TP7 : Autour des Probabilités

1 Quelques outils

Simulation de variables aléatoires

En ce qui concerne la simulation d'expériences aléatoires, on utilisera la fonction `random()` du module `random` , importé par : `import random as rd` . Cette fonction renvoie un nombre (pseudo)aléatoire dans $[0;1[$.
On trouve bien sûr dans des bibliothèques Python des fonctions fournissant des simulations de toutes sortes de lois (par exemple `rd.randint(a,b)` simule une loi uniforme sur $\{a,\dots,b\}$). Le but n'étant pas de prendre en main ces modules, on se contentera des deux commandes simples ci-dessus.
On peut simuler d'autres expériences aléatoires à partir ce celles-ci. Par exemple, pour simuler une variable aléatoire suivant un loi de Bernoulli de paramètre `p` :

- Première méthode :

```
def simulBer(p):
    a=rd.random()
    if a<p:
        return 1
    else :
        return 0
```

- Deuxième méthode (on a également importé numpy (`as np`)) :

```
def simulBer2(p):
    a=rd.random()
    return int(np.floor(a+p))
```

Diagramme en bâtons

Pour visualiser une loi discrète, on produira des diagrammes en bâtons à l'aide de la fonction `bar` du module `matplotlib.pyplot` . Exemple :

```
import matplotlib.pyplot as plt
X=[x for x in range(5)]
Y=[1,2,1,2,1]
plt.bar(X,Y,width=0.1)
```

Pour superposer les diagrammes représentant deux lois sur un même graphique (pour les comparer par exemple), il sera pratique de jouer sur les couleurs et de décaler légèrement les diagrammes. Sans effacer le graphique obtenu précédemment, on peut lui en superposer un nouveau par :

```
X1=[x-0.1 for x in X]
Y1=[1,3,2,0,2]
plt.bar(X1,Y1,width=0.1,color='r')
```

2 Lois usuelles

Loi binomiale

Écrire une fonction `simulBin(n,p)` qui simule la réalisation d'une variable aléatoire de loi binomiale de paramètres n et p. On pourra utiliser la fonction `simulBer` .
Tester cette fonction en calculant la moyenne empirique obtenue sur un grand nombre de tirages.

Loi géométrique

Écrire une fonction `simulGeom(p)` qui simule la réalisation d'une variable aléatoire de loi géométrique de paramètre p. On pourra utiliser la fonction `simulBer` .
Tester cette fonction en calculant la moyenne empirique obtenue sur un grand nombre de tirages.

3 Simulation et représentation d' une loi

Pour obtenir une approximation d'une loi (à espace d'état fini), on simule un grand nombre de fois une variable aléatoire distribuée suivant cette loi, puis on estime les fréquences empiriques de chaque valeur. De même, si l'on veut estimer l'espérance d'une loi, on l'approche par la moyenne empirique sur un grand nombre de simulations (tout cela repose sur la loi des grands nombres).
On illustre cela sur l'exemple suivant.
On dispose d'une urne contenant n boules où n est un nombre entier tel que $n \geq 2$. Ces boules sont numérotées $1, 2, \ldots, n$, et on les extrait au hasard, une par une, et sans remise dans l'urne. Pour $1 \leq k \leq n$, on désigne par Y_k la variable aléatoire valant 1 si la k-ème boule tirée porte le numéro k, et 0 sinon. Enfin, on note X la variable aléatoire somme : $X = Y_1 + Y_2 + \cdots + Y_n$. On s'intéresse ici au cas où $n = 3$.

1. Étude théorique

   (a) Écrire tous les résultats possibles de l'expérience précédente (pour $n = 3$), c'est-à-dire tous les triplets (i, j, k) indiquant un ordre possible dans lequel on a retiré les boules (i représente le numéro que porte la première boule tirée, j celui de la deuxième boule tirée et k celui de la troisième boule tirée).

