Modélisation en probabilités

Pierre-Loïc Méliot

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).

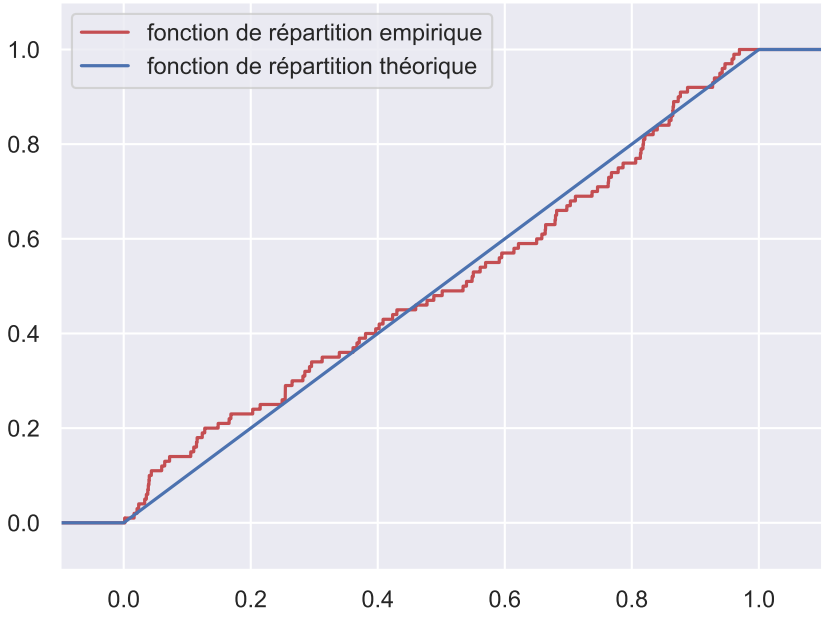

FIG. 1.1. Fonctions de répartition théorique et empirique d'un échantillon de taille 100 pour la loi uniforme sur $[0, 1]$.

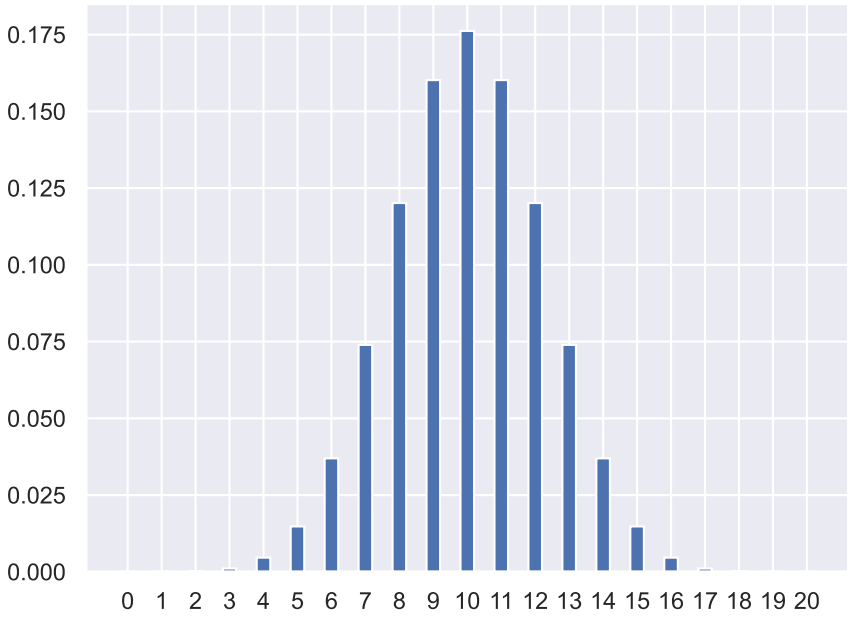

FIG. 1.2. Histogramme de la loi $\mathrm{Bin}(n = 20, p = 0.5)$.

- 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.

2. Méthodes de simulation de variables réelles

Soit X une variable aléatoire réelle de loi $\mathbb{P}_X$. La fonction de répartition

$$F_X(t) = \mathbb{P}_X[(-\infty, t]] = \mathbb{P}[X \leq t]$$

caractérise la loi de X. En effet, elle détermine les valeurs de $\mathbb{P}_X$ sur la classe stable par intersection des intervalles $(-\infty, t]$, et la tribu engendrée par ces intervalles est la tribu des boréliens, donc par le lemme de classe monotone, F_X détermine $\mathbb{P}_X$. Remarquons par ailleurs que :

(1) La fonction de répartition est croissante de $\mathbb{R}$ vers $[0, 1]$.

(2) Cette fonction est continue à droite en tout point : $\lim_{s \to t,\, s>t} F_X(s) = F_X(t)$.

(3) On a $\lim_{t \to -\infty} F_X(t) = 0$ et $\lim_{t \to +\infty} F_X(t) = 1$.

Nous verrons plus loin que toute fonction avec ces trois propriétés est la fonction de répartition d'une variable aléatoire réelle.

1. Simulation par inversion.

Soit $F : \mathbb{R} \to [0, 1]$ une fonction croissante et continue à droite, de limites 0 et 1 en $\pm\infty$. On appelle *inverse généralisée* de F la fonction $F^{-1} : (0, 1) \to \mathbb{R} \sqcup \{\pm\infty\}$ définie par

$$F^{-1}(u) = \inf\left(\{t \,|\, F(t) \geq u\}\right),$$

avec par convention $\inf(\emptyset) = +\infty$. Si la fonction F est continue strictement croissante, alors F^{-1} est simplement la bijection réciproque de F.

(1) Pour $0 < u < 1$, notons

$$A_u = \{t \,|\, F(t) \geq u\}.$$

Montrer que $A_u \neq \emptyset$ et que $A_u \neq \mathbb{R}$. Si $s \in A_u$ et $t > s$, montrer que $t \in A_u$. En déduire que $F^{-1}(u) \in \mathbb{R}$ et que $A_u = [F^{-1}(u), +\infty)$.

(2) Soit U une variable de loi uniforme sur $[0, 1]$. Calculer la fonction de répartition de la variable aléatoire $F^{-1}(U)$. En déduire que les trois propriétés listées au début du chapitre caractérisent les fonctions $F : \mathbb{R} \to [0, 1]$ qui sont des fonctions de répartition.

(3) Soit X une variable aléatoire à valeur réelle et F_X sa fonction de répartition. Pour U uniforme sur $[0, 1]$, quelle est la loi de $F_X^{-1}(U)$? En déduire une méthode pour simuler une variable aléatoire qui a la même loi que X.

(4) Retrouver la méthode de simulation des variables exponentielles de paramètre λ.

(5) La *loi de Cauchy* est la loi à densité

$$f(x) = \frac{1}{\pi(1 + x^2)}\, \mathrm{d}x$$

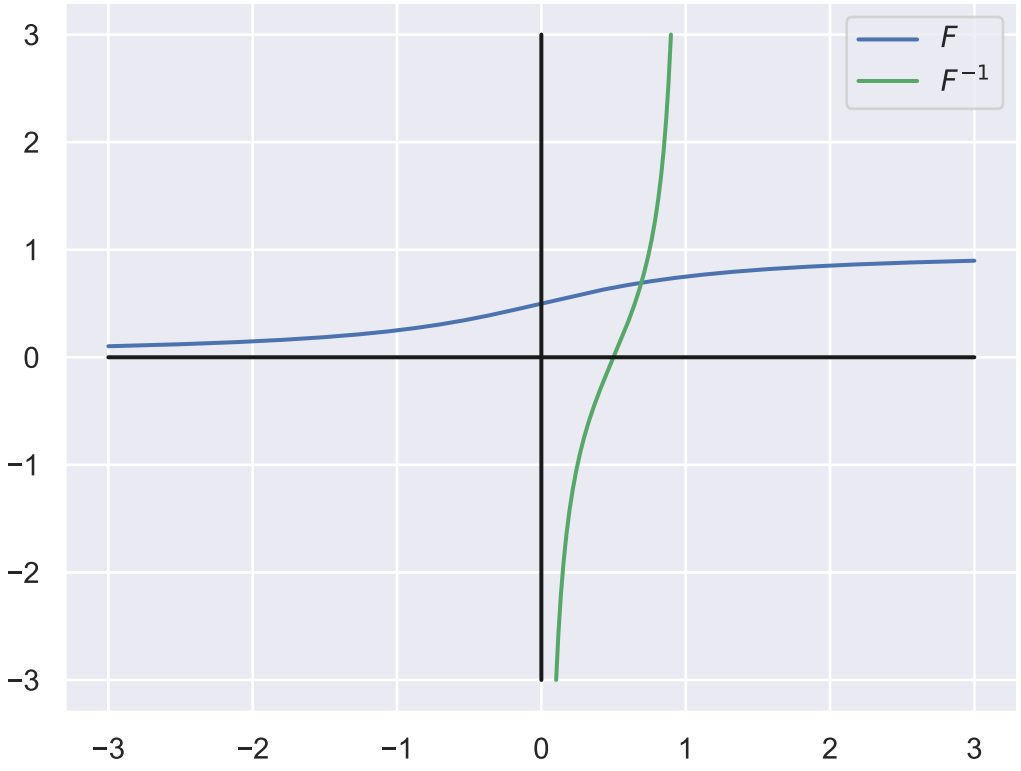

FIG. 2.1. Inverse d'une fonction de répartition.

sur $\mathbb{R}$. Calculer la fonction de répartition correspondante $F(x)$, puis son inverse généralisée $F^{-1}(x)$. Écrire un programme qui simule des variables aléatoires suivant la loi de Cauchy.

(6) Trouver m tel que si X suive la loi de Cauchy, alors $\mathbb{P}[|X| \geq m] \leq 0.05$. Dessiner sur un même graphique la fonction de répartition empirique d'un échantillon de $N = 10, 100, 1000$ variables de Cauchy indépendantes, et la fonction $F(x)$ calculée à la question précédente.

La méthode de *simulation par inversion* permet ainsi d'engendrer n'importe quelle variable aléatoire réelle *dont la fonction de répartition est calculable, ainsi que son inverse*. Pour traiter les cas où ceci n'est pas possible, on combinera cette méthode avec la méthode dite du *rejet*.

2. Méthode de rejet pour des variables à support compact.

Soit X une variable aléatoire dont la loi à un support compact inclus dans un intervalle $[a, b]$, et a une densité bornée $f : [a, b] \to [0, K]$.

(1) Écrire un programme `uniforme(a, b)` qui simule une variable de loi $\mathrm{Unif}([a, b])$, et donc de densité

$$g(x) = \frac{1_{a \leq x \leq b}}{b - a}.$$

Écrire ensuite un programme `point_uniforme(a, b, K)` qui renvoie un couple (X, Y) avec X et Y indépendants, $X \sim \mathrm{Unif}([a, b])$ et $Y \sim \mathrm{Unif}([0, K])$.

(2) On considère le cas où $[a,b]=[1,2]$ et $f(x)=C(x^2-1)\,1_{(1\leq x\leq 2)}$. Calculer C pour que f soit une densité de probabilité, puis K pour que f prenne ses valeurs dans $[0,K]$.

(3) Écrire un programme `nuage_points(N, f, a, b, K)` qui prend en argument une densité $f:[a,b]\to[0,K]$, tire au hasard N points indépendants $(X_1,Y_1),\dots,(X_N,Y_N)$ dans le rectangle $[a,b]\times[0,K]$, et les dessine, en mettant en bleu les points sous la courbe $y=f(x)$, et en vert les points au-dessus de la courbe. On pourra utiliser la commande `ax.scatter(X, Y, c=list_of_colors)` pour dessiner un nuage de points d'abscisses contenues dans le vecteur X, et d'ordonnées contenues dans le vecteur Y. Utiliser ce programme avec $N=200$ et la fonction f de la question précédente.

Dans le contexte de la question précédente, on note :

$$M=\inf(\{i\geq 1\,|\,f(X_i)>Y_i\})$$

et on dit que (X_M,Y_M) est le *point accepté*.

(4) Quelle est la loi de M ? Calculer ensuite

$$\mathbb{P}[M=n \text{ et } X_M\leq t]$$

pour un paramètre $t\in[a,b]$. En déduire que M et X_M sont indépendants, et que X_M a la loi de densité f.

(5) Écrire un programme `nuage_points_accept(N, f, a, b, K)` qui simule N points acceptés, les dessine dans le plan (ainsi que le graphe de la fonction f), et renvoie le vecteur formé des N abscisses de ces points. Tester ce programme avec la même fonction f que précédemment et $N=1000$. Dessiner sur un même graphique la fonction de répartition empirique du vecteur obtenu, et la fonction de répartition théorique F correspondant à f.

On a ainsi une méthode générale de simulation pour toute loi à densité bornée et à support compact. La troisième section étend ce principe, en retirant ces deux hypothèses sur la densité.

3. Méthode de rejet dans le cas général.

Soient f et g deux densités de probabilité sur $\mathbb{R}$, telles que $f(x)\leq\frac{1}{p}\,g(x)$ pour un certain $p\in(0,1)$ et tout x dans $\mathbb{R}$. On tire au hasard $X_1,X_2,\dots$ indépendants de densité g (par exemple avec la méthode d'inversion), puis $Y_1,Y_2,\dots$ définis par :

$$Y_i=\frac{1}{p}\,g(X_i)\,U_i,$$

où les U_i sont variables aléatoires indépendantes entre elles et indépendantes des variables X_i, de loi uniforme sur $[0,1]$. Remarquons que la construction de la section précédente correspondait au cas où $g(x)=\frac{1_{a\leq x\leq b}}{b-a}$ et $p=\frac{1}{K(b-a)}$.

On pose comme précédemment :

$$M=\inf(\{i\geq 1\,|\,f(X_i)>Y_i\}).$$

(1) En toute généralité, montrer que M suit une loi géométrique de paramètre p, et que X_M est indépendant de M, et a la loi de densité f.

(2) Écrire un programme `nuage_points_2(N, f, g, p, gen_g)` qui :

- prend en paramètre un entier N, deux densités f et g, un paramètre $p\in(0,1)$ tel que $f(x)\leq\frac{1}{p}\,g(x)$, et un générateur de variables aléatoires sous la densité g.

- tire au hasard N points $(X_1, Y_1), \dots, (X_N, Y_N)$ et les dessine dans le plan, en mettant en bleu les points sous la courbe $y = f(x)$ et en vert les points au-dessus de cette courbe. On mettra également sur le dessin les graphes des fonctions $f(x)$ et $\frac{1}{p}\, g(x)$.
- renvoie le vecteur formé des abscisses des points (X_i, Y_i) acceptés, c'est-à-dire tels que $f(X_i) > Y_i$.

(3) On pose $g(x) = \frac{1}{2}\, \mathrm{e}^{-|x|}$. Montrer que si $E \sim \mathrm{Exp}(1)$ et S est une variable indépendante de E, à valeurs dans $\{\pm 1\}$ et avec $\mathbb{P}[S = 1] = \mathbb{P}[S = -1] = \frac{1}{2}$, alors $X = ES$ suit la loi de densité g.

(4) Soit

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

la densité de la *loi normale (ou gaussienne)* $\mathrm{N}(0, 1)$. Trouver un paramètre p tel que $f(x) \leq \frac{1}{p}\, g(x)$, et utiliser le programme `nuage_points_2` avec ces fonctions, ce paramètre p, un générateur adéquat pour la loi g, et $N = 500$.

(5) Modifier le programme `nuage_points_2` pour que le paramètre N donne le nombre de points acceptés du résultat. Dessiner sur un même graphique la fonction de répartition empirique du vecteur obtenu (toujours avec $N = 500$ et les fonctions f et g ci-dessus), et la fonction de répartition théorique de la gaussienne, qui est donnée par la commande `scs.norm.cdf` (appliquée à un *array* `numpy` de valeurs).

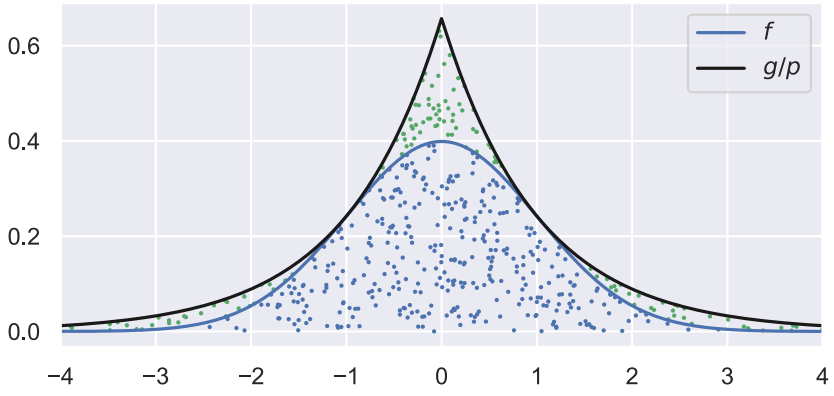

FIG. 2.2. Méthode de rejet : on tire des abscisses aléatoires de loi à densité à g, des ordonnées aléatoires $Y_i = \frac{1}{p}\, g(X_i)\, U_i$, et on accepte les abscisses des points placés sous la courbe $y = f(x)$.

3. Convergence presque sûre

Les précédents chapitres ont expliqué comment, à partir de la commande `random.random()`, on pouvait engendrer un échantillon $(X_1, X_2, \dots, X_N)$ de N variables aléatoires suivant une loi donnée (pas forcément la loi uniforme sur $[0,1]$). On s'intéresse maintenant aux propriétés statistiques de ces échantillons, en particulier lorsque N est grand.

1. Loi des grands nombres.

La *loi des grands nombres* est le résultat suivant : si $(X_1, X_2, \dots, X_n)$ est un n-échantillon de variables réelles suivant toutes la même loi μ et si μ admet un moment d'ordre 1 :

$$\int_{\mathbb{R}} |x|\, \mu(\mathrm{d}x) < +\infty,$$

alors la suite des *moyennes empiriques* $M_n = \frac{X_1+X_2+\cdots+X_n}{n}$ tend presque sûrement lorsque n tend vers l'infini vers

$$m = \mathbb{E}[X_1] = \int_{\mathbb{R}} x\, \mu(\mathrm{d}x).$$

Ainsi, $\mathbb{P}[\lim_{n\to\infty} M_n = m] = 1$.

(1) Écrire un programme qui prend un échantillon $(X_1, X_2, \dots, X_N)$ et un paramètre m, et qui dessine le graphe de la suite des moyennes $(M_n)_{1\leq n\leq N}$, et le compare à la valeur m (représentée par une droite horizontale d'ordonnée m).

FIG. 3.1. Convergence presque sûre des moyennes empiriques d'une suite de variables i.i.d. vers la moyenne théorique.

(2) Utiliser ce programme avec un échantillon de taille $N = 1000$ de variables indépendantes et exponentielles de loi Exp(1). On prendra $m = \mathbb{E}[X_1]$. La loi des grands nombres est-elle bien mise en évidence ?

