Probabilités, Statistiques, Combinatoire : TM4

Simulation de lois de probabilités

L'objectif de cette feuille est de *simuler* des lois classiques de variables aléatoires d'une part, et d'*observer* le résultat de ces expériences, en les répétant un grand nombre de fois, d'autre part.

- **Simulation :** vous allez écrire des fonctions que nous appellerons *simulateurs*. Nous appellerons *simulateur* d'une expérience, une **fonction** (sans arguments) qui simule un tirage selon l'expérience demandée, et retourne le résultat. Des appels successifs à la fonction retourneront des résultats indépendants. L'un des objectifs de cette feuille est d'écrire des simulateurs pour diverses expériences ; autant que possible, on cherchera à *réutiliser ces simulateurs pour en construire d'autres.*
- **Observation :** en plus des simulateurs, un des objectifs de cette feuille est d'écrire des fonctions qui ont pour rôle d'appeler un simulateur un grand nombre de fois, et de retourner une structure de données qui indique, pour chaque valeur retournée au moins une fois par le simulateur, la *proportion* des tirages qui ont donné cette valeur. On s'attend à ce que, pour des grands nombres de tirages, cette proportion devienne proche de la probabilité attribuée à chaque valeur.
- **Générateurs :** dans la suite de la feuille, vous écrirez de nouveaux types de fonctions que nous appellerons *générateurs* : un générateur sera une fonction qui retourne un simulateur (une fonction qui retourne une fonction, donc). Oui, PYTHON est un langage qui possède aussi des traits fonctionnels.

Voici, à titre d'exemple, un simulateur pour le lancer d'un dé équilibré à six faces, qui doit donc retourner une valeur aléatoire uniforme sur l'ensemble $[[1, 6]]$:

```
def simuleDe6():
    return random.randint(1,6)
```

Il est facile de le combiner avec lui-même pour écrire un simulateur du lancer de **deux** dés (indépendants) à six faces, qui doit retourner un couple de valeurs aléatoires uniformes sur l'ensemble $[[1, 6]]$ (donc, une valeur aléatoire uniforme sur l'ensemble $[[1, 6]] \times [[1, 6]]$:

```
def simule2De6():
    x = simuleDe6()
    y = simuleDe6()
    return (x,y)
```

On appellera *simulateur* d'une expérience, une **fonction** (sans arguments) qui simule un tirage selon l'expérience demandée, et retourne le résultat. Des appels successifs à la fonction retourneront des résultats indépendants. L'un des objectifs de cette feuille est d'écrire des simulateurs pour diverses expériences ; autant que possible, on cherchera à réutiliser ces simulateurs pour en construire d'autres.

Afin de **tester** empiriquement les simulateurs, on procédera de la manière suivante : on appellera le simulateur un grand nombre n de fois ($n = 100$, 1000, un million : c'est la machine qui fait le travail), et on résumera les résultats en retournant une structure de données indiquant, pour chaque "valeur", le *nombre de fois* qu'elle a été obtenue. En divisant ces nombres par n, on obtiendra des *fréquences empiriques*, dont on espère qu'elles seront proches des probabilités qui définissent le modèle.

Dans cette séance, vous serez amené à écrire ces deux types de fonctions, ainsi que des générateurs de simulateurs. Les fonctions dont le nom commence par `simule` sont supposées

être des simulateurs ; les fonctions dont le nom commence par `teste`, sont des testeurs ; enfin, celles dont le nom commence par `genere` sont des générateurs de simulateurs.

Les dictionnaires de Python

Pour manipuler les fréquences empiriques d'apparition de valeurs, nous allons utiliser une structure de données de *dictionnaire*, fournie par le langage PYTHON.

Le dictionnaire est un conteneur – il n'y a pas d'ordre sur les données. Celles-ci sont organisées en couples de forme `clé : valeur` ; l'affichage par défaut d'un dictionnaire se fait entre accolades {...}, ce qui permet de le distinguer visuellement d'une liste ou d'un tuple.

