Python – travaux pratiques

Les slides du cours, avec l'aide Python [1], doivent constituer une documentation suffisante. De nombreux tutoriels sont accessibles depuis la page du cours :

`http://www.cmap.polytechnique.fr/~benaych/aleatoire_index.html`

Un corrigé sera mis en ligne après la fin des TP sur cette page. Ne pas hésiter à le consulter !

Démarrage

1. Dans un terminal, créer un répertoire de travail (par exemple : `mkdir tp_python`) et s'y placer (`cd tp_python`).
2. Lancer **spyder** par la commande `spyder` ou, *au choix*, **jupyter notebook** par la commande `jupyter notebook`
3. Option **spyder** : on peut, alors, taper les commandes dans la fenêtre de commande, les programmes étant, quant à eux, écrits dans l'éditeur de texte intégré, puis exécutés. Option **jupyter notebook** : le notebook est fait de cellules qui peuvent être des cellules de texte (pour des explications) et des cellules de code (que l'on peut executer).

Cet énoncé est **long**. Nous conseillons aux élèves de commencer par les exercices 1 et 2, puis d'**avancer selon leur choix** : les exercices 3 et 8 ne sont pas les plus faciles, mais sont peut-être les plus divertissants, alors que les exercices 4, 5, 6 et 7, plus scolaires, préparent directement les élèves aux projets des cours MAP311-312.

Dans les indications données ici, il est implicite que les programmes commencent par

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

1 Prise en main

a) On considère n cartes numérotées de 0 à $n-1$ et triées par ordre croissant.

1. On rappelle que l'aide Python pour une fonction appelée `ma_fonction` s'obtient en tapant : `? ma_fonction` dans la console.

— on supprime la première carte,
— on met la suivante à l'autre extrémité du paquet,
— on recommence jusqu'à ce qu'il n'en reste qu'une.

Ecrire un programme qui affiche la liste des cartes supprimées dans l'ordre de leurs suppressions, ainsi que la carte restante.
Indication : on utilisera une boucle `while` et la récursivité n'est pas utile.

b) Soit f définie par $f(x) = \frac{\sin(x)}{x}$ si $x \neq 0$ et $f(0) = 1$. Tracer son graphe sur $[-a, a]$ en rouge, pour $a = 20\pi$ par exemple. On pourra ajouter un titre et une légende.
Indication : π s'obtient avec `np.pi`, la commande `plt.plot(x,y,color="r",label="ma legende")` affiche en rouge la courbe affine par morceaux reliant les points d'abscisses `x` et d'ordonnées `y`, `plt.legend(loc='best')` affiche la légende définie plus haut avec `label` (en position optimale) et `plt.title("mon titre")` donne un titre à la figure.

c) Une *suite de Fibonacci aléatoire* est une suite réelle (u_n) vérifiant, pour tout $n \geq 0$, $u_{n+2} = u_{n+1} + s_n u_n$ où (s_n) est une suite de variables aléatoires indépendantes prennant les valeurs ± 1 avec probabilités $1/2, 1/2$. Tracer sur un même graphique les n premiers termes de suites de Fibonacci aléatoires choisies indépendamment (càd correspondant à des suites (s_n) indépendantes) avec éventuellement des premiers termes différents.
Indication : on utilisera `2*np.random.binomial(1,.5,n)-1` pour générer `n` variables aléatoires indépendantes prenant les valeurs ± 1 avec probabilités $1/2, 1/2$.

2 Loi des Grands Nombres et Théorème Central Limite

a) Afin d'illustrer la *Loi des Grands Nombres* (cf cours ou slides de l'amphi Python), visualiser la suite $S_n = \frac{X_1 + \cdots + X_n}{n}$ pour X_i une suite de variables aléatoires indépendantes de loi uniforme sur $[-1, 1]$.
Indication : pour `x`$= [x_1, \ldots, x_n]$ vecteur, `np.cumsum(x)` est le vecteur

$$[x_1,\ \ x_1 + x_2,\ \ x_1 + x_2 + x_3, \ldots,\ \ x_1 + \cdots + x_n]$$

des sommes cumulées des coordonnées de `x`.

