1. Variables aléatoires et leurs représentations

Ce cours présente quelques modèles aléatoires élémentaires : en particulier, les suites de nombres aléatoires, les arbres aléatoires, et les graphes aléatoires. Les propriétés simples de ces objets seront expliquées et le plus souvent démontrées, et elles seront observées dans des simulations numériques et graphiques s'appuyant sur `Python` et ses divers modules (`numpy`, `scipy`, `matplotlib`, `networkx`). Pour la programmation en Python, on renvoie à la documentation officielle (https://docs.python.org/fr/3) et aux nombreux exemples qui seront donnés dans chaque chapitre. On conseille l'utilisation de feuilles de calcul Jupyter (documentation : https://jupyter.org) ; pour ouvrir une feuille Jupyter, on tapera `jupyter notebook` dans un terminal, puis on cliquera sur le bouton "New". Une alternative consiste à utiliser l'environnement de développement Spyder (documentation : https://www.spyder-ide.org). Tous les programmes présentés par la suite présupposent l'import des bibliothèques suivantes :

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

Optionnellement, on pourra rajouter

```
import seaborn as sns
sns.set_theme()
```

pour avoir des graphiques un peu plus élégants. On importera ponctuellement d'autres bibliothèques selon les besoins des programmes. Les dessins seront tous produits par `Matplotlib` : c'est l'outil standard pour les graphiques et le calcul scientifique dans `Python`. On renvoie à l'annexe pour des détails sur les modules employés ; des explications seront également apportées au fur et à mesure.

1. Variables uniformes sur $[0, 1]$.

Le point de départ de toutes nos simulations est la possibilité d'obtenir des variables aléatoires indépendantes de *loi uniforme* sur $[0, 1]$, avec la commande `random.random(size=N)`.

(1) Produire un échantillon de taille $N = 5$ de variables uniformes sur le segment $[0, 1]$. Le résultat est un *array* `numpy` unidimensionnel ; on peut aussi créer des matrices (*array* bidimensionnel) avec l'argument `size=(l, c)`, où l est le nombre de lignes, et c le nombre de colonnes. Produire une matrice aléatoire de taille 10×5 dont les entrées sont des variables uniformes sur $[0, 1]$ et indépendantes.

Le résultat de la commande `random.random(size=N)` est un vecteur $(X_1, \dots, X_N)$ de nombres réels compris entre 0 et 1, avec les propriétés suivantes :

- *loi uniforme* : pour tout indice $i \in [\![1, N]\!]$ et tout intervalle $[a_i, b_i] \subset [0, 1]$, on a $\mathbb{P}[a_i \leq X_i \leq b_i] = b_i - a_i$.

- *indépendance* : pour tous intervalles $[a_1, b_1], \dots, [a_n, b_n]$,

$$\mathbb{P}[\forall i \in [\![1, N]\!],\ a_i \leq X_i \leq b_i] = \prod_{i=1}^{N} \mathbb{P}[a_i \leq X_i \leq b_i].$$

Comment `Python` produit-il un tel vecteur ? Il n'y a pas de composant du processeur qui jette des pile-ou-face très rapidement pour obtenir les bits aléatoires des nombres $X_1, \dots, X_N$ (ce serait techniquement possible avec un ordinateur *quantique*). On utilise plutôt un *générateur de nombres pseudo-aléatoires*, qui renvoie une suite qui est déterministe mais qui a les mêmes propriétés statistiques qu'une suite de variables aléatoires uniformes, du moins si l'on observe un échantillon de taille raisonnable.

L'un des premiers algorithmes de ce type est l'algorithme de génération par congruence linéaire. Soit a, b, M trois entiers, avec M très grand. On fixe un entier $x_0 \in [\![0, M-1]\!]$ (la *graine* de l'algorithme), et on définit par récurrence :

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

Posons alors $X_n = \frac{x_n}{M}$; la suite $(X_1, \dots, X_N)$ est à valeurs dans l'intervalle $[0, 1]$, et pour des choix judicieux de a et b (en particulier, tels que la transformation $x \mapsto ax + b$ dans $\mathbb{Z}/M\mathbb{Z}$ soit une permutation d'ordre élevé, idéalement M), cette suite est bien répartie dans $[0, 1]$ et ressemble à une suite de variables uniformes indépendantes.

