Simulation de variables aléatoires discrètes

1 Généralités sur la simulation d'une variable aléatoire

1.1 Principe

Par définition, un nombre aléatoire n'est pas prévisible. Vouloir donc utiliser un ordinateur pour obtenir des nombres aléatoires est paradoxal, ou plutôt impossible. En effet, l'ordinateur ne peut appliquer qu'une formule prédéfinie qui lui est fournie sous la forme d'un algorithme.

La génération de variables aléatoires s'accomplit donc en général au moyen de deux étapes de nature différente:

- la génération de nombres au hasard U_1, U_2,... jouant le rôle de variables aléatoires indépendantes et identiquement distribuées (ce que nous noterons i.i.d. dans la suite) uniformes sur l'intervalle $[0, 1]$;
- la transformation des nombres U_1, U_2, ... précédents en des nombres susceptibles de jouer le rôle des variables aléatoires intervenant dans la définition de la loi, et qui ne sont en général ni i.i.d. ni uniformes sur $[0, 1]$.

Ainsi, la simulation informatique de variables aléatoires, aussi complexes soient elles, repose sur la simulation de variables aléatoires i.i.d. très simples, auxquelles sont appliquées des transformations adéquates. La variable aléatoire de base est celle de loi uniforme sur $[0, 1]$.

1.2 Notions de vocabulaire

Avant d'entrer dans les détails, il convient de détailler le vocabulaire.

Expérience aléatoire: Se dit d'une expérience dont on ne peut prévoir le résultat.

Machines déterministes: Les ordinateurs actuels sont des machines déterministes. En d'autres termes, cela veut dire que les algorithmes que nous écrivons sont régis par la règle suivante: pour une même entrée, l'algorithme produit toujours la même sortie.

1.3 L'aléatoire et le pseudo-aléatoire

Des définitions du vocabulaire, il est donc important de noter que **la notion de phénomène aléatoire est incompatible avec la prédictabilité des résultats issus du déterminisme.**

Ce constat est sans équivoque: l'aléatoire pur ne peut être codé en machine. On va donc devoir se contenter d'une forme affaiblie de l'aléatoire conciliable avec le déterminisme des machines. Il s'agira de coder de **l'aléatoire à résultats prédictibles**. C'est exactement ce que permettent les générateurs pseudo-aléatoires.

1.4 Générateur pseudo-aléatoire

Un générateur pseudo-aléatoire est caractérisé par un triplet $(S,\ f,\ s)$:

- S est un ensemble fini (de cardinal grand);
- f est une application $f : S \to s$;
- s est un élément de S ($s \in S$) appelé « graine » (*seed* en anglais).

Un tel générateur fournit une suite de nombres de S notée (x_n):

- $x_0 = s$: le premier nombre fourni est la graine;
- $\forall n \in \mathbb{N}, x_{n+1} = f(x_n)$.

La suite (x_n) obtenue est déterminée de manière unique par sa valeur initiale s. On obtient ainsi une suite d'éléments de S.

Pour que ces résultats soient facilement exploitables, on utilise généralement une fonction $g : S \to [0, 1[$ afin de transporter les valeurs de la suite (x_n) dans $[0,\ 1[$: on obtient ainsi une suite $(g(x_n))$ d'éléments dans $[0,\ 1[$.

Ainsi, à s fixé, un générateur pseudo-aléatoire fournira toujours la même suite de réels $(g(x_n))$.

Quels sont les avantages du pseudo-aléatoire?

- Les simulations sont reproductibles;
- En jouant sur la définition de la fonction g, on peut définir la répartition des valeurs fournies par le générateur. Ce dernier point permet d'envisager la simulation de variables suivant des lois de probabilités usuelles.