b) Faire de même avec des variables aléatoires Y_i de loi de Cauchy (càd de densité $\frac{1}{\pi}\frac{1}{1+x^2}$), qui s'obtiennent avec la formule $Y_i = \tan(\pi X_i/2)$. La suite $\frac{Y_1 + \cdots + Y_n}{n}$ semble-t-elle converger ? Pourquoi ?

c) Calculer la moyenne μ et l'écart-type σ des X_i et verifier, afin d'illustrer le *Théorème Central Limite* (cf cours ou slides de l'amphi Python), que $\sqrt{n}(S_n - \mu)/\sigma$ converge en loi, lorsque $n \to \infty$, vers $\mathcal{N}(0, 1)$: on se fixera une grande valeur de n, on simulera un grand nombre de fois $\sqrt{n}(S_n - \mu)/\sigma$, on en tracera l'histogramme et on le comparera à la densité $(2\pi)^{-1/2} e^{-\frac{x^2}{2}}$ de $\mathcal{N}(0, 1)$.

Indication : pour `x` un tableau bidimensionnel de taille $n \times p$, `np.sum(x,axis=1)` donne le tableau unidimensionnel de taille n dont les coordonnées sont les sommes des lignes de `x`.

3 Battage de cartes par le *riffle shuffle*

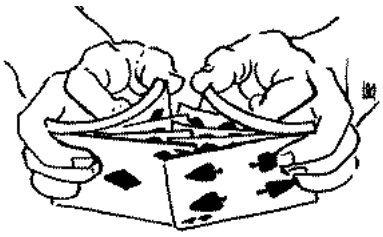
On s'intéresse ici à la question simple de savoir combien de fois il faut mélanger un jeu de cartes pour qu'il soit *suffisamment mélangé*. Cette question a donné lieu à de nombreux travaux scientifiques, notamment les articles du mathématicien Persi Diaconis, probabiliste de renommée mondiale et ancien prestidigitateur professionnel, dont les travaux sur la question (cf bibliographie) ont eu pour conséquence une modification de la réglementation à Las Vegas.

On considère un jeu de n cartes numérotées de 0 à $n-1$. Au départ, elles sont ordonnées par ordre croissant et on cherche à les *battre* (càd appliquer une permutation aléatoire de $\{0,\dots,n-1\}$) de façon à ce que les cartes soient réparties de la façon la plus aléatoire possible. La solution optimale consisterait à appliquer une permutation aléatoire choisie de façon uniforme, mais cela est bien sûr hors de portée, le choix d'une telle permutation devant se faire parmi les $n!$ permutations possibles, ce qui est beaucoup trop si n vaut, par exemple, 52. On opère en conséquence à un battage "artisanal" (mais réglementé lorsque l'on se trouve dans un cercle de jeu public). Nous allons ici étudier un exemple de battage, le *riffle shuffle* : nous proposerons un modèle probabiliste pour celui-ci et étudierons, via un programme Python, le caractère uniforme de la répartition des cartes qu'il produit. Nous chercherons notamment le nombre minimum de *riffle shuffles* à réaliser pour que le paquet de cartes soit suffisamment mélangé.

Le *riffle shuffle* (cf figure ci dessous) se décompose en deux étapes que l'on modélise ainsi :

1. on coupe le jeu en deux paquets de taille k et $n-k$: les k premières cartes du paquet et les $n-k$ suivantes, k étant choisi de manière aléatoire suivant une loi Binomiale de paramètres n et $1/2$,
2. on entrelace alors les deux paquets de la manière suivante :
   (a) si on a n_1 cartes dans le paquet 1 et n_2 cartes dans le paquet 2, on choisit d'abord une carte :
   — la première carte du paquet 1 avec probabilité $\dfrac{n_1}{n_1+n_2}$,
   — la première carte du paquet 2 avec probabilité $\dfrac{n_2}{n_1+n_2}$,
   (b) on continue l'expérience, en actualisant n_1 et n_2 à chaque étape, jusqu'à épuisement des deux paquets.

1. Ecrire une fonction `RS` qui s'applique à un objet de type `list`, appelé `J` (il sera pratique de supposer que `J` ne contient que des valeurs 2 à 2 distinctes), qui renvoie un objet de type `list`, appelé `R`, correspondant à un *riffle shuffle* appliqué à `J` (`R` contiendra donc les mêmes éléments que `J`, mais dans un autre ordre).



