Correction - TP 4

Simulation de variables aléatoires discrètes

Exercice 1

1. Tout d'abord, `rd.random()` donne un nombre réel aléatoire de $]0,1[$ (la probabilité d'obtenir 0 ou 1 est nulle). Donc `n*rd.random()` donne un nombre réel aléatoire de $]0,n[$. Donc Donc `np.floor(n*rd.random())` donne un nombre entier aléatoire de $[\![0,n-1]\!]$.

   Soit $k \in [\![0,n-1]\!]$. Alors `np.floor(n*rd.random())` est égal à k si `n*rd.random()` est compris entre k et $k+1$ donc si `rd.random()` est compris entre $\frac{k}{n}$ et $\frac{k+1}{n}$. Avec la propriété 1, la probabilité que le réel renvoyé par `rd.random()` soit compris entre $\frac{k}{n}$ et $\frac{k+1}{n}$ vaut $\frac{k+1}{n} - \frac{k}{n} = \frac{1}{n}$.

   La commande `np.floor(n*rd.random())` simule donc bien une loi uniforme $\mathcal{U}([\![0,n-1]\!])$.

2. (a) Voici la fonction demandée :

```
def uniforme(a,b):
    return a+(b-a)*np.floor(n*rd.random())
```

   (b) Voici la fonction demandée :

```
def Uniforme(a,b,N):
    v = np.zeros(N) ;
    for k in range(N)
        v[k] = uniforme(a,b)
    return v
```

Exercice 2

On effectue des lancers de dés successifs à l'aide de la commande `rd.random(1,7)` qui simule une réalisation d'une loi $\mathcal{U}([\![1,6]\!])$.

Il faut qu'à chaque lancer, on garde une trace du numéro obtenu pour savoir si on l'obtient de nouveau ensuite. On va pour cela créer un vecteur `u = np.zeros(6)` avec des zéros pour commencer (puisqu'on a obtenu zéro fois chaque entiers $1, \dots, 6$), la composante `u[k-1]` correspondant au nombre de fois où l'entier k est apparu. À chaque fois qu'on lance le dé avec la commande `k = rd.random(1,7)`, on augmente la $k-1$-ème composante du vecteur `u` de 1 à l'aide de la commande `u[k-1] = u[k-1]+1`.

On continue à lancer le dé tant qu'on ne tombe pas sur un numéro déjà obtenu, ce qu'on testera avec la condition `np.max(u)<=1`. On va donc faire une boucle `while` avec cette condition. On stockera enfin le nombre de lancers de dés dans la variable `n` qu'on renverra à la fin.

On propose donc le script suivant (on nous demande un programme et pas une fonction !) :

```
u = np.zeros(7)
n = 0
while np.max(u) <=1 :
    k = rd.randint(1,7)
    u[k-1] = u[k-1]+1
    n = n+1
print(n)
```

Exercice 3

1. On peut procéder ainsi :

```
def bernoulli(p):
    u = 0
```

```
    if rd.random()<=p:
        u = 1
    return u
```

2. On procède comme précédemment :

```
def Bernoulli(p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = bernoulli(p)
    return v
```

Exercice 4

La condition `rd.random()<3/7` est réalisée avec une probabilité de 3/7. Elle correspond à la probabilité de l'évènement ¡¡ tirer une boule blanche au premier tirage ¿¿. Si cet évènement est réalisé, alors on attribue à X la valeur 1, sinon on lui attribue la valeur 0.

À la suite du premier tirage, il reste dans l'urne $3-X$ boules blanches et 6 boules en tout dans l'urne, soit une proportion de $(3-X)/6$ boules blanches dans l'urne. On a donc une probabilité de $(3-X)/6$ de tirer une boule blanche au deuxième tirage. Si `rd.random()<(3-X)/6` est réalisé, ce qui arrive avec une probabilité de $(3-X)/6$, on attribue alors à Y la valeur 1. Sinon, on lui attribue la valeur 0.

On peut maintenant compléter le programme comme suit.

```
if rd.random()<3/7 :
     X = 1
else :
     X = 0
if rd.random()< (3-X)/6 :
     Y = 1
else :
     Y = 0
print (X,Y)
```

Exercice 5

1. On peut procéder comme suit, en utilisant la fonction `Bernoulli` définie précédemment.

```
def binomiale(n,p):
    u = Bernoulli(p,n)
    return np.sum(u)
```

   Une autre possibilité est d'utiliser la commande `np.sum(rd.random(n)<=p)`.

```
def binomiale(n,p):
    return np.sum(rd.random(n)<=p)
```

2. Toujours la même méthode.

```
def Binomiale(n,p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = binomiale(n,p)
    return v
```

3. On peut procéder ainsi :

```
>>> u = Binomiale(10,0.2,10000)
>>> np.mean(u)
```

On obtient (par exemple) `2.0421`. Cela correspond approximativement à ce qu'on attend. En effet, l'espérance d'une variable suivant une loi binomiale de paramètres $n = 10$ et $p = 0,2$ est $n \times p = 2$.

