TP1 : Simulation de phénomènes aléatoires

Quelques références en Python :

1. `docs.python.org/fr/3` (en français) et `docs.python.org/3` (en anglais). La documentation officielle de Python. Pour les débutants, nous recommandons en particulier les chapitres 1 à 4 du tutoriel, qui devraient suffire en ce qui concerne les fonctionnalités de base.

2. `github.com/jrjohansson/scientific-python-lectures`. Introduction to Scientific Computing in Python par Robert Johansson. Une excellente introduction pédagogique (en anglais) à `numpy`, `matplotlib` et `scipy`, sous pdf ou comme notebooks IPython, visionnable directement sur le web. Les chapitres 1 à 5 du pdf ou les cours (lectures) 1 à 4 sont largement suffisants comme prérequis.

3. `scipy.org/docs.html`. La documentation officielle de `numpy`, `matplotlib` et `scipy` peut servir comme référence, mais semble être trop exhaustive pour une première utilisation.

Nous recommandons l'utilisation de **Spyder** ou de **Jupyter** pour programmer en Python.

1 Bases de la simulation aléatoire

Toutes les simulations aléatoires reposent sur un élément de base : la simulation par Python de variables aléatoires uniformes.
Avant toute chose, on inclut les bibliothèques adéquates :

```
import math
import numpy as np
import scipy.stats as scs
import matplotlib.pyplot as plt
import numpy.random as random
```

Pour invoquer le générateur de nombres aléatoires on utilisera la commande `random.rand()`. Des appels successifs à cette fonction renvoient des réalisations indépendantes de la loi uniforme sur $[0;1]$ (appelée aussi mesure de Lebesgue sur $[0;1]$). Essayer la séquence de commandes suivante :

```
for i in range(10): # 10 appels successifs de la fonction rand
    print(random.rand())
print(f"Un vecteur de nombres aléatoires de taille 4 : {random.rand(4)}.")
print(f"Une matrice de nombres aléatoires de taille 2 par 3 : {random.rand((2,3))}.")
```

1.1 Comment Python fait-il pour engendrer ces nombres aléatoires ?

Comment Python simule-t-il une suite de variables aléatoires $X_1, X_2, \dots$ indépendantes et uniformes sur l'intervalle $[0;1]$? En quelques mots, cela revient à engendrer des entiers dits « pseudo-aléatoires » $N_1, N_2, \dots$ indépendants uniformes dans $E = \{0;1;\dots;M-1\}$ et à renvoyer $X_i = N_i/M$.
Comment fonctionne ce générateur de nombres « pseudo-aléatoires » ? Dans un ordinateur, il n'y a pas de petits bonhommes qui lancent très très vite des pièces avant d'en transmettre le résultat au processeur. En réalité, lorsqu'on demande un nombre aléatoire à Python il nous renvoie un terme d'une suite déterministe. La manière dont procède Python est savante. Commençons par décrire une vieille méthode relativement efficace. C'est l'algorithme de génération par congruence linéaire et c'est la manière dont les générateurs de nombres aléatoires fonctionnaient autrefois. On se donne trois paramètres entiers a, b et M, où M est un grand entier, par exemple $2^{31}-1$. À la n-ième demande d'un nombre au générateur, il renvoie le terme x_n défini par

$$x_{n+1} = ax_n + b \bmod M.$$

Le n-ième appel à la fonction `rand()` renvoie $\frac{x_n}{M}$ qui est bien dans $[0;1]$. Étant donné le premier terme x_0 (appelé « graine » ou « seed » en anglais), la suite des aléas est entièrement fixée. Cependant un choix habile de a, b et M donnera l'impression d'une suite de variables aléatoires, et on supposera toujours les deux propriétés suivantes :

(i) Un appel à la fonction `rand` donne une réalisation d'une variable aléatoire de loi uniforme sur $[0;1]$.

(ii) Les appels successifs à la fonction `rand` fournissent une suite de variables aléatoires indépendantes.