Par exemple `{ (1,4): 3, 6: 'badPassword'}` est un dictionnaire avec deux clés ; l'une des clés est le tuple `(1,4)` (auquel est associée la valeur 3), l'autre clé est l'entier `6` (auquel est associée comme valeur, une chaîne de caractères). Les clés peuvent être des entiers (comme dans un tableau), mais aussi des tuples, ou des chaînes de caractères (on ne peut pas utiliser de liste, ni aucun objet modifiable, comme clé d'un dictionnaire).

Pour initialiser un dictionnaire vide, on peut utiliser `monDict = {}`.

Pour ajouter un couple clé-valeur à un dictionnaire, on utilise la clé comme indice, comme pour accéder à une valeur d'une liste ou d'un tupple (à ceci près que les clés n'ont pas besoin d'être des entiers) :

```
monDict[(1,2)] = 4
```

Si la clé n'existe pas encore dans le dictionnaire, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe `=` ; si la clé est déjà présente, l'ancienne valeur est remplacée par la nouvelle.

Pour accéder à une valeur précise, on utilise la même syntaxe qu'avec les listes :

```
a = monDict[c]
```

affectera la valeur associée à la clé `c` à la variable `a` – ou provoquera une erreur, si la clé n'existe pas dans le dictionnaire.

Pour tester si une clé appartient au dictionnaire, et aussi pour parcourir toutes les clés du dictionnaire, on utilise le mot-clé `in` comme pour les listes :
`(1,3) in monDict` vaut `False`, `(1,4) in monDict` vaut `True`.

De même, on peut faire une itération sur les clés d'un dictionnaire avec `for` :

```
for c in monDict:
    print(monDict[c])
```

affiche la valeur associée à chaque clé présente dans le dictionnaire (dans un ordre qui n'est pas forcément celui qu'on souhaiterait ; la seule chose garantie est que la valeur associée à chaque clé sera affichée une fois).

Enfin, on peut récupérer l'ensemble des clés d'un dictionnaire `monDict` avec la méthode `keys()` (qui retourne un itérateur ; on peut le convertir en liste) :

```
L = list(monDict.keys())
```

forme une liste L dont les éléments seront toutes les clés présentes dans le dictionnaire `monDict`.

Une remarque générale sur les programmes demandés : Les exercices de cette feuille sont tous réalisables en quelques lignes, par des programmes extrêmement simples. En revanche, ils ne mènent pas à des algorithmes de simulation de complexité optimale ; notamment, pour simuler des lois binomiales dont le paramètre k est grand, ou des lois de Poisson dont le paramètre x est grand, ils deviennent assez inefficaces. Évitez, dans vos tests,

les valeurs plus grandes que 20 ou 30 pour ces paramètres, ou alors limitez le nombre n de répétitions en conséquence.

4.1 Simulation de lois de probabilités

Voici une fonction `testeDe6(n)`.

```
def testDe6(n):
   freq={}
    for i in range(n):
        resultat = simuleDe6()
        if resultat in freq :
            freq[resultat]+=1
        else:
            freq[restulat]=1
    for valeur in freq:
        freq[valeur] = freq[valeur]/n
    return freq
```

Cette fonction retourne un dictionnaire contenant les *fréquences empiriques* d'apparition des valeurs de 1 à 6 après n lancers d'un dé, calculées avec l'algorithme suivant :

— Initialiser un dictionnaire vide.
— Effectuer n lancers ; pour chaque lancer, si le résultat est déjà une clé présente dans le dictionnaire, incrémenter de 1 la valeur associée ; sinon, ajouter la clé et lui associer la valeur 1.
— Avant de retourner, *normaliser* le dictionnaire, c'est-à-dire diviser toutes les valeurs (nombres d'occurrences des différentes clés) par n pour obtenir des valeurs dont la somme soit 1.