2. Urnes de Bernoulli–Laplace.

De façon générale, on dit qu'une suite de variables aléatoires $(M_n)_{n\in\mathbb{N}}$ *converge presque sûrement* vers une variable X si $\mathbb{P}[\lim_{n\to\infty} M_n = X] = 1$. La loi des grands nombres assure la convergence presque sûre lorsque M_n est une moyenne empirique de variables indépendantes et de même loi admettant une espérance m, avec une limite $X = m$. Il y a de nombreux autres résultats de convergence presque sûre en probabilité, avec une limite X qui peut être aléatoire. Cet exercice présente un tel exemple, à partir d'un modèle d'urne.

(1) On considère une urne qui contient initialement $B_0 = 1$ boule blanche et $R_0 = 1$ boule rouge. À chaque instant $n \geq 0$, on tire uniformément l'une des $n + 2$ boules de l'urne au temps n (indépendamment de tout ce qui s'est passé avant), et on la remplace par deux boules de la même couleur pour obtenir l'urne au temps $n + 1$. On a donc deux suites croissantes $(B_n)_{n\geq 0}$ et $(R_n)_{n\geq 0}$ d'entiers aléatoires, avec $n+2 = B_n + R_n$ pour tout n. Écrire un programme `urn(N)` qui tire au hasard la suite $(B_n)_n$ jusqu'au rang N. On pourra calculer les probabilités conditionnelles $\mathbb{P}[B_{n+1} = k + 1 \,|\, B_n = k]$ et $\mathbb{P}[B_{n+1} = k \,|\, B_n = k]$.

(2) On note $M_n = \frac{B_n}{n+2}$; c'est la proportion de boules blanches au rang n. Dessiner la fonction $n \mapsto M_n$ pour plusieurs tirages de la suite, et pour $n \in [\![0, N]\!]$ avec $N = 1000$. La suite $(M_n)_{n\in\mathbb{N}}$ converge-t-elle presque sûrement ?

(3) Dessiner la fonction de répartition empirique de 1000 tirages de la variable aléatoire M_{1000}. Que peut-on conjecturer ?

3. Moyennes de variables de Cauchy.

Cet exercice examine le cas où l'hypothèse de la loi des grands nombres d'existence d'un moment d'ordre 1 n'est plus vérifiée. Ainsi, on considèrera des variables de Cauchy, dont on rappelle que la densité est la fonction $f(x) = \frac{1}{\pi(1+x^2)}$. Un échantillon de taille N de variables aléatoires de Cauchy peut être obtenu avec la commande `scs.cauchy.rvs(size=N)`.

(1) Soit $(X_1, X_2, \dots, X_N)$ un N-échantillon de variables de Cauchy, et $(M_1, M_2, \dots, M_N)$ la suite des moyennes empiriques correspondantes. Dessiner plusieurs fois la fonction $n \mapsto M_n$ sur l'intervalle $[\![1, N]\!]$, avec $N = 5000$. Qu'observe-t-on ? Y-a-t'il une convergence presque sûre ?

(2) On note μ_n la loi de $\frac{X_1+\cdots+X_n}{n}$, où les X_i sont des variables aléatoires de Cauchy indépendantes. On souhaite déterminer cette loi μ_n, et sa fonction de répartition F_n. Écrire un programme `loi_empirique_moyenne_cauchy(N, n)` qui construit un échantillon $Z_1, \dots, Z_N$ de variables aléatoires indépendantes de loi μ_n, et qui trace la fonction de répartition empirique de cet échantillon.

(3) Utiliser ce programme pour dessiner sur un même graphique des versions approchées des fonctions de répartition F_1, F_2 et F_{10}. Que peut-on conjecturer sur la loi μ_n de $\frac{X_1+\cdots+X_n}{n}$ en fonction de n ?

(4) Relier la conjecture de la question précédente aux observations de la première question.

4. Théorème de Glivenko–Cantelli.

Soit μ une loi de variables aléatoires réelles, $X_1, \dots, X_n$ des réalisations indépendantes de cette loi, et F_n^X la fonction de répartition empirique de cet échantillon. Le *théorème de Glivenko–Cantelli* assure que lorsque n tend vers l'infini, on a

$$\|F_n^X - F_\mu\|_\infty \to_{\text{presque sûrement}} 0.$$

On s'est déjà servi implicitement de ce résultat, en approximant une fonction de répartition ou un histogramme théorique par l'objet empirique correspondant obtenu à partir d'un échantillon de grande taille. Les premières questions de l'exercice donnent une preuve complète du théorème, et les questions suivantes s'intéressent à la vitesse de convergence.

(1) On souhaite d'abord montrer que pour tout $x \in \mathbb{R}$, $F_n^X(x)$ converge presque sûrement vers $F_\mu(x)$. Relier ce résultat à la loi des grands nombres pour des variables de Bernoulli de paramètre $p \in (0,1)$. Si $B_1, \dots, B_n$ sont des variables indépendantes de loi $\mathrm{Ber}(p)$ et $S_n = B_1 + \cdots + B_n$, montrer que la convergence presque sûre $\frac{S_n}{n} \to p$ est impliquée par la condition suivante :

$$\mathbb{E}\Big[(S_n - np)^4\Big] = O(n^2).$$

On pourra considérer la série $\sum_{n\geq 1} \mathbb{P}\Big[\Big|\frac{S_n}{n} - p\Big| \geq n^{-\frac{1}{8}}\Big]$, et utiliser une forme de l'inégalité de Markov.

(2) Montrer que $\mathbb{E}[S_n] = np$ et $\mathbb{E}[(S_n)^2] = np + n(n-1)p^2$. Pour le second moment, on pourra développer $(\sum_{i=1}^n B_i)^2$ et regrouper les termes B_iB_j selon que $i = j$ ou $i \neq j$. En exploitant cette idée, montrer qu'on a de même

$$\mathbb{E}[(S_n)^3] = np + 3\,n^{\downarrow 2}p^2 + n^{\downarrow 3}p^3;$$
$$\mathbb{E}[(S_n)^4] = np + 7\,n^{\downarrow 2}p^2 + 6\,n^{\downarrow 3}p^3 + n^{\downarrow 4}p^4$$

avec $n^{\downarrow k} = n(n-1)\cdots(n-k+1)$.

(3) Dans `Python`, on peut manipuler des polynômes en n et p avec les commandes suivantes :

```
from sympy import var, expand
var("n,p")
M1 = n*p
M2 = n*p + n*(n-1)*(p**2)
```

Alors, la commande `expand(M2)` renvoie $n^2p^2 - np^2 + np$. En utilisant les formules pour les moments d'ordre 1 à 4, calculer $\mathbb{E}[(S_n - np)^4]$, et en déduire la convergence ponctuelle des fonctions de répartition empiriques.

(4) Notons F la fonction de répartition théorique de la loi uniforme. On suppose établie la convergence presque sûre $\|F_n^U - F\|_\infty \to 0$ pour les fonctions de répartition empiriques de variables $U_1, U_2, \dots, U_n$ indépendantes et uniformes sur $[0,1]$. En utilisant la méthode de simulation par inversion, en déduire la convergence presque sûre $\|F_n^X - F_\mu\|_\infty \to 0$ dans le cas général, c'est-à-dire pour les fonctions de répartition empiriques de variables $X_1, X_2, \dots, X_n$ indépendantes et de loi μ.

(5) Soit $(f_n)_{n\in\mathbb{N}}$ une suite de fonctions de $\mathbb{R}$ vers $[0,1]$ qui sont croissantes. On suppose que $(f_n)_{n\in\mathbb{N}}$ converge ponctuellement vers une fonction $f : \mathbb{R} \to [0,1]$ qui est continue (et bien sûr croissante). Montrer alors que $\|f_n - f\|_\infty \to 0$ (c'est le *théorème de Dini*), et conclure la preuve du théorème de Glivenko–Cantelli.

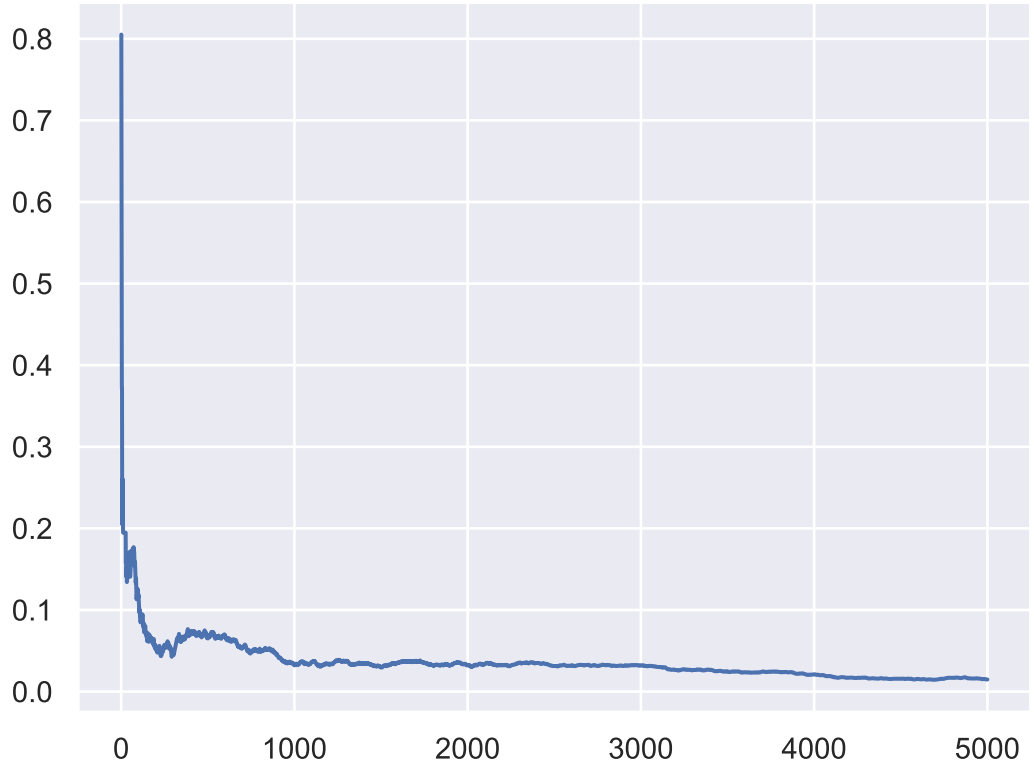

FIG. 3.2. Glivenko–Cantelli : décroissance presque sûre vers 0 de $\|F_n - F\|_\infty$.

Dans toute la suite, F_n désigne la fonction de répartition empirique d'un échantillon $U_1, \dots, U_n$ de variables indépendantes uniformes dans $[0,1]$, et $(U_{(1)} \leq U_{(2)} \leq \cdots \leq U_{(n)})$ désigne le réordonnement croissant de ces variables.

(6) Montrer que

$$\|F_n - F\|_\infty = \max\left(\max_{k=1,\dots,n}\left\{\frac{k}{n} - U_{(k)}\right\}, \max_{k=1,\dots,n}\left\{U_{(k)} - \frac{k-1}{n}\right\}\right).$$

Écrire un programme `distance_loi_uniforme(N)` qui tire au hasard un échantillon $U_1, \dots, U_N$, et qui renvoie le vecteur des distances $\|F_n - F\|_\infty$ avec $n \in [\![1, N]\!]$. On prendra garde au fait qu'au temps n, on doit utiliser le réordonnement croissant de $U_1, \dots, U_n$, et pas les n premiers termes du réordonnement croissant de $U_1, \dots, U_N$.

(7) Tracer le graphe de

$$n \mapsto \|F_n - F\|_\infty$$

pour $n \in [\![1, 5000]\!]$, et vérifier qu'il semble y avoir une décroissance en $n^{-\alpha}$ pour un certain exposant α. Déterminer la valeur de cet exposant en traçant le graphe de

$$n \mapsto \frac{\log \|F_n - F\|_\infty}{\log(n+1)}.$$

(8) Tracer la différence renormalisée $n^\alpha(F_n(x) - F(x))$ sur l'intervalle $[0,1]$, pour $n = 5000$. Commenter.

4. Convergence en loi

La convergence en loi est une notion de convergence pour les variables aléatoires qui est plus faible que la convergence presque sûre, et qui capture l'idée suivante : $(X_n)_{n\in\mathbb{N}}$ *converge en loi* (ou *en distribution*) vers une variable X si les probabilités relatives à X_n (mais pas les valeurs de X_n) convergent vers les probabilités relatives à X. Pour des raisons topologiques, il faut faire un peu attention à la définition précise de convergence des probabilités. Les conditions suivantes s'avèrent être équivalentes pour une suite de variables aléatoires réelles $(X_n)_{n\in\mathbb{N}}$:

(1) *Convergence des observables.* Pour toute fonction continue bornée $f : \mathbb{R} \to \mathbb{R}$,

$$\mathbb{E}[f(X_n)] \to_{n\to\infty} \mathbb{E}[f(X)].$$

(2) *Convergence des probabilités.* Pour toute partie mesurable A telle que $\mathbb{P}[X \in \partial A] = 0$,

$$\mathbb{P}[X_n \in A] \to_{n\to\infty} \mathbb{P}[X \in A].$$

(3) *Convergence des fonctions de répartition.* Pour tout point t tel que F_X soit continue en t,

$$F_{X_n}(t) \to_{n\to\infty} F_X(t).$$

(4) *Convergence des transformées de Fourier.* Pour tout réel ξ,

$$\mathbb{E}[e^{i\xi X_n}] \to_{n\to\infty} \mathbb{E}[e^{i\xi X}].$$

On admettra l'équivalence de ces conditions, et on dira si elles sont vérifiées que $(X_n)_{n\in\mathbb{N}}$ converge en loi vers X (notation : $X_n \rightharpoonup_{n\to\infty} X$). Notons que si $(X_n)_{n\in\mathbb{N}}$ est une suite de variables aléatoires qui converge presque sûrement vers X, alors par le théorème de convergence dominée, on a convergence des observables : la convergence presque sûre est donc plus forte que la convergence en loi. Si les variables X_n et X sont à valeurs dans $\mathbb{Z}$, alors la convergence en loi est simplement donnée par :

(5) *Convergence locale.* Pour tout $k \in \mathbb{Z}$,

$$\mathbb{P}[X_n = k] \to_{n\to\infty} \mathbb{P}[X = k].$$

Si $X \sim \mu$, on notera aussi la convergence en loi $X_n \rightharpoonup_{n\to\infty} \mu$.

1. Convergence vers la loi de Poisson.

On fixe un paramètre $\lambda > 0$ et on travaille avec des entiers $n \geq \lambda$.

(1) Dresser les histogrammes empiriques de N tirages $X_1, \dots, X_N$ d'une loi binomiale $\mathrm{Bin}(n, \frac{\lambda}{n})$ avec $N = 1000$, $n = 50, 1000$ et $\lambda = 1, 3, 5$. Comparer avec les histogrammes des lois de Poisson $\mathrm{Poi}(\lambda)$.

(2) Montrer avec le critère de convergence locale qu'on a effectivement $\mathrm{Bin}(n, \frac{\lambda}{n}) \rightharpoonup_{n\to\infty} \mathrm{Poi}(\lambda)$.

(3) Donner une autre preuve reposant sur la convergence des transformées de Fourier.

2. Théorème central limite.

Le *théorème central limite* est un énoncé de convergence en loi pour la différence renormalisée entre une moyenne empirique de variables indépendantes et de même loi, et la moyenne théorique. Ainsi, si $X_1, X_2, \ldots$ sont des variables indépendantes dont la loi commune μ admet un moment d'ordre 2 :

$$\int_{\mathbb{R}} x^2 \, \mu(\mathrm{d}x) < +\infty,$$

et si $M_n = \frac{X_1+X_2+\cdots+X_n}{n}$ et $m = \mathbb{E}[X_1]$, alors

$$\sqrt{n}\left(\frac{M_n - m}{\sigma}\right) \rightharpoonup_{n\to\infty} \mathrm{N}(0,1),$$

où

$$\sigma^2 = \left(\int_{\mathbb{R}} x^2 \, \mu(\mathrm{d}x)\right) - \left(\int_{\mathbb{R}} x \, \mu(\mathrm{d}x)\right)^2$$

désigne la variance de la loi μ, et $\mathrm{N}(0,1)$ est la loi gaussienne de densité $f(x) = \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{x^2}{2}}$ sur $\mathbb{R}$. La preuve repose sur le critère des transformées de Fourier, et sur le développement de Taylor $\mathbb{E}[\mathrm{e}^{\mathrm{i}\xi(X_1-m)}] = \exp\left(-\frac{\sigma^2\xi^2}{2} + o(1)\right)$.

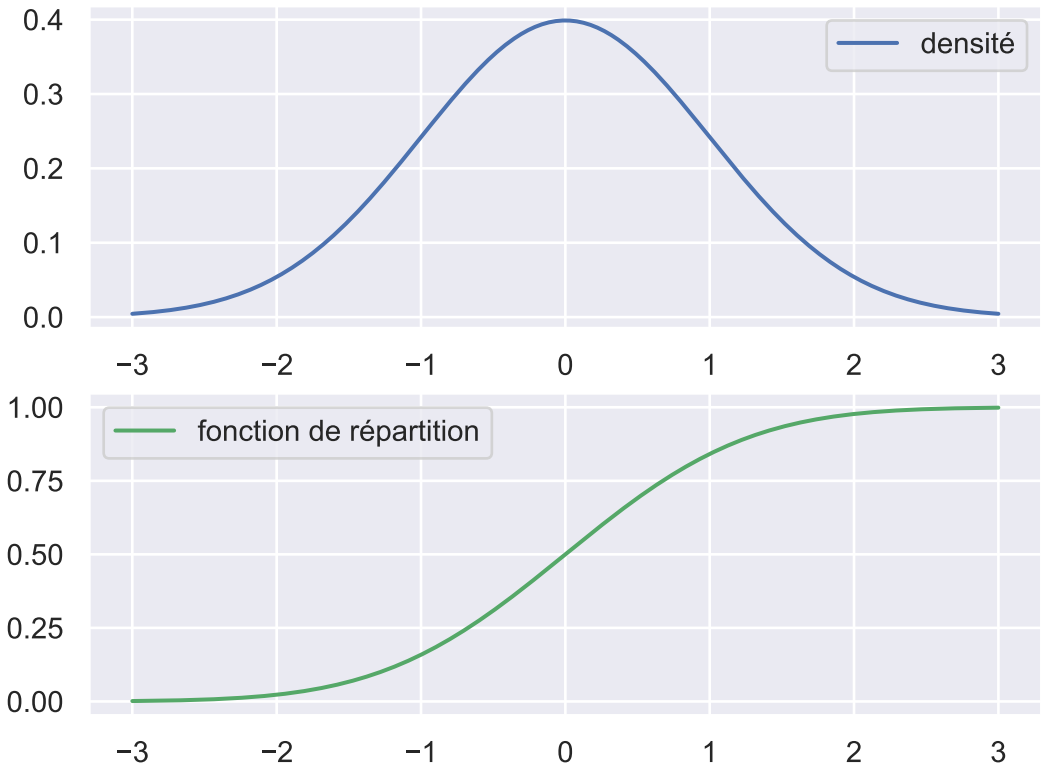

FIG. 4.1. Densité et fonction de répartition de la loi normale $\mathrm{N}(0,1)$.