Remarque 1. Comme expliqué plus haut, l'algorithme pour engendrer des nombres pseudo-aléatoires utilisé par Python et la plupart des logiciels modernes n'est pas celui détaillé ci-dessus, et il est relativement plus savant. Si les nombres renvoyés ici peuvent sembler aléatoires, des tests simples permettent de se rendre compte qu'il n'en est rien. L'algorithme utilisé par Python est appelé « Mersenne Twister » (développé par Makoto Matsumoto et Takuji Nishimura en 1997) et la suite de nombres ainsi engendrée ressemble beaucoup plus à une suite de variables aléatoires uniformes et indépendantes. Néanmoins elle ne l'est pas tout à fait puisqu'il s'agit d'une suite périodique. Il ne s'agit pas d'une récurrence à un terme $N_i = f(N_{i-1})$ comme dans l'algorithme présenté plus haut, mais il s'agit toujours d'une suite définie par récurrence. La période $2^{19937}-1$ est un nombre premier (de Mersenne) tellement gigantesque qu'en pratique on ne s'en rendra jamais compte. Ceci dit, X_1 et $X_{2^{19937}}$ ne sont clairement pas indépendants puisqu'ils sont égaux.
Ainsi, les nombres aléatoires que nous allons utiliser dans ces simulations ne le sont pas vraiment : ils sont déterministes mais nous n'aurons pas de moyen de les distinguer d'une suite aléatoire. Nous considérerons la fonction `rand` comme une boîte noire qui nous renvoie réellement des nombres indépendants et uniformes sur $[0;1]$.

Remarque 2. Pour se rendre compte que les nombres générés ne sont pas vraiment aléatoires, il suffit de constater que si on prend la même « seed » x_0, alors la suite générée est la même. Taper la suite de commandes suivante :

```
print(random.rand())
random.seed(seed=1) #ici on fixe la valeur de la graine
print(random.rand())
print(random.rand())
random.seed(seed=1)
print(random.rand())
print(random.rand())
```

1.2 Premières simulations : des points dans le plan

Exercice 1. —

1. À l'aide de la fonction `rand`, écrire une fonction `uniforme_continue(a,b)` qui renvoie une réalisation d'une variable de loi uniforme sur l'intervalle $[a;b]$. Tester cette fonction sur 10 appels.

2. Écrire une fonction `test_disque(N)` qui :

   (a) Simule N réalisations indépendantes de points $Z_i = (X_i, Y_i)$ avec X_i et Y_i indépendants de loi uniforme sur $[-1;1]$, et stocke le résultat dans un tableau de taille $2 \times N$ (pour initialiser un tel tableau on pourra utiliser `Z = np.zeros((2,N))`).

   (b) Trace le nuage de points dans le plan (la fonction `plt.scatter(x,y)` trace le nuage de points dont les abscisses sont dans le vecteur `x` et les ordonnées dans le vecteur `y`).

(c) Trace la fonction qui à k dans $\{1, 2, \dots, N\}$ associe

$$f(k) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}_{\|Z_i\|_2 \leqslant 1},$$

c'est-à-dire la proportion des k premiers points qui sont à distance au plus 1 de l'origine.

(d) Crée deux listes Z^+ et Z^-, l'une contenant seulement les points de norme $\leqslant 1$ et l'autre ceux de norme > 1 ; puis trace les 2 nuages de points associés dans des couleurs différentes.

Tester le programme pour différentes valeurs de N. À votre avis, vers quelle valeur m tend la proportion de points de norme $\leqslant 1$ lorsque N tend vers l'infini ? Modifier le programme pour qu'il superpose au graphe de la proportion la droite d'équation $y = m$.

2 Représentation graphique d'une probabilité discrète

Soit $E = \{e_1, e_2, \dots\}$ une famille finie ou dénombrable de réels 2 à 2 distincts et $(p_e)_{e \in E}$ une famille de réels dans $[0; 1]$ dont la somme vaut 1. La mesure $\mu = \sum_{e \in E} p_e \delta_e$ est la mesure de probabilité discrète sur l'ensemble E telle que $\mu(\{e\}) = p_e$ pour tout e dans E.
On représente souvent cette mesure par un diagramme en bâtons, la hauteur du bâton d'abscisse e correspondant à $\mu(\{e\}) = p_e$.
Dans `matplotlib.pyplot`, la fonction `bar` permet de tracer des diagrammes en bâtons : `bar(x, h, width)` trace des bâtons de hauteur `h[i]` dont les côtés gauche sont d'abscisse `x[i]` (`x` et `h` sont des vecteurs de même taille). On peut spécifier la largeur des barres par le paramètre optionnel `width` et centrer les barres en `x[i]` en fixant le paramètre optionnel `align="center"`.
Voici comment tracer le diagramme en bâtons de la loi binomiale :

```
n = 10
p = 0.2
x = np.array(range(n+1))
y = [math.comb(n,i) * (1-p)**(n-i) * p**i for i in x]
width = 0.1
fig = plt.figure()
plt.bar(x, y, width, align="center")
plt.scatter(x, y, color="black")
plt.xlabel("$e$")
plt.ylabel("$p_e$")
plt.title(f"Diagramme en bâtons de la loi binomiale $B({n},{p})$.")
plt.show()
```