La valeur retournée de `testeDe6(5)` pourrait donc ressembler à { `1 : 0.2, 2 : 0.4, 3 : 0.2, 6 : 0.2`} (les clés sont les résultats possibles de l'expérience ; les valeurs sont les fréquences observées, sous formes de flottants)

1. Tester la fonction `testeDe6(n)` avec des valeurs de n comme $10, 100, 1000, 1000000$, chacune à plusieurs reprises. Commenter les résultats obtenus (les résultats sont-ils conformes aux attentes ? de quel ordre de grandeur, en fonction du paramètre n, semble être l'écart entre la fréquence observée et la "vraie" probabilité ?)
2. En réutilisant la fonction `simuleDe6`, écrire une fonction `simulePremierSix()` qui simule le lancer répété d'un dé et retourne le nombre total de lancers nécessaires pour obtenir le résultat 6 pour la première fois. Quelle est la loi de probabilité théorique associée à cette expérience ?
3. Écrire une fonction `testePremierSix(n)` comparable à la fonction `testeDe6()`, mais pour l'expérience consistant à attendre le premier six, à répéter `n` fois ; comme pour la fonction `testeDe6()`, expérimenter avec des valeurs de `n` de 10, 100, 1000 et 1000000 et commenter les résultats obtenus.
4. Faire la même chose avec l'expérience correspondant à lancer trois dés, et à retourner le triplet de valeurs obtenues : écrire une fonction `simuleTroisDes()` qui simule cette expérience, et une fonction `testeTroisDes(n)` qui retourne un dictionnaire contenant

les fréquences observées des différents résultats possibles pour n tirages. Quelles sont les fréquences que vous vous attendez à observer lorsque la fonction `testeTroisDes` est utilisée avec de grandes valeurs de n ? Valider expérimentalement cette attente.

Il devrait être évident à ce stade que les définitions de toutes les fonctions de test se ressemblent beaucoup. La seule différence est le simulateur utilisé pour obtenir les résultats ; le fait d'utiliser un dictionnaire pour représenter les résultats permet de ne pas se limiter à des simulateurs qui retournent des valeurs entières.

Il va de soi qu'il vaut mieux définir une fonction de test générique, à laquelle on peut passer le simulateur en paramètre.

5. Écrire une fonction `frequencesEmpiriques(sim,n)` qui, étant donné un simulateur `sim` et un entier n, retourne un dictionnaire contenant les fréquences empiriques des résultats observés à l'issue de n appels à `sim`. Tester votre fonction, avec comme simulateur `simuleDe6`, `simulePremierSix` et `simuleTroisDes`.

4.2 Simulateurs génériques

Jusqu'ici, chaque expérience nouvelle que l'on souhaite simuler demande d'écrire un simulateur spécifique, alors même que certains paramètres pourraient changer sans que le code à écrire ne change significativement : la fonction `simuleDe10()` (pour un dé équilibré à 10 faces : valeur uniforme sur $[[1, 10]]$) serait quasiment la même que `simuleDe6()`. On pourrait passer des paramètres aux simulateurs, mais cela nuirait à la généricité des testeurs.

En PYTHON, comme dans les langages fonctionnels classiques, une fonction peut non seulement prendre comme paramètre une fonction (c'est le cas de votre testeur `frequencesEmpiriques`), mais aussi retourner une fonction. Nous allons exploiter cette possibilité, et écrire des fonctions (que nous appellerons "générateurs") qui vont retourner des simulateurs, en utilisant les paramètres de la fonction pour fixer les paramètres du simulateur.