(1) Écrire un programme qui tire un échantillon $(X_1, \dots, X_N)$ de variables aléatoires de même loi, et qui calcule le vecteur formée par toutes les moyennes recentrées et renormalisées :

$$\left(Z_n = \sqrt{n} \left(\frac{\frac{X_1+\cdots+X_n}{n} - m}{\sigma} \right) \right)_{n \in [\![1,N]\!]}.$$

On pourra faire des essais avec différentes lois : par exemple, la loi uniforme sur $[0,1]$, la loi exponentielle $\mathrm{Exp}(1)$, et la loi de Bernoulli $\mathrm{Ber}(\frac{1}{2})$.

(2) Tracer pour la loi choisie le graphe de $n \mapsto Z_n$ pour $n \in [\![1, 5000]\!]$. A-t-on une convergence presque sûre ?

(3) Tracer la fonction de répartition empirique d'un échantillon de $M = 1000$ variables Z_{1000}, et la comparer à la fonction de répartition de la gaussienne, qui est obtenue par la commande `scs.norm.cdf`.

(4) On tire au hasard $p \in [0,1]$ avec la commande `random.random()`, puis on observe des variables de Bernoulli $B_1, B_2, \dots, B_n$ de paramètre p, qui reste inconnu. À l'aide du théorème central limite, construire un *estimateur* $\hat{p}_n = \hat{p}_n(B_1, \dots, B_n)$ du paramètre p, puis un *intervalle de confiance* $[\hat{p}_{n,-}, \hat{p}_{n,+}]$ tel que

$$\lim_{n\to\infty} \mathbb{P}[p \in [\hat{p}_{n,-}, \hat{p}_{n,+}]] \geq 0.98.$$

Tester cet intervalle de confiance avec un échantillon de taille $n = 1000$.

La dernière question est la base théorique de la théorie des sondages.

3. Variables géométriques renormalisées.

Soit $\lambda > 0$ un paramètre fixé, et X_n une variable aléatoire géométrique de paramètre $p = \frac{\lambda}{n}$:

$$\mathbb{P}[X_n = k] = (1-p)^{k-1}\, p \text{ pour tout entier } k \geq 1.$$

On pose $Y_n = \frac{X_n}{n}$.

(1) Écrire un programme qui simule la variable Y_n. Dresser la fonction de répartition empirique de cette variable pour $n = 3, 10, 100$.

(2) Comparer le cas $n = 100$ avec la fonction de répartition théorique d'une variable réelle de loi exponentielle $\mathrm{Exp}(\lambda)$. Que peut-on conjecturer ?

(3) Que vaut $\mathbb{P}[Y_n \geq s]$ pour $s \in \mathbb{R}_+$? En déduire une preuve de la conjecture de la question précédente.

4. Maximum de variables aléatoires exponentielles.

Dans cet exercice, $X_1, X_2, \dots$ est une suite de variables aléatoires indépendantes de loi $\mathrm{Exp}(1)$, et on s'intéresse à la suite des maxima :

$$L_n = \max(X_1, X_2, \dots, X_n).$$

(1) Écrire un programme qui trace le graphe de $n \mapsto \frac{L_n}{\log n}$ pour $n \in [\![2, 5000]\!]$. A-t-on une convergence presque sûre ?

(2) Quelle est la fonction de répartition de L_n ? Montrer que

$$\sum_{n\geq 1} \mathbb{P}[L_n \leq (1-\varepsilon)\log n] < +\infty$$

et que

$$\sum_{n\geq 1} \mathbb{P}[L_{\alpha_n} \geq (1+\varepsilon)\log \alpha_n] < +\infty$$

pour tout $\varepsilon > 0$, et pour une suite croissante $(\alpha_n)_{n\geq 1}$ appropriée. En déduire une preuve de la conjecture de la question précédente. On pourra s'appuyer sur le lemme de Borel-Cantelli pour contrôler

$$\liminf_{n\to\infty} \frac{L_n}{\log n} \quad \text{et} \quad \limsup_{n\to\infty} \frac{L_{\alpha_n}}{\log \alpha_n},$$

et pour la limite supérieure de L_n (au lieu de L_{α_n}), on pourra utiliser la croissance de la suite $(L_n)_{n\geq 1}$.

(3) Tracer maintenant le graphe de $n \mapsto L_n - \log n$ pour $n \in [\![1, 5000]\!]$. A-t-on une convergence presque sûre ?

(4) Calculer la limite de $\mathbb{P}[L_n \leq \log n + t]$, et en déduire la convergence en loi de la suite $(L_n - \log n)_{n\geq 1}$. La loi limite est appelée *loi de Gumbel*.

(5) Dessiner sur un même graphique la fonction de répartition empirique d'un échantillon de $N = 1000$ tirages de la variable $L_{1000} - \log 1000$, et la fonction de répartition de la loi de Gumbel.

5. Chaînes de Markov

Soit $\mathfrak{X}$ un ensemble fini, dont on numérote les éléments de 0 à $N-1$, avec $N = \operatorname{card}\mathfrak{X}$: $\mathfrak{X} = \{x_0, x_1, \dots, x_{N-1}\}$. Une *matrice stochastique* (ou *matrice de transition*) sur $\mathfrak{X}$ est une matrice carrée de taille $N \times N$ à coefficients positifs, dont on note $P(x_i, x_j)$ l'élément sur la i-ième ligne et la j-ième colonne, et telle que :

$$\forall i \in \{0, 1, \dots, N-1\},\ \sum_{j=0}^{N-1} P(x_i, x_j) = 1.$$

Autrement dit, chaque ligne de la matrice P est une mesure de probabilité sur $\mathfrak{X}$. Une *chaîne de Markov* de matrice P est une suite de variables aléatoires $(X_n)_{n\in\mathbb{N}}$ avec chaque $X_n \in \mathfrak{X}$, et telle que pour tout $n \geq 0$ et tout $(n+1)$-uplet $(a_0, a_1, \dots, a_n) \in \mathfrak{X}^{n+1}$, on a :

$$\mathbb{P}[X_0 = a_0, X_1 = a_1, \dots, X_n = a_n] = \mathbb{P}[X_0 = a_0]\, P(a_0, a_1)\, P(a_1, a_2) \cdots P(a_{n-1}, a_n).$$

L'idée est la suivante :

- on tire au hasard X_0 suivant une certaine loi π_0 sur $\mathfrak{X}$ (la *loi initiale* de la chaîne).
- si $X_0, X_1, \dots, X_{n-1}$ sont construits, alors X_n est obtenu en se plaçant en $X_{n-1} = a_{n-1}$, et en effectuant une transition vers $X_n = a_n$ avec a_n choisi suivant la ligne a_{n-1} de la matrice P.

En effet, la formule définissant une chaîne de Markov est équivalente à la formule de probabilité conditionnelle :

$$\mathbb{P}[X_n = a_n \,|\, X_0 = a_0, \dots, X_{n-1} = a_{n-1}] = P(a_{n-1}, a_n).$$

On peut donc voir une chaîne de Markov de matrice P comme une marche aléatoire sur un graphe dont les sommets sont les éléments de l'*espace des états* $\mathfrak{X}$, et dont les arêtes orientées sont les $(x \to y)$ avec $P(x, y) > 0$ (*transitions possibles*).

1. Programmes généraux.

Pour tirer au hasard un élément d'un *array* `L` avec une distribution discrète de probabilités `P`, on pourra utiliser la commande `random.choice(L, p=P)`. Si à la place de `L` on met un entier `n`, la liste des choix possibles sera $[\![0, n-1]\!]$.

(1) Écrire deux programmes `verif_proba(pi)` et `verif_matrix(P)` qui vérifient qu'un vecteur est une probabilité, et qu'une matrice est stochastique. Pour tester l'égalité de nombres réels avec décimales, on pourra utiliser `np.isclose`, qui évite les imprécisions dues à l'écriture décimale. Vérifier ces programmes avec $\pi_0 = (0.2, 0.5, 0.3)$ et

$$P = \begin{pmatrix} \frac{1}{3} & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{4} & 0 & \frac{3}{4} \\ 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}.$$

(2) Écrire un programme `trajectoire_markov(pi0, P, n)` qui prend en argument un vecteur de probabilités de taille N, une matrice stochastique de taille $N \times N$, et un entier n, et qui calcule les $n+1$ premiers pas $(X_0, X_1, \dots, X_n)$ d'une chaîne de

Markov sur $\mathfrak{X} = [\![0, N-1]\!]$, avec loi initiale π_0 et matrice stochastique P. On tirera donc au hasard X_0 suivant π_0, puis les transitions suivant les lignes de la matrice P. Expérimenter avec les paramètres π_0 et P de la question précédente, et $n = 30$.

(3) Pour $n \geq 0$, on note π_n la loi marginale d'une chaîne de Markov $(X_n)_{n\in\mathbb{N}}$ de matrice P sur un espace d'états $\mathfrak{X}$. C'est donc le vecteur

$$\pi_n = (\mathbb{P}[X_n = x_1], \mathbb{P}[X_n = x_2], \dots, \mathbb{P}[X_n = x_N])$$

si $\mathfrak{X} = \{x_1, \dots, x_N\}$. Montrer qu'on a pour tout $n \geq 0$:

$$\pi_n = \pi_0 \, P^n,$$

où le produit à droite est le produit matriciel d'un vecteur ligne de taille N avec une matrice carrée $N \times N$. Écrire un programme `loi_marginale(pi0, P, n)` qui calcule le vecteur π_n. Pour effectuer des produits matriciels, on pourra utiliser les commandes `np.matmul` et `np.linalg.matrix_power`.

(4) Avec les mêmes paramètres que précédemment, dessiner sur un même graphique l'histogramme empirique de $N = 1000$ tirages de X_{30}, et l'histogramme théorique de π_{30}, que l'on calculera avec le programme `loi_marginale`.

(5) Toujours avec la même loi initiale et la même matrice de transition, comparer les vecteurs π_{30} et π_{300}. Que peut-on conjecturer pour

$$\lim_{n\to\infty} \mathbb{P}[X_n = x] \, ?$$

(6) En utilisant la méthode `np.linalg.eig` et la transposition `P.T` de matrices, trouver les valeurs propres $\lambda_1, \lambda_2, \lambda_3$ de P, et des vecteurs propres à gauche ν_1, ν_2, ν_3 tels que $\nu_i P = \lambda_i \nu_i$. En déduire une preuve de la conjecture de la question précédente, pour les chaînes de Markov de matrice P introduite à la question (1).

(7) Dessiner sur un même graphique l'histogramme de la loi limite déterminée à la question précédente, et l'histogramme empirique des $N = 1000$ premières valeurs d'une trajectoire de la chaîne de Markov. En déduire une autre conjecture concernant la limite presque sûre de

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

2. Marche aléatoire sur le cercle.

Les conjectures des dernières questions de l'exercice précédent sont résolues par les résultats suivants, que l'on admettra. Soit $\mathfrak{X}$ un ensemble fini et P une matrice stochastique sur $\mathfrak{X}$. On dit que P est :

- *irréductible* si, pour tous états $x, y \in \mathfrak{X}$, il existe $n \geq 1$ tel que $P^n(x, y) > 0$. Autrement dit, le graphe orienté associé à la matrice P est connexe.
- *à diagonale non nulle* s'il existe un état $x \in \mathfrak{X}$ tel que $P(x, x) > 0$.

Les trois théorèmes ci-dessous décrivent le comportement en temps long d'une chaîne de Markov finie irréductible :

(a) Si P est irréductible, alors il existe un unique vecteur de probabilité π tel que $\pi P = \pi$. On dit que π est la *loi invariante* ou *stationnaire* d'une chaîne de Markov de matrice stochastique P. Cette loi charge tout l'espace $\mathfrak{X}$: $\forall x \in \mathfrak{X},\ \pi(x) > 0$.

(b) *Théorème ergodique.* Si P est irréductible et π est sa loi stationnaire, alors pour toute loi initiale π_0, étant donnée une chaîne de Markov $(X_n)_{n\in\mathbb{N}}$ de loi initiale π_0 et de

matrice de transition P, la loi empirique

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

converge presque sûrement vers π lorsque N tend vers l'infini. Autrement dit, pour tout état $x \in \mathfrak{X}$, la fréquence de visites $\mu_N(x)$ tend p.s. vers $\pi(x)$.

(c) *Convergence des lois marginales.* Si P est irréductible et à diagonale non nulle, alors pour toute loi initiale π_0, on a aussi convergence des lois marginales :

$$\pi_n = \pi_0 P^n \to_{n\to\infty} \pi$$

dans l'espace $\mathbb{R}^{\mathfrak{X}}$. Autrement dit, pour tout état $x \in \mathfrak{X}$, $\pi_n(x)$ tend vers $\pi(x)$.

On admettra ces résultats ; l'exercice qui suit illustre ces résultats pour la *marche aléatoire sur un cercle* (discrétisé).

(1) On se place sur $\mathfrak{X} = \mathbb{Z}/N\mathbb{Z}$ avec $N \geq 3$; pour les simulations, on pourra prendre par exemple $N = 10, 50$. Écrire un programme `marche_aleatoire_cercle(N, n, k=1)` qui dessine k trajectoires indépendantes de la marche aléatoire sur $\mathbb{Z}/N\mathbb{Z}$, dont la matrice de transition est :

$$P(k,l) = \begin{cases} \frac{1}{3} & \text{si } l = k,\ k-1 \text{ ou } k+1 \bmod N, \\ 0 & \text{sinon.} \end{cases}$$

Ainsi, la marche aléatoire fait à chaque étape un pas négatif avec probabilité $\frac{1}{3}$, un pas positif avec probabilité $\frac{1}{3}$, et reste au même endroit avec probabilité $\frac{1}{3}$. Pour réduire modulo N un entier r, on utilisera la commande `r % N`. Le paramètre `n` du programme est le temps jusqu'auquel on dessinera les trajectoires. Pour la loi initiale, on pourra prendre $X_0 = \lfloor \frac{N}{2} \rfloor$ presque sûrement. Tester le programme avec $n = 1000$, $k = 3$.

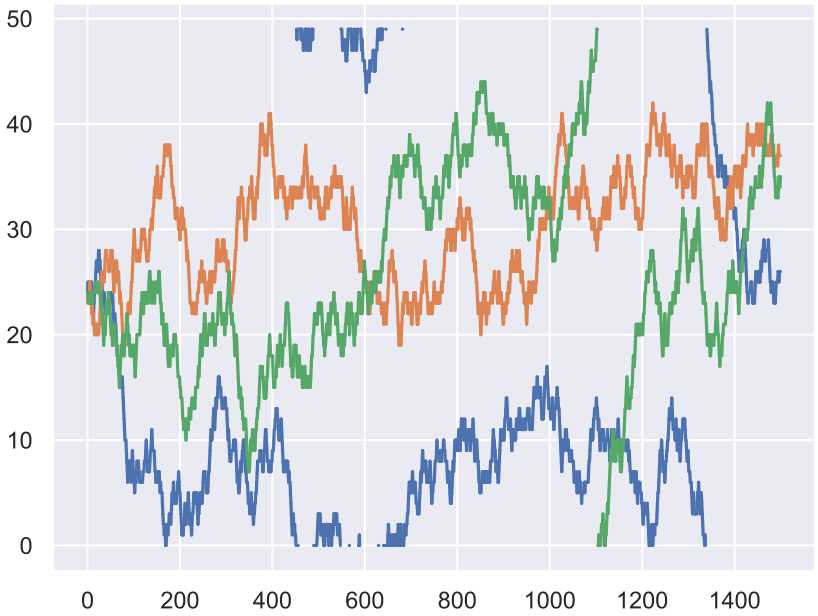

FIG. 5.1. Marches aléatoires sur le cercle.

(2) Quelle est la loi stationnaire de cette chaîne de Markov ? Dessiner sur un même graphique toutes les fonctions $k \mapsto \frac{1}{k}\sum_{i=1}^{k} 1_{(X_k=x)}$, avec $k \in [\![0, 1000]\!]$ et $x \in \mathbb{Z}/N\mathbb{Z}$, $N = 10$. Commenter ce dessin.

(3) Dessiner aussi sur un même graphique l'histogramme de la loi stationnaire, et l'histogramme empirique de 10000 tirages de la variable X_{10000}. Commenter les histogrammes obtenus.

(4) Soit $T = \inf(\{n \in \mathbb{N} \,|\, \{X_0, X_1, \dots, X_n\} = \mathbb{Z}/N\mathbb{Z}\})$ le *temps de couverture* de l'espace des états par la marche aléatoire. Écrire un programme qui simule cette variable aléatoire. Dessiner la fonction de répartition empirique d'un échantillon de taille $M = 1000$ de cette variable aléatoire, avec $N = 10, 50$. Commenter.

3. Urnes d'Ehrenfest.

On fixe un entier $N \geq 1$, et on considère une boîte contenant N particules numérotées et tombant dans deux compartiments A et B. À chaque étape :

- on tire au hasard l'une des particules numérotées de 1 à N, chaque particule étant équiprobable.
- avec probabilité $\frac{1}{2}$, on fait passer la particule tirée au hasard de son compartiment vers l'autre compartiment (donc, de A vers B si elle était en A et de B vers A si elle était en B).
- avec probabilité $\frac{1}{2}$, on laisse la boîte invariante.

On note X_n le nombre de particules dans le compartiment A au temps n ; alors, il y a $N - X_n$ particules dans le compartiment B au temps n. On admet que si les choix décrits ci-dessus sont fait indépendamment à chaque étape, alors $(X_n)_{n\in\mathbb{N}}$ est une chaîne de Markov sur $\mathfrak{X} = \{0, 1, \dots, N\}$ (attention, par rapport aux questions précédentes, $\mathfrak{X}$ est maintenant de cardinal $N + 1$).

(1) Calculer la matrice de transition de la chaîne définie ci-dessus. On remarquera que $P(k, l) = 0$ si $l \notin \{k-1, k, k+1\}$, de sorte qu'il suffit de calculer les valeurs $P(k, k-1)$, $P(k, k)$ et $P(k, k + 1)$.

(2) Trouver l'unique vecteur de probabilité π qui vérifie pour tous $x, y \in \{0, 1, \dots, N\}$ l'*équation de réversibilité*

$$\pi(x)\, P(x, y) = \pi(y)\, P(y, x).$$

Notant $\alpha = \pi(0)$, on pourra essayer de calculer en fonction de α les valeurs $\pi(1)$, $\pi(2)$, etc. pour déterminer π à un coefficient multiplicatif près ; puis, utiliser la condition $\sum_{x=0}^{N} \pi(x) = 1$ pour trouver la valeur de α. Montrer que le vecteur π ainsi obtenu est la loi invariante de la matrice de transition P.

(3) Écrire un programme `matrice_ehrenfest(N)` qui construit la matrice de transition P de taille $(N + 1) \times (N + 1)$, et un autre programme `loi_invariante_ehrenfest(N)` qui construit le vecteur π déterminé par la question 9. Vérifier l'équation $\pi P = \pi$ pour différentes valeurs de N. En utilisant la commande `np.linalg.eig`, énoncer une conjecture sur la liste des valeurs propres de la matrice P.

