Objectifs de la séance : apprendre à utiliser Python et ses modules dans un contexte de simulations aléatoires.

Ouvrir un terminal, puis lancer `Python/Jupyter` avec la commande `jupyter notebook`. Créer une feuille de calcul `Python 3`. Pour exécuter le contenu d'une cellule de la feuille de calcul, taper (Shift+Enter). Dans la première cellule, importer les modules suivants :

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

Le module `Numpy` aide à la manipulation de vecteurs et à l'exécution de fonctions mathématiques ; le module `Scipy` définit notamment les distributions de probabilité usuelles ; et le module `Matplotlib` permettra de faire tous les dessins.

Opérations, fonctions, méthodes. On commence par rappeler les bases de `Python` et de son module `Numpy`, qui est essentiel pour les mathématiques.

(1) Calculer

$$3+2+5 \quad ; \quad \sin\left(\frac{\pi}{3}\right) \quad ; \quad \sqrt{1-\sin^2\left(\frac{\pi}{3}\right)}-\cos\left(\frac{\pi}{3}\right).$$

Les fonctions mathématiques comme sin ou $\sqrt{\cdot}$ sont implantées par le module `Numpy` (on utilisera respectivement `np.sin` et `np.sqrt`).

(2) On stocke ou modifie une variable nommée x avec la commande `x = val`, où `val` est la valeur que l'on veut donner à x. Recalculer la troisième quantité de la première question en stockant d'abord une variable égale `x` à $\sin\left(\frac{\pi}{3}\right)$. Pour afficher la valeur d'une variable `x`, on utilise la commande `print(x)`, ou simplement `x`.

(3) Le module `Numpy` introduit aussi les vecteurs et les matrices. Avec la commande `np.array?`, demander la documentation de la classe `array`. Créer une variable `M` égale à la matrice $\left(\begin{smallmatrix}1&2\\0&1\end{smallmatrix}\right)$.

(4) On peut appliquer une fonction à tous les termes d'un vecteur ou d'une matrice stockée dans un `array Numpy`. Par exemple, `np.cos(np.pi/3 * M)` calcule la matrice

$$\begin{pmatrix}\cos\left(\frac{\pi}{3}\right) & \cos\left(\frac{2\pi}{3}\right)\\ 0 & \cos\left(\frac{\pi}{3}\right)\end{pmatrix}.$$

Effectuer ce calcul.

(5) Tous les objets dans `Python` ont des méthodes qui leurs sont associées, et qui dépendent du type de l'objet en question. Une méthode `method` d'un objet `x` est appelée par la commande `x.method()`. Pour avoir la liste des méthodes associées à un objet `x`, on peut taper `x.` + (Tab). Calculer la transposée de la matrice `M`.

(6) Les outils d'algèbre linéaire sont introduits par le sous-module `np.linalg` ; on renvoie à https://numpy.org/doc/stable/reference/routines.linalg.html. Calculer le déterminant, les valeurs propres et l'inverse de la matrice `M`.

(7) Une fonction $f(x)$ d'une variable x est définie dans `Python` comme suit :

```
def f(x):
    instructions
    return (le résultat)
```

Créer une fonction $g(x) = \sqrt{1 - \exp(-x)}$, et calculer ses valeurs en $0, 0.1, 0.2, \ldots, 1$. On pourra utiliser la commande `np.linspace(a, b, N)`, qui crée un vecteur de taille N avec des valeurs régulièrement espacées entre a et b.

Dessins. La syntaxe générale pour faire un dessin avec `Matplotlib` est :

```
fig, ax = plt.subplots()
instructions de dessin: ax.dessin(args)
paramètres pour afficher: ax.parametres(args)
plt.show()
```

(1) Pour dessiner une fonction $f(x)$ sur un intervalle $[a, b]$, on utilise un échantillonnage régulier `xx` de cet intervalle, et la commande `ax.plot(xx, f(xx))`.

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

Dessiner le graphe de la fonction $g(x) = \sqrt{1 - \exp(-x)}$ sur l'intervalle $[0, 5]$. Trouver les options d'affichage pour que la courbe soit dessinée en rouge, pour que les axes des abscisses et des ordonnées contiennent respectivement les intervalles $[-0.1, 5.1]$ et $[-0.1, 1.1]$, pour que les graduations de l'axe des abscisses soient en tous les entiers entre 0 et 5, et pour qu'une légende de la fonction g apparaisse.