Voici par exemple une fonction qui "fabrique" un simulateur de dés équilibrés :

```
def genereDe(faces):
    def sim():
        return random.randint(1,faces)
    return sim
```

Un tel générateur de simulateurs peut être utilisé de la manière suivante, pour simuler des dés de différents types :

```
d6 = genereDe(6)
d10 = genereDe(10)
freq6 = frequencesEmpiriques(d6,10000)
freq10 = frequencesEmpiriques(d10,10000)
```

6. Écrire un générateur `genereBernoulli(p)`, qui retourne un simulateur pour la loi de Bernoulli de paramètre p. Le simulateur fera appel à la fonction de bibliothèque `random.random()`, comme pour les simulations de jeux de ballon du TM3.
7. Écrire un générateur `genereSomme(sim1,sim2)`, qui prend en entrée deux simulateurs et retourne un simulateur pour la variables aléatoire correspondant à la somme de deux variables aléatoires indépendantes, une simulée par chacun des deux simulateurs paramètres. Utiliser votre générateur pour obtenir, de la manière la plus simple possible, un simulateur pour la somme de deux dés classiques (à 6 faces).

8. Écrire un générateur `genereBinomiale(k,p)`, qui retourne un simulateur de la loi binomiale de paramètres (k, p). **Rappel :** pour simuler une variable binomiale de paramètres (k, p), il suffit de faire la somme de k variables de Bernoulli indépendantes, de paramètre p. Remarquez que le **même** simulateur de Bernoulli peut être utilisé, car les appels successifs au simulateur retournent des valeurs indépendantes.
   Observez les fréquences empiriques pour diverses combinaisons de k et p, en les comparant aux probabilités prévues.
9. Écrire un générateur `genereGeometrique(p)`, qui retourne un simulateur de la loi géométrique de paramètre p. **Rappel :** pour simuler une variable géométrique, il suffit de simuler une suite de variables de Bernoulli de paramètre p jusqu'à la première apparition d'un 1, et retourner le nombre total de répétitions.
   Observez les fréquences empiriques pour diverses valeurs de p.

4.2.1 Lois de Poisson : l'œuf et la poule

Les exercices précédents vous ont incités à écrire des simulateurs de lois de probabilités en les construisant à partir de simulateurs plus simples, avec deux "cas de base" qui sont le tirage uniforme dans un intervalle d'entiers et le tirage d'une variable de Bernoulli, pour lesquels vous faites appel aux fonctions de la bibliothèque `random`.

Ce genre de construction ne permet pas d'écrire de simulateur pour les lois de Poisson, qui n'ont pas de représentation simple en fonction de Bernoulli ou d'uniformes. En revanche, on peut utiliser des propriétés remarquables des lois de Poisson :

— si deux variables aléatoires indépendantes sont des variables de Poisson, leur *somme* est également une variable de Poison (dont le paramètre est la somme des paramètres) ; cette propriété se généralise à la somme de plus de deux variables de Poisson indépendantes (cf exercice de la feuille TD6).

— si N est une variable de Poisson de paramètre x, et qu'une fois connue la valeur de N on prend la somme de N variables de Bernoulli indépendantes de même paramètre p, le résultat est une variable de Poisson de paramètre xp.

Ensemble, ces deux propriétés permettent d'écrire des simulateurs de variables de Poisson pour n'importe quel paramètre, pour peu que l'on dispose d'un premier simulateur de variable de Poisson pour un paramètre connu.

La fonction `simulePoisson1()` fournie, est un simulateur de variable de Poisson de paramètre 1 – il n'est pas demandé d'essayer de comprendre comment il fonctionne.

10. Écrire une fonction `generePoissonEntier(n)` qui retourne un simulateur de variable de Poisson de paramètre n, pour un paramètre $n > 0$ **entier**, en utilisant `simulePoisson1` et la propriété sur les sommes de variables de Poisson.
11. Écrire une fonction `generePoissonFrac(p)` qui, pour un paramètre p qui doit être compris entre 0 et 1, retourne un simulateur de variable de Poisson de paramètre p ; là encore, vous utiliserez `simulePoisson1`, et la deuxième propriété des variables de Poisson.
12. Enfin, écrire une fonction `generePoisson(x)` qui retourne un simulateur de variable de Poisson, pour un paramètre $x > 0$ quelconque. Vous réutiliserez pour cela les fonctions précédentes.
    La fonction *partie entière* est `math.floor` ; une fonction *partie fractionnaire* est fournie dans le fichier (fonction `frac`).