Une preuve de la convergence des lois marginales d'une chaîne irréductible à diagonale non nulle repose sur le fait algébrique suivant : sous cette hypothèse, toutes les valeurs propres complexes de P sont comprises dans le disque unité, et l'unique valeur propre de module 1 est $\lambda = 1$; elle est associée à un sous-espace propre de dimension 1 qui est engendré par un vecteur de probabilité (la loi invariante).

(4) Avec $N = 10$, comparer l'histogramme de la loi π avec l'histogramme empirique de 1000 tirages de la variable X_{100} (on pourra partir de $X_0 = 0$). Commenter.

4. Marche aléatoire sur la droite.

Le concept de chaîne de Markov s'étend sans difficultés aux espaces des états $\mathfrak{X}$ infinis dénombrables : c'est la même définition, avec une fonction de transition $P : \mathfrak{X}^2 \to [0,1]$ telle que $\sum_{y\in\mathfrak{X}} P(x,y) = 1$ pour tout $x \in \mathfrak{X}$. Dans cet exercice, on considère l'exemple le plus simple de chaîne infinie : la *marche aléatoire simple symétrique* sur $\mathbb{Z}$.

(1) Soit $(\xi_n)_{n\geq 1}$ une suite de variables indépendantes avec, pour tout $n \in \mathbb{N}$, $\mathbb{P}[\xi_n = 1] = \mathbb{P}[\xi_n = -1] = \frac{1}{2}$. On pose $X_n = \xi_1 + \xi_2 + \cdots + \xi_n$, avec par convention $X_0 = 0$. Montrer que $(X_n)_{n\in\mathbb{N}}$ est une chaîne de Markov sur $\mathfrak{X} = \mathbb{Z}$, de matrice de transition :

$$P(k,l) = \begin{cases} \frac{1}{2} & \text{si } l = k-1 \text{ ou } l = k+1, \\ 0 & \text{sinon.} \end{cases}$$

(2) Dessiner des trajectoires de cette marche aléatoire sur l'intervalle de temps $[\![0, N]\!]$, avec $N = 100$, puis avec $N = 100000$. En utlisant la commande `ax.set_aspect()`, faire en sorte que les abscisses et les ordonnées soient représentées avec un rapport abscisse / ordonnée égal à $1/\sqrt{N}$. Commenter les courbes obtenues.

(3) Pour $N \geq 1$ et $t \geq 0$, on pose $X_t^{(N)} = \frac{1}{\sqrt{N}} X_{\lfloor Nt \rfloor}$. Montrer que pour t fixé, $X_t^{(N)}$ converge en loi vers une variable X_t dont on précisera la loi. Modifier le programme de la question précédente pour dessiner $t \mapsto X_t$ sur un intervalle de temps $[0,T]$ arbitraire, en utilisant un ordre d'approximation N suffisamment élevé (par exemple $N = 100000$). Expérimenter avec $T = 10$.

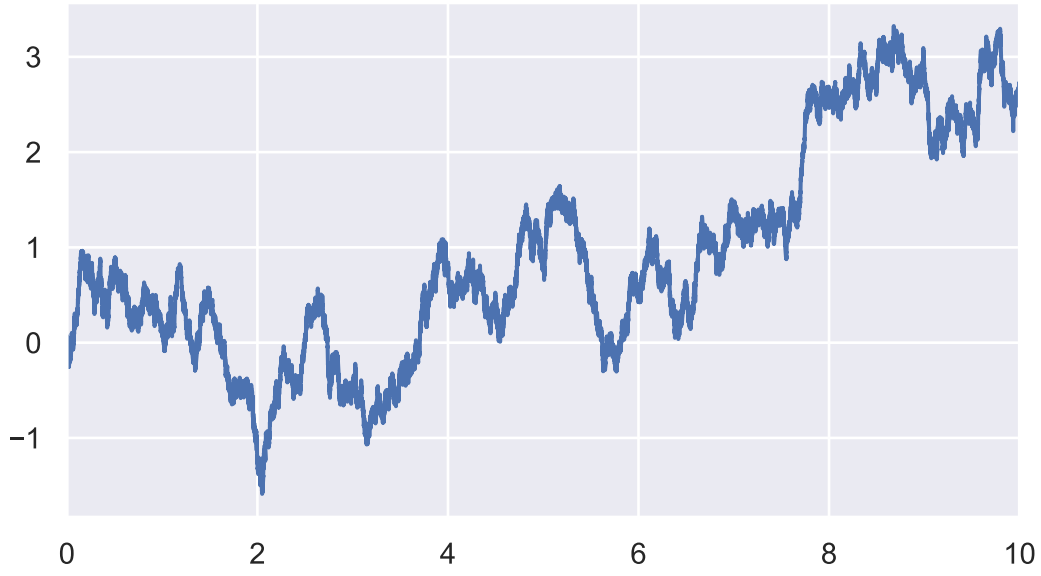

FIG. 5.2. Marche aléatoire symétrique sur $\mathbb{Z}$, renormalisée par un facteur $\frac{1}{N}$ en abscisse et $\frac{1}{\sqrt{N}}$ en ordonnée.

(4) Les courbes aléatoires que l'on obtient ont la propriété de *récurrence* :

$$\mathbb{P}[\mathrm{card}(\{n \geq 1 \mid X_n = 0\}) = +\infty] = 1.$$

De façon équivalente, si $R_0 = \inf\{n \geq 1 \,|\, X_n = 0\}$, alors ce *temps de retour en* 0 est presque sûrement fini :

$$\mathbb{P}[R_0 < +\infty] = 1.$$

Écrire un programme qui dresse l'histogramme empirique de la loi de R_0. Comparer cet histogramme à la loi théorique :

$$\forall m \geq 0,\ \ \mathbb{P}[R_0 = 2m+2] = \frac{1}{2^{2m+1}\,(m+1)} \binom{2m}{m};$$

cette formule peut être établie par un argument combinatoire.

On peut modifier le modèle en introduisant un paramètre $p \in (0,1)$, et en changeant les probabilités des pas :

$$\mathbb{P}[\xi_n = 1] = p \qquad ; \qquad \mathbb{P}[\xi_n = -1] = 1-p.$$

(5) Montrer que si $p \neq \frac{1}{2}$, alors $(X_n)_{n\in\mathbb{N}}$ tend presque sûrement vers $+\infty$ ou vers $-\infty$, en fonction du signe de $p - \frac{1}{2}$.

(6) On suppose $p < \frac{1}{2}$. Dessiner quelques trajectoires de cette marche non symétrique. Comme $\lim_{n\to\infty} X_n = -\infty$ avec probabilité, on peut définir sans ambiguïté la variable

$$M = \max(\{X_n,\ n \in \mathbb{N}\}).$$

Écrire un programme `maximum_marche_aleatoire(p, N=10000)` qui simule cette variable, en l'approchant par la variable $M_N = \max(\{X_n,\ n \leq N\})$. Dresser l'histogramme empirique de cette variable aléatoire, par exemple avec $p = \frac{1}{3}$ et $p = \frac{1}{4}$. Quelle loi semble suivre la variable aléatoire M ?

6. Arbres aléatoires

On s'intéresse à un modèle d'arbres aléatoires qui a une structure markovienne, et qui joue un rôle très important à la fois dans la théorie des graphes aléatoires, et pour les applications en biologie. Expliquons d'abord comment encoder et dessiner des arbres dans `Python`. Un *arbre enraciné* est un graphe $T = (V, E)$ connexe, avec un sommet distingué $r \in V$ (la *racine* de l'arbre), et tel que pour tout autre sommet $v \in V$, il existe un unique chemin de longueur minimale reliant r à v. Cette condition implique que T n'a pas de *cycle*, et que $|V| = |E| + 1$. La *profondeur* d'un sommet v de l'arbre est la distance de graphe entre r et v. Le *profil* de l'arbre est le vecteur

$$\begin{aligned} p(T) &= (p_0, p_1, ...) \\ &= (\text{nombre de sommets de profondeur } 0, \text{nombre de sommets de profondeur } 1, ...). \end{aligned}$$

Dans ce qui suit, on travaillera avec des arbres enracinés dont les sommets de profondeur d sont étiquetés par les paires d'entiers (d, k) avec $k \in [\![1, p_d]\!]$.

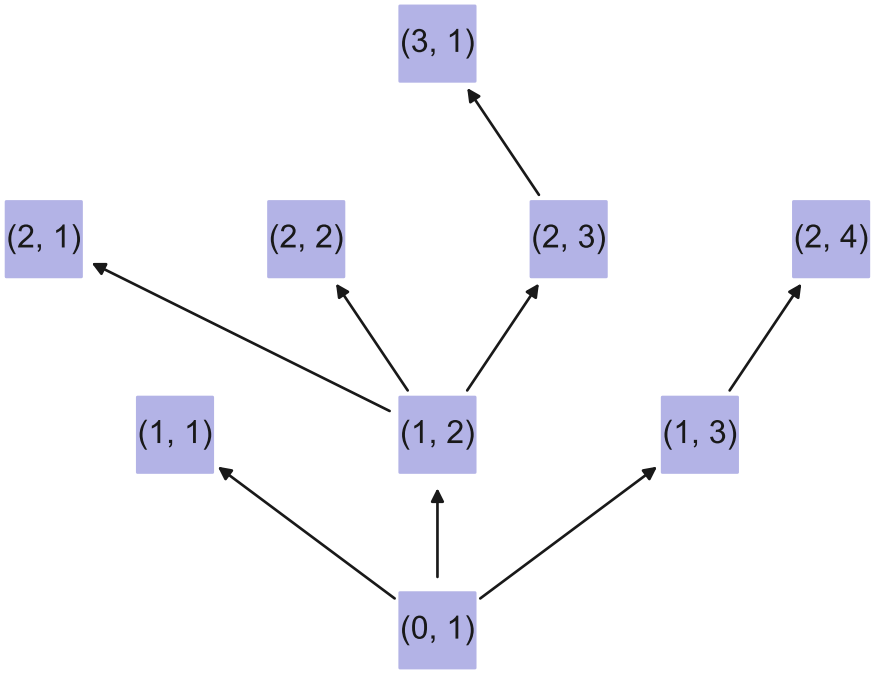

FIG. 6.1. Un arbre enraciné avec 9 sommets.

Le programme suivant, que l'on pourra télécharger, définit une classe `RootedTree` conforme aux définitions données ci-dessus.

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

```
class RootedTree:
    def __init__(self, max_size=10**6):
        self.depth = - np.ones(max_size, dtype=np.int64)
        self.row_position = np.zeros(max_size, dtype=np.int64)
        self.parent = np.empty(max_size, dtype=object)
        self.children = np.empty(max_size, dtype=object)
        self.depth[0], self.row_position[0], self.children[0] = 0, 1, []
        self.limit, self.profile, self.nodes_number = max_size, [1], 1

    def __repr__(self):
        return f"Rooted tree with {self.nodes_number} vertices"

    def find_index(self, d, k):
        I = (self.depth==d)[:self.nodes_number]
        I *= (self.row_position==k)[:self.nodes_number]
        return int(np.nonzero(I)[0][0])

    def add_children(self, parent, c):
        ip = self.find_index(*parent)
        if len(self.profile) - 1 == self.depth[ip]:
            self.profile.append(0)
        for i in range(self.nodes_number, self.nodes_number + c):
            self.depth[i] = self.depth[ip] + 1
            self.profile[self.depth[i]] += 1
            self.row_position[i] = self.profile[self.depth[i]]
            self.parent[i], self.children[i] = ip, []
        self.children[ip] += list(range(self.nodes_number, self.nodes_number + c))
        self.nodes_number += c

    def networkx(self):
        T = nx.DiGraph({i:self.children[i] for i in range(self.nodes_number)})
        for i in range(self.nodes_number):
            T.nodes[i]["label"] = (int(self.depth[i]), int(self.row_position[i]))
        return T

    def layout(self):
        return {i:np.array([- (self.profile[self.depth[i]] + 1)/2 + self.row_position[i],
                self.depth[i]]) for i in range(self.nodes_number)}

    def draw_on_ax(self, ax0, with_labels=False):
        ax0.set_axis_off()
        T = self.networkx()
        if with_labels:
            nx.draw_networkx(T, ax=ax0, pos=self.layout(), node_shape="s",
                        node_size=800, node_color=np.array([[0.7,0.7,0.9]]),
                        labels={i:T.nodes[i]["label"] for i in range(self.nodes_number)})
        else:
            nx.draw_networkx(T, ax=ax0, pos=self.layout(), node_shape="s",
                        node_size=300, node_color=np.array([[0.7,0.7,0.9]]),
                        with_labels=False)
```

Détaillons le contenu de ce programme :

- La commande `RootedTree()` crée un arbre avec une racine d'étiquette $(0, 1)$, à laquelle on pourra récursivement greffer de nouveaux sommets. Il stocke les informations dans un tableau avec quelques lignes et un nombre de colonnes indiqué par l'argument `max_size`, qui par défaut prend la valeur 10^6. Si l'on souhaite manipuler des arbres plus grands, il faudra donc le préciser à l'avance.

- Si `T` est un arbre, on ajoute c enfants au sommet d'étiquette (d, k) avec la commande `T.add_children((d,k), c)`. Ainsi, l'arbre à 9 sommets ci-dessus est obtenu avec :

```
T = RootedTree()
T.add_children((0, 1), 3)
T.add_children((1, 2), 3)
T.add_children((2, 3), 1)
T.add_children((1, 3), 1)
```

- Les commandes `T.nodes_number` et `T.profile` donnent le nombre de sommets et le profil de l'arbre.
- La méthode `networkx` fait le lien avec le module `NetworkX`, et la méthode `layout` calcule les positions des sommets de l'arbre pour un dessin. On utilisera simplement `T.draw_on_ax(ax0)` pour dessiner l'arbre sur une sous-figure `ax0` d'une figure `Matplotlib` ; les étiquettes pourront être affichées avec `with_labels=True`. Ainsi, la figure ci-dessus est obtenue avec :

```
fig, ax0 = plt.subplots()
T.draw_on_ax(ax0, with_labels=True)
plt.show()
```

1. Modèle de Galton–Watson.

Soit X une variable aléatoire à valeurs dans l'ensemble des entiers $\mathbb{N}$. On suppose X intégrable, et on note $m = \mathbb{E}[X]$. Étant donnée une famille $(X_{n,k})_{n\geq 0, k\geq 1}$ de variables indépendantes et toutes de même loi que X, l'*arbre de Galton–Watson* associé à la variable X est l'arbre enraciné aléatoire dont le k-ième sommet de profondeur n a $X_{n,k}$ enfants. Par exemple, l'arbre dessiné précédemment était une réalisation d'un arbre de Galton–Watson avec :

$$\begin{array}{llllll} X_{0,1} = 3 & ; & & & & \\ X_{1,1} = 0 & ; & X_{1,2} = 3 & ; & X_{1,3} = 1 & ; \\ X_{2,1} = 0 & ; & X_{2,2} = 0 & ; & X_{2,3} = 1 & ; & X_{2,4} = 0 & ; \\ X_{3,1} = 0. & & & & & \end{array}$$

En pratique, un arbre de Galton–Watson peut être infini : il est possible qu'à chaque génération n, le nombre Z_n d'individus de profondeur n soit non nul, et que l'une des variables $X_{n,k}$ avec $k \in [\![1, Z_n]\!]$ soit non nulle, de sorte que Z_{n+1} soit encore non nul. Pour rester dans le cadre des arbres enracinés finis, on appellera donc *arbre de Galton–Watson associé à la variable X et coupé à la n-ième génération* le sous-arbre de l'arbre de Galton–Watson obtenu en ne conservant que les sommets de profondeur inférieure ou égale à n.

(1) Écrire un programme `GaltonWatson(n, X)` qui prend en argument une profondeur n et une variable aléatoire X (définie dans `scipy.stats` et qu'on peut simuler avec `X.rvs()`), et qui renvoie une simulation de l'arbre de Galton–Watson associé à la variable X et coupé à la n-ième génération. Dessiner plusieurs exemples, avec $n = 10$, $X \sim \mathrm{Poi}(\lambda)$ et $\lambda \in \{0.75, 1, 1.25, 1.5\}$. Commenter ces dessins. L'arbre s'éteint-il avant la n-ième génération ? S'il ne s'éteint pas, quel est son comportement ?

(2) On note Z_n le nombre d'individus de n-ième génération ; le profil de l'arbre obtenu par la commande `GaltonWatson(n, X)` est donc $(Z_0 = 1, Z_1, \ldots, Z_n)$. Montrer que

Z_n vérifie la relation de récurrence :

$$Z_{n+1} = \sum_{k=1}^{Z_n} X_{n,k}.$$

En déduire un programme `GWZ(n, X)` qui calcule la suite $(Z_0, Z_1, \dots, Z_n)$ sans s'appuyer sur l'arbre correspondant.

(3) On note $P(k,l) = \mathbb{P}[X_1 + \cdots + X_k = l]$, où les X_i sont des copies indépendantes de la variable X. Montrer que la suite $(Z_n)_{n\in\mathbb{N}}$ de la question précédente est une chaîne de Markov sur $\mathbb{N}$ de matrice de transition P. Si $X \sim \mathrm{Poi}(\lambda)$, que vaut $P(k,l)$? En déduire un programme encore plus simple `GWZ_poisson(n, L)` qui calcule une simulation de $(Z_0, Z_1, \dots, Z_n)$ lorsque la variable de reproduction X suit une loi de Poisson de paramètre `L`.

2. Probabilité d'extinction.

Étant donné un arbre de Galton–Watson de variable de reproduction X, on note $p_n = \mathbb{P}[Z_n = 0]$ la probabilité pour que l'arbre s'éteigne avant la n-ième génération. On note par ailleurs

$$G(s) = \mathbb{E}[s^X] = \sum_{k=0}^{\infty} \mathbb{P}[X = k]\, s^k$$

la fonction génératrice de la variable X. Cette fonction est bien définie au moins pour $s \in [0,1]$.

(1) Calculer $G(s)$ lorsque $X \sim \mathrm{Poi}(\lambda)$.

(2) On note $G_n(s) = \mathbb{E}[s^{Z_n}] = \sum_{k=0}^{\infty} \mathbb{P}[Z_n = k]\, s^k$. Que valent G_0 et G_1 ?

(3) En décomposant l'espérance

$$\mathbb{E}[s^{Z_{n+1}}] = \sum_{k=0}^{\infty} \mathbb{P}[Z_n = k]\ \mathbb{E}\Big[s^{\sum_{j=1}^{k} X_{n,j}}\Big],$$

écrire une relation de récurrence vérifiée par les fonctions G_n.

(4) Quel est le lien entre la fonction G_n et la probabilité p_n ? En déduire que :

$$p_n = \underbrace{(G \circ G \circ \cdots \circ G)}_{n \text{ compositions}}(0).$$

(5) Pour $n = 5$, $X \sim \mathrm{Poi}(\lambda)$ et $\lambda \in \{0.75, 1, 1.25\}$, calculer p_n, et comparer ces valeurs aux probabilités empiriques d'extinction d'arbres de Galton-Watson avec ces paramètres (on prendra par exemple des échantillons de taille 1000).