(2) Superposer au graphe précédent celui de la fonction $h(x) = \sqrt{1 + x}\,\sin\frac{x}{2}$, dessinée en bleu. C'est l'un des principes importants du dessin avec `Matplotlib` : on peut réaliser des dessins complexes en listant des opérations simples, qu'on applique successivement à `ax`.

(3) Dessiner sur le même graphique les courbes des fonctions $f_\alpha(x) = \sin(\alpha x)$ avec $\alpha \in \{1, 2, 3, 4, 5\}$. On pourra utiliser une boucle `for` :

```
for x in L:
    instructions(x)
```

où `L` est un objet `Python` muni d'un itérateur, par exemple une liste ou un `array Numpy`.

Nous verrons dans un instant d'autres types de dessin dans le contexte de simulation de variables aléatoires.

Variables aléatoires. Les lois usuelles sont implantées par le module `Scipy`. Par exemple, `G = scs.norm(0, 1)` définit la loi gaussienne, et on peut avoir un échantillon de N variables indépendantes de loi $\mathcal{N}(0, 1)$ avec la commande `G.rvs(N)`.

(1) Trouver avec la documentation https://docs.scipy.org/doc/scipy/reference/stats.html les commandes pour obtenir des échantillons de taille 20 des lois suivantes :

- loi binomiale $\mathcal{B}(n,p)$ avec $n=10$ et $p=0.3$;
- loi exponentielle $\mathcal{E}(\lambda)$ avec $\lambda=3$;
- loi gaussienne $\mathcal{N}(m,\sigma^2)$ avec $m=1$ et $\sigma^2=2$;
- loi de Poisson $\mathcal{P}(\lambda)$ avec $\lambda=3$.

Engendrer aussi une matrice A de taille 5×5 dont les entrées sont des variables aléatoires indépendantes de loi $\mathcal{N}(0,1)$.

(2) Si `scs.loi` est une distribution de probabilité à densité, sa densité $f(x)$ en x est obtenue avec la commande `scs.loi.pdf(x)`, et la fonction de répartition correspondante $F(x)=\int_{-\infty}^{x} f(s)\,ds$ est obtenue avec la commande `scs.loi.cdf(x)`. Dessiner sur deux graphiques différents la densité et la fonction de répartition de la loi normale $\mathcal{N}(0,1)$.

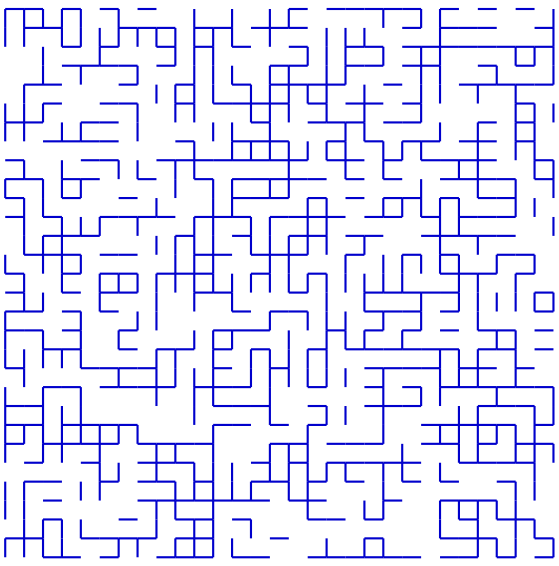
(3) Si $(X_n)_{1\leq n\leq N}$ est un échantillon d'une loi de fonction de répartition F, la *fonction de répartition empirique* est l'approximation de F définie par :

$$F_N(x)=\frac{1}{N}\sum_{n=1}^{N} 1_{(X_n\leq x)}.$$

Le théorème de Glivenko–Cantelli garantit que presque sûrement, F_N converge en norme infinie vers F lorsque N tend vers l'infini. Vérifier ce théorème avec la loi normale $\mathcal{N}(0,1)$ et un échantillon de taille 200. On pourra utiliser la commande `ax.ecdf(E)` pour tracer la fonction de répartition empirique d'un échantillon E.

