Échantillonnage des distributions de probabilité

1. Introduction

On s'intéresse à la génération de nombres pseudo aléatoires suivant une distribution de probabilité non uniforme. On dispose d'un générateur de nombres pseudo aléatoires qui délivre une séquence de nombres entiers compris entre 0 et M, avec en principe la même probabilité pour chacun de ces nombres (distribution uniforme). On utilisera le générateur du module python `random`. Ces nombres entiers sont convertis (par division par M) en nombres à virgule flottante compris entre 0 et 1 avec la fonction `random.random()`, ou bien en nombres entiers sur un intervalle quelconque avec la fonction `random.randint(a,b)`.

On considère N évènements référencés par un indice k variant de 0 à $N-1$. La probabilité de l'évènement k est noté p_k et on a :

$$\sum_{k=0}^{N-1} p_k = 1 \tag{1}$$

Cette condition est appelée *condition de normalisation des probabilités*. Dans certains cas, on dispose de probabilités non normalisés, dont la somme n'est pas égale à 1. Il suffit en principe de les diviser par leur somme pour obtenir des probabilités normalisées mais il peut arriver que le calcul de la somme soit trop complexe à effectuer.

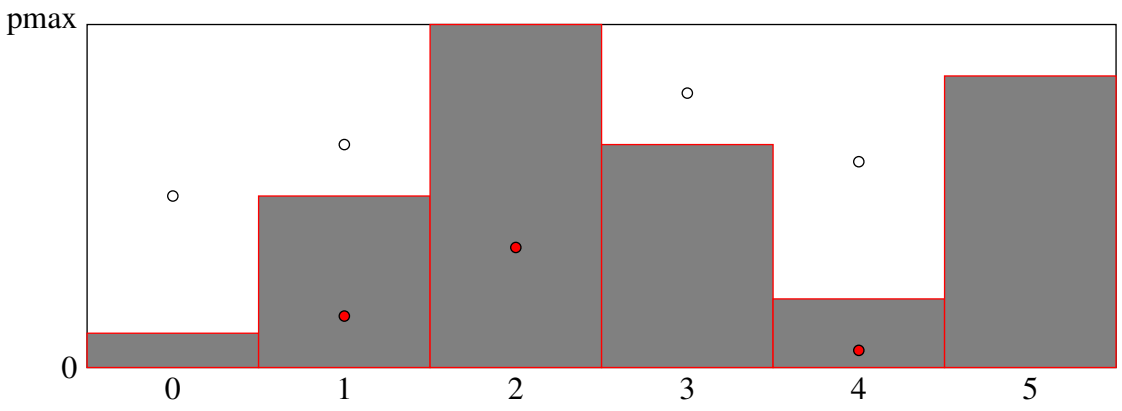
On cherche à tirer des nombres entiers aléatoires compris entre 0 et $N-1$ avec la distribution de probabilité (p_k).

Nous allons voir trois méthodes pour y parvenir : la méthode du rejet, la méthode d'inversion de la fonction de répartition et la méthode de Metropolis.

On verra aussi comment ces méthodes peuvent s'appliquer aux distributions de probabilité continues discrétisées.

2. Méthode du rejet

Soit p_{max} la plus grande probabilité de la distribution (p_k). La figure suivante montre une représentation graphique d'une distribution à 6 évènements.



La méthode du rejet consiste à tirer tout d'abord un nombre aléatoire entier compris entre 0 et $N-1$ avec une probabilité uniforme. Soit k le nombre tiré. Un nombre aléatoire réel x est tiré avec une densité de probabilité uniforme dans l'intervalle $[0, p_{max}]$. Si $x < p_k$ alors le nombre k est accepté, sinon il est rejeté.