2 Simulation d'une v.a.r en Python

2.1 La fonction `random`

La fonction `random`, de la bibliothèque `random`, implémente un générateur pseudo-aléatoire. Cette fonction retourne un nombre de type `float` aléatoirement choisi dans $[0; 1[$ selon une distribution uniforme, c'est à dire que ce nombre a une probabilité $\mathrm{b} - \mathrm{a}$ d'appartenir à un intervalle $]\mathrm{a}, \mathrm{b}[\subset [0, 1[$, donc 1 chance sur 10 d'être dans $[0.2, 0.3]$ (le fait que cet intervalle soit ouvert ou fermé ne change pas la probabilité car le fait de choisir un nombre précis $\mathrm{a} \in [0, 1[$ est de probabilité nul (bien que possible)).

Exercice 2: A la découverte de `random`!

Après avoir importé le module `random`, évaluer `random.random()`. Qu'obtient-on? Comparer avec le résultat de votre voisin.

Exercice 3: Explorons un peu plus `random`!

Évaluer la commande `random.seed(0)`. Évaluer alors plusieurs fois de suite la commande `random.random()`. Qu'obtient-on ? Comparer avec le résultat de votre voisin.

Exercice 4: On veut des détails!

Expliquer en détails mais de façon brève les résultats obtenus.

Le générateur pseudo-aléatoire implementé dans la fonction `random` vérifie les propriétés suivantes:

- Si $X(\omega)$ désigne le résultat de l'appel `random()`, alors X est une v.a.r. de loi uniforme sur $[0, 1[$.
- Si $X_1(\omega)...X_n(\omega)$ désignent les résultats de n appels successifs de `random()`, alors les variables aléatoires $X_1, ..., X_n$ sont mutuellement indépendantes.

En mathématiques, on travaille souvent avec des variables aléatoires, mais en informatique, on simule **une réalisation** de ces variables aléatoires (ce qui correspond à la notion d'observation en statistique).

2.2 Diagrammes en bâtons en Python

En guide d'illustration, supposons que l'on a:

- une v.a.r. X pouvant prendre les valeurs $X(\Omega) = \{1, 2, 5, 7, 8, 10\}$.
- une liste d'observations obs = [1, 10, 7, 5, 2, 7, 8, 5, 2, 2].

Cette situation se représente par un diagramme en bâtons contenant 6 classes (chaque valeur 1, 2, 3, 5, 7, 10 produit une classe). Chacune de ces valeurs est portée en abscisse.

En ordonnée, on porte l'effectif de chaque classe, c'est à dire le nombre d'individus que comporte chaque classe. Dans notre exemple, on obtient:

- effectif de la classe 1: 1
- effectif de la classe 2: 3
- effectif de la classe 5: 2
- effectif de la classe 7: 2
- effectif de la classe 8: 1
- effectif de la classe 10: 1

Ce qui correspond au tableau des effectifs: [1, 3, 2, 2, 1, 1].

Exercice 4: fonction position

Ecrire une fonction `position` qui prends en paramètre une liste `L` et un élément `elt` de la liste et qui renvoie la position de cet élément dans la liste.

Exercice 5: fonction de calcul des effectifs

Étant donnée une liste `cl` contenant les valeurs de chaque classe et une liste `obs` contenant une liste d'observation, écrire une fonction `calcul_effectif` qui renvoie le tableau des effectifs de chaque classe de l'observation.

Pour tracer un diagramme en bâtons dans Python, nous utiliserons la fonction `bar` qui provient du module `matplotlib.pyplot`. Nous importerons dans la suite du TP ce module en utilisant l'alias `plt` (`import matplotlib.pyplot as plt`).

La fonction `bar` s'appelle généralement avec deux arguments `absc` et `ord`:

- `absc` désigne la liste des points sur lesquels les bâtons vont s'appuyer;
- `ord` désigne la hauteur dans l'ordre de chaque bâton.

Par défaut, chaque bâton est de largeur `0.8` mais on peut la changer en ajoutant comme paramètre d'appel par exemple `width = 0.2`. On peut aussi modifier la couleur des bâtons en ajoutant le paramètre d'appel `color = 'b'`.

2.3 Histogrammes en Python

De même, pour tracer un histogramme dans Python, on peut utiliser la fonction `hist` provenant du module `matplotlib.pyplot`. De plus, cette fonction peut en même temps réaliser sur le même graphique la courbe de la densité de la probabilité si on lui fournit la formule de celle-ci et en activant l'option `density`.