L'algorithme utilisé par Python est appelé *Mersenne Twister* ; il a été développé par Makoto Matsumoto et Takuji Nishimura en 1997, et c'est une version plus élaborée de l'idée décrite ci-dessus, avec une suite qui n'est pas récurrente à un seul terme, et qui repose sur des fonctions non linéaires. La période de la suite $(X_1, X_2, ...)$ est le nombre premier de Mersenne $2^{19937} - 1$; elle est suffisamment grande pour qu'on ne soit jamais concrètement confronté à la répétition $X_1 = X_{2^{19937}}$. La suite produite par le Mersenne Twister a la propriété d'uniforme répartition suivante : avec $k = 623$ et $n = 32$, si l'on considère toutes les suites de bits

$$\left(\overline{X_i}^n, \overline{X_{i+1}}^n, \dots, \overline{X_{i+k-1}}^n\right), \quad 1 \leq i \leq 2^{19937} - 1$$

avec $\overline{y}^n = (n$ premiers bits du nombre réel $y)$, alors les $2^{nk} = 2^{19936}$ suites de bits possibles apparaissent toutes 2 fois au cours d'une période, sauf la suite avec des 0 pour chaque bit qui apparaît une seule fois. La suite a de nombreuses autres bonnes propriétés qui la rendent très difficile à distinguer par des tests statistiques d'une suite de variables uniformes indépendantes (notamment, la suite du Mersenne Twister est bien meilleure statistiquement qu'une suite obtenue par congruence linéaire). Par ailleurs, l'algorithme Mersenne Twister admet plusieurs variantes adaptées à des besoins spécifiques (production de réels avec plus de bits de précision, cryptographie, utilisation minimale d'espace mémoire, etc.).

Lors du premier appel de `random.random()`, une graine X_0 est choisie, en convertissant l'heure du système en un nombre réel dans $[0, 1]$. On peut néanmoins spécifier une autre graine avec la commande `random.seed(a)`, qui prend en argument un entier `a`.

(2) Comparer les résultats des deux commandes suivantes :

```
print(random.random(size=5)) ;
print(random.random(size=5)) ;
```

et

```
random.seed(1337) ; print(random.random(size=5)) ;
random.seed(1337) ; print(random.random(size=5)) ;
```

(3) À partir d'un réel aléatoire X de loi uniforme sur $[0,1]$, on produit aisément un entier aléatoire N de loi uniforme parmi les valeurs $k \in [\![a, b-1]\!]$ en considérant $N = \lfloor a+(b-a)X \rfloor$. Dans `Python`, la commande `random.randint(a, b, size=N)` produit un échantillon de taille N de variables entières uniformes dans $[\![a, b-1]\!]$. Expérimenter cette commande avec diverses valeurs de a, b, N.

Nous ne détaillerons pas plus les détails algorithmiques du Mersenne Twister ; du point de vue de l'expérimentateur probabiliste, tout se passe comme si `random.random()` était une "boîte noire" qui produisait effectivement des variables uniformes sur $[0,1]$ et indépendantes. Tous les autres objets aléatoires que nous rencontrerons pourront être construits à partir de ces réels aléatoires uniformes dans $[0,1]$.

(4) Une représentation graphique possible d'un échantillon de réels dans $[0,1]$ est par un nuage de points sur un segment :

```
fig, ax = plt.subplots()
alea = random.random(size = N) ;
ax.plot([0, 1], [0, 0], color="k")
ax.plot([0, 0], [-0.03, 0.03], color="k")
ax.plot([1, 1], [-0.03, 0.03], color="k")
ax.scatter(alea, np.zeros(N), color="red")
ax.set_axis_off()
ax.set_aspect(1)
plt.show()
```

Expérimenter avec $N = 10, 50, 100$. L'uniformité est-elle bien lisible sur cette représentation ?