(4) Si `scs.loi` est une distribution de probabilité sur l'ensemble des entiers, on la représente plutôt avec un diagramme en bâtons / histogramme. Les valeurs $\mathbb{P}[X=k]$ sont obtenues avec la commande `scs.loi.pmf(k)`, et l'histogramme avec la commande `ax.bar(np.arange(a, b+1), scs.loi.pmf(np.arange(a, b+1)))`, où $[\![a,b]\!]$ est l'intervalle sur lequel on veut dessiner l'histogramme. Dessiner l'histogramme de la loi de Poisson de paramètre $\lambda=3$.

(5) La commande `ax.plot(X, Y)` relie dans le plan les points dont l'abscisse est dans la liste `X`, et dont l'ordonnée est dans la liste `Y`. Virtuellement, on peut absolument tout dessiner avec des suites de telles instructions, en découpant toute figure en petits traits. Écrire une fonction `percolation(n, p)` qui garde chaque arête de la grille $[\![0,n]\!]\times[\![0,n]\!]$ avec probabilité p indépendamment pour chaque arête, et qui affiche le résultat.



Commenter les dessins obtenus avec $n = 50$ et $p \in \{0.3, 0.4, 0.5, 0.6\}$.

Chaînes de Markov. Attaquons-nous finalement à la programmation de diverses chaînes de Markov : marches aléatoires, processus de branchement, *etc.*

(1) Écrire un programme `marche(n, p)` qui affiche les n premières étapes d'une marche aléatoire sur $\mathbb{Z}$ de paramètre p. Vérifier expérimentalement que si $p > \frac{1}{2}$, alors la marche aléatoire tend presque sûrement vers $+\infty$; tandis que si $p = \frac{1}{2}$, alors la marche semble revenir infiniment souvent en 0.

(2) Écrire un programme `marche_cercle(n, N)` qui calcule les n premières étapes d'une marche aléatoire sur $\mathbb{Z}/N\mathbb{Z}$, issue de 0 et avec probabilités de transition

$$P(k,k) = P(k,k-1) = P(k,k+1) = \frac{1}{3}.$$

Pour calculer dans `Python` la réduction modulo N d'un entier a, on utilisera la commande `a % N`. Vérifier avec un histogramme empirique que pour n grand, la loi de la position X_n est quasi-uniforme sur $\mathbb{Z}/N\mathbb{Z}$ (on pourra par exemple prendre $n = 100$, $N = 10$ et un échantillon de taille 10000).

(3) Soit μ une mesure de probabilité sur $\mathbb{N}$. Le *processus de Galton–Watson* de loi μ est la suite aléatoire $(X_n)_{n\in\mathbb{N}}$, telle que X_{n+1} est la somme de X_n variables aléatoires indépendantes de loi μ. Il représente l'évolution d'une population dont chaque individu a aléatoirement $Y \sim \mu$ enfants, indépendamment pour chaque individu. Calculer la matrice de transition de cette chaîne de Markov, et écrire un programme `galton_watson(mu, n)` qui dépend d'une distribution de probabilité `mu` et qui donne les n premières générations du processus, partant de $X_0 = 1$. Tester ce programme avec des lois de Poisson de paramètre $\lambda \in \{1, 1.5, 2\}$.

(4) On peut définir une classe `Markov` qui s'initialise à partir d'une matrice stochastique (de taille finie) P, et qui a des méthodes pour calculer toutes les quantités de la théorie des chaînes de Markov, en particulier des trajectoires aléatoires $(X_n)_{0\leq n\leq N}$ issues d'un point X_0 donné. Si P est de taille N, l'espace des états sera $[\![0, N-1]\!]$.

```
class Markov:

    def __init__(self, P):
        self.transition = P
        self.size = P.shape[0]

    def trajectoire(self, n, x0):
        res = np.zeros(n+1, dtype=np.int64)
        res[0] = x0
        for i in range(n):
            res[i+1] = np.random.choice(self.size, p=self.transition[res[i], :])
        return res
```

La commande `C = Markov(P)`, qui prend en argument un `array P` bidimensionnel, crée l'espace et la matrice de transition ; et la commande `C.trajectoire(n, x0)` calcule la trajectoire jusqu'au temps n. Utiliser cette classe pour programmer le modèle des urnes d'Ehrenfest, et pour reprogrammer la marche aléatoire sur le cercle. Rajouter des méthodes pour calculer la loi marginale au temps n sachant la loi de départ, et la mesure empirique d'une trajectoire ; pour déterminer si la matrice de transition est irréductible ; et pour calculer dans ce cas la mesure invariante de la chaîne. Vérifier la validité du théorème ergodique pour les deux modèles précités.