(6) La *probabilité d'extinction* d'un arbre de Galton-Watson associé à la variable aléatoire X est $p = \lim_{n\to\infty} p_n$. Montrer que :

$$(m > 1) \Leftrightarrow (p < 1).$$

On pourra remarquer que la fonction G est croissante convexe, et que $m = G'(1)$. Écrire un programme `proba_extinction(L)` qui calcule (une approximation de) p lorsque X suit une loi de Poisson de paramètre `L`. On pourra agrémenter ce programme d'une représentation graphique du calcul.

(7) Si $X \sim \mathrm{Poi}(1)$, montrer par une simulation que $\lim_{n\to\infty} n(1 - p_n) = 2$. Question bonus : démontrer ce résultat.

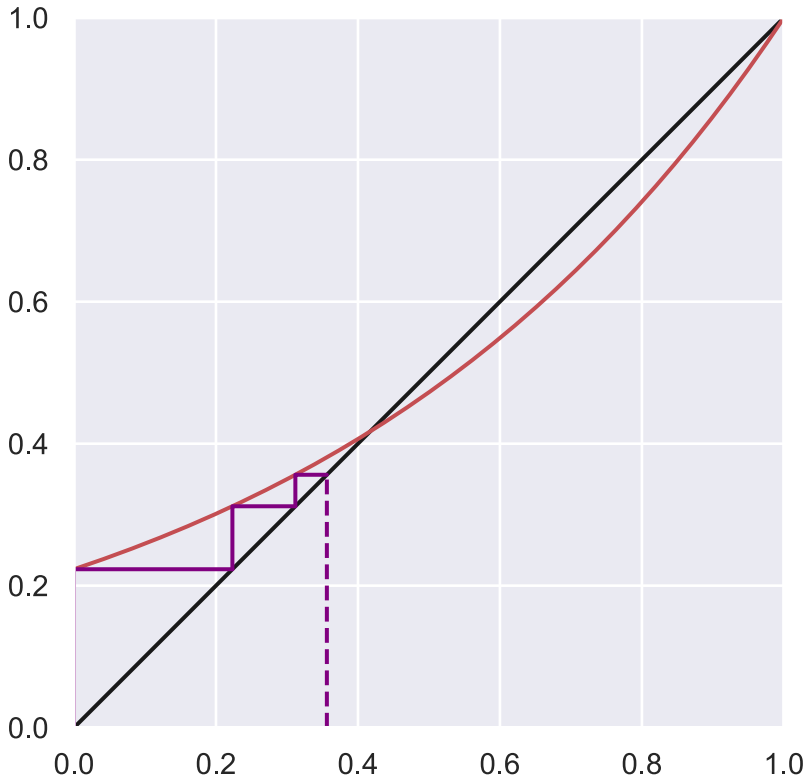

FIG. 6.2. La probabilité d'extinction au temps n est obtenue en itérant n fois la fonction génératrice de X en 0. Ici, $X \sim \mathrm{Poi}(1.5)$ et $n = 3$; $p_3 = 0.3562$.

On peut montrer que si un arbre de Galton-Watson ne s'éteint pas, alors $\lim_{n\to\infty} Z_n = +\infty$ (sauf dans le cas exceptionnel où $X = 1$ presque sûrement). Autrement dit, on a soit extinction, soit croissance de l'arbre avec un profil de plus en plus large.

$$\mathbb{P}\Big[\lim_{n\to\infty} Z_n = 0\Big] + \mathbb{P}\Big[\lim_{n\to\infty} Z_n = +\infty\Big] = 1.$$

3. Arbres surcritiques et critiques.

Un arbre de Galton–Watson est dit *surcritique* (respectivement, *critique*) si $m > 1$ (respectivement, si $m = 1$).

(1) Montrer que $\mathbb{E}[Z_n] = (G_n)'(1)$. En utilisant la relation de récurrence démontrée dans la section précédente, en déduire la valeur de $\mathbb{E}[Z_n]$ en fonction de n et de m.

(2) On pose

$$M_n = \frac{Z_n}{m^n}.$$

Si $X \sim \mathrm{Poi}(\lambda)$ avec $\lambda \in \{0.75, 1, 1.25, 1.5\}$, dessiner sur l'intervalle de temps $[\![0, N]\!]$ des trajectoires de $(M_n)_{n\in\mathbb{N}}$, avec $N = 100$. Illustrer le résultat suivant : lorsque n tend vers l'infini, $(M_n)_{n\in\mathbb{N}}$ converge presque sûrement vers une variable aléatoire M.

(3) Avec $\lambda = 1.25$ et $n = 100$, dessiner la fonction de répartition empirique de M_n, qui approxime celle de la limite M. Comparer avec la valeur de p calculée dans la section précédente, et la valeur de la fonction de répartition de M en 0. Commenter.

(4) D'après la dernière question de la section précédente, si $\lambda = 1$, alors $Z_n > 0$ avec probabilité d'ordre $\frac{2}{n}$. On note Z'_n une variable aléatoire de loi :

$$\mathbb{P}[Z'_n = k] = \begin{cases} \frac{\mathbb{P}[Z_n=k]}{\mathbb{P}[Z_n>0]} & \text{si } k > 0, \\ 0 & \text{si } k = 0. \end{cases}$$

La variable Z'_n représente donc la taille de la n-ième génération d'un arbre de Galton-Watson critique conditionnellement à la non-extinction au temps n. Comme $\mathbb{P}[Z_n > 0]$ ne décroît pas trop vite, on peut simuler Z'_n en tirant au hasard des variables Z_n jusqu'à obtenir un résultat strictement positif. Dessiner la fonction de répartition empirique de la variable $X_n = \frac{Z'_n}{n}$, par exemple avec $n = 100$. Comparer avec la fonction de répartition d'une variable de loi Exp(2). Que peut-on conjecturer ?

7. Graphes aléatoires

Dans ce chapitre, on s'intéresse à un autre modèle fondamental de graphes aléatoires : les graphes d'Erdős–Rényi. On utilisera `NetworkX` pour manipuler tous les graphes dans `Python` ; on renvoie à l'annexe pour une description des principales commandes utiles. En particulier, si `A` est une matrice d'adjacence, le graphe correspondant sera créé et affiché avec les commandes :

```
G = nx.Graph(A)
fig, ax0 = plt.subplots()
nx.draw_networkx(G, ax=ax0)
plt.show()
```

1. Modèle d'Erdős–Rényi.

Si $n \geq 1$ et $p \in (0,1)$, le *graphe d'Erdős–Rényi* de paramètres n et p est le graphe aléatoire $G = (V, E)$ dont l'ensemble des sommets est $V = [\![1, n]\!]$, et dont chaque arête possible $\{i, j\}$ avec $1 \leq i < j \leq n$ apparaît dans E avec probabilité p ; les variables de Bernoulli $B_{i,j}$ correspondantes sont supposées indépendantes. On note $G \sim \mathrm{G}(n, p)$ pour indiquer qu'un graphe aléatoire est un graphe d'Erdős–Rényi de paramètres n et p.

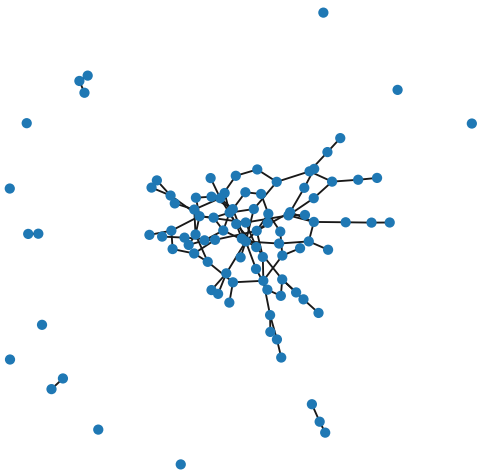

FIG. 7.1. Un graphe d'Erdös–Rényi de paramètres $n = 100$ et $p = 0.01$.

(1) Quelle est la loi de $|E|$ si $G = (V, E) \sim \mathrm{G}(n, p)$? Que vaut $\mathbb{E}[|E|]$? Comment choisir p en fonction de n pour que $|V|$ et $|E|$ soient du même ordre ?

(2) Écrire des programmes `erdos_renyi(n, p)` et `dessin_erdos_renyi(n, p)` qui construisent et dessinent un graphe aléatoire d'Erdős–Rényi de paramètres n et p. On pourra utiliser l'argument `node_size` de `nx.draw_networkx` pour ajuster la taille des sommets et avoir des graphes lisibles. Expérimenter avec les paramètres $n = 100$ et $p \in \{0.005, 0.008, 0.01, 0.015, 0.02, 0.03\}$, et décrire les graphes obtenus.

(3) Pour mieux comprendre l'évolution des graphes d'Erdős–Rényi lorsque n est fixé et p augmente, on peut *coupler* les graphes $\mathrm{G}(n,p)$ comme suit :

- on tire des variables indépendantes uniformes $U_{i,j} \in [0,1]$;
- le graphe $\mathrm{G}(n,p)$ est obtenu en posant $B_{i,j} = 1_{(U_{i,j} \leq p)}$.

Avec cette construction, $\mathrm{G}(n,p_1)$ est un sous-graphe de $\mathrm{G}(n,p_2)$ si $p_1 \leq p_2$; ceci permet de mieux comparer les graphes d'Erdős–Rényi de paramètres p différents. Écrire un programme `dessin_liste_erdos_renyi(n, P)` qui affiche plusieurs graphes d'Erdős–Rényi de paramètre p dans une liste `P`. Pour pouvoir suivre les sommets, on pourra :

- utiliser le même *Layout* pour les sommets de tous les graphes. Ainsi,

```
pos0 = nx.spring_layout(G)
...
nx.draw_networkx(H, pos=pos0)
```

fixe les positions de sommets dans un graphe G (on peut par exemple choisir le graphe de paramètre p maximal dans la liste `P`), puis impose ces positions aux sommets des autres graphes H.

- ou alors, faire évoluer le *Layout* au fur et à mesure ; on pourra explorer la documentation de `nx.spring_layout`.

Expérimenter avec $n = 100$ et un échantillonnage P des intervalles $[0.003, 0.007]$, $[0.007, 0.02]$ et $[0.02, 0.06]$.

2. Composantes isolées.

Dans cette section, $p = \frac{c}{n}$ avec $c \in [0,1)$. On va voir que dans ce cas, le graphe d'Erdős–Rényi est constitué de petites composantes connexes qui sont presque toutes des arbres.

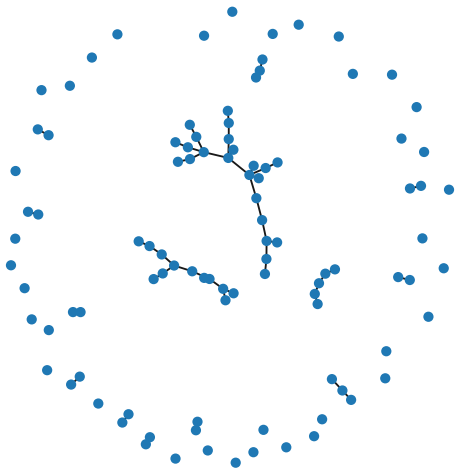

FIG. 7.2. Aspect d'un graphe d'Erdős–Rényi avant l'apparition d'une composante géante : $p = \frac{c}{n}$ avec $c < 1$. Toutes les composantes connexes sont de petits arbres.

(1) Si `G` est un graphe `NetworkX` et `L` est une partie de l'ensemble des sommets de `G`, `G.subgraph(L)` est le graphe induit par `G` sur cette partie. En utilisant la commande `nx.connected_components(G)`, écrire un programme `proportion_arbre` qui calcule la proportion de sommets d'un graphe $G = (V,E)$ qui sont dans des composantes

connexes qui sont des arbres :

$$P_{\text{arbre}} = \sum_{c \text{ composante connexe de } G} 1_{(c \text{ est un arbre})} \frac{|c|}{|V|}.$$

On pourra tester si un graphe est un arbre avec `nx.is_tree`.

(2) Si les graphes $\mathrm{G}(n,p)$ sont couplés comme dans la section précédente, montrer que la proportion d'arbres est une fonction (aléatoire) décroissante de p. Avec $n = 10000$ et $p = \frac{c}{n}$, dessiner la fonction $c \mapsto P_{\text{arbre}}(\mathrm{G}(n, \frac{c}{n}))$ pour $c \in [0,2]$. Qu'observe-t-on ?

(3) Si $I = \{i_1, i_2, \dots, i_k\}$ est une partie de taille $k \geq 1$ de $[\![1,n]\!]$ et T est un arbre sur cet ensemble de sommets, montrer que la probabilité pour que :

- I soit une composante connexe de $\mathrm{G}(n,p)$ (donc, déconnectée de $[\![1,n]\!] \setminus I$) ;
- le graphe induit par $\mathrm{G}(n,p)$ sur I est T

est donné par la formule :

$$p_k = p^{k-1}(1-p)^{k(n-k)+\frac{(k-2)(k-1)}{2}}.$$

On admet la formule combinatoire suivante (*formule de Cayley*) : le nombre d'arbres sur un ensemble de sommets de taille k est k^{k-2}. En déduire une formule pour l'espérance de T_k, le nombre de composantes connexes de $\mathrm{G}(n,p)$ qui sont des arbres de taille k.

(4) Montrer que si $c > 0$, alors pour tout $k \geq 1$,

$$\lim_{n\to\infty} \frac{\mathbb{E}[T_k(\mathrm{G}(n, \frac{c}{n}))]}{n} = \frac{1}{c} \frac{k^{k-2}}{k!} (c\mathrm{e}^{-c})^k.$$

Pour $|z| < \mathrm{e}^{-1}$, on pose $T(z) = \sum_{k=1}^{\infty} \frac{k^{k-1}}{k!} z^k$; la série est convergente par la formule de Stirling. Comme k^{k-1} est le nombre d'arbres étiquetés enracinés sur k sommets (c'est-à-dire avec une racine distinguée), $T(z)$ est la série génératrice exponentielle des arbres enracinés. Or, un arbre enraciné peut être vu comme une racine reliée à une union éventuellement vide d'autres arbres enracinés ; ceci implique la formule

$$T(z) = z\,\mathrm{e}^{T(z)}$$

pour tout z complexe de module inférieur à e^{-1}. Ceci implique en particulier que

$$\lim_{\substack{s\to\mathrm{e}^{-1} \\ s<\mathrm{e}^{-1}}} T(s) = 1$$

et que T établit une bijection croissante entre $[0, \mathrm{e}^{-1})$ et $[0,1)$. Si $c = T(s)$ avec $c \in [0,1)$, on a alors :

$$\frac{1}{c} \sum_{k=1}^{\infty} \frac{k^{k-1}}{k!} (c\mathrm{e}^{-c})^k = \frac{1}{T(s)} T\big(T(s)\,\mathrm{e}^{-T(s)}\big) = \frac{T(s)}{T(s)} = 1.$$

(5) Quel est le lien entre la proportion P_{arbre} et les variables T_k, $k \geq 1$? Déduire du calcul ci-dessus le fait suivant : si $c \in [0,1)$, alors

$$\lim_{n\to\infty} \mathbb{E}\Big[P_{\text{arbre}}\Big(\mathrm{G}\Big(n, \frac{c}{n}\Big)\Big)\Big] = 1.$$

(6) Montrer aussi que si $c < 1$, alors il n'y a pas de *composante géante* dans $\mathrm{G}(n,p)$: pour tout $\varepsilon > 0$,

$$\lim_{n\to\infty} \mathbb{P}[\mathrm{G}(n,p) \text{ contient une composante connexe de taille plus grande que } \varepsilon n] = 0.$$

3. Composante géante.

Dans cette section, $p = \frac{c}{n}$ avec $c > 1$ fixé. Dans ce cas, le graphe d'Erdős–Rényi est constitué d'une composante géante de taille $O(n)$, et d'autres composantes connexes beaucoup plus petites.

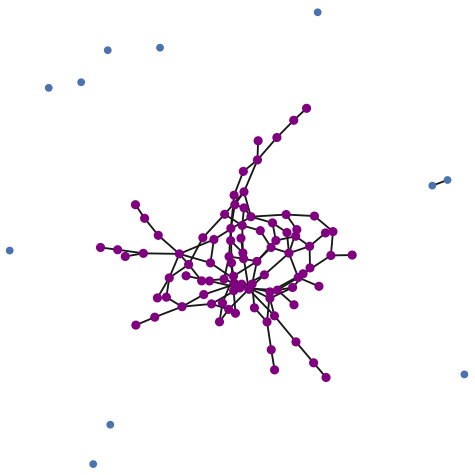

FIG. 7.3. Aspect d'un graphe d'Erdős–Rényi après l'apparition d'une composante géante : $p = \frac{c}{n}$ avec $c > 1$. Il y a une composante géante d'ordre $O(n)$, et les autres composantes sont de taille $O(\log n)$.

(1) Modifier le programme `dessin_erdos_renyi` pour que la plus grande composante connexe (ou l'une des plus grandes composantes connexes s'il y en a plusieurs de même taille) soit affichée avec une autre couleur. Expérimenter avec $n = 100$ et $p = \frac{c}{n}$ avec $c \in \{0.005, 0.006, \dots, 0.015\}$.

(2) Si les graphes $\mathrm{G}(n,p)$ sont couplés comme précédemment, montrer que la fonction aléatoire

$$p \mapsto \frac{\text{taille de la plus grande composante connexe de } \mathrm{G}(n,p)}{n}$$

est croissante. Dessiner cette fonction pour $n = 10000$ et $p = \frac{c}{n}$ avec c variant entre $[0.5, 3]$. Qu'observe-t-on ?

(3) Remarquons que la fonction $x \mapsto x\mathrm{e}^{-x}$ est croissante sur $[0,1]$ et décroissante sur $[1,+\infty)$; par conséquent, si $c > 1$, il existe un unique c^* tel que $c^*\mathrm{e}^{-c^*} = c\mathrm{e}^{-c}$. Comparer la fonction précédente sur $[1,3]$ à la fonction $c \mapsto (1 - \frac{c^*}{c})$. Pour trouver c^*, on pourra utiliser `fsolve`, qu'on importera avec `from scipy.optimize import fsolve`.

(4) Vérifier par la simulation que les autres composantes connexes de $\mathrm{G}(n, \frac{c}{n})$ avec $c > 1$ sont d'ordre $O(\log n)$. Ainsi, il y a une unique composante connexe géante pour $c > 1$.

On peut montrer que si $c > \log n$, alors la composante géante absorbe tous les sommets : le graphe est connexe avec grande probabilité pour $p > \frac{\log n}{n}$. Plus précisément, si $c = \log n + d$ avec $d \in \mathbb{R}$ fixé, alors

$$\lim_{n\to\infty} \mathbb{P}\Big[\mathrm{G}\Big(n, \frac{c}{n}\Big) \text{ est connexe}\Big] = \mathrm{e}^{-\mathrm{e}^{-d}}.$$

8. Percolation de Bernoulli

Dans ce dernier chapitre, on généralise la construction des graphes d'Erdős–Rényi en fixant à l'avance une structure pour les graphes aléatoires, par exemple une grille. Si $G = (V, E)$ est un graphe fini fixé, le *graph percolé* G_p de paramètre p est le graphe aléatoire :

- avec le même ensemble de sommets V,
- avec chaque arête $e \in E$ conservée avec probabilité p, indépendamment pour chaque arête.