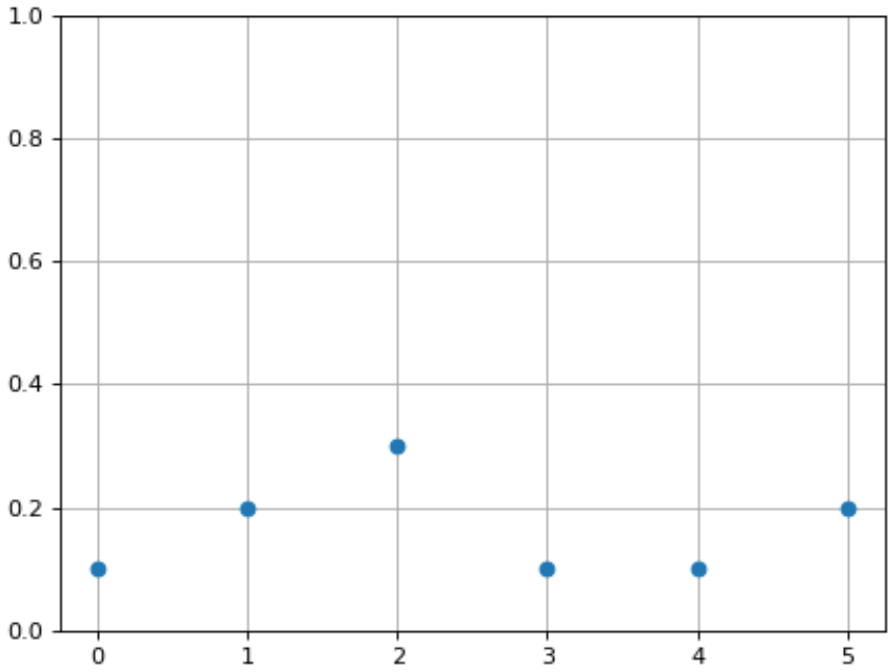
Graphiquement, cela revient à lancer des jetons sur le rectangle $[0, N-1] \times [0, p_{max}]$ et à rejeter les jetons qui tombent en dehors des rectangles coloriés. Sur la figure précédente, 7 tirages sont effectués, et seulement 3 sont acceptés (coloriés en rouge).

On remarque que cette méthode fonctionne aussi avec des probabilités non normalisées.

[1] Écrire une fonction `random_rejet(N,proba,pmax)` qui effectue un tirage aléatoire d'un nombre entier compris entre 0 et N-1, avec les probabilités définies dans la liste `proba`. La valeur maximale des probabilités est aussi donnée en argument.

[2] Tester cette fonction avec les probabilités `proba=[1,2,3,1,1,2]` en effectuant un million de tirages tout en construisant un histogramme, c'est-à-dire à un tableau à N éléments dont l'élément d'indice k contient le nombre de tirages donnant la valeur k. Les valeurs de l'histogramme seront finalement divisées par le nombre de tirages, ce qui permet d'obtenir les valeurs des probabilités (normalisées). Tracer l'histogramme avec : `plot(histogramme,"o")`.

```
def random_rejet(N,proba,pmax):
    fin = False
    while not fin:
        k = random.randint(0,N-1)
        x = random.random()*pmax
        fin = x <= proba[k]
    return k

proba = [1,2,3,1,1,2]
random.seed(143652)
pmax = max(proba)
N = len(proba)
histogramme = [0]*N
Ntir = int(1e6)
for i in range(Ntir):
    k = random_rejet(N,proba,pmax)
    histogramme[k] += 1
histogramme = numpy.array(histogramme)/Ntir
figure()
plot(histogramme,"o")
ylim(0,1)
grid()
```



La méthode du rejet est acceptable lorsque le taux de rejet est faible, c'est-à-dire lorsque la distribution est presque uniforme. Elle devient très inefficace lorsque le taux de rejet est élevé.