La loi des variables $X_1, \dots, X_N$ obtenues par la commande `random.random()` est uniforme sur $[0,1]$. D'un point de vue théorique, ceci veut dire que :

- les X_i sont des fonctions mesurables

$$X_i : (\Omega, \mathscr{F}) \to ([0,1], \text{Boréliens}([0,1]))$$

définies sur un espace de probabilité $(\Omega, \mathscr{F}, \mathbb{P})$.

- pour tout i, la *loi* de X_i, qui est l'image par X_i de $\mathbb{P}$ et qui est donc une probabilité sur $\mathbb{R}$, est la mesure de Lebesgue sur $[0,1]$:

$$\mathbb{P}_{X_i} = \mathbb{P} \circ (X_i)^{-1} = \text{Leb}_{[0,1]}.$$

Nous verrons plus loin d'autres lois de variables aléatoires réelles. Pour représenter une probabilité μ sur $\mathbb{R}$, on utilise généralement la *fonction de répartition* de μ, qui est la fonction croissante $\mathbb{R} \to [0,1]$ définie par

$$F_\mu(x) = \mu\left((-\infty, x]\right) = \int_{(-\infty,x]} \mu(\mathrm{d}s).$$

Les propriétés fines de ces fonctions seront étudiées dans le Chapitre 2 ; on verra en particulier que F_μ caractérise la probabilité μ. Si $\mu = \mathbb{P}_X$ est la loi d'une variable X, on notera $F_\mu = F_X$.

(5) Quelle est la fonction de répartition de la loi uniforme sur $[0,1]$? La représenter graphiquement avec `Python/Matplotlib`.

On peut aussi représenter un échantillon de nombres réels $(X_1, \dots, X_N)$ avec une fonction de répartition, dite *fonction de répartition empirique*. La *loi empirique* de l'échantillon est

$$\mu_N = \frac{1}{N} \sum_{i=1}^{N} \delta_{X_i},$$

où δ_x désigne le Dirac en x (mesure de masse totale 1, qui attribue le poids 1 à toute partie mesurable contenant x et le poids 0 à toute partie mesurable ne contenant pas x). La loi empirique μ_N est une loi aléatoire, et la fonction de répartition empirique de l'échantillon $(X_1, \dots, X_N)$ est $F_N = F_{\mu_N}$. Ainsi,

$$F_N(x) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{(X_i \leq x)}.$$

(6) On note $(X_{(1)}, \dots, X_{(N)})$ le réordonnement croissant de l'échantillon $(X_1, \dots, X_N)$; ce sont les mêmes valeurs, mais avec $X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(N)}$. Montrer que $F_N(x)$ est donnée par :

$$F_N(x) = \begin{cases} 0 & \text{si } x < X_{(1)}; \\ \frac{i}{N} & \text{si } X_{(i)} \leq x < X_{(i+1)},\ 1 \leq i \leq N-1; \\ 1 & \text{si } x \geq X_{(N)}. \end{cases}$$

Par conséquent, on peut dessiner la fonction de répartition d'un échantillon $X = (X_1, \dots, X_N)$ avec la suite de commandes suivante :

```
fig, ax = plt.subplots()
X.sort() %
N = len(X) %
ax.step(X, (np.arange(N)+1)/N, where="post", color="r") %
ax.step([min(X)-0.1, min(X), min(X)], [0, 0, 1/N], color="r")
ax.plot([max(X), max(X)+0.1], [1, 1], color="r")
ax.set_xlim([min(X)-0.1, max(X)+0.1])
ax.set_ylim([-0.1, 1.1])
plt.show()
```

Dessiner sur le même graphique la fonction de répartition de la loi uniforme, et la fonction de répartition empirique d'un échantillon de taille $N = 10, 100, 1000$ de variables uniformes indépendantes sur $[0, 1]$. Commenter.