Le graphe d'Erdős–Rényi $\mathrm{G}(n, p)$ est le cas particulier de graphe percolé avec $G = K_n$, le graphe complet à n sommets. Dans ce qui suit, on s'intéressera au cas où

$$G = C_n = [\![-n, n]\!]^2$$

est la grille carrée de côté $2n$, avec (i, j) connecté à (i', j') si et seulement si $|i-i'|+|j-j'| = 1$.

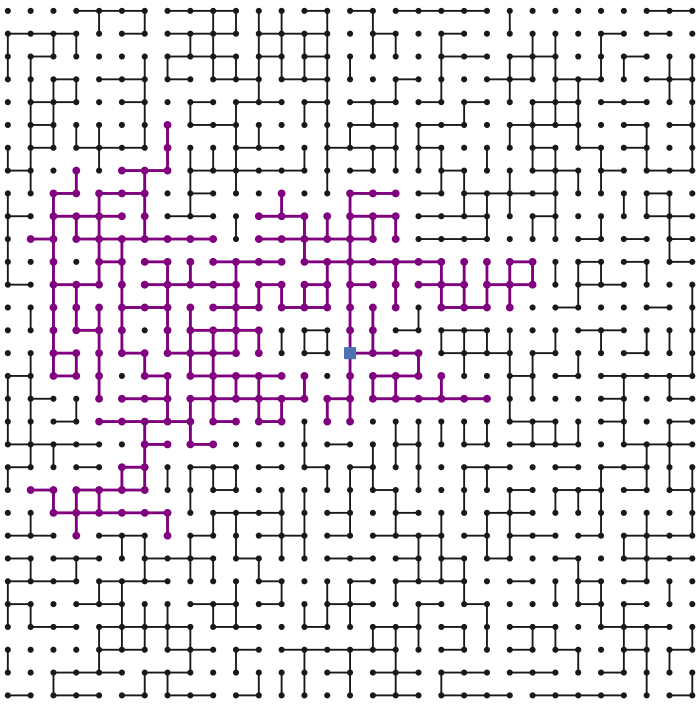

FIG. 8.1. Percolation sur la grille carrée. Ici, $n = 15$ et $p = 0.45$; la composante connexe de (0.0) apparaît en violet.

1. Simulation dans $\mathbb{Z}^2$.

(1) Écrire un programme `percolation(n, p)` qui prend en paramètre une taille n et un paramètre $p \in (0, 1)$, et qui renvoie le graphe aléatoire percolé $C_{n,p}$ (sous la forme d'un graphe `NetworkX`). On pourra faire en sorte que les sommets du graphe soient identifiés par leurs coordonnées dans $\mathbb{Z}^2$ (les sommets d'un graphe `NetworkX` peuvent être n'importe quel objet `Python`, par exemple une paire d'entiers). Écrire un autre

programme `dessin_percolation(n, p)` qui dessine $C_{n,p}$, en respectant la géométrie de la grille $\mathbb{Z}^2$.

(2) Expérimenter les programmes précédents avec $n = 25$ et p dans $[0, 1]$. Commenter les graphes obtenus, en particulier l'aspect de leurs composantes connexes.

(3) Modifier le programme de dessin pour que la composante connexe qui contient le point $(0, 0)$ apparaisse dans une autre couleur. Pour déterminer la composante connexe du centre de la grille, on pourra utiliser la commande `nx.node_connected_component`.

(4) Une quantité importante est la *probabilité de connexion au bord* :

$$\begin{aligned} q_{n,p} &= \mathbb{P}_p\left[(0,0) \leftrightarrow \partial C_n\right] \\ &= \mathbb{P}_p[\text{la composante connexe contenant } (0,0) \text{ touche le bord de la grille carrée } C_n]. \end{aligned}$$

Écrire un programme `proba_connexion(n, p, N)` qui estime cette probabilité sur un échantillon de N simulations de $C_{n,p}$. Tester ce programme avec $n = 30$ et différentes valeurs de p.

(5) En utilisant un couplage approprié des graphes percolés $C_{n,p}$, montrer que $p \mapsto q_{n,p}$ est croissant. En déduire un programme efficace pour dessiner le graphe de $p \mapsto q_{n,p}$. Commenter ce graphe lorsque $n = 50$.

(6) Montrer que $n \mapsto q_{n,p}$ est décroissante.

La dernière question permet de définir sans ambiguïté $q_p = \lim_{n\to\infty} q_{n,p} = \mathbb{P}_p[(0, 0) \leftrightarrow \infty]$: c'est la probabilité pour qu'un chemin partant de 0 aille à l'infini dans le graphe percolé.

2. Transition de phase.

La percolation en dimension 2 (et en fait, en toute dimension $d \geq 2$) subit une *transition de phase* lorsque p augmente, vis-à-vis de la probabilité de connexion à l'infini :

$$\begin{cases} q_p = 0 \text{ si } p < p_c; \\ q_p > 0 \text{ si } p > p_c, \end{cases}$$

avec $p_c \in (0, 1)$ probabilité *critique*.

(1) Conjecturer la valeur de p_c à partir des simulations.

(2) On souhaite montrer que $p_c > 0$: autrement dit, $q_p = 0$ pour p assez petit. Montrer que :

$$q_p \leq p^n \,\#(\text{chemin partant de } (0,0) \text{ et de longueur } n)$$

pour tout $n \geq 1$ (par chemin, on entend une suite de sommets distincts, avec les sommets consécutifs qui sont voisins). En déduire que si $p < \frac{1}{3}$, alors $q_p = 0$. Donner une borne inférieure pour p_c.

(3) On souhaite montrer que $p_c < 1$: autrement dit, $q_p > 0$ pour p assez grand. La percolation de paramètre p dans $\mathbb{Z}^2$ induit une percolation de paramètre $1 - p$ dans le *graphe dual* $\mathbb{Z'}^2 = \mathbb{Z}^2 + (\frac{1}{2}, \frac{1}{2})$:

$$\left(i + \frac{1}{2}, j - \frac{1}{2}\right) \leftrightarrow \left(i + \frac{1}{2}, j + \frac{1}{2}\right) \text{ dans } \mathbb{Z'}^2 \quad \Leftrightarrow \quad (i, j) \nleftrightarrow (i + 1, j) \text{ dans } \mathbb{Z}^2;$$

$$\left(i - \frac{1}{2}, j + \frac{1}{2}\right) \leftrightarrow \left(i + \frac{1}{2}, j + \frac{1}{2}\right) \text{ dans } \mathbb{Z'}^2 \quad \Leftrightarrow \quad (i, j) \nleftrightarrow (i, j + 1) \text{ dans } \mathbb{Z}^2.$$

Autrement dit, on met une arête dans le graphe dual $\mathbb{Z}'^2$ si et seulement si elle ne croise pas une arête dans le graphe $\mathbb{Z}^2$. Remarquons que alors si $(0,0)$ n'est pas connecté à l'infini dans $\mathbb{Z}^2$, alors il existe un chemin clos l'entourant dans le graphe dual.

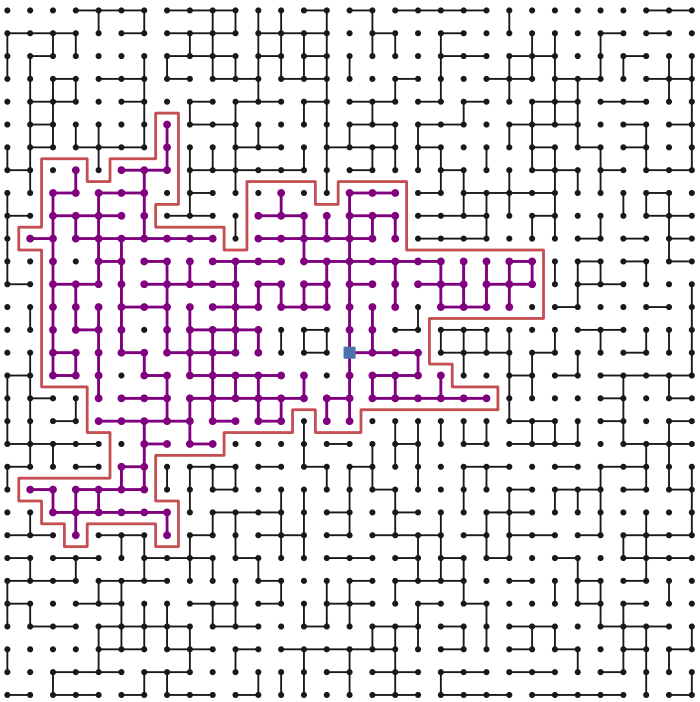

FIG. 8.2. Si $(0,0)$ n'est pas connecté à l'infini dans $(\mathbb{Z}^2)_p = \bigcup_{n=1}^{\infty} \uparrow C_{n,p}$, alors dans le graphe dual $(\mathbb{Z}'^2)_{1-p}$, il y a un chemin clos entourant $(0,0)$.

À partir de cette observation, montrer que

$$1 - q_p \leq \sum_{n=1}^{\infty} (1-p)^{2n} \, \#(\text{circuit dual entourant } (0,0) \text{ et de longueur } 2n).$$

Montrer que le nombre de circuits entourant $(0,0)$ et de longueur $2n$ est plus petit que

$$n 3^{2n-1}.$$

On pourra remarquer qu'un tel circuit contient un segment $(k+\frac{1}{2}, -\frac{1}{2}) \leftrightarrow (k+\frac{1}{2}, \frac{1}{2})$ avec $k \in [\![0, n-1]\!]$. Conclure.

Modules Python

Cette annexe propose une introduction succinte aux modules les plus employés dans les 8 chapitres de ce cours. On renvoie aussi aux documentations officielles :

- `numpy` et `scipy` (calcul scientifique) : https://numpy.org et https://scipy.org.
- `matplotlib` (dessin scientifique) : https://matplotlib.org.
- `networkx` (manipulation de graphes) : https://networkx.org.

Si ces modules ne sont pas déjà installés (ou ne sont pas à jour), `pip3 install module` dans un terminal peut remédier à cela.

1. Numpy.

Le module `numpy` introduit une structure de données essentielle pour tous nos calculs : les *arrays* (tableaux).

```
>>> L = np.array([1, 3.5, -2]) ; L

array([ 1. , 3.5, -2. ])
```

La syntaxe de manipulation des listes s'étend aux *arrays* : par exemple, pour récupérer le i-ième élément d'un *array* `L`, on utilisera la commande `L[i]`, et pour récupérer les éléments d'indices entre a et $b-1$, on utilisera la commande `L[a:b]`. Pourquoi alors utiliser les *array* à la place des listes Python ? Il y a au moins trois bonnes raisons :

(1) Les données d'un *array* sont stockées contiguement en espace mémoire, alors que les données d'une liste ne le sont pas. Ceci fait que les calculs sur un *array* sont en général beaucoup plus rapides. Par exemple, considérons 100 000 réels aléatoires stockés sous la forme d'un *array* `A` ou d'une liste `L`. Les commandes suivantes calculent le temps requis pour faire la somme de ces nombres :

```
>>> A = random.random(size=100000)
>>> L = list(A)
>>> timeit (np.sum(A))

13.6 µs ± 27.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

>>> timeit (sum(L))

1.83 ms ± 880 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

La même opération avec un *array* prend moins de 100 fois moins de temps !

(2) Les *arrays* peuvent être unidimensionnels (comme des listes), mais aussi multidimensionnels, et en particulier bidimensionnels (comme des matrices).

```
>>> L = np.array([[1, 3.5, -2], [-4, 0, 2.25]])
```

```
>>> L.shape

(2,3)

>>> L[:,2]

array([-2.  ,  2.25])
```

C'est par exemple utile si l'on veut considérer un échantillon $M_1, \dots, M_N$ de moyennes empiriques, chaque M_i étant la moyenne de n variables aléatoires $X_{i,1}, \dots, X_{i,n}$. Si les variables $X_{i,j}$ sont uniformes sur $[0,1]$, on peut obtenir l'échantillon des moyennes empiriques en réduisant un tableau de taille $N \times n$:

```
>>> N, n = 10, 100
>>> alea = random.random(size=(N, n))
>>> np.mean(alea, axis=1)

array([0.44985482, 0.57135158, 0.54310497, 0.46410955, 0.54345499,
       0.47592304, 0.47515065, 0.49792214, 0.45701027, 0.47436332])
```

Par ailleurs, les tableaux bidimensionnels permettent d'effectuer efficacement des opérations d'algèbre linéaire, grâce aux commandes du sous-module `numpy.linalg`. Dans ce cours, ce sera en particulier utile pour manipuler les matrices de transition des chaînes de Markov.

(3) Les *arrays* sont adaptés à l'application de fonctions à chacune de leurs entrées (vectorisation des opérations). Voyons quelques exemples :

- on peut additionner des *arrays* ensemble (terme à terme), ou additionner un nombre à chaque terme d'un *array*.

```
>>> L = np.array([[1, 3.5, -2], [-4, 0, 2.25]])
>>> L1, L2 = L[0,:], L[1:,]
>>> L1 + L2

array([[-3.  ,  3.5 ,  0.25]])

>>> L1 + 2

array([3. , 5.5, 0. ])
```

- de même, on peut multiplier des *arrays* ensemble (terme à terme), ou multiplier un *array* par un nombre.

```
>>> L1 * L2

array([[-4. ,  0. , -4.5]])

>>> 2 * L2

array([[-8. ,  0. ,  4.5]])
```

- on peut appliquer une fonction à tous les termes d'un *array*. La plupart des fonctions mathématiques usuelles ont des versions `numpy` adaptées à cet usage : `np.sqrt` pour la racine carrée, `np.exp` pour l'exponentielle, etc.

```
>>> np.exp(L1 + L2**2)

array([[2.41549528e+07, 3.31154520e+01, 2.13809428e+01]])

>>> (L1 >= 0)

array([ True,  True, False])
```

Une application importante de ces techniques est le dessin du graphe d'une fonction F. Si l'on veut dessiner sur l'intervalle $[a, b]$ le graphe $y = F(x)$, on peut : utiliser `xx = np.linspace(a, b, n)` pour créer n points régulièrement espacés entre a et b (typiquement avec $n = 500$) ; appliquer F à ce vecteur avec la commande vectorielle `yy = F(xx)` ; et finalement dessiner le graphe de la fonction avec la commande `ax.plot(xx, yy)`.

Les autres modules que nous employerons ont pour arguments ou pour résultats des *arrays* : par exemple, les variables aléatoires de `scipy.stats` produisent par la méthode `.rvs(size=N)` un N-échantillon donné sous la forme d'un *array*, et les fonctions de dessin de données de `matplotlib.pyplot` prennent comme arguments des *arrays* de données. C'est une autre bonne raison pour employer systématiquement cette structure de données.

2. Scipy.

Le module `scipy` contient de très nombreuses fonctions et algorithmes mathématiques : intégration numérique, algèbre linéaire, interpolation, etc. Nous utiliserons presque exclusivement les fonctions de probabilités et de statistiques du sous-module `scipy.stats`.

Les distributions de probabilité suivantes sont implantées dans `scipy.stats` :

- loi de Bernoulli $\mathrm{Ber}(p)$: `scs.bernoulli(p)`.
- loi binomiale $\mathrm{Bin}(n, p)$: `scs.binom(n, p)`.
- loi de Poisson $\mathrm{Poi}(\lambda)$: `scs.poisson(L)`.
- loi géométrique $\mathrm{Geom}(p)$: `scs.geom(p)`.
- loi uniforme $\mathrm{Unif}([a, b])$: `scs.uniform(loc = a, scale = b-a)`. Par défaut, $[a, b] = [0, 1]$.
- loi exponentielle $\mathrm{Exp}(\lambda)$: `scs.expon(scale = 1/L)`. Par défaut, $\lambda = 1$.
- loi normale $\mathrm{N}(m, \sigma^2)$: `scs.norm(loc = m, scale = sigma)`. Par défaut, $m = 0$ et $\sigma^2 = 1$.
- loi de Cauchy $\mathrm{Cau}(c)$: `scs.cauchy(scale = c)`. Par défaut, $c = 1$.

Cette liste est non exhaustive (il y a plus de 100 distributions déjà implantées), mais largement suffisante pour nos simulations. Étant donnée une loi `law` comme ci-dessus, différentes méthodes permettent de simuler des variables aléatoires avec cette loi, et d'obtenir des objets mathématiques reliés à la loi.

- La commande la plus utile est `law.rvs(size = N)`, qui crée un N-échantillon de variables indépendantes. On peut aussi utiliser le paramètre `size = (l, c)` pour créer une matrice de variables indépendantes, avec l lignes et c colonnes.

```
>>> scs.norm.rvs(size=5)
```

```
array([-2.77972731,  0.1897642 , -1.02439086, -0.02655945,  1.57445863])
```

- Les commandes `law.mean()` et `law.var()` donnent la moyenne et la variance (théoriques) de la loi.

```
>>> scs.norm.mean(), scs.norm.var()

(np.float64(0.0), np.float64(1.0))
```

- La commande `law.cdf` renvoie la fonction de répartition de la loi. On peut l'appliquer à un réel, ou aux termes d'un *array*.

```
>>> F = scs.norm.cdf
>>> F(2)

np.float64(0.9772498680518208)

>>> F(np.array([0,1,2,3]))

array([0.5       , 0.84134475, 0.97724987, 0.9986501 ])
```

- Dans le cas discret, `law.pmf` est la fonction sur les entiers qui donne la probabilité $\mathbb{P}[X = k]$, et dans le cas continu, `law.pdf` est la fonction sur les réels qui donne la densité de probabilité $f_X(x)$.