Exercice 2. — La loi géométrique de paramètre p est la loi d'une variable aléatoire X à valeurs entières strictement positives telle que $\mathbb{P}(X = k) = (1 - p)^{k-1} p$.

1. Écrire un programme qui dépend de p et de m et qui trace le diagramme en bâtons de la loi géométrique de paramètre p restreinte à l'intervalle $[1; m]$.

2. Montrer que si X suit une loi géométrique de paramètre p, et si $m \in \mathbb{N}$, alors $\mathbb{P}(X > m) = (1 - p)^m$. En déduire une valeur $m(p)$ tel que $\mathbb{P}(X > m(p)) \leqslant 0.05$.

3. Tester votre programme pour différentes valeurs de p avec $m = m(p)$. Expliquer pourquoi c'est un choix raisonnable de troncature.

3 Mesure empirique

Si on a un échantillon $X_1, X_2, \ldots, X_n$ obtenu par simulation de la loi μ, on peut s'intéresser à la distribution empirique $\widehat{\mu}_n$ qui est la mesure de probabilité sur E qui donne comme probabilité à $\{e\}$ la proportion des X_i qui valent e, c'est-à-dire :

$$\widehat{\mu}_n(\{e\}) = \frac{\mathrm{Card}(\{1 \leqslant i \leqslant n : X_i = e\})}{n}.$$

Vérifier qu'il s'agit bien d'une mesure de probabilité sur E.
Supposons par exemple donnés $X_1, \ldots, X_{1000}$ dans $\{1, 2, 3, 4\}$, et notons pour j dans $\{1, 2, 3, 4\}$:

$$N_j = \mathrm{Card}(\{1 \leqslant i \leqslant 1000 : X_i = j\}).$$

Si sur une observation de cet échantillon on a $N_1 = 450$, $N_2 = 350$, $N_3 = 50$ et $N_4 = 150$, alors le diagramme en bâtons de la loi empirique sera donné par :

```
x = [1, 2, 3, 4]
N = [450, 350, 50, 150]
n = sum(N)
frequence = [x/n for x in N]
width = 0.05
fig = plt.figure()
plt.xlabel("$e$")
plt.ylabel("$p_e$")
plt.title("Diagramme en bâtons de la loi empirique")
plt.bar(x, frequence, width, align="center")
plt.scatter(x, frequence, color="black")
plt.show()
```

Exercice 3. — Soit $Y = \max(X_1, X_2, X_3)$, où X_1, X_2, X_3 sont les résultats de 3 jets indépendants d'un dé équilibré à 6 faces.

1. Quelle est la loi de Y ? On pourra calculer $\mathbb{P}(Y \leqslant n)$.
2. Écrire un programme qui simule le résultat d'un dé à 6 faces (c'est-à-dire renvoie un nombre entre 1 et 6 à chaque appel avec une chance sur 6 pour chacun des résultats). On pourra réutiliser la fonction `uniforme_continue`.
3. Écrire un programme qui dépend de n et qui :
   (a) Simule n réalisations indépendantes de la variables aléatoire Y: $Y_1, \ldots, Y_n$.
   (b) Calcule $N_j = \mathrm{Card}(\{1 \leqslant i \leqslant n : Y_i = j\})$ pour j dans $\{1, 2, \ldots, 6\}$.
   (c) Trace sur un même graphique le diagramme en bâtons de la loi de Y et le diagramme de la loi empirique calculée à partir des N_j.
4. Tester le programme pour différentes valeurs de n. Qu'observez-vous ?