Exercice 6

On commence par lancer un dé équilibré à n faces, ce qui revient à exécuter la commande `rd.randint(1,n+1)` (loi uniforme sur $[\![1, n]\!]$) et on stocke le résultat dans la variable `k`. On effectue alors n tirages avec remise dans l'urne U_k, et on compte le nombre de boules blanches obtenues.
Ceci correspond à une loi binomiale $\mathcal{B}(n, \frac{k}{n})$ ($\frac{k}{n}$ étant la proportion de boules blanches dans l'urne k), et correspond à exécuter la commande `rd.binomial(n,k/n)`.

On propose donc la fonction suivante.

```
def simul(n):
    k = rd.randint(1, n+1)
    x = rd.binomial(n, k/n)
    return x
```

Exercice 7

1. On peut procéder comme suit :

```
def geom(p):
    u = 1
    while rd.random()>p:
        u=u+1
    return u
```

   On peut remplacer la commande `rd.random()>p` par `bernoulli(p)==0`.

2. Toujours pareil.

```
def Geom(p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = geom(p)
    return v
```

3. On peut procéder ainsi :

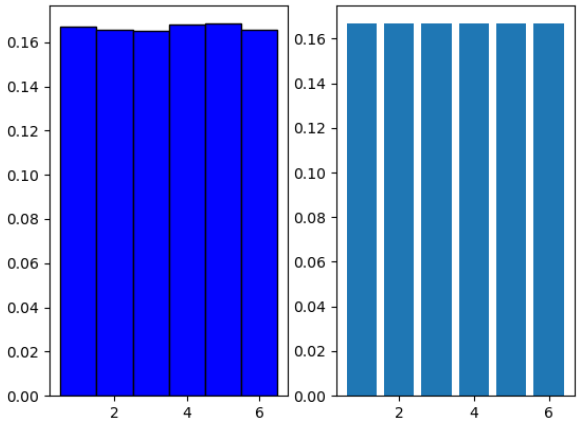
```
>>> u = Geom(0.2,10000)
>>> np.mean(u)
```

   On obtient (par exemple) `5.0013`. Cela correspond approximativement à ce qu'on attend. En effet, l'espérance d'une variable suivant une loi géométrique de paramètre $p = 0,2$ est $\frac{1}{p} = 5$.

Exercice 8

Le script commence par créer et stocker dans la variable `x` un échantillon de taille 100000 à l'aide de la fonction `uniforme` définie au début du TP, avec $n = 1$ et $m = 7$. Elle trace ensuite le diagramme en bâton des fréquences empiriques de cet échantillon. Pour cela, on définit les classes dans la variable `c`. Elle trace enfin le diagramme en bâton des probabilités théoriques. En particulier, le vecteur `v` contient `[1/6,1/6,1/6,1/6,1/6,1/6]`, qui correspond bien aux probabilités théoriques d'une loi uniforme sur $[\![1, 6]\!]$.

En exécutant le code proposé, on obtient la représentation graphique suivante :



Les diagrammes en bâtons étant très similaires, on peut en conclure que la fonction `uniforme` renvoie bien ce que l'on souhaite : des entiers de 1 à 6 avec une fréquence de 1/6 environ. La simulation de la loi $\mathcal{U}(\llbracket 1,6 \rrbracket)$ à l'aide de la fonction `uniforme(1,7)` est bien pertinente.

Exercice 9

1. En exécutant ces commandes pour $n = 3$, on obtient `c = np.array([1,3,3,1])`, et pour $n = 4$, `c = np.array([1,4,6,4,1])`. On reconnait la liste des coefficients binomiaux $\binom{n}{k}$. Tentons de l'expliquer.

   `np.arange(n,0,-1)` est le vecteur `[n,n-1,...,2,1]`. Et donc :

   `np.arange(n,0,-1)/np.arange(1,n+1)`

   est le vecteur `[n/1,(n-1)/2,...,2/(n-1),1/n]`. Par conséquent, si on applique à ce vecteur la fonction `np.cumprod`, le vecteur obtenu a pour k-ème composante :

   $$\frac{n}{1} \times \cdots \times \frac{n-k+1}{k} = \frac{n(n-1)\dots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}\binom{n}{k}.$$

   Il nous manque alors dans le vecteur le cas où $k = 0$, qu'on ajoute à l'aide de la commande `np.ones(n+1)`, dont la 0-ième composante vaut 1.

