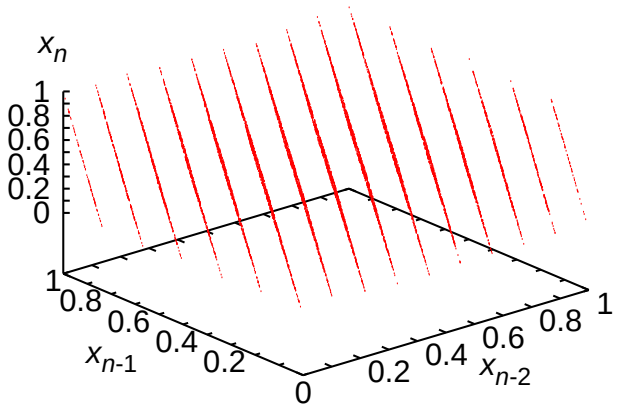
Absence des corrélations

```
import matplotlib.pyplot as plt # importer le module Matplotlib comme "plt"
a, c, m = 57, 33, 4096     # Valeurs des parametres
def random(): # Le generateur :
  random.RNG = (a*random.RNG+c) % m  # la prescription
  return random.RNG

random.RNG = 123456
x = [] # une liste pour l'abcisse
for i in range(0,m): # une periode
   x.append(random()) # tirer et ajouter une valeur a la liste

# Creer un liste decale dans y en mettant le premier element x[0] a la fin
y = x[1:]+[x[0]]

plt.plot(x, y, 'ro')       # creer le graphe
plt.xlabel("I_n-1")        # appellation de l'abscisse
plt.ylabel("I_n")          # appellation de l'ordonnee
plt.xlim(0,m)              # assurer des bonnes limites pour l'abscisse
plt.ylim(0,m)              # assurer des bonnes limites pour l'ordonnee
plt.show()                 # montrer le graphe
```

RANDU IBM

Si on regarde les multiplets $(I_n, I_{n-1}, \ldots, I_{n-d-1})$ en d dimensions, il y a un maximum de $\sqrt[d]{m}$ hyperplans possibles Marsaglia, PNAS, 1968

Je vous montre encore un exemple mauvais : le générateur RANDU avec $a = 2^{16} + 3 = 65\,539$, $m = 2^{31} = 2\,147\,483\,648$, $c = 0$ qu'était le standard sur les supercalculateurs IBM pendant 1960 et 1975



$$x_n = I_n/(m-1)$$

Si vous tracez les les triplets (I_n, I_{n-1}, I_{n-2}) dans $d = 3$ dimensions pour ce générateur. Évidemment, on trouve fortement moins plans des $\sqrt[3]{2^{31}} \approx 1290$ possibles, il sont seulement 15

RANDU IBM

La raison est simple

$$I_{n+2} = (2^{16} + 3)I_{n+1} = (2^{16} + 3)^2 I_n$$

$$I_{n+2} = (2^{32} + 6 \times 2^{16} + 9)I_n = [6(2^{16} + 3) - 9]I_n$$

parce que 2^{32} mod(2^{31}) = 0 et donc

$$I_{n+2} = 6I_{n+1} - 9I_n$$

Cette relation donne des corrélations macroscopiques entre I_{n+2}, I_{n+1}, I_n

Une modification des valeurs de I_{n+1}, I_n de l'ordre de 0.01, change la valeur de I_{n+2} d'au plus 0.15.

Pour avoir un "bon" générateur, on souhaite une relation avec des coefficients beaucoup plus grands que 6 et 9, de telle manière qu'une petite modification de I_{n+1}, I_n change complètement la valeur de I_{n+2}, , pour donner l'illusion d'un tirage vraiment aléatoire.

La moyenne en statistique

Il sera aussi utile de parler un peu d'une notion en statistique : la moyenne. Supposons qu'un valeur x se réalise avec une probabilité $p(x)$. Dans ce cas, la moyenne d'une fonction $f(x)$ est définit par

$$\langle f(x) \rangle := \sum_x f(x)\, p(x)\,.$$

Prenons p. ex. un dé :

$$p(1) = p(2) = p(3) = p(4) = p(5) = p(6) = \frac{1}{6}\,.$$

Évidemment, cette probabilité satisfait à la condition de normalisation de toute probabilité

$$\sum_{x=1}^{6} p(x) = 1\,.$$

D'un autre côté on trouve pour la valeur moyenne d'un dé

$$\langle x \rangle = \sum_{x=1}^{6} x\, p(x) = \sum_{x=1}^{6} x\, \frac{1}{6} = \frac{1}{6} \sum_{x=1}^{6} x = \frac{1}{6}\, \frac{6 \cdot 7}{2} = \frac{7}{2}\,.$$