```
>>> M = scs.poisson(3).pmf
>>> M(np.arange(10))

array([0.04978707, 0.14936121, 0.22404181, 0.22404181, 0.16803136,
       0.10081881, 0.05040941, 0.02160403, 0.00810151, 0.0027005 ])

>>> f = scs.norm.pdf
>>> f(np.array([0,1,2,3]))

array([0.39894228, 0.24197072, 0.05399097, 0.00443185])
```

Une fois un échantillon de variables aléatoires obtenu, on peut utiliser les méthodes de `numpy` pour calculer les statistiques de cet échantillon (par exemple la moyenne et la variance empirique avec `np.mean()` et `np.var()`), et les méthodes de `matplotlib.pyplot` pour représenter cet échantillon (par exemple avec `ax.ecdf()` pour la fonction de répartition empirique, et avec `ax.hist()` pour l'histogramme empirique dans le cas discret).

3. Matplotlib.

On utilise `Matplotlib` pour tous les dessins ; c'est un module très puissant et configurable, mais sa syntaxe n'est pas la plus claire. En effet, si l'on veut juste une figure avec le dessin d'un cercle, il faut écrire :

```
import matplotlib
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.add_patch(matplotlib.patches.Circle((0,0), 1))
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
ax.set_aspect(1)
```

```
ax.set_axis_off()
plt.show()
```

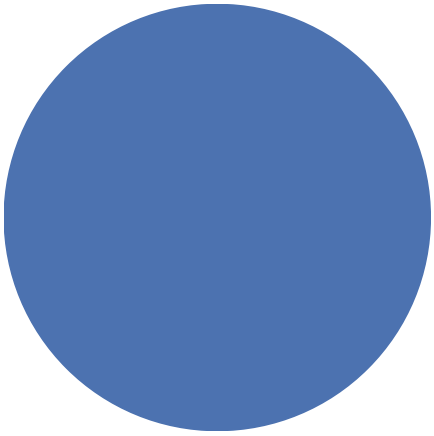

FIG. 3. Un cercle dessiné par `Matplotlib`. Après l'import des bibliothèques, il faut 7 lignes de commande...

Dans ce qui suit, on employera toujours les mêmes commandes d'amorce d'une figure : ce n'est pas la seule façon de faire, mais c'est relativement simple et clair par rapport aux autres façons.

Amorce. Dans l'exemple ci-dessus, la première commande `fig, ax = plt.subplots()` crée :

- une figure `fig`, qui sera dessinée par la dernière ligne de commande `plt.show()`. La dite figure n'apparaît plus jamais dans le programme. À la fin, on peut aussi utiliser `plt.savefig("path/to/file.pdf")` pour sauvegarder la figure à l'emplacement spécifié.
- un *Axis* `ax`, qui est une sous-figure de `fig`, sur laquelle des *Artists* `artist` viendront dessiner divers objets. On détaillera plus loin comment dessiner sur cette sous-figure : toutes les commandes seront du type `ax.artist(args)`.

Une figure peut contenir un unique *Axis*, ou plusieurs dans un *array* d'*Axes*. Si l'on veut plusieurs sous-figures l'une au-dessus de l'autre dans un tableau (respectivement, l'une à côté de l'autre), on utilisera l'amorce `fig, axs = plt.subplots(n)` (respectivement, l'amorce `fig, axs = plt.subplots(1, n)`), avec n égal au nombre de sous-figures souhaitées. On agira alors sur la k-ième figure avec `axs[k].artist(args)`. Si l'on veut carrément un tableau de $l \times c$ sous-figures, on utilisera l'amorce `fig, axs = plt.subplots(l, c)`. On agira alors sur la sous-figure d'indices (i, j) avec `axs[i,j].artist(args)`. Par exemple :

```
fig, axs = plt.subplots(1, 3)
axs[0].add_patch(matplotlib.patches.Circle((0,0), 1, color="r"))
axs[1].add_patch(matplotlib.patches.Circle((0,0), 1, color="g"))
axs[2].add_patch(matplotlib.patches.Circle((0,0), 1, color="b"))
for i in range(3):
    axs[i].set_xlim(-1, 1)
    axs[i].set_ylim(-1, 1)
    axs[i].set_aspect(1)
    axs[i].set_axis_off()
plt.show()
```

produit la figure suivante.

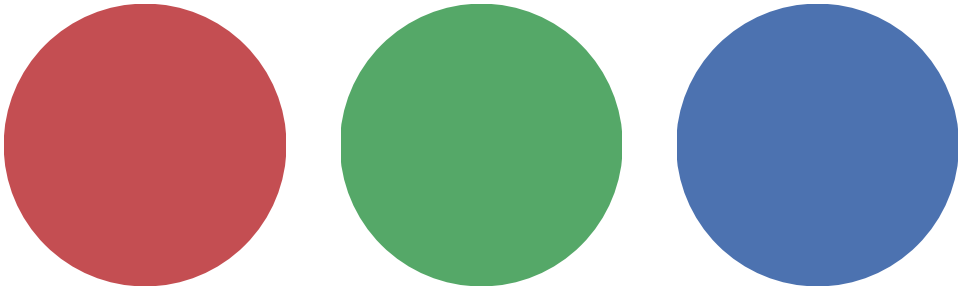

FIG. 4. Trois sous-figures Matplotlib contenant chacune un cercle.

Taille, espacement, titres. La taille de la figure peut être donnée en argument optionnel : `plt.subplots(figsize=(a, b))` force la figure à occuper un rectangle de taille $a \times b$ (par défaut, la taille d'une figure est 6.4×4.8, en pouces). Une fois la taille de la figure fixée, Matplotlib place les sous-figures avec un certain espacement. Le choix par défaut est en général assez mauvais : par exemple,

```
fig, axs = plt.subplots(2, 2) ;
plt.show()
```

donne ceci.

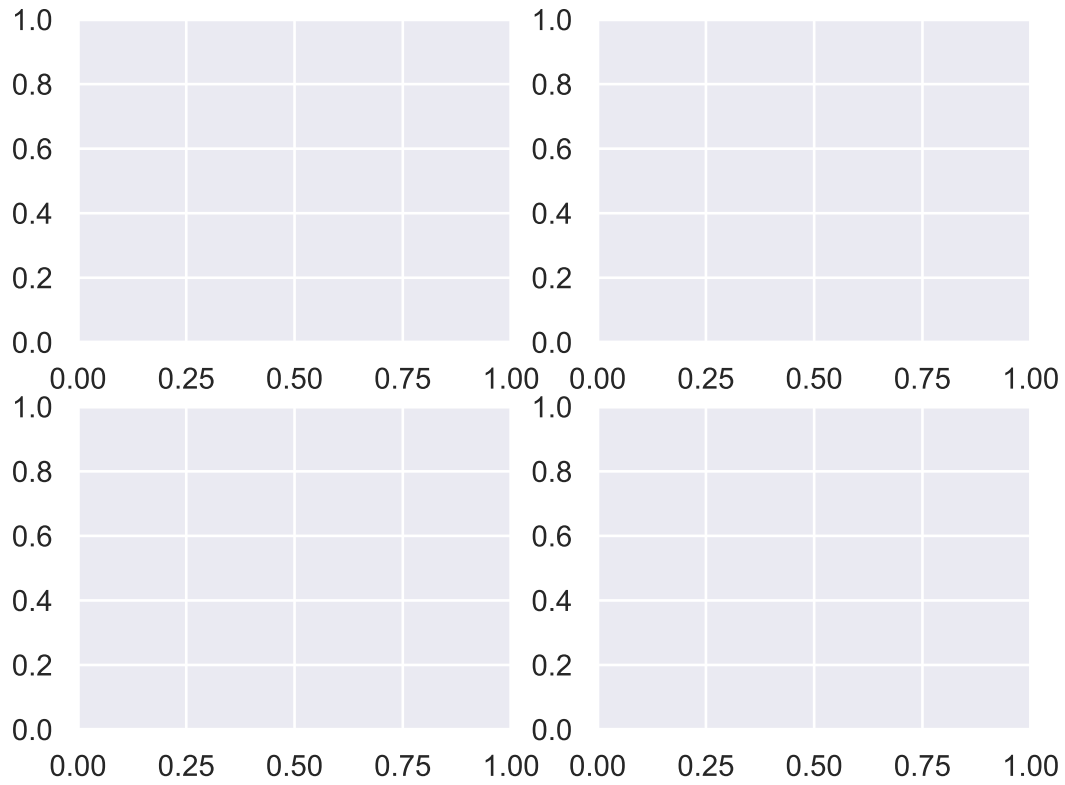

FIG. 5. Espacement par défaut d'un tableau de sous-figures.

On peut rajouter de l'espacement entre les sous-figures grâce à la commande

```
plt.subplots_adjust(wspace=x, hspace=y)
```

avec des valeurs réelles pour x et y. Par exemple, $x = y = 0.5$ donne :

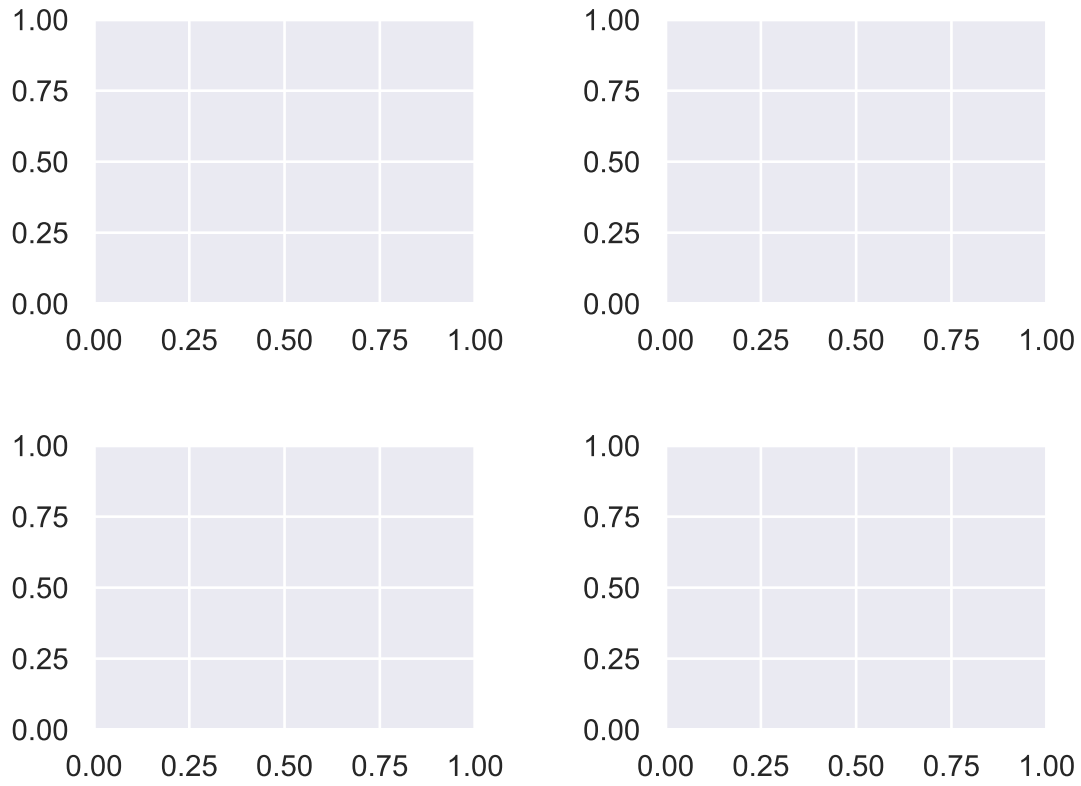

FIG. 6. Espace ajusté entre les sous-figures.

Pour chaque sous-figure `ax`, les commandes

```
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
```

avec des valeurs réelles pour les bornes $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ déterminent la boîte $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ dans laquelle les *Artists* agissant sur `ax` ont le droit de dessiner. Sans ces commandes, `Matplotlib` essaie de deviner une boîte appropriée. Par ailleurs, par défaut, le module `Matplotlib` représente les sous-figures en essayant de remplir toute la figure qui les contient. C'est un bon aspect en général, mais par conséquent, les échelles des deux axes (abscisse et ordonnée) ne sont pas en général respectées (par exemple, une unité en abscisse peut avoir la même taille que deux unités en ordonnée). Si l'on veut fixer le ratio entre abscisse et ordonnée, on utilise `ax.set_aspect(r)`, avec $r = 1$ pour que les deux axes aient la même échelle.

Le titre de la sous-figure `ax` est spécifié par `ax.set_title("titre")`. Si l'on a plusieurs sous-figures et si l'on veut un titre global, on utilisera `plt.suptitle("titre global")`.

Axes, légendes. Voyons maintenant comment spécifier les décorations usuelles d'une sous-figure `ax`.

- Les labels des deux axes sont spécifiés par `ax.set_xlabel("label des abscisses")` et `ax.set_ylabel("label des ordonnées")`.

- La graduation sur chaque axe peut être fixée avec les commandes `ax.set_xticks(L)` et `ax.set_yticks(L)`, où `L` est une liste ou un *array*. En particulier, pour avoir une graduation de l'axe des abscisses en chaque entier de $[\![0, n]\!]$, on utilisera la commande `ax.set_xticks(np.arange(n+1))`. De même, pour obtenir une graduation avec N points entre a et b, on utilisera `ax.set_xticks(np.linspace(a, b, N))`.
- Si on veut retirer les axes, on utilisera `ax.set_axis_off()`.
- Les options des commandes `set_xlabel`, `set_ylabel` et `set_title` permettent de positionner exactement comme l'on veut les labels des axes et les titres des sous-figures.
- Si un *Artist* `artist` agit sur une sous-figure `ax` par la commande

```
ax.artist(args, label="str", color=c)
```

alors le dessin effectué par `artist` aura la couleur donnée par l'argument `color`. Les couleurs noir, rouge, vert, bleu sont données respectivement par les caractères `"k"`, `"r"`, `"g"` et `"b"`. Il y a d'autres couleurs simples données par des mots clés (par exemple, `"purple"`), et on peut sinon donner un triplet RGB $(\rho, \gamma, \beta) \in [0, 1]^3$. La commande `ax.legend()` ajoute ensuite à la sous-figure `ax` un petit encart avec les labels et couleurs des différents *Artists*.

Par exemple, voici un dessin optimal du graphe de la fonction $x \mapsto \frac{x}{2}$ sur l'intervalle $[0, 2]$:

```
fig, ax = plt.subplots()
ax.plot([0,2], [0,1], label="$y=\\frac{x}{2}$", color="b")
ax.set_xlim(0, 2)
ax.set_ylim(0, 1)
ax.set_xlabel("abscisse", labelpad=10, loc="right")
ax.set_ylabel("ordonnée", labelpad=15, loc="top")
ax.set_title("graphe d'une fonction", pad=30)
ax.set_xticks(np.linspace(0, 2, 5))
ax.set_yticks(np.linspace(0, 1, 5))
ax.legend()
ax.set_aspect(1)
plt.show()
```

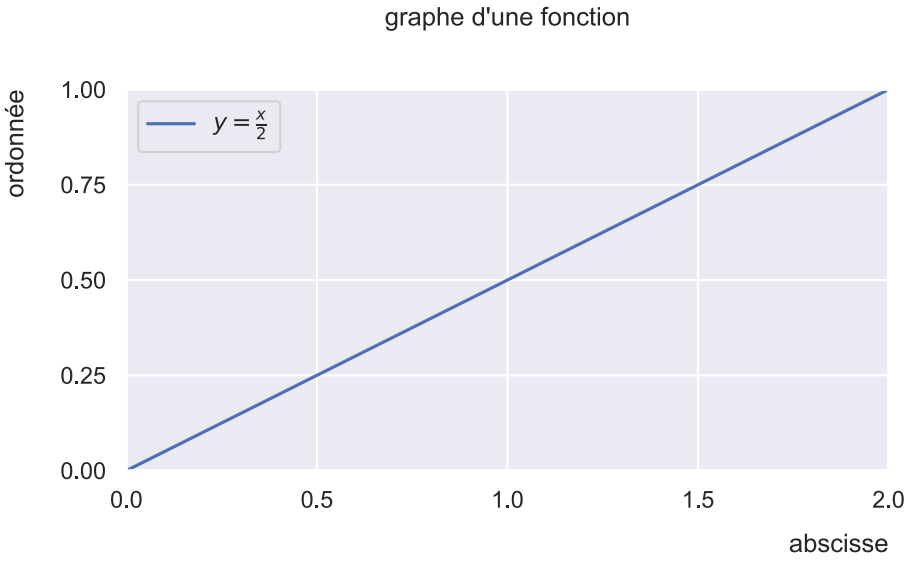

FIG. 7. Graphe de la fonction $x \mapsto \frac{x}{2}$ avec `Matplotlib`.

Lignes, graphes. On peut relier des points $(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)$ en utilisant la commande `ax.plot(x, y)`, où `x` est la liste des abscisses $(x_1, \ldots, x_l)$ donnée sous forme de liste ou d'*array*, et `y` est la liste des ordonnées $(y_1, \ldots, y_l)$. Divers arguments optionnels permettent d'ajuster l'épaisseur, la forme, la couleur, etc. du trait. Voici par exemple le résultat de :

```
ax.plot([0, 3, 2, 0], [0, 0, 2, 1], color="orange", linestyle="--")
```

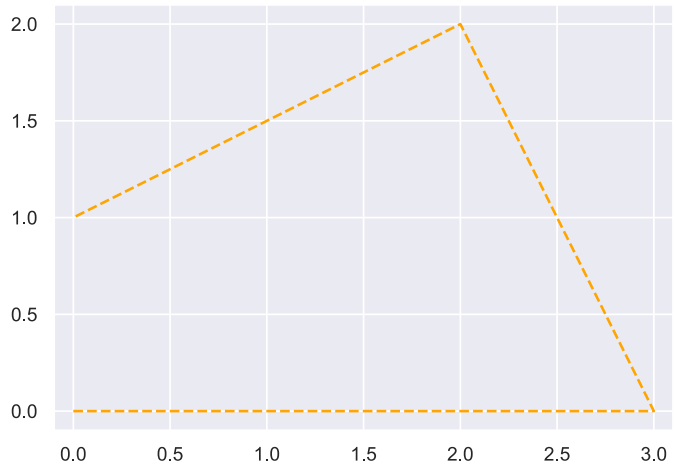

FIG. 8. Une ligne brisée dessinée avec la commande `plot`.

Si l'on veut dessiner le graphe d'une fonction $y = f(x)$, on peut utiliser une ligne reliant de nombreux points (x_i, y_i) avec $y_i = f(x_i)$. Plus précisément, la fonction :

```
def draw_function(f, a, b, samples=500):
    xx = np.linspace(a, b, samples)
    fig, ax = plt.subplots()
    ax.plot(xx, f(xx))
    plt.show()
```

dessine entre deux bornes a et b le graphe de la fonction f. Voici le résultat de cette commande avec `f = (lambda x : np.sin(x) * np.exp(-x/10))` et $[a, b] = [-5, 5]$.

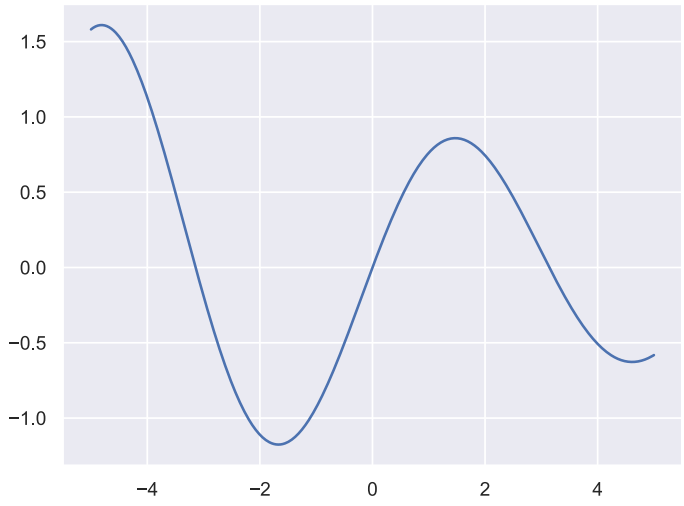

FIG. 9. Le graphe d'une fonction, approché par une ligne brisée avec de nombreux points.

Sur le même principe, on peut dessiner n'importe quelle courbe paramétrée $\gamma(t) = (x(t), y(t))$ avec $t \in [0, T]$. Il suffit de créer un échantillonnage `E = np.linspace(0, T, samples)` et de relier les points avec `ax.plot(x(E),y(E))`. Il peut y avoir des petits problèmes de lissage (la courbe dessinée a des irrégularités non souhaitées), qu'on peut généralement résoudre en augmentant le nombre `samples`.

Représentation de données. Pour conclure cette introduction à `Matplotib`, voyons comment représenter des données, qui sont par exemple obtenues à partir d'une expérience aléatoire :

- nuage de points. Si l'on a un ensemble de points du plan $((x_1, y_1), (x_2, y_2), \dots, (x_l, y_l))$, on peut le dessiner avec `ax.scatter(x, y)`. Par exemple,