2. On commence par générer un échantillon de taille $N = 100000$ grâce à la fonction `Binomiale`, et tracer l'histogramme (diagramme en bâtons) des fréquences empiriques de l'échantillon.

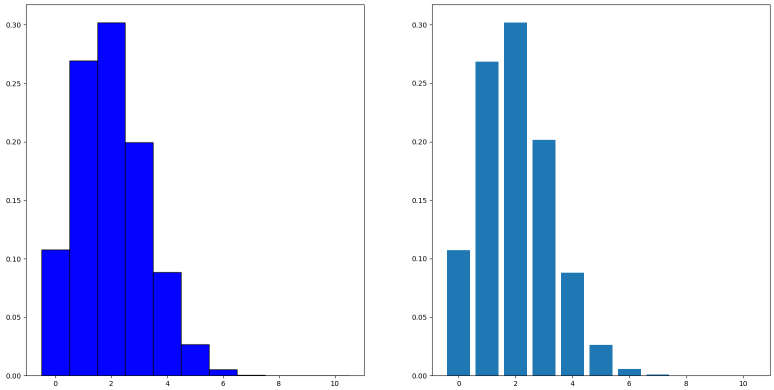
```
# Echantillon
p = 0.2
n = 10
N = 100000
x = Binomiale(n,p,N)

#DeB des fréquences
c = np.arange(-0.5,11)
plt.subplot(1,2,1)
plt.hist(x,c,density='True',edgecolor='k',color='blue',label="DeB
 des fréq")
```

   On trace ensuite le diagramme en bâtons des probabilités théoriques. Pour cela, on construit un vecteur `v` dont la k-ème composante est $\binom{10}{k}p^k(1-p)^{10-k}$.

```
#DeB des probas théoriques
s = np.arange(0,11,1)
c = np.ones(n+1)
c[1:(n+1)] = np.cumprod(np.arange(n,0,-1)/np.arange(1,n+1))
u = p**np.arange(0,11)
v = (1-p)**np.arange(10,-1,-1)
t = c*u*v
```

```
18
19 plt.subplot(1,2,2)
20 plt.bar(s,t,label="DeB des prob theo")
21 plt.show()
```

On obtient la représentation graphique suivante.



On observe que les diagrammes en bâtons sont pratiquement identique. Il est donc pertinent de simuler la loi $\mathcal{B}(10, 0.2)$ à l'aide de la fonction `binomiale`.

Exercice 10

1. On sait que si $X \hookrightarrow \mathcal{G}(p)$, alors $X(\Omega) = \mathbb{N}^*$ et $P(X = k) = p \times (1-p)^{k-1}$. On peut par exemple donner l'instruction suivante :

```
>>> t = p*(1-p)**np.arange(0,20)
```

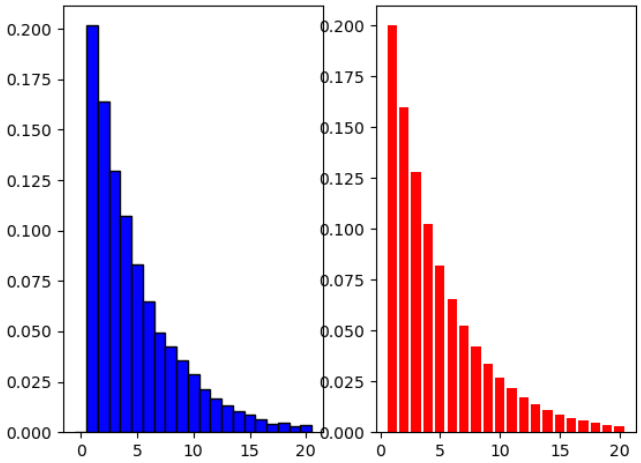
2. On commence par générer un échantillon de taille $N = 100000$ grâce à la fonction `Geom`, et tracer l'histogramme (diagramme en bâtons) des fréquences empiriques de l'échantillon.

```
1 # Echantillon
2 p = 0.2
3 N = 100000
4 x = Geom(p,N)
5
6 #DeB des fréquences
7 c = np.arange(-0.5,21)
8 plt.subplot(1,2,1)
9 plt.hist(x,c,density='True',edgecolor='k',color='blue',label="DeB
   des fréq")
```

On trace ensuite le diagramme en bâtons des probabilités théoriques :

```
10 #DeB des probas théoriques
11 s = np.arange(1,21)
12 t = p*(1-p)**np.arange(0,20)
13
14 plt.subplot(1,2,2)
15 plt.bar(s,t,label="DeB des prob theo")
16 plt.show()
```

On obtient la représentation graphique suivante.