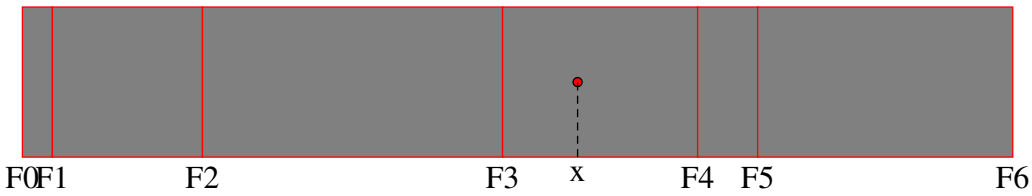
3. Inversion de la fonction de répartition

La fonction de répartition (ou fonction de distribution cumulative) est définie par :

$$F_0 = 0 \tag{2}$$
$$F_1 = p_0 \tag{3}$$
$$F_2 = p_0 + p_1 \tag{4}$$
$$\ldots \tag{5}$$
$$F_n = \sum_{k=0}^{n-1} p_k \tag{6}$$
$$\ldots \tag{7}$$
$$F_N = \sum_{k=0}^{N-1} p_k \tag{8}$$

Plus précisément, la fonction de répartition associe à l'entier n ($0 \leq n \leq N$) le nombre F_n, qui est la probabilité d'obtenir un évènement dans l'intervalle $[0, n-1]$. Le dernier éléments est F_N, qui vaut 1 si les probabilités sont normalisées (mais ce n'est pas nécessaire pour l'algorithme d'inversion). Les probabilités étant positives, la fonction de partition est croissante.

Voyons tout d'abord une approche graphique. Au lieu de disposer des rectangles de hauteur p_k côte à côte, on empile horizontalement des rectangles de largeur p_k.



Des tirages sont faits dans le rectangle (de largeur F_N) : un nombre réel x aléatoire est généré dans l'intervalle $[0, F_N]$ avec une densité de probabilité uniforme. Il faut déterminer dans quel rectangle le point correspondant est situé, c'est-à-dire trouver l'indice k tel que :

$$F_k \leq x < F_{k+1} \tag{9}$$

L'indice k est alors le nombre aléatoire obtenu pour ce tirage. La détermination de k revient à chercher l'indice k correspondant à une certaine valeur de la fonction de partition, c'est pourquoi on parle d'inversion de la fonction de partition.

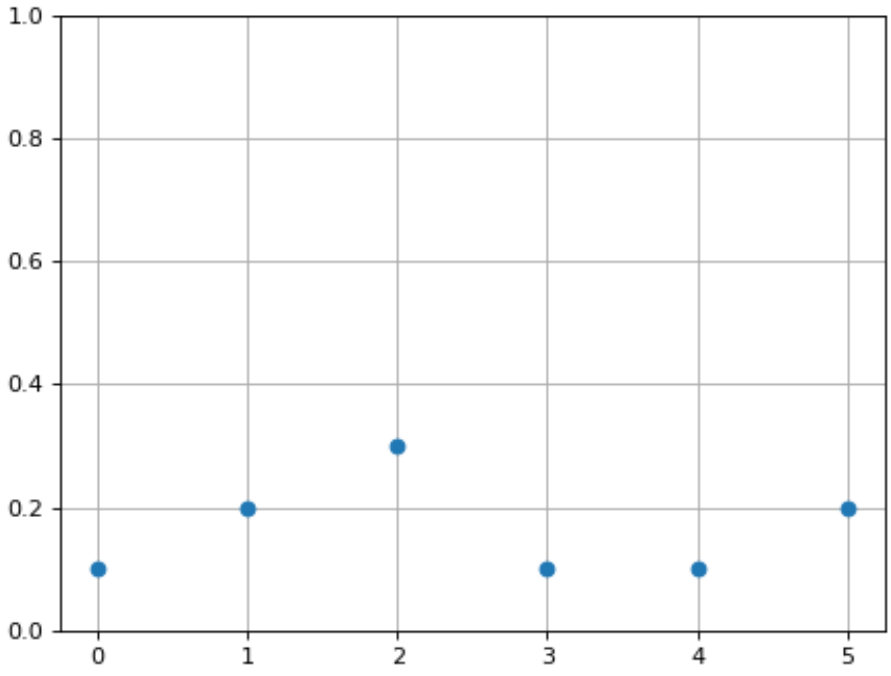
[3] Écrire une fonction `fonction_repartition(N,proba)` qui renvoie le tableau des N+1 valeurs de la fonction de répartition d'une distribution de probabilité donnée.