Indication : On pourra utiliser `np.random.binomial`, `np.random.choice`, `list.append`, `list.remove`, `list.extend`.

2. Ecrire une fonction `Trajectoire` qui s'applique à un objet de type `list`, appelé `J0` et à un `int` appelé `T`, qui renvoie un objet de type `list`, de longueur `T+1`, dont le premier élément est `J0` et dont les éléments se déduisent successivement les uns des autres par application de `RS`.

3. On a donc une fonction qui, à un ordre initial, que l'on choisira `J0=range(n)`, pour `n=52`, associe la suite J_0 =`J0`, J_1=`RS(J0)`, ..., J_t=`RS`(J_{t-1}), ..., J_T, aléatoire, de ses *riffle shuffle* successifs. On cherche à mesurer comment la loi de J_t se rapproche de la loi uniforme sur l'ensemble des permutations de $\{0,\dots,n-1\}$ lorsque t augmente. Si la loi de J_t était la loi uniforme sur l'ensemble des permutations de $\{0,\dots,n-1\}$, alors la loi de chaque coordonnée $J_t(k)$ serait la loi uniforme sur $\{0,\dots,n-1\}$, càd que l'on aurait, pour tous k,i,

$$\mathbb{P}(J_t(k)=i) \;=\; \frac{1}{n}.$$

On va alors mesurer le caractère uniforme de la distribution de J_t avec l'indicateur

$$I(J_t) := \max\{|n\mathbb{P}(J_t(k)=i)-1|\,;\,(k,i)\in\{1,\dots,n-1\}^2\}$$

Calculer, pour $n=52, T=10$, une estimation $\hat{I}(J_t)$ de $I(J_t)$ pour $t\in\{0,\dots,T\}$, afficher le graphe de $\hat{I}(J_t)$ en fonction de t et donner le t minimum pour avoir $\hat{I}(J_t)<1$ (si un tel t est trouvé).
Indication : Pour estimer $\mathbb{P}(J_t(k)=i)$, on simulera un grand nombre de trajectoires et on pourra utiliser, par exemple, la fonction `np.bincount` : pour `v` un vecteur d'entiers de $\{0,\dots,n-1\}$, `np.bincount(v,minlength=n)`=$[b_0,\dots,b_{n-1}]$ où pour tout k, b_k est le nombre d'apparitions de k dans `v`.

4. Pour J_{unif} une permutation aléatoire uniforme de `J0=range(n)`, on a $I(J_{\text{unif}})=0$. Néanmoins, ce à quoi il est légitime de comparer les estimations $\hat{I}(J_t)$ n'est pas $I(J_{\text{unif}})$ mais l'estimation de $\hat{I}(J_{\text{unif}})$ obtenue avec un échantillon de même taille de permutations aléatoires uniformes indépendantes. Dans le même programme, calculer cette estimation $\hat{I}(J_{\text{unif}})$ et faire apparaître, sur le graphique précédent, une ligne horizontale d'ordonnée $\hat{I}(J_{\text{unif}})$.
Indication : Pour simuler des permutations aléatoires uniformes indépendantes, on

utilise `np.random.shuffle`, comme par exemple dans le programme suivant, donné ici car l'usage de cette fonction n'est pas absolument évident.

```
import numpy as np
n=5
p=10
x=range(n)
Samples=[]
for i in xrange(p):
    np.random.shuffle(x)
    y=x[:]
    Samples.append(y)
print(Samples)
```

Bibliographie :

P. Diaconis. *The cutoff phenomenon in finite Markov chains*. Proc. Natl. Acad. Sci. USA. Vol 93, pp 1659–1664, 1996.

D. Bayer, P. Diaconis. *Trailing the dovetail shuffle to its lair*. Annals of Applied Probability. Vol 2(2), pp. 294–313, 1992.