On observe que les diagrammes en bâtons sont pratiquement identique. Il est donc pertinent de simuler la loi $\mathcal{G}(0.2)$ à l'aide de la fonction `geom`.

Exercice 11

1. Sachant $(N = k)$, on réalise k expérience de Bernoulli indépendantes dont la probabilité de succès est toujours égale 0.2. Donc $C \hookrightarrow \mathcal{B}(k, 0.2)$.

2. Voici le programme demandé :

```
1 def simulC():
2     N = rd.poisson(10)
3     C = rd.binomial(N,0.2)
4     return(C)
```

3. On ajoute à la suite du programme précédent les instructions :

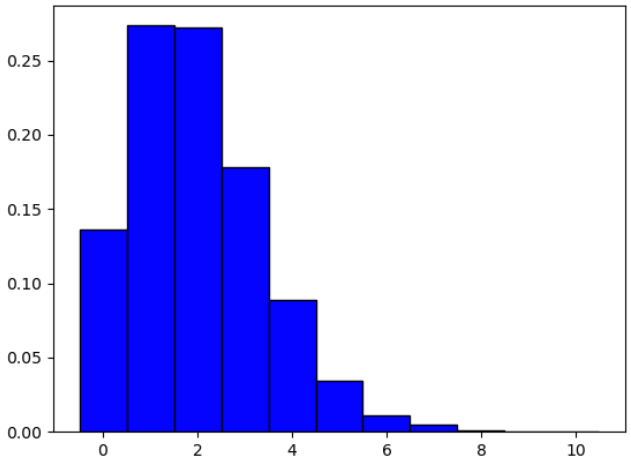
```
6 C = np.zeros(10000)
7 for k in range(10000):
8     C[k] = simulC()
```

4. (a) Après exécution de la commande `np.mean(C<=10)`, on obtient comme résultat `1.`. Ceci signifie que tous les éléments de `C` sont entre 0 et 10. On peut donc définir des classes entre 0 et 10. Donc `c = np.arange(-0.5,11)`.

   (b) Pour tracer le diagramme en bâtons des fréquences :

```
10 c = np.arange(-0.5,11)
11 plt.hist(C,c,density='True',edgecolor='k',color='blue',label=
   "DeB des fréq")
12 plt.show()
```

On obtient la représentation graphique suivante.



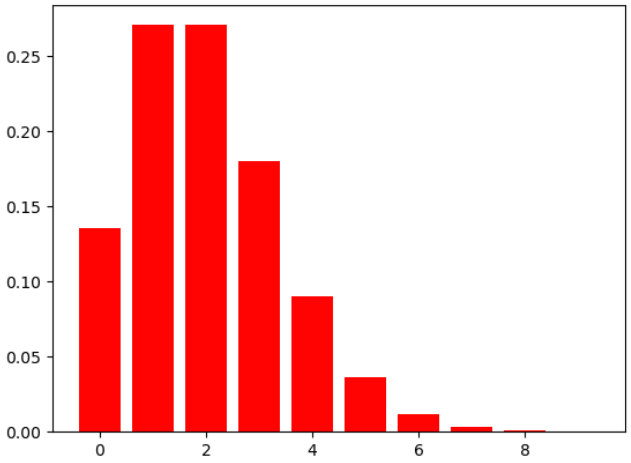
5. (a) On utilise les instructions suivantes :

```
t = np.zeros(10)
t[0] = np.exp(-2)
for k in range(1,10):
    t[k] = np.exp(-2)*(2**k)/np.prod(np.arange(1,k+1))
```

(b) On trace le diagramme en bâtons associé :

```
14 s = np.arange(0,10)
15 t = np.zeros(10)
16 t[0] = np.exp(-2)
17 for k in range(1,10):
18     t[k] = np.exp(-2)*(2**k)/np.prod(np.arange(1,k))
19 plt.bar(s,t,label="DeB des prob theo")
20 plt.show()
```

On obtient la représentation graphique suivante.



On remarque que les deux diagrammes en bâtons obtenus aux questions 4.(b) et 5.(b) sont semblables. On peut donc conjecturer que $C \hookrightarrow \mathcal{P}(2)$.

Exercice 12

1. On découpe l'intervalle $[0,1]$ en 6 sous-intervalles $I_k =]\frac{k-1}{6}, \frac{k}{6}]$ de même longueur $1/6$ où $k = 1, \ldots, 6$ (on prendra $I_1 = [0, 1/6]$).