[4] Écrire une fonction `random_inversion(repartition)` qui effectue un tirage aléatoire en utilisant la fonction de répartition donnée. L'indice k sera recherché en parcourant les éléments du tableau `repartition` dans l'ordre.

[5] Tester cette fonction en calculant un histogramme pour un million de tirages.

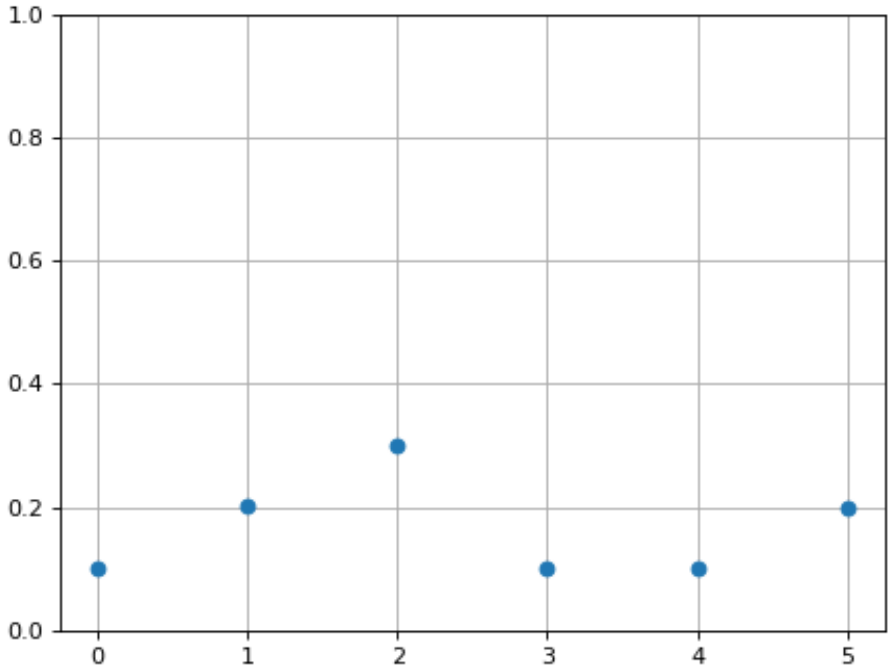
[6] Lorsque N est grand, il est préférable de rechercher l'indice k par dichotomie. Écrire une fonction `random_inversion_dichotomie(repartition)` qui utilise cet algorithme puis la tester.

```
def fonction_repartition(N,proba):
    repartition = [0]
    for k in range(1,N+1):
        repartition.append(repartition[k-1]+proba[k-1])
    return repartition

def random_inversion(repartition):
    x = random.random()*repartition[-1]
    k = 0
    while x>=repartition[k+1]:
        k += 1
    return k

repartition = fonction_repartition(N,proba)
N = len(proba)
histogramme = [0]*N
Ntir = int(1e6)
for i in range(Ntir):
    k = random_inversion(repartition)
    histogramme[k] += 1
histogramme = numpy.array(histogramme)/Ntir
figure()
plot(histogramme,"o")
ylim(0,1)
grid()
```




```
def random_inversion_dichotomie(repartition):
    x = random.random()*repartition[-1]
    kmin = 0
```

```
    kmax = len(repartition)-1
    fin = False
    while not fin:
        k = (kmin+kmax)//2
        if repartition[k+1]<=x:
            kmin = k
        elif x<repartition[k]:
            kmax = k
        else:
            fin = True
    return k

histogramme = [0]*N
Ntir = int(1e6)
for i in range(Ntir):
    k = random_inversion_dichotomie(repartition)
    histogramme[k] += 1
histogramme = numpy.array(histogramme)/Ntir
figure()
plot(histogramme,"o")
ylim(0,1)
grid()
```



Cette méthode peut être utilisée pour échantillonner une distribution de probabilité continue. On considère par exemple une variable aléatoire réelle à densité (notée x), définie sur l'intervalle $[0, 1]$, dont la densité de probabilité est donnée par :

$$p(x) = 2\cos^2(4\pi x) \tag{10}$$

La probabilité d'obtenir une valeur de x dans l'intervalle $[a, b]$ est :

$$P(a \leq x \leq b) = \int_a^b p(x)dx \tag{11}$$

La distribution être tout d'abord être discrétisée, c'est-à-dire convertie en une distribution de probabilités discrètes à N valeurs, réparties régulièrement sur l'intervalle $[0, 1]$.