La fonction `hist` prends généralement 2 arguments ou 3 dans le cas ou la densité veut être affichée. Ces arguments sont:

- `x`: une liste de valeurs.
- `bins`: Si `bins` est un nombre entier, il définit le nombre d'intervalles de largeur égale dans la plage.

Dans le cas où l'argument `density` égale à `True`, dessine et renvoie une densité de probabilité. Pour ensuite afficher cette densité de probabilité, il faut utiliser les données obtenues du précédent calcul et l'afficher à l'aide de la formule de la densité de probabilité.

Remarques: Dans le module `random` et `numpy`, nous avons la fonction random() qui retourne un nombre de type `float` aléatoirement choisi dans [0, 1[selon une distribution uniforme. `random.random()` est toujours utilisée sans argument, par contre celle du module `numpy` peut être utilisée avec 1 argument:

- `numpy.random.random(4)` retourne une liste de type numpy.nd.array de 4 nombre aléatoires choisis dans [0; 1[selon une distribution uniforme
- `numpy.random.random((3,5))` retourne un tableau de type numpy.nd.array de 3 lignes 5 colonnes de nombre aléatoires choisis dans [0, 1[selon une distribution uniforme
- `numpy.random.random((4, 3, 5))` retourne un "tableau" à trois dimensions de type numpy.nd.array de taille $4 \times 3 \times 5$ de nombres aléatoires choisis dans [0, 1[selon une distribution uniforme.

3 Simulation de lois de probabilités

3.1 Simulation d'une v.a.r. suivant une loi discrète usuelle

Pour les lois discrètes usuelles, nous présenterons ici uniquement les méthodes simples de simulation de v.a.r. à partir de la situation d'application et ne faisant pas intervenir la méthode d'inversion de la fonction de répartition.

3.1.1 Loi uniforme discrète

3.1.1.1 Simulation à l'aide de la fonction `random`

On considère le programme suivant:

```
import random
import math

def uniforme(a, b):
    return (a + math.floor(random.random() * ((b-a) + 1)))
```

Exercice 6: Fonction uniforme

Quel est le rôle de la fonction `uniforme` ? Expliquer en quelques phrases.

Remarque: La fonction `random.randomint` fournit le même résultat.

3.1.1.2 Diagrammes en bâtons associés

Il s'agit maintenant de comparer:

- Le diagramme en bâtons obtenu par `N` observations de la simulation de la loi uniforme;
- Avec le diagramme en bâtons représentant les fréquences théoriques.

Pour plus de simplicité, considérons initialement une loi uniforme sur $[1, n]$.

On considère le programme suivant:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Paramètres
5  N = 1000        # 1000 simulations
6  n = 4           # loi uniforme sur [[1,4]]
7  width = 0.35    # largeur des bandes
8
9  # Résultat de la simulation
10 obs = [uniforme(1,n) for k in range(N)]
11
12 # Tableau des effectifs des observations
13 cl = np.linspace(1, n, n)
14 effectif = calcul_effectif(cl, obs)
15
16 # Tableau de distribution de probabilité (valeurs théoriques)
17 P = np.zeros(n)
18
19 for k in range(n):
20     P[k] = 1/n
```

Exercice 7: Fonction np.linspace

A quoi sert la fonction `np.linspace`?

Exercice 8: Boucle for et comprehension de liste

On utilise dans le code précédent à la ligne 10, une compréhension de liste. Comment peut-on remplacer ce code par une boucle `for` pour obtenir le même résultat?

Exercice 9: Tracé des deux diagrammes

Veuillez compléter le programme suivant (au niveau des pointillés).