La commande `ax.ecdf(X)` trie et dessine d'un coup la fonction de répartition empirique d'un échantillon ; elle remplace les trois lignes ci-dessus qui se finissent par `%`.

2. Variables géométriques.

Si X est une variable aléatoire à valeurs entières (dans $\mathbb{Z}$), alors sa fonction de répartition est constante par morceaux sur chaque intervalle $[k, k+1)$:

$$\forall x \in [k, k+1),\ F_X(x) = F_X(k) = \sum_{l \leq k} \mathbb{P}[X = l].$$

Il est alors commun de représenter la loi de X par son *histogramme* au lieu de sa fonction de répartition. L'histogramme d'une loi μ sur $\mathbb{Z}$ est la diagramme en bâtons avec un bâton de hauteur $\mu(\{k\})$ en face de chaque entier k. D'un point de vue pratique :

- On choisit un intervalle $I = [\![m, M]\!]$ sur lequel représenter la loi. Si μ est à support fini, on peut prendre le plus petit intervalle contenant ce support. Sinon, on choisit m et M de sorte que $\mu(\{k \in \mathbb{Z},\ k < m\})$ et $\mu(\{k \in \mathbb{Z},\ k > M\})$ soit des petites probabilités (typiquement, de somme inférieure à 0.05).

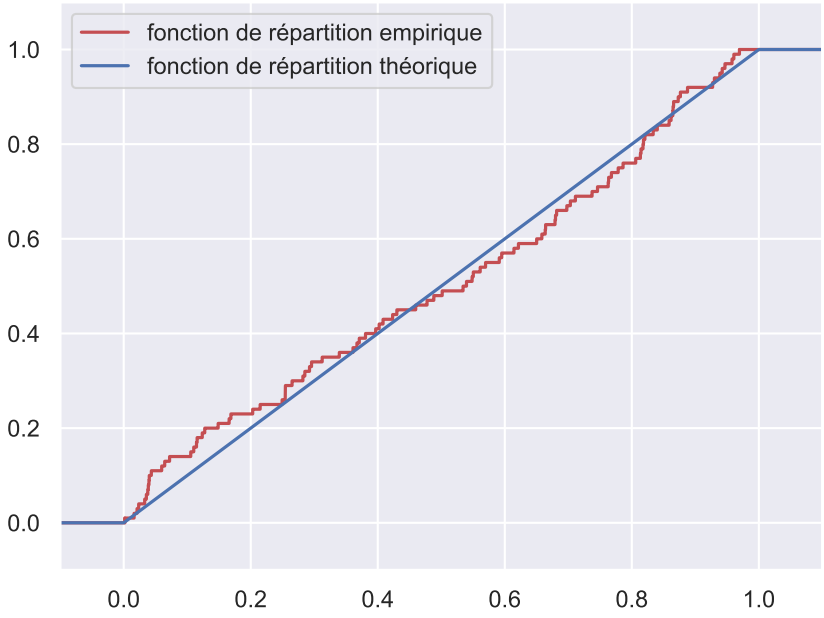

FIG. 1.1. Fonctions de répartition théorique et empirique d'un échantillon de taille 100 pour la loi uniforme sur $[0, 1]$.