Soit $\Delta x = \frac{1}{N}$ l'intervalle de discrétisation. La probabilité p_k est la probabilité d'obtenir une valeur comprise entre $k\Delta x$ et $(k+1)\Delta x$, c'est-à-dire :

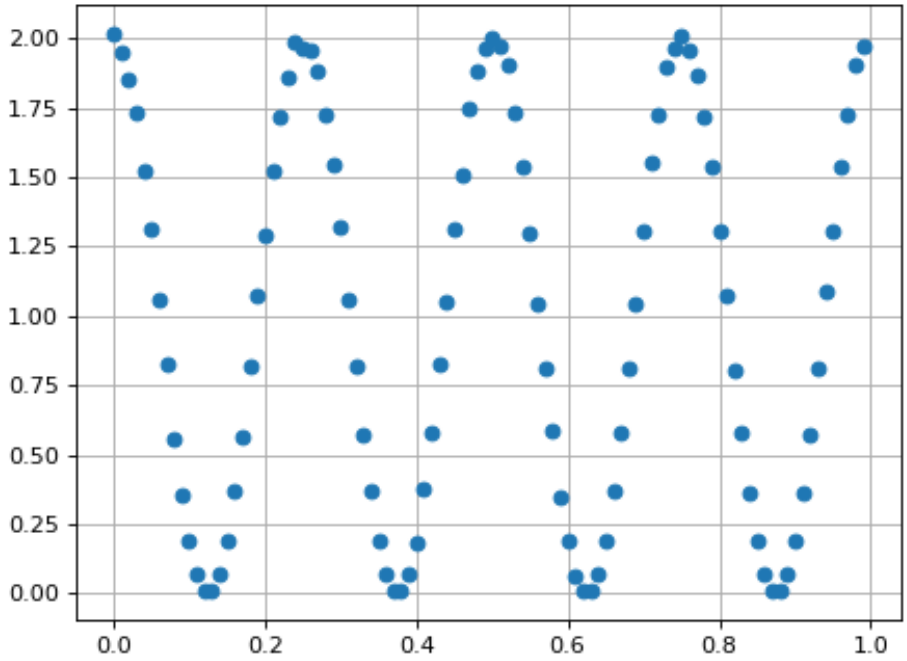
$$p_k = \int_{k\Delta x}^{(k+1)\Delta x} p(x)dx \approx p(k\Delta x)\Delta x \tag{12}$$

L'histogramme normalisé donne les probabilités p_k. Pour retrouver la densité de probabilité à partir de cet histogramme, il faut diviser les valeurs par Δx.

[7] Générer le tableau `proba` des probabilités p_k correspondant à la densité (10), avec $N = 100$.

[8] Effectuer un million de tirage en calculant l'histogramme. Tracer la densité de probabilité en fonction de x à partir de cet histogramme.

```
N=100
k = numpy.arange(N)
x = k/N
delta_x = 1/N
proba = 2*numpy.cos(4*numpy.pi*k/N)**2*delta_x
repartition = fonction_repartition(N,proba)
histogramme = [0]*N
Ntir = int(1e6)
for i in range(Ntir):
    k = random_inversion_dichotomie(repartition)
    histogramme[k] += 1
histogramme = numpy.array(histogramme)/Ntir*N
figure()
plot(x,histogramme,"o")
grid()
```



4. Méthode de Metropolis

La méthode de Metropolis a été introduite pour effectuer des calculs de physique statistique. Voir Algorithme de Metropolis pour une présentation détaillée de la méthode. On se contente ici de donner un exemple d'utilisation pour obtenir l'échantillonnage d'une distribution non uniforme.