Variables indépendants

Si deux variables aléatoires x, y sont indépendants, on trouve pour le produit de deux observables

$$\langle f(x) \cdot g(y) \rangle = \langle f(x) \rangle \cdot \langle g(y) \rangle$$

Prenons p. ex. deux dés et calculons la moyenne de produits des valeurs :

$$\begin{aligned} \langle x \cdot y \rangle &= \sum_{x=1}^{6} \sum_{y=1}^{6} x\, y\, p(x,y) = \sum_{x=1}^{6} \sum_{y=1}^{6} x\, y\, p(x) \cdot p(y) \\ &= \left(\sum_{x=1}^{6} x\, p(x) \right) \cdot \left(\sum_{y=1}^{6} x\, p(y) \right) = \langle x \rangle \cdot \langle y \rangle = \left(\frac{7}{2} \right)^2 . \end{aligned}$$

Ici, le point central est que $p(x,y) = p(x) \cdot p(y)$, soit que les deux dés sont vraiment indépendants.

Approximation de la probabilité

Parfois, on ne connait pas la probabilité vraie, mais on a seulement N réalisations x_i en sachant qu'elles suivent la probabilité inconnue $p(x_i)$. Dans ce cas, on peut approcher la moyenne par

$$\langle f(x)\rangle \approx \langle f(x)\rangle_N := \frac{1}{N}\sum_{k=1}^{N} f(x_k) = \frac{1}{N}\sum_{i=1}^{M} N_i f(x_i)\,; \qquad p_N(x_i) = \frac{N_i}{N}$$

ou N_i est le nombres de foix que le valeur x_i a été trouvé et M les possibles valeurs de la variable x.
L'approximation de la probabilité

$$p_N(x_i) = \frac{N_i}{N}$$

est la frequence de occurence del valeur x_i.
Pourtant, l'approximations pour la moyenne et les probabilités sont normalement l'objet d'un erreur.
Cette erreur devient plus petit, plus grand N devienne.

Méthode de Fibonacci différée

Revenons à un deuxième exemple des générateurs de nombres aléatoires : la méthode Fibonacci différée utilise un mémoire plus grande et la relation de récurrence

$$I_n = I_{n-p} \star I_{n-q} \quad \{I_1, I_2, \ldots, I_{n-p}, \ldots, I_{n-q} \ldots, I_{n-1}\}$$

Le symbole $\star$ peut signifier des opérations différentes. On utilise souvent l'« ou exclusif bit à bit ». En Python, cette opération est réalisée par « ^ ».

ou exclusif bit à bit - XOR

Un XOR au niveau des bits prend deux séquences de bits de longueur égale et exécute l'opération OU exclusif logique sur chaque paire de bits correspondants.
Avec XOR nous effectuons la comparaison de deux bits, valant 1 si les deux sont différents et 0 s'ils sont identiques.

```
a=60        00111100
b=240       11110000
c=a^b       11001100
```

$c = 204$

Méthode de Fibonacci différée

Cette méthode a plusieurs avantages. D'abord, grâce à la construction avec deux termes précédentes, la périodicité peut être beaucoup plus long que la période d'un générateur congruentiel linéaire.
Pour la mise en œuvre on a besoin d'une forme de mémoire. Pour démontrer le principe en Python, vous pouvez utiliser des listes.

```
def random():
  global In          # variable GLOBALE In
  FibAleatoire = ...
  # ajouter nouveau nombre au debut
  In = [FibAleatoire] + In[:p-1]
  return FibAleatoire
```

Ici nous supposons qu'il y a une liste de longueur $p \geq q$ que contient l'histoire
`In =` $[I_{n-1}, \ldots, I_{n-q}, \ldots I_{n-p}]$

Le générateur doit encore être initialisé au début avec $\max(p, q)$ nombres. Pour cette tâche on peut utiliser un générateur congruentiel linéaire.
Dans la fonction, le mot clé « `global` » declare cette liste une variable globale qui est définie à l'extérieur de la fonction.