   (b) Indiquer la valeur prise par la variable aléatoire X pour chacun des résultats possibles, et en déduire la loi puis l'espérance de X.

2. Simulation

   (a) Compléter la fonction suivante afin qu'elle simule la réalisation d'un tirage de la variable aléatoire X.

```
def simulX():
    a1=rd.random() # pour simuler le premier tirage
    if a1<1./3: # premier tirage :  boule num\'ero 1
        a2=rd.random()# pour simuler le deuxi\`eme tirage
        if a2< ... # deuxi\`eme tirage : boule num\'ero 2
            X=...
        else :
            X=...
    elif ... :
                ... (plusieurs lignes)

    else :
                ... (plusieurs lignes)
    return X
```

   (b) Écrire un programme qui simule $N = 100$ (puis augmenter) fois cette variable aléatoire et calcule la moyenne empirique.

   (c) Compléter le programme suivant afin qu'il simule $N = 100$ tirages de la variable alétoire X, affiche la loi empirique de X obtenue par ces tirages (c'est à dire la proportion de chacune des valeurs obtenues) sous forme d'un diagramme bâtons.

```
l=[0,1,2,3] #valeurs prises par la v.a.
C=[0 for x in l] #  compte le nombre de fois
                 # o\`u l'on obtient chaque valeur
for i in range(N):
    Xi=simulX()
    ...
Y=[....]
plt.bar(...,...,width=0.1)
```

Modifier ensuite le programme précédent pour qu'il superpose le diagramme en bâtons de la loi de X à celui de la loi empirique obtenue par simulation.

4 Méthode de Monte Carlo

La méthode de Monte-Carlo est une méthode de calcul d'intégrale (dans un sens large) basée sur la loi des grands nombres : on va approcher $E(X)$ par une moyenne empirique $\frac{1}{n}\sum_{k=1}^{n} X_k$, où X_k est une simulation de la variable aléatoire X.
On connaît bien sûr d'autres méthodes (rectangles, trapèzes ...), mais dans certaines circonstances (en dimension supérieure par exemple), ces méthodes sont coûteuses en calculs.

Premier exemple : calcul d'une valeur approchée de π

On s'éloigne un peu du cadre de votre cours de probabilité pour considérer une loi uniforme sur le carré $[0,1]^2$ (on ne soulèvera pas de difficultés théoriques sur les probabilités). Une variable aléatoire suit cette loi lorsque, pour tout sous-ensemble A de $[0,1]^2$ (vérifiant certaines conditions), la probabilité que X appartienne à A est la surface de A.
Étant donnée une variable aléatoire X suivant cette loi, on considère la variable aléatoire Y valant 1 si X est dans le quart de disque D de rayon 1 contenu dans $[0,1]^2$ et 0 sinon.
La variable aléatoire Y suit une loi de Bernoulli de paramètre $\pi/4$ (qui est également son espérance).
Le programme suivant simule N réalisation de Y et affiche la moyenne empirique obtenue ($\times 4$) :

```
N=10000
S=0.
for i in range(N):
    x,y = rd.random(),rd.random()
    if x**2+y**2<1:
        S+=1
print(4*S/N)
```

À partir de la loi géométrique

On souhaite calculer, pour $p \in]0,1[$, la somme de la série $S = \sum_{k=1}^{+\infty} \frac{1}{k} p^k$ (remarque : somme qui vaut $-\ln(1-p)$; il y a donc bien sûr d'autres moyens de la calculer).
On remarque que (en notant $p = 1-q$) :

$$\sum_{k=1}^{+\infty} \frac{1}{k} p^k = \frac{p}{q} \sum_{k=1}^{+\infty} \frac{1}{k} q(1-q)k = \frac{p}{q} E\left(\frac{1}{X}\right) \ ;$$

où X suit une loi géométrique de paramètre q.
On a déjà programmé une fonction simulant une loi géométrique.
Compléter le programme suivant afin qu'il calcule et affiche une valeur approchée de S suivant la méthode de Monte-Carlo :

```
p = 0.4 ; q = 1-p
N=10000
T = 0.
for i in range(N):
    X = ...
    T +=  ...
moy = ...
print(...
```