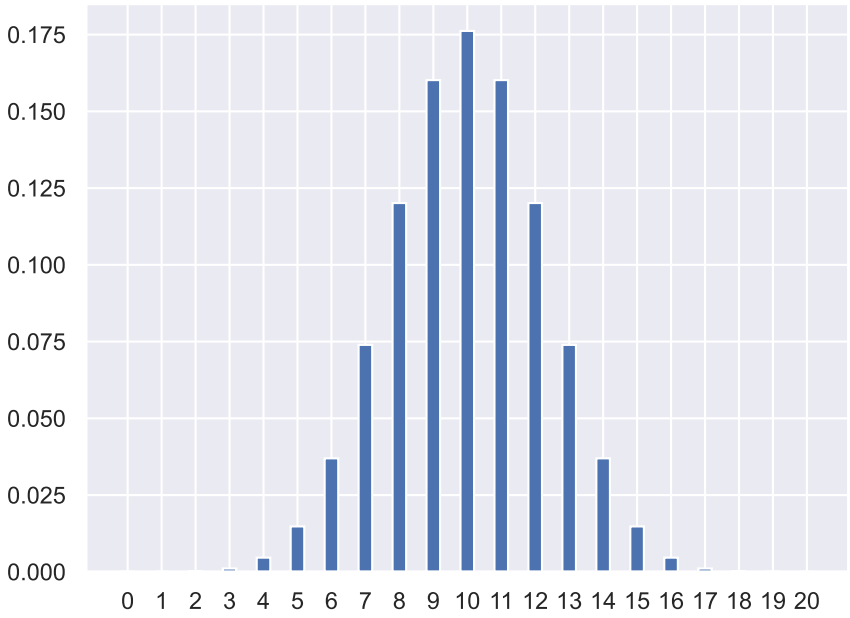

FIG. 1.2. Histogramme de la loi $\mathrm{Bin}(n = 20, p = 0.5)$.

- Puis, on trace le diagramme avec la commande `ax.bar(I, mu)`, où I est l'intervalle précédemment choisi, et μ le vecteur des probabilités correspondant aux valeurs entières dans ces intervalles.

On a par exemple représenté ci-dessus l'histogramme de la loi binomiale $\mathrm{Bin}(n = 20, p = 0.5)$.

(1) On considère la *loi géométrique* $\mu = \mathrm{Geom}(p)$ de paramètre p : c'est la loi du numéro X du premier succès dans une suite d'expériences de Bernoulli indépendantes de paramètre p. Rappeler pourquoi on a :

$$\mu(k) = (1-p)^{k-1}\, p$$

pour tout $k \geq 1$. Que vaut la fonction de répartition F_μ ?

(2) Écrire un programme qui dépend de p et de m et qui trace l'histogramme de la loi géométrique de paramètre p restreinte à l'intervalle $[1, m]$.

(3) En utilisant la valeur de F_μ, trouver une valeur $m(p)$ telle que si $X \sim \mathrm{Geom}(p)$, alors $\mathbb{P}[X > m(p)] \leq 0.05$. Réécrire le programme de la question précédente pour qu'il choisisse par défaut $m = m(p)$.

(4) À partir d'une variable $U \sim \mathrm{Unif}([0,1])$ de loi uniforme sur $[0,1]$, comment construire une expérience de Bernoulli de paramètre p ? En déduire un programme qui tire un échantillon de N variables indépendantes toutes de loi $\mathrm{Geom}(p)$.

(5) Dans la suite, on pourra utiliser la commande `scs.geom(p).rvs(size=N)` à la place du programme de la question précédente. Dessiner sur le même graphique l'histogramme de la loi $\mathrm{Geom}(0.5)$, et l'histogramme empirique d'un échantillon $(X_1, \dots, X_N)$ de taille $N = 10, 100, 1000$ de variables indépendantes ayant cette loi. Cet histogramme empirique mettra en face de chaque entier k une barre de taille $\frac{N_k}{N}$, avec

$$N_k = \mathrm{card}(\{i \in [\![1, N]\!],\ X_i = k\}).$$

Pour compter le nombre de valeurs égales à k dans un *array* numpy `X`, on utilisera la commande `np.count_nonzero(X==k)`. Pour comparer deux histogrammes, on pourra utiliser l'astuce suivante : les arguments optionnels `align="edge", width=a` de la commande `ax.bar()` placent les bâtons à gauche des entiers correspondants si `a` est un réel négatif, et à droite des entiers correspondants si `a` est un réel positif.