```
fig, ax = plt.subplots()
x = np.linspace(0, 1, 100) + scs.norm(0, 0.1).rvs(size=100)
y = 0.5*x + 0.2 + scs.norm(0, 0.1).rvs(size=100)
ax.scatter(x, y, color=np.where(y>0.5*x+0.2, "g", "b"))
plt.show()
```

dessine un nuage de points autour de la droite d'équation $y = \frac{x}{2} + \frac{1}{5}$, avec des erreurs d'observation gaussiennes. On a mis comme paramètre de couleur une liste, qui donne la couleur verte si $y_i > \frac{x_i}{2} + \frac{1}{5}$ et la couleur bleue sinon.

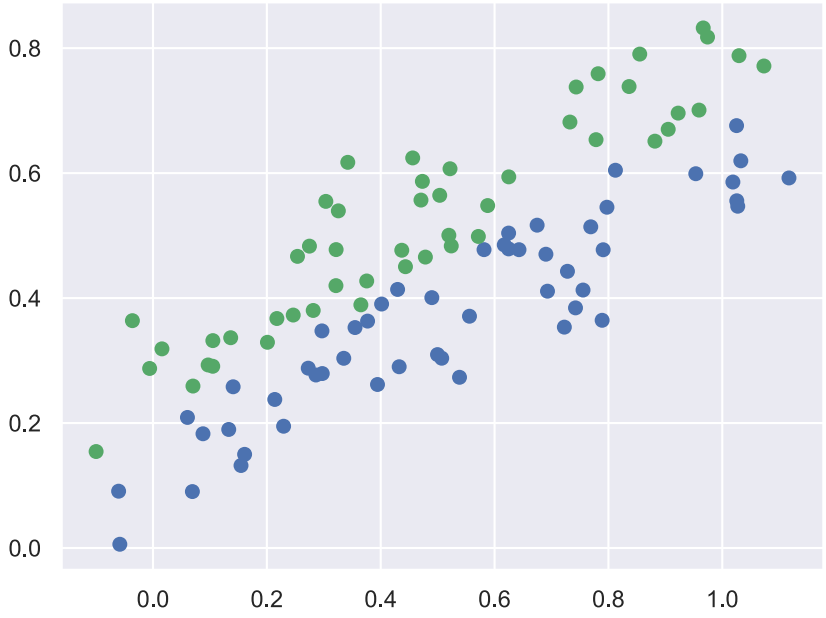

FIG. 10. Un nuage de points dessiné avec la commande `scatter`.

- fonction de répartition empirique. Si l'on a un échantillon `X` (sous forme d'*array*), on peut en représenter la fonction de répartition avec la commande `ax.ecdf(X)`. Par exemple,

```
fig, ax = plt.subplots()
X = scs.norm.rvs(size=100)
ax.ecdf(X)
ax.set_xlim(min(X), max(X))
ax.set_ylim(-0.1, 1.1)
plt.show()
```

dessine la fonction de répartition empirique d'un vecteur gaussien de taille $N = 100$.

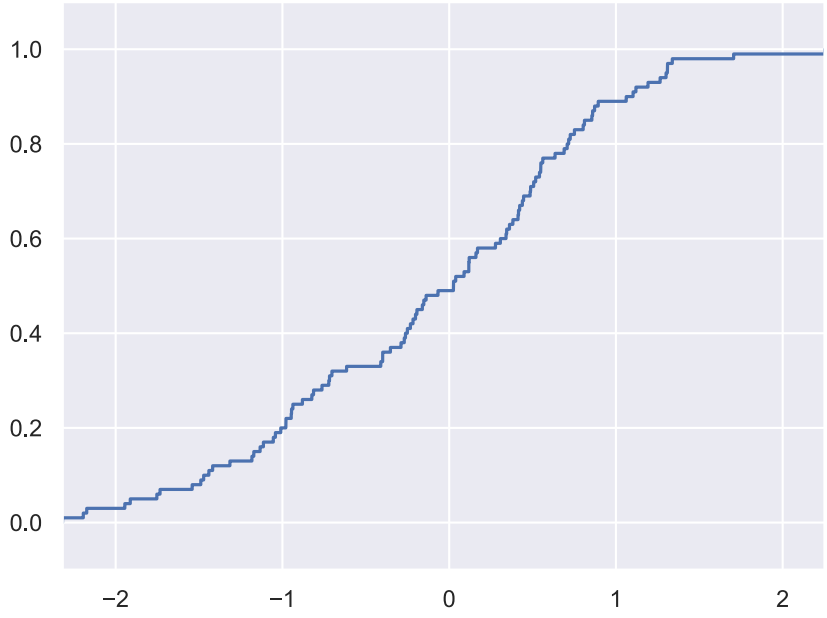

FIG. 11. Une fonction de répartition empirique obtenue avec la commande `ecdf`.

4. NetworkX.

Le module `NetworkX` permet de manipuler des graphes (paires $G = (V, E)$ formées d'un ensemble de sommets V et d'un ensemble E d'arêtes), de les dessiner et de calculer diverses statistiques sur ces objets. On ajoutera à la liste d'imports usuelle la commande :

```
import networkx as nx
```

Construction de graphes. Un graphe vide est créé avec la commande `nx.Graph()`. Ses ensembles de sommets et d'arêtes sont alors vides :

```
>>> G = nx.Graph()
>>> G.nodes(), G.edges()

(NodeView(()), EdgeView([]))
```

On peut ajouter des sommets au graphe avec la commande `G.add_nodes_from(S)`, où `S` est une liste de sommets ; et des arêtes avec la commande `G.add_edges_from(E)`, où `E` est une liste de paires de sommets. Par exemple, on crée un triangle comme suit :

```
>>> G.add_nodes_from([0, 1, 2])
>>> G.add_edges_from([(0,1), (0,2), (1,2)])
>>> G.nodes(), G.edges()

(NodeView((0, 1, 2)), EdgeView([(0, 1), (0, 2), (1, 2)]))

>>> G.number_of_nodes(), G.number_of_edges()

(3, 3)
```

On expliquera plus loin comment dessiner sur une figure `Matplotlib` une représentation de ce graphe.

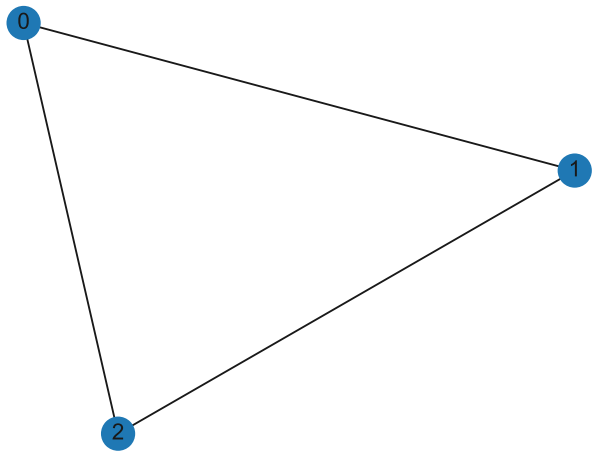

FIG. 12. Un triangle encodé avec NetworkX et dessiné avec Matplotlib.

La construction précédente est tout à fait appropriée lorsque le graphe est construit récursivement, et peut croître au fur et à mesure d'un algorithme. Alternativement, on peut définir un graphe en donnant un dictionnaire dont les clés sont les sommets, et dont les entrées sont les listes des voisins :

```
>>> H = nx.Graph({0 : [1,2], 1: [0,2], 2: [0,1]})
>>> nx.is_isomorphic(G, H)

True
```

Pour définir un graphe, on peut aussi utiliser la matrice d'adjacence. Si `G` est un graphe `NetworkX`, sa matrice d'adjacence est obtenue avec la commande `nx.to_numpy_array(G)`. Réciproquement, si `A` est un *array* carré dont les entrées sont dans $\{0, 1\}$, alors `nx.Graph(A)` renvoie le graphe dont la matrice d'adjacence est `A`.

```
>>> A = nx.to_numpy_array(G)
>>> A

array([[0., 1., 1.],
       [1., 0., 1.],
       [1., 1., 0.]])

>>> K = nx.Graph(A)
nx.is_isomorphic(G, K)

True
```

La classe `nx.Graph` est adaptée à la manipulation de graphes *simples* (pas d'arêtes multiples), éventuellement avec des *boucles* basées en les sommets, et *non orientés* (les arêtes sont des paires $\{v, w\}$ de sommets, sans prendre en compte l'ordre). Le module `NetworkX` définit également des classes permettant de manipuler des *multigraphes* (graphes avec éventuellement des arêtes multiples), ou des *graphes orientés*. En particulier, pour les graphes orientés, on utilisera la classe `nx.DiGraph`. Ainsi, la commande `nx.DiGraph({0 : [1], 1: [2], 2: [0]})` crée un triangle orienté dont les arêtes forment un cycle. On peut oublier l'orientation des arêtes d'un graphe orienté `D` avec la commande `D.to_undirected()`.

Manipulation. On a déjà expliqué comment ajouter des sommets ou des arêtes à un graphe `G`. Les méthodes `G.remove_nodes_from` et `G.remove_edges_from` correspondent aux opérations

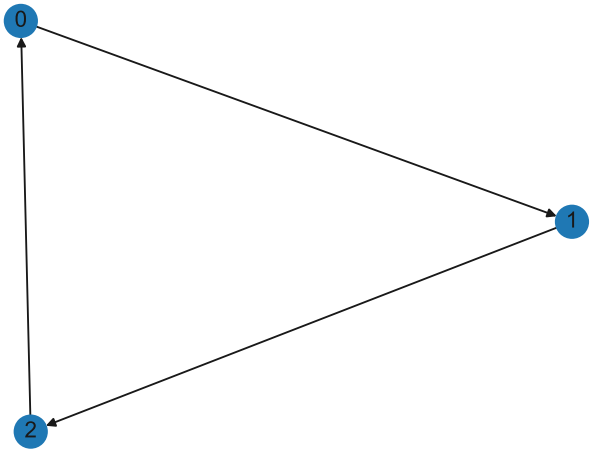

FIG. 13. Un triangle orienté avec NetworkX.

inverses de suppression d'un sommet ou d'une arête. Les commandes `G.number_of_nodes()` et `G.number_of_edges()` calculent le nombre de sommets et le nombre d'arêtes du graphe `G`. Dans tout ce qui suit, on manipulera le graphe dessiné ci-dessous :

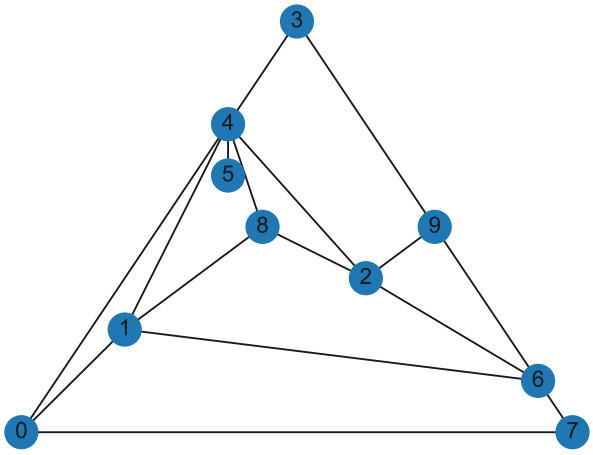

FIG. 14. Un graphe non orienté avec 10 sommets et 16 arêtes.

Chaque sommet ou arête de `G` peut être étiqueté : les étiquettes sont des paires (clé, valeur) d'un dictionnaire. Par exemple, si l'on attribue aux arêtes de `G` des poids aléatoires avec la commande :

```
for e in G.edges():
    G.edges[e]["weight"] = random.randint(0, 10)
```

alors on obtient le graphe étiqueté dessiné ci-après (où on a juste représenté les arêtes et leurs poids). Chaque sommet `v` du graphe a déjà par défaut un dictionnaire, dont les clés sont les voisins de `v`, et dont l'entrée associée à la clé `w` est le dictionnaire des étiquettes de l'arête (v, w) (par défaut, ce dictionnaire est vide).

```
>>> G[3]

AtlasView({4: {'weight': 0}, 9: {'weight': 3}})
```

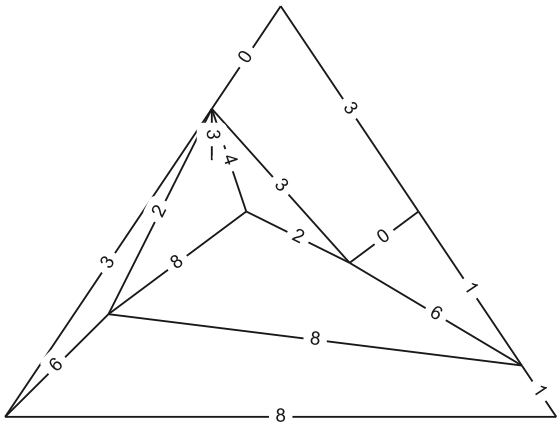

FIG. 15. Chaque arête ou sommet d'un graphe encodé avec NetworkX peut être étiqueté, avec une ou plusieurs étiquettes stockées dans un dictionnaire.

La somme de tous les poids des arêtes vis-à-vis des étiquettes de label "weight" est obtenue avec la méthode size(weight="weight"). Par exemple :

```
>>> G.size(weight="weight")

58.0
```

Algorithmes. Une longue liste d'algorithme plus ou moins complexes est implantée dans NetworkX. Donnons-en quelques uns :

- Les commandes nx.is_bipartite(G), nx.is_connected(G), nx.is_planar(G) et nx.is_tree(G) testent si G est un graphe bipartite, un graphe connexe, un graphe planaire et un arbre. Pour notre exemple :

```
>>> nx.is_bipartite(G), nx.is_connected(G), nx.is_planar(G), nx.is_tree(G)

(False, True, True, False)
```

- La commande nx.connected_components(G) renvoie un itérateur dont les objets sont les parties de l'ensemble des sommets de G qui en forment les composantes connexes. On peut aussi demander directement la composante connexe contenant un sommet v avec nx.node_connected_component(G, v).

```
>>> nx.node_connected_component(G, 0)

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- On peut tester l'isomorphisme entre deux graphes G et H (existence d'une bijection entre les sommets préservant l'adjacence) avec la commande nx.is_isomorphic(G, H).
- On peut facilement calculer des distances entre les sommets d'un graphe G. Par exemple, nx.shortest_path(G, v, w) trouve un chemin de longueur minimale entre les deux sommets v et w, et nx.shortest_path_length(G, v, w) calcule la distance de graphe entre ces sommets (la longueur minimale d'un chemin). Ces deux fonctions peuvent prendre en argument un poids sur les arêtes.

```
>>> nx.shortest_path_length(G, 0, 6)
```

```
2

>>> nx.shortest_path(G, 0, 6)

[0, 1, 6]

>>> nx.shortest_path_length(G, 0, 6, weight="weight")

7

>>> nx.shortest_path(G, 0, 6, weight="weight")

[0, 4, 3, 9, 6]
```

Dessin. On peut dessiner un graphe `G` encodé avec `NetworkX` en utilisant tout simplement la commande `nx.draw(G)`. Toutefois, il est préférable d'utiliser une syntaxe un peu plus compliquée, mais qui donne un bien meilleur contrôle du dessin, et qui permet de mieux comprendre ce qui se passe. On conseille ainsi :

```
fig, ax0 = plt.subplots()
pos0 = nx.spring_layout(G)
nx.draw_networkx(G, pos=pos0, ax=ax0, with_labels=True)
ax0.set_axis_off()
plt.show()
```

Expliquons en détail cette suite d'instructions :

- Le dessin du graphe `G` sera posé sur une figure `Matplotlib`, qu'on introduit comme décrit dans la section précédente. Ceci permet de combiner toute la puissance du module `NetworkX` avec les bonnes pratiques de dessin vues précédemment (sous-figures, titres, légendes, etc). Dans la commande principale `nx.draw_networkx`, la sous-figure sur laquelle on agit est précisée par l'argument `ax=ax0`.
- Pour dessiner un graphe, il faut savoir où placer ses sommets. Plusieurs choix sont possibles : aléatoirement dans le plan, en essayant d'avoir le moins d'intersections possibles entre les arêtes, sur un cercle, etc. La commande `pos0 = nx.spring_layout(G)` calcule un dictionnaire dont les clés sont les sommets, et dont les entrées sont les coordonnées où placer chaque sommet. Par exemple, les dessins précédents avaient pour *Layout* :

```
>>> nx.planar_layout(G)

{0: array([-1.        , -0.39759036]),
 1: array([-0.63855422, -0.15662651]),
 2: array([ 0.20481928, -0.03614458]),
 3: array([-0.03614458,  0.56626506]),
 4: array([-0.27710843,  0.3253012 ]),
 5: array([-0.27710843,  0.20481928]),
 6: array([ 0.80722892, -0.27710843]),
 7: array([ 0.92771084, -0.39759036]),
 8: array([-0.15662651,  0.08433735]),
 9: array([0.44578313, 0.08433735])}
```

On précise les emplacements des sommets avec l'argument `pos=pos0` dans la commande principale `nx.draw_networkx`. On peut définir soit-même à la main un dictionnaire contenant les emplacements, ou utiliser les commandes `nx.spring_layout(G)`,

`nx.planar_layout(G)` et `nx.circular_layout(G)` pour produire automatiquement des emplacements *équilibrés* (les sommets sont placés en se repoussant les uns les autres, comme s'il y avait des ressorts), *planaires* (si le graphe est planaire, les arêtes du dessin ne s'intersectent pas) et *circulaires* (les sommets sont placés sur un cercle). On renvoie à la documentation de `NetworkX` pour d'autres *Layouts* classiques.

- On peut ensuite ajouter de très nombreux arguments optionnels ; par exemple, l'argument `with_labels = True` force les numéros des sommets à apparaître. On peut préciser la taille des sommets, leur forme, leur couleur, l'épaisseur des arêtes, etc.
- Si l'on veut dessiner séparément les sommets et les arêtes, on peut utiliser les commandes `nx.draw_networkx_nodes` et `nx.draw_networkx_edges` ; c'est parfois utile pour gérer précisément les options d'affichage. Dans ce cas, il est essentiel d'avoir stocké à l'avance dans un *array* `pos0` les emplacements des sommets ; on utilise alors ce même *array* comme argument des différentes commandes de dessin.