```
labels = ['1', '2', '3', '4']
x = np.arange(len(labels))
fig, ax = plt.subplots()

rects1 = ax.bar(x - width / 2, ... , width, label = 'Théorique')
rects2 = ax.bar(x + width / 2, ... , width, label = 'Observé')

ax.set_xticks(x, labels)
ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)
ax.set_ylim(0.0,0.35)

ax.legend()
fig.tight_layout()

plt.show()
```

Exécutez l'ensemble des codes pour obtenir le diagramme.

Exercice 10: De [1,n] à [a,b]

Comment pourrait-on adapter le programme précédent afin d'obtenir le diagramme associé à la simulation d'une loi uniforme discrète sur $[a, b]$ et plus seulement sur $[1, n]$?

3.1.2 Loi binomiale

3.1.2.1 Simulation à l'aide de la fonction `random`

Exercice 11: Fonction `Bernoulli(p)`

Ecrire une fonction `Bernoulli(p)` simulant une variable aléatoire de loi $\mathcal{B}(p)$.

Note: Une variable aléatoire suivant une loi de Bernoulli de paramètre p prend la valeur 1 avec probabilité p et 0 avec la probabilité 1 − p. Pour simuler un résultat d'une telle loi, il faut pouvoir obtenir un nombre 1 avec une probabilité p en n'utilisant que la fonction `random`. On sait que cette fonction retourne un nombre appartenant à [0, p] avec une probabilité p. Donc il suffit d'effectuer une évaluation de la fonction `random` et de retourner 1 si ce résultat est dans [0, p] et 0 sinon. On modélise ainsi bien un résultat aléatoire suivant une loi de Bernoulli.

Exercice 12: La loi binomiale

Que signifie $X \hookrightarrow \mathcal{B}(n, p)$? Dans quel type d'expérience cette loi est-elle utilisée? Préciser:
a) $X(\Omega) =$
b) $\forall k \in X(\Omega), \mathbb{P}(X = k) =$

Exercice 13: Fonction `binomiale(n,p)`

Ecrire une fonction `binomiale(n,p)` simulant une variable aléatoire de loi $\mathcal{B}(n, p)$.

Note: Soit $X \hookrightarrow \mathcal{B}(n,p)$, on sait que X représente le nombre de succès dans une répétition de n épreuves identiques et indépendantes, la probabilité de succès étant de p. Donc pour la modéliser il suffit de simuler n résultats X_i d'une loi de Bernoulli, ces résultat valent 1 avec probabilité p et le nombre de succès (résultat égal à 1) est alors $X = \sum_{i=1}^{n} X_i$ qui suit bien une loi $\mathcal{B}(n,p)$.

Remarque: La fonction `np.random.binomial(n,p)` fournit le même résultat.

3.1.2.2 Diagrammes en bâtons associés

Il s'agit maintenant de comparer:

- Le diagramme en bâtons construit à partir de 1000 simulations indépendantes;
- Avec le diagramme en bâtons corresponsant à la loi $\mathcal{B}(40, 0.3)$ (fréquences théoriques).

On considère le programme suivant:

```
# Paramètres
N = 1000
n = 40
p = 0.3

# Valeurs observées (résultat de la simulation)
obs = []
for k in range(N):
    obs = obs + [binomiale(n, p)]

# Tableau des effectifs des observations
cl = np.linspace(0, n, n + 1)
effectif = calcul_effectif(cl, Obs)
```

```
14
15 # Tableau de la distribution de probabilité (valeurs théoriques)
16 P = np.zeros(n + 1)
17 for k in range(n + 1):
18     comb = ...
19     P[k] = comb * (p ** k) * ((1-p)**(n-k))
```

Exercice 14: D'une boucle `for` à une compréhension de liste

Aux lignes 8-9, nous utilisons une boucle `for`, comment pouvons nous la remplacer par une compréhension de liste?

Exercice 15: Compléter le programme

Compléter la ligne 18 du progamme
comb =

Il convient maintenant de tracer les diagrammes en bâtons correspondants.

Exercice 16: Tracé du diagramme

Compléter le programme suivant pour tracer les diagrammes

```
x = np.linspace(0, n, ...)

fig, ax = plt.subplots()
rects1 = ax.bar(x - width / 2, ... , width, label = 'Théorique')
rects2 = ax.bar(x + width / 2, ... , width, label = 'Observé')

ax.set_ylim(0.0, 0.16)
ax.legend()

fig.tight_layout()
plt.show()
```