Méthode de Monte-Carlo - loi faible des grands nombres

Corrigé Code de partage avec Capytale : d31a-9959885

On utilisera les bibliothèques suivantes :

```
import numpy as np
import numpy.random as rd
```

Une méthode de Monte-Carlo consiste à faire un grand nombre de réalisations numériques d'un phénomène aléatoire pour approcher une espérance. L'idée est fondée sur la loi faible des grands nombres, dont la traduction pratique consiste à dire que, étant donnée une variable admettant un moment d'ordre 2, $(x_k)_{k\in\mathbf{N}}$ une suite de réalisations indépendantes de la loi de X, alors, pour n « grand » :

$$\frac{1}{n}\sum_{k=1}^{n} x_k \approx E(X)$$

et plus n augmente, meilleure est l'approximation d'après la loi faible des grands nombres

Comme nous l'avons vu, la valeur $\dfrac{1}{n}\displaystyle\sum_{k=1}^{n} x_k$ est appelée **moyenne empirique**.

Exercice 1

1. créer une fonction `Tirage1(p)` qui simule la loi $\mathscr{B}(p)$ modélisant le jeu du Pile ou Face avec une probabilité p d'obtenir Pile sur un lancer.

   Il s'agit simplement de simuler une loi de Bernoulli de paramètre p

   ```
   def Tirage1(p):
       return rd.binomial(1,p)
   ```

2. créer une fonction `Simulation(N, p)` qui simule N lancers et renvoie la fréquence d'apparition de Pile sur ces N lancers.

   On utilisera une boucle `for` et un compteur Pile qui comptera le nombre de Pile.

   On vérifiera que la valeur renvoyée se rapproche de p lorsqu'on augmente la valeur de N

   On crée un compteur et on réalise simplement 100 fois la fonction précédente (pour cela on aurait pu utiliser une loi binomiale) et on ajoute le résultat de la fonction `Tirage1` (qui renvoie 1 en cas de succès). La fonction renvoie le nombre de « succès » divisé par le nombre d'expérience (la fréquence).

   ```
   def Simulation(N, p):
       c=0
       for i in range(N):
           c=c+Tirage1(p)
     return c/N
   ```

   On peut ensuite tester, par exemple avec `Simulation(100,0.7)`, `Simulation(1000,0.7)`...

Exercice 2

On considère une urne contenant n boules noires et b boules blanches ($b \geqslant 2$). On effectue des tirages sans remise d'une boule dans l'urne et on note :
- X le rang d'apparition de la première boule blanche,
- Y le rang d'apparition de la deuxième boule blanche.

1. Définir une fonction `SimX(n, b)` qui simule une réalisation de la variable X

   Attention il ne s'agit pas d'une loi géométrique puisque les tirages successifs se font sans remise.
   On peut tester par exemple avec `[SimX(8,2) for k in range(10)]`

```
def SimX(n, b):
    t=n+b # total de boules
    c=1 # compteur de tirages
    while rd.randint(1,t+1)>b : # on considère que les b premières sont les
   blanches
        t=t-1 # on retire une boule (noire)
        c=c+1
    return c
```

2. Compléter la fonction suivante simulant une réalisation de Y

   Une fois la première boule blanche, il s'agit à nouveau de la même expérience avec des conditions initiales différentes : si x est le rang d'apparition de la première boule blanche, cela signifie qu'une boule blanche a été tirée et $x-1$ noires, il reste alors $b-1$ boules blanches et $n-(x-1)$ boules noires, d'où le programme suivant (qui réutilise la fonction précédente) :

```
def SimY(n, b):
    x=SimX(n,b)
    return x+SimX(n-x+1,b-1)
```

   On peut tester par exemple avec `[SimY(8,2) for k in range(10)]`
   Dans ce cas, Y ne peut prendre que des valeurs entre 2 et N

3. Vérifier que, pour tous entiers k et ℓ tels que $1 \leqslant k \leqslant \ell$:

$$\binom{\ell}{k} = \frac{\ell}{k}\binom{\ell-1}{k-1}$$

   Pour démontrer l'égalité, il suffit d'utiliser la formule des coefficients binomiaux avec les factorielles.

   Puis définir une fonction récursive `coefbin(k, l)` qui calcule le coefficient binomial $\binom{\ell}{k}$

   On utilise la formule précédente et il faut donner une condition d'arrêt (si $k = 0$) :

```
def coefbin(k,l):
    if k==0:
        return 1
    else :
        return l/k*coefbin(k-1,l-1)
```

4. On considère la fonction suivante :

```
def simulationX(n, b):
    L=np.zeros(n+2)
    for i in range(10000):
        x= SimX(n, b)
        L[x] +=1/10000
    return L
```

(a) Que contient le vecteur ligne `L` ?

Le vecteur ligne contient les fréquences de chaque rang d'apparition après 10 000 expériences

(b) On montre que : $P(X=k) = \frac{\binom{n-k+b}{b-1}}{\binom{n+b}{b}}$ pour $k \in \{1, \dots n+1\}$

Ecrire une fonction `LoiX(n, b)` renvoyant un vecteur ligne contenant les valeurs $P(X = 0), \dots, P(X = n)$
On utilisera la fonction `coefbin` et on remarquera que $P(X = 0) = 0$

On utilise la formule ci-dessus et on crée de manière itérative les termes de la liste (on aurait aussi pu le faire avec l'écriture synthétique) :

```
def LoiX(n, b):
    L=[0]
    for k in range(1,n+2):
        L.append(coefbin(b-1,n-k+b)/coefbin(b,n+b))
    return L
```

On peut tester avec `LoiX(8,2)`

(c) Vérifier le résultat précédent par simulation à l'aide de la fonction `simulationX`

On exécute `simulationX(8,2)` et on vérifie que les résultats de la simulation (de l'expérience) sont proches des valeurs théoriques de la question précédente.
Pour bien faire, on représente les deux séries de valeurs à l'aide de diagrammes en bâtons.

```
x=[k for k in range(10)]
y1=LoiX(8,2)
y2=simulationX(8,2)
import matplotlib.pyplot as plt
plt.bar(x,y1)
plt.bar(x,y2,width=0.5)
plt.show()
```

5. On veut compléter la fonction `simulationX` pour approcher la loi de Y

(a) Compléter cette fonction.

```
def SimulationXY(n, b):
    LXY=np.zeros([n+3, n+3])
    for i in range(10000):
        x= SimX(n, b)
        y = ...
        LXY[x, y]= ...
    return LXY
```

(b) On montre que : $P(Y=i) = (i-1) \times \frac{\binom{n+b-i}{b-2}}{\binom{n+b}{b}}$ pour $i \in \{2, \dots n+2\}$

Vérifier ce résultat par simulation.

6. Compléter la fonction suivante pour simuler la loi du couple (X, Y)

```
def SimulationXY(n, b):
    LXY=np.zeros([n+3, n+3])
    for i in range(10000):
        x= SimX(n, b)
        y = ...
        LXY[x, y]= ...
    return LXY
```