2. On détermine une réalisation t d'une loi uniforme sur $[0,1]$ à l'aide de la fonction `rd.random()`. On détermine l'intervalle I_k contenant t, et on renvoie l'entier k. On obtient la fonction suivante.

```
def unif():
    t = rd.random()
    k = 1
    while t>k/6:
        k = k+1
    return k
```

Lorsque la boucle `while` s'arrête, la variable k satisfait $t \leq \frac{k}{6}$ et $t > \frac{k-1}{6}$, donc t appartient à l'intervalle I_k. Cet évènement se produit avec une probabilité de $\dfrac{1}{6}$. On renvoie alors la valeur k.

Exercice 13

1. Selon la méthode d'inversion expliquée ci-dessus, on renverra la valeur $k = 0$ si t appartient à l'intervalle $I_0 = [0, \frac{\lambda^0}{0!}e^{-\lambda}]$, et la valeur $k \in \mathbb{N}^*$ si t appartient à l'intervalle $I_k = \left]\displaystyle\sum_{i=0}^{k-1} \frac{\lambda^i}{i!}e^{-\lambda}, \sum_{i=0}^{k} \frac{\lambda^i}{i!}e^{-\lambda}\right]$.
2. On cherche à l'aide d'une boucle `while` l'intervalle I_k contenant t. On renvoie alors l'entier k. Notons que les variables u et s contiennent respectivement $\dfrac{\lambda^k}{k!}$ et $\displaystyle\sum_{i=0}^{k} \frac{\lambda^i}{i!}e^{-\lambda}$ à chaque nouveau passage dans la boucle.
3. On procède comme d'habitude.

```
def Poisson(lbd,N):
    X = np.zeros(N)
    for k in range(N):
        X[k] = poisson(lbd)
    return X
```

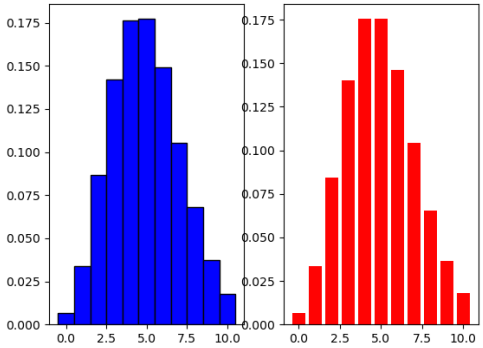
4. On utilise le code suivant pour tracer le diagramme en bâtons des fréquences empiriques

```
# Echantillons
lbd = 5
N = 100000
x = Poisson(lbd,N)

#DeB des fréquences
c = np.arange(-0.5,11)
plt.subplot(1,2,1)
plt.hist(x,c,density='True',edgecolor='k',color='blue',label="DeB
 des fréq")
```

On trace ensuite le diagramme en bâtons des probabilités théoriques. Pour cela, il faut construire un vecteur `u` contenant en k-ème composante $\frac{\lambda^k}{k!}e^{-\lambda}$. Il faut traiter le $k = 0$ à part, ce qu'on fait avec la première commande. Le vecteur `v` contient ensuite les λ^k, et le vecteur `w` les $k!$. Reste alors à faire les opérations coefficients par coefficients.

```
#DeB des probas théo
u = np.exp(-lbd)*np.ones(11)
v = lbd**np.arange(1,11)
w = np.cumprod(np.arange(1,11))
u[1:11] = np.exp(-lbd)*v/w

s = np.arange(11)

plt.subplot(1,2,2)
plt.bar(s,u,color='red',label="DeB des prob theo")
plt.show()
```

On obtient la représentation graphique suivante :



Les diagrammes en bâtons étant presque identiques, on en déduit que les simulations renvoyées par la fonction `poisson` sont pertinentes.

Exercice 14

On propose la fonction suivante.

```
def geom3(p):
    t = rd.random()
    k = 1
    u = p
    s = u
    while s<t :
        k = k+1
        u = u*(1-p)
        s = s+u
    return k
```

La variable `u` contient à chaque étape $(1-p)^{k-1}p$. La variable s contient à chaque étape $\sum_{i=0}^{k}(1-p)^{i-1}p$. La boucle `while` permet de déterminer l'entier k satisfaisant :

$$\sum_{i=0}^{k-1}(1-p)^{i-1}p < t \leq \sum_{i=0}^{k}(1-p)^{i-1}p.$$

Et on a bien une probabilité de $\sum_{i=0}^{k}(1-p)^{i-1}p - \sum_{i=0}^{k-1}(1-p)^{i-1}p = (1-p)^{k-1}p$ que cela se réalise.