Dans les questions ci-dessus, on a vu comment utiliser des variables uniformes sur $[0,1]$ et indépendantes pour construire une variable $X \sim \mathrm{Geom}(p)$. Plus généralement, si X suit une loi discrète μ sur $\mathbb{Z}$, le programme suivant produit à partir de $U \sim \mathrm{Unif}([0,1])$ une simulation de X :

- Avec probabilité 1, il existe un unique entier $k \in \mathbb{Z}$ tel que

$$\sum_{j=-\infty}^{k-1} \mu(\{j\}) < U \leq \sum_{j=-\infty}^{k} \mu(\{j\}).$$

On calcule cet entier $X = k$ en faisant les sommes partielles des probabilités $\mu(\{j\})$, jusqu'à ce qu'on dépasse U.

- Alors, l'entier X obtenu a bien pour loi μ, car

$$\begin{aligned}\mathbb{P}[X = k] &= \mathbb{P}\left[\sum_{j=-\infty}^{k-1} \mu(\{j\}) < U \leq \sum_{j=-\infty}^{k} \mu(\{j\})\right] \\ &= \left(\sum_{j=-\infty}^{k} \mu(\{j\})\right) - \left(\sum_{j=-\infty}^{k-1} \mu(\{j\})\right) = \mu(\{k\}).\end{aligned}$$

Ce n'est pas forcément l'algorithme le plus élégant ou le plus rapide pour une loi discrète donnée, mais il fonctionne en toute généralité.

3. Variables de Poisson.

Une variable aléatoire X suit la *loi de Poisson* de paramètre $\lambda > 0$ si elle est à valeur entières positives ou nulles, et si

$$\mathbb{P}[X = n] = \mathrm{e}^{-\lambda}\frac{\lambda^n}{n!}$$

pour tout entier n. On notera alors $X \sim \mathrm{Poi}(\lambda)$.

(1) Montrer que $\mathbb{E}[X] = \sum_{n=0}^{\infty} n\,\mathbb{P}[X = n] = \lambda$, et que $\mathrm{var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = \lambda$. Ceci implique qu'une variable de Poisson de paramètre λ prend avec grande probabilité des valeurs d'ordre λ.

(2) Plus précisément, déterminons un seuil $k\lambda$ avec $k \geq 1$ tel que $\mathbb{P}[X \geq k\lambda]$ soit toujours petit si $X \sim \mathrm{Poi}(\lambda)$. Montrer qu'on a toujours :

$$\mathbb{P}[X \geq k\lambda] \leq \frac{\mathbb{E}[\mathrm{e}^{tX}]}{\mathrm{e}^{tk\lambda}}, \quad t \geq 0,$$

et que par ailleurs, $\mathbb{E}[\mathrm{e}^{tX}] = \mathrm{e}^{\lambda(\mathrm{e}^t - 1)}$. En déduire en optimisant que

$$\mathbb{P}[X \geq k\lambda] \leq \mathrm{e}^{\lambda(k-1-k\log k)} \leq \mathrm{e}^{-\frac{\lambda(k-1)\log k}{2}}.$$

Montrer que si $(k-1)\lambda = \max(6, 2\lambda)$, alors la borne obtenue est plus petite que 0.05. Ainsi, on pourra dessiner les histogrammes pour $k \in [\![0, \max(3\lambda, 6+\lambda)]\!]$.

(3) En utilisant la méthode générale décrite à la fin de l'exercice précédent, écrire un programme qui tire au hasard des variables sous la loi de Poisson $\mathrm{Poi}(\lambda)$. Dessiner sur un même graphique l'histogramme de la loi $\mathrm{Poi}(3)$, et l'histogramme empirique d'un échantillon de $N = 1000$ tirages de variables suivant cette loi. On pourra utiliser `from scipy.special import factorial` pour avoir une fonction factorielle.