5 Processus de Galton-Watson

Le processus que nous allons étudier a été proposé dans les années 1870 pour étudier l'extinction des noms de familles.
On part de la génération 0 où un certain nom est porté par un individu. On suppose qu'à chaque génération, chaque individu donne naissance à un certain nombre d'enfants (garçons si l'on étudie les noms de famille dans le contexte de l'époque) selon une loi de probabilité identique pour tous les individus, et disparaît ensuite. En notant N_n le nombre d'individus à la n-ème génération et $X_{n,j}$ le nombre de descendants du j-ème individu de la n-ème génération, on a donc $N_0 = 1$ et

$$N_{n+1} = \sum_{j=1}^{N_n} X_{n,j} \ ;$$

les $(X_{n,j})_{(n,j)\in\mathbb{N}^2}$ formant une famille de variables aléatoires identiquement distribuées, supposées indépendantes. On supposera de plus que ces V.A. possèdent un moment d'ordre 1, noté m, et qu'elles ne sont pas constantes.
On note, pour $k \in \mathbb{N}$, $p_k = P(X_{n,j} = k)$ et f la fonction génératrice commune des $X_{n,j}$.
Le but de ce qui suit est d'étudier l'évolution de la probabilité que le nom soit éteint à la génération n : $P(N_n = 0)$.

Étude expérimentale

On donne une loi de probabilité sous forme d'une liste `L` telle que $p_k = L[k]$.

1. Écrire une fonction `simul_X(L)` qui simule la réalisation du tirage d'une variable aléatoire de loi `L` .

2. Écrire une fonction `simul_N(n,L)` qui simule le processus décrit ci-dessus durant n générations et renvoie la valeur de N_n.

3. Compléter le programme dans le fichier TP-GW-ACompleter.py afin qu'il simule M fois le processus décrit ci-dessus durant une durée T, qu'il estime la valeur empirique de la probabilité $P(N_n = 0)$ pour $0 \leq n \leq T$ et qu'il trace la suite de ces valeurs.

4. Essayer avec `L1=[0.3,0.5,0.2]` , `L2=[0.2,0.5,0.3]` . Qu'observe-t-on ?

Étude théorique

1. On se donne une V.A. N à valeurs dans $\mathbb{N}$, de fonction génératrice G_N, une suite $(X_k)_{k\in N}$ de V.A. à valeurs dans $\mathbb{N}$ indépendantes entre elles et indépendantes de N, identiquement distribuées, de fonction génératrice commune G_X. On définit une nouvelle V.A. Y par :

$$Y = \sum_{k=1}^{N} X_k \ .$$

   Montrer qu'alors $G_Y = G_N \circ G_X$.

2. En déduire que la fonction génératrice de N_n est f composée n fois. Que vaut alors $P(N_n = 0)$?

3. Tracer sur $[0;1]$ les graphiques des fonctions génératrices correspondant aux deux exemples précédents. Quelle est la différence entre ces deux graphiques et en quoi explique-t-elle la différence de comportement du processus.

4. Montrer que l'on a l'alternative suivante :

   - Soit $m > 1$ et la limite de $P(N_n = 0)$ est strictement inférieure à 1.
   - Soit $m \leq 1$ et la limite de $P(N_n = 0)$ est égale à 1.

6 Permutations aléatoires

On revient sur l'exemple du paragraphe 3 : on dispose d'une urne contenant n boules où n est un nombre entier tel que $n \geq 2$. Ces boules sont numérotées $0, 1, 2, \ldots, n-1$ (on prend cette convention pour s'adapter à l'indexation des listes python), et on les extrait au hasard, une par une, et sans remise dans l'urne. Pour $0 \leq k \leq n-1$, on désigne par Y_k la variable aléatoire valant 1 si la k-ème boule tirée porte le numéro k, et 0 sinon. Enfin, on note X la variable aléatoire somme : $X = Y_0 + Y_2 + \cdots + Y_{n-1}$.