3.1 Variables géométriques

On va comparer plusieurs manières de simuler une variable géométrique de paramètre p.

1. De manière générale, pour simuler une variable aléatoire X à valeurs entières telle que $\mathbb{P}(X = k) = p_k$ pour tout k, on tire une variable uniforme U sur $[0; 1]$ et on renvoie l'entier k tel que

$$p_0 + \cdots + p_{k-1} < U < p_0 + \cdots + p_k$$

2. On tire des variables aléatoires de Bernoulli de paramètre p et on s'arrête à la première fois qu'on tombe sur 1.

3. On pose $\lambda = -(\ln(1-p))^{-1}$ et on renvoie $\lceil -\lambda \ln U \rceil$, où U est une variable uniforme sur $[0;1]$ (la fonction partie entière par excès $\lceil x \rceil$ est obtenue par la commande `math.ceil(x)`).

4. On utilise une fonction déjà présente dans Python. Chercher dans Google « geometric law python » pour découvrir comment.

Exercice 4. —

1. On considère la première méthode évoquée.

   (a) Expliquer pourquoi la méthode proposée simule bien une loi géométrique de paramètre p.

   (b) Écrire une fonction `geometrique1(p)` qui renvoie une réalisation d'une variable discrète de loi géométrique de paramètre p selon la cette méthode évoquée.

   (c) Tester votre algorithme : tracer sur une même figure le diagramme en bâtons de la loi géométrique sur l'intervalle $[1;m]$, et ceux des distributions empiriques obtenues par cette méthode. Choisir de manière adéquate le paramètre p, la taille des échantillons et m.

   (d) Pour évaluer la rapidité de votre algorithme, utiliser :

```
from time import time
N = 10000
p = 0.5
t1 = time()
[geometrique1(p) for i in range(N)]
t2 = time()
temps1 = t2 - t1
print(f"La méthode 1 a pris {temps1} secondes.")
```

2. Reprendre la question précédente pour les méthodes 2, 3 et 4 (en changeant de manière adéquate le nom de l'algorithme).

3. Que dire de la comparaison des différentes méthodes en termes de rapidité ?

3.2 Variables de Poisson

La variable aléatoire X suit la loi de Poisson de paramètre λ si X est à valeur entières positives ou nulles et si

$$\mathbb{P}(X = n) = \mathrm{e}^{-\lambda}\frac{\lambda^n}{n!}.$$

Il y a plusieurs méthodes pour simuler une variable de loi de Poisson :

1. À nouveau, on peut utiliser la première méthode générale décrite pour la loi géométrique.

2. On tire des variables aléatoires U_i uniformes sur $[0;1]$ et on renvoie la variable

$$X = \inf(\{n \in \mathbb{N} : U_0 U_1 \cdots U_n \leqslant \mathrm{e}^{-\lambda}\}).$$

3. On utilise une fonction déjà présente dans Python.

Exercice 5. —

1. Implémenter ces 3 méthodes de simulations et comparer leur efficacité.

2. Tracer sur un même diagramme : le diagramme en bâtons de la loi de Poisson, et ceux des distributions empiriques obtenues par chacune des 3 méthodes. Choisir de manière adéquate le paramètre λ et la taille des échantillons pour bien montrer la convergence.

3. Écrire un algorithme `convolution(l, m)` qui à partir de votre méthode de simulation préférée de la loi de Poisson simule $X + Y$, où X et Y sont indépendants de lois de Poisson de paramètres respectifs l et m. Tracer sur une même figure le diagramme en bâtons de cette distribution empirique et celui de la loi de Poisson de paramètre $l + m$. Qu'en conclure ?

4. Écrire un algorithme `decimation(l, p)` qui, étant donnés $0 < p < 1$ et $l > 0$, engendre X de loi de Poisson de paramètre l ; puis, X étant fixé, renvoie Y de loi la binomiale de paramètre (X, p). Ainsi, pour chacun des X individus obtenus par simulation d'une loi de Poisson, on le garde avec probabilité p et on l'enlève sinon ; le nombre d'individus gardés est Y. Tracer sur une même figure le diagramme en bâtons de la distribution empirique de la variable Y, et celui de la loi de Poisson de paramètre pl. Qu'en conclure ?