(4) On propose une autre méthode de simulation d'une variable suivant la loi $\mathrm{Poi}(\lambda)$: on tire au hasard des variables aléatoires $U_0, U_1, \ldots$ indépendantes et uniformes sur $[0, 1]$, et on renvoie :

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

Traiter la même question que ci-dessus avec cette nouvelle méthode. Question subsidiaire : pourquoi est-ce que cela marche ?

(5) Écrire un programme qui prend en paramètres deux réels positifs λ_1 et λ_2, et qui simule $X + Y$, où X et Y sont indépendants de lois de Poisson de paramètres respectifs λ_1 et λ_2. Tracer sur une même figure l'histogramme empirique d'un échantillon de taille $N = 1000$, et celui de la loi de Poisson de paramètre $\lambda_1 + \lambda_2$. Qu'en conclure ?

Avec `scipy`, on pourra plus tard utiliser la commande `scs.poisson(L).rvs(size=N)` pour engendrer N variables de Poisson de paramètre L et indépendantes.

4. Variables exponentielles.

On revient maintenant à des variables dont la loi a une fonction de répartition continue. Un cas particulier est celui où F_μ est dérivable :

$$F_\mu(x) = \int_{-\infty}^{x} f(s)\,\mathrm{d}s$$

pour une fonction positive mesurable $f : \mathbb{R} \to \mathbb{R}_+$ avec $\int_{-\infty}^{\infty} f(s)\,\mathrm{d}s = 1$. On dit alors que f est la *densité* de la loi μ ; elle est liée à la fonction mesurable $F = F_\mu$ par l'équation $F'(x) = f(x)$.

Le cas où $f(x) = \mathbb{1}_{(0\leq x\leq 1)}$ nous ramène aux variables uniformes sur $[0,1]$. Dans ce dernier exercice, on considère le cas d'une densité sur $\mathbb{R}_+$ qui décroît exponentiellement vite :

$$f(x) = \lambda\, \mathrm{e}^{-\lambda x}\, \mathbb{1}_{(x\geq 0)}$$

pour un certain paramètre $\lambda > 0$. On notera $X \sim \mathrm{Exp}(\lambda)$ si X est variable aléatoire dont la loi a pour densité f ; on dit alors que X suit une *loi exponentielle* de paramètre λ.

(1) Calculer la fonction de répartition d'une variable $X \sim \mathrm{Exp}(\lambda)$, ainsi que $1 - F(x) = \mathbb{P}[X > x]$.

(2) En déduire que si U suit une loi uniforme sur $[0,1]$, alors $X = -\frac{\log U}{\lambda}$ suit une loi $\mathrm{Exp}(\lambda)$. Écrire un programme qui prend en argument λ et N et renvoie N variables indépendantes exponentielles de paramètre λ.

(3) Dessiner sur un même graphique la fonction de répartition de la loi $\mathrm{Exp}(1)$, ainsi que la fonction de répartition empirique de $N = 10, 100, 1000$ variables exponentielles indépendantes avec cette loi. On choisira convenablement un intervalle pour l'axe des abscisses, de sorte que $\mathbb{P}[X \notin I] \leq 0.05$. Pour dessiner une fonction continue $y = F(x)$ sur un intervalle $I = (a, b)$, on pourra utiliser la commande `xx=np.linspace(a, b, 500)` pour avoir un échantillon régulier de points de l'intervalle, puis `ax.plot(xx, F(xx))` pour dessiner la fonction. Commenter les graphiques obtenus.

Avec `scipy`, on pourra plus tard utiliser la commande `scs.expon(scale=1/L).rvs(size=N)` pour engendrer N variables exponentielles de paramètre L et indépendantes (attention, l'argument `scale` doit être égal à $\frac{1}{L}$, et non L). La méthode de simulation de la question (2) sera généralisée dans le Chapitre 2, et la convergence des fonctions de répartition empiriques vers la fonction de répartition théorique F sera expliquée dans le Chapitre 3.