La méthode consiste à générer une suite de nombres aléatoires. Un nombre aléatoire ne dépend que du nombre précédent (chaîne de Markov). Un nombre k' s'obtient à partir du précédent k de la manière suivante :

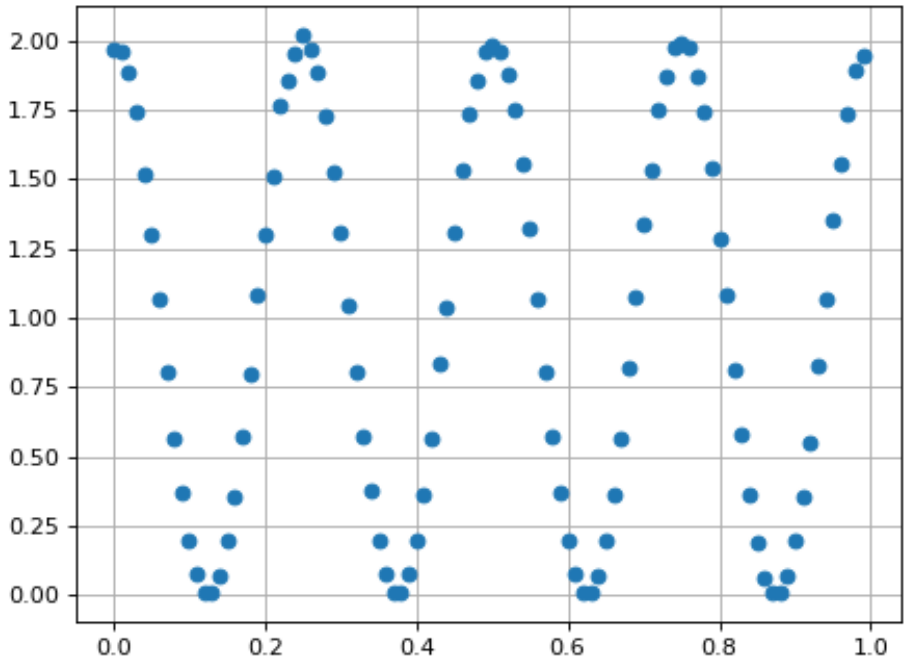
- ▷ Tirage aléatoire d'un nombre k' à partir de k avec une probabilité de sélection. Dans le cas présent, le plus simple est de tirer k' uniformément sur l'intervalle $[0, N-1]$ (sélection indépendante de l'état précédent).
- ▷ Si $p_{k'} > p_k$, le nombre k' est accepté.
- ▷ Si $p_{k'} < p_k$, le nombre k' est accepté avec la probabilité $p_{k'}/p_k$. Pour cela, on tire un nombre aléatoire réel dans l'intervalle [0,1] avec une distribution uniforme et on le compare à ce rapport.
- ▷ Si k' est refusé alors $k' = k$.

Il faut remarquer que si le nouveau nombre k' est refusé alors le nombre suivant est identique au précédent, ce qui est différent de la méthode du rejet, où un nouveau tirage doit être fait à chaque fois qu'une valeur est rejetée.

[9] Écrire une fonction `ramdom_metropolis(N,proba,k)` qui renvoie la valeur de k' pour un k donné.

[10] Utiliser cette fonction pour faire un million de tirages avec la densité de probabilité (10). Tracer la densité de probabilité.

```
def random_metropolis(N,proba,k):
    i = random.randint(0,N-1)
    if proba[i] >= proba[k]:
        return i
    else:
        x = random.random()
        if x < proba[i]/proba[k]:
            return i
        return k

histogramme = [0]*N
Ntir = int(1e6)
k = N//2
for i in range(Ntir):
    histogramme[k] += 1
    k = random_metropolis(N,proba,k)
histogramme = numpy.array(histogramme)/Ntir*N
figure()
plot(x,histogramme,"o")
grid()
```



La méthode d'échantillonnage de Metropolis est intéressante lorsque le nombre N est extrêmement grand. Elle a l'inconvénient de nécessiter un certain nombre d'itérations avant d'obtenir des nombres distribués effectivement avec les probabilités données.