Corrélations

Si les termes I_n sont indépendants, on doit trouver

$$\langle I_n\, I_{n-r}\, I_{n-s}\rangle_N = \langle I_n\rangle_N \langle I_{n-r}\rangle_N \langle I_{n-s}\rangle_N = \langle I_n\rangle_N^3$$

pour tout $r, s > 0$, $r \neq s$.
Ici $\langle\cdot\rangle_N$ signifie la moyenne sur beaucoup de réalisations.
Donc si on consider

$$0 \leq x_n = \frac{I_n}{m-1} \leq 1$$

et

$$\langle x\rangle = \frac{1}{2} \qquad \langle x\rangle^3 = \frac{1}{8}$$

exercice pour la maison

Vérifiez si il est vrai pour RANDU ; pour le le générateur congruentiel linéaire avec $a = 57$ $m = 4096$ et $c = 33$ et aussi pour le générateur Fibonacci différée « R250 »avec $p = 250$, $q = 147$ et l'ou exclusif bit à bit. Utilisez un générateur congruentiel linéaire avec $a = 65\,539$, $m = 2^{31} = 2\,147\,483\,648$ et $c = 0$ pour initialisation.

Le générateur de NumPy

Essayer le generateur de NumPy version plus efficace (avec des listes)

```
import numpy as np                  # importer le module comme "np"

# afficher quelques nombres aleatoirs
for i in range(0,10):
print np.random.random_integers(1,6)

Ni = [0, 0, 0, 0, 0, 0]  # vecteur et compter
Nsamples = 100000   # nombre de valeurs

for i in range(0, Nsamples):
  r = np.random.random_integers(1,6) # "de"
  Ni[r-1] += 1                              # compter la frequence de r
# Attention : les six valeurs sont dans Ni[0] a Ni[5]

for r in range(0,6):   # et afficher le resultat
  print "frequence de", r+1, Ni[r]/float(Nsamples)
```

Nombres aléatoires non uniformes

Supposons que j'ai un générateur de nombres aléatoires qui donne nombres x distribués dans $[0, 1]$ avec une fonction de distribution de probabilité (FDP) $q(x)$, et je veux obtenir avec une transformation de la variable $y = y(x)$ des nombres aléatoires y distribués comment une autre FDP $p(y)$ défini dans un domaine $[a, b]$.

La probabilité est conservée

Therefore

1. $\int_0^1 q(x)dx = \int_a^b p(y)dy = 1$ —- $q(x)$ et $p(y)$ sond FDP
2. La probabilité d'avoir un nombre aléatoire dans $[x, x + dx]$ est donnée par $q(x)dx$
3. Ceci est conservée si je fais une transformation de variables de x à y
4. Donc $\boxed{q(x)dx = p(y)dy}$ avec $y = y(x)$

Nombres aléatoires non uniformes

Pour simplifier notre dérivation, supposons que x soit un nombre aléatoire distribué uniformemente dans $[0, 1]$ alors $q(x) \equiv 1$. Ainsi nous avons

$$dx = p(y)dy \rightarrow x = \int_a^y p(y')dy' = P(y)$$

où $P(y)$ est la fonction de probabilité cumulative. Nous pouvons inverser cette relation et obtenir

$$\boxed{y = P^{-1}(x)}$$

qui donne le nombre aléatoire souhaité y avec le FDP $p(y)$.

Un exemple simple

Je veux des nombres aléatoires y répartis comment $p(y) = y/2$ dans l'intervalle $[0, 2]$, donc

$$x = P(y) = \int_0^y \frac{y'}{2} dy' = \frac{y^2}{4}$$

et

$$y = P^{-1}(x) = \sqrt{4x}$$

et puisque $x \in [0, 1]$ puis $y \in [0, 2]$ comme souhaité.

Distribution exponentielle

Considérons une distribution de probabilité importante, celle exponentielle

$$p(y) = a\mathrm{e}^{-ay} \quad \text{with} \quad y \in [0; \infty)$$

la distribution cumulative est

$$P(y) = \int_0^y p(y')dy' = 1 - \mathrm{e}^{-ay}$$

Donc

$$x = 1 - \mathrm{e}^{-ay} \rightarrow (1 - x) = \mathrm{e}^{-ay} \rightarrow y = \frac{-\ln(1-x)}{a}$$

puisque $1 - x$ est une variable aléatoire dans $[0, 1]$ ainsi que x, nous pouvons enfin écrire

$$\boxed{y = \frac{-\ln(x)}{a}}$$

Distribution exponentielle

```
import numpy as np
import matplotlib.pyplot as plt

def ranexp (a):
    x=np.random.random_sample()
    y= -np.log(x)/a
    return y

N=1000000
data=[]
for i in range (1,N):
    z=ranexp(0.5)  #a=0.5
    data.append(z)

plt.hist(data, bins=200, range=(0,10),normed=1)
# data contains the number of times you have the random number
# bins is the number of bins you want
# range fix the extrema
# normed tells that you want a histogram with area one

# data contient le nombre de fois que vous avez le nombre aléatoire
```