6.1 Simulation

La façon dont nous avons effectué la simulation en cours dans le cas $n = 3$ est beaucoup trop lourde pour le cas n quelconque. Pour effectuer celle-ci, on remarque que le tirage sans remise des n boules de l'urne consiste en fait à effectuer une permutation aléatoire de $\{0, \ldots, n-1\}$ (de manière équidistribuée), et que la variable aléatoire X compte le nombre de points fixes de cette permutation. On explique ci-dessous la simulation de telles permutations aléatoire.
Ce qui suit décrit une méthode pour créer une permutation aléatoire sur $E = \{0, \ldots, n-1\}$.
Si l'on se donne des variables aléatoires T_0, ..., T_{n-1}, définies sur un espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$, indépendantes et identiquement distribuées de loi uniforme sur $[0; 1]$. On peut alors définir pour chaque ω de Ω la permutation $\sigma(\omega)$ de E définie par :

$$\sigma(\omega)(k) = Card\{i \in E, T_i(\omega) < T_k(\omega)\} \ ;$$

$\sigma(\omega)(k)$ est donc le rang de $T_k(\omega)$ (compté à partir de 0) une fois les $T_i(\omega)$ rangés dans l'ordre croissant. Il est clair que la V.A. σ ainsi définie prend ses valeurs dans l'ensemble des permutations de E (en admettant que la probabilité que deux tirages soient égaux est nulle). On admettra par ailleurs qu'elle est équidistribuée.
Ainsi, pour simuler un tirage aléatoire équidistribué de permutations, il suffit de faire un tirage de n simulations de variables aléatoires suivant une loi uniforme sur $[0, 1]$ et d'appliquer la procédure expliquée précédemment. On admettra par ailleurs que la répétition de commande `rd.random()` permet de simuler le tirage de variables aléatoires indépendantes.
Prenons un exemple pour fixer les idées (avec $n = 4$). On fait appel 4 fois à la fonction `rd.random()` et on range les nombres obtenus par ordre d'apparition dans une liste. Par exemple on obtient $T = [0.24, 0.87, 0.43, 0.32]$. On construit alors la liste σ telle que $\sigma[k]$ est le rang de $T[k]$ dans la liste T (en commençant à 0). Ici on obtient $\sigma = [0, 3, 2, 1]$ (qui représente bien une permutation).

1. Écrire une fonction `PermutAlea(n)` qui simule un tirage et renvoie la permutation σ qui lui est associée de la façon décrite précédemment (c'est à dire en créant une liste `L` de n nombres aléatoires compris entre 0 et 1 et en comptant pour chaque `k` le nombre d'indices `i` tels que `L[i]<L[k]`. Cette permutation sera présentée sous forme d'une liste `sigma` de longueur n telle que `sigma[k]` $=\sigma(k)$.

2. Quelle est la complexité de la fonction précédente en fonction de n ? (On considèrera l'appel à la fonction `random()` comme une opération élémentaire.)
   Peut-on améliorer cette complexité ? Si oui, comment ? (On ne demande pas d'écrire l'algorithme mais d'en expliquer le principe.)

3. Écrire un programme répétant un grand nombre de fois l'expérience aléatoire précédente et permettant d'obtenir une valeur approchée de l'espérance de X.

6.2 Étude théorique

1. Indiquer ce que représente concrètement l'événement $[X = k]$. en terme de permutation aléatoire

2. Déterminer la loi de la variable aléatoire Y_1 et son espérance.

3. De manière générale, déterminer la loi de la variable aléatoire Y_k et son espérance.

4. Déterminer l'espérance de la variable aléatoire X. Cela confirme-t-il le résultat obtenu expérimentalement ?