4 Lois continues

Dans ces exercices, on se propose d'identifier la loi d'une variable aléatoire X en traçant l'histogramme d'un grand nombre, $X_1, \ldots, X_n$, de réalisations indépendantes de X et en le comparant à la densité supposée de X. On rappelle qu'un *histogramme* d'un échantillon est un diagramme en colonnes exprimant la répartition des valeurs de cet échantillon dans divers intervalles (la renormalisation est faite de façon à ce que l'aire totale vaille 1). Commande Python :

```
plt.hist(mon-echantillon,bins=mon-nombre-de-colonnes,normed=1)
```

On peut aussi remplacer le paramètre `mon-nombre-de-colonnes` par le vecteur des abscisses (ordonnées dans l'ordre croissant) des bases des colonnes.

a) Soient X et Y des variables aléatoires de loi uniforme sur $[-1, 1]$ indépendantes. Illustrer, via un histogramme, le fait que $X + Y$ a pour densité

$$\rho(x) := \frac{1}{4} \max\{2 - |x|, 0\}.$$

Indication : utiliser `np.random.rand` pour simuler X et Y.

b) Soient X et Y des variables aléatoires de loi exponentielle de paramètre 1, indépendantes. Illustrer, via un histogramme, le fait que $X+Y$ a pour densité $\mathbb{1}_{x\geq 0}xe^{-x}$.
Indication : utiliser `np.random.exponential`

c) Illustrer la propriété *d'absence de mémoire* des lois exponentielles : si X suit une loi exponentielle, alors pour tout $t>0$, la loi de $X-t$ sachant que $X>t$ est la loi de X (c'est la raison pour laquelle on utilise ces lois pour modéliser les durées de vie de composants *sans usure*).
Indication : pour `X` vecteur de type `numpy`, `X[X>t]` est le vecteur `X` dans lequel on n'a gardé que les coordonnées `>t`.

5 Simulation par la méthode de rejet

La *loi de Wigner* est la loi de support $[-2,2]$ et de densité $\frac{1}{2\pi}\sqrt{4-x^2}$. Simuler un grand nombre de variables aléatoires de loi de Wigner, représenter l'histogramme associé et le comparer à la densité.
Indication : Pour simuler une variable aléatoire de densité f de support $[a,b]$ et telle que $0\leq f\leq M$, on peut utiliser une suite $(U_n,V_n)_{n\geq 1}$ de couples indépendants de variables aléatoires telles que pour tout n, U_n,V_n sont indépendantes et de lois uniformes sur respectivement $[a,b]$, $[0,M]$, poser

$$\tau=\min\{n\geq 1\,;\,V_n\leq f(U_n)\},$$

et la variable aléatoire U_τ suit alors la loi de densité f. C'est la méthode de *simulation par rejet*. Notons que plus M est petit, plus cette méthode est rapide, on a donc intérêt à choisir $M=\|f\|_\infty$.

6 Méthode de Monte-Carlo

a) Calculer une estimation de la probabilité de l'événement $\sin(X)>1/2$ pour X de loi $\mathcal{N}(0,1)$. On utilisera la méthode de Monte-Carlo (càd que l'on simulera un grand nombre variables aléatoires indépendantes de loi $\mathcal{N}(0,1)$ et on calculera la proportion de cas où l'événement est réalisé).
Indication : si `x` est un vecteur, `y=(np.sin(x)>1/2)` désigne le vecteur de booléens dont les coordonnées valent `True` ou `False` selon que les coordonnées correspondantes de `np.sin(x)` soient ou non $>1/2$. Par ailleurs, les vecteurs de booléens se convertissent naturellement en vecteurs de 0 et de 1 si on leur applique une fonction dont les arguments sont des nombres (ex : `np.mean`).

b) Soit $(X_i)_{i\geq 1}$ des variables aléatoires indépendantes de loi uniforme sur $[0,1]$. Estimer n_0, la plus petite valeur de l'entier n telle que

$$\mathbb{P}(X_1 + \cdots + X_n < 5) \ \leq \ 0.2$$

Indication : l'inégalité de Markov (cf cours) nous permet d'affirmer que $n_0 \leq 20$. On pourra donc simuler, pour $m \gg 1$, $m \times 20$ variables aléatoires, utiliser les fonctions `np.cumsum(., axis=1)` et `np.sum(., axis=0)`, ainsi que la fonction `np.argwhere`.

7 Lois discrètes

Un calcul simple montre que lorsque n tend vers l'infini, la loi de binomiale $B(n, \lambda/n)$ tend vers la loi de Poisson de paramètre λ. En pratique, on assimile $B(n,p)$ à la loi de Poisson de paramètre np dès que $np^2 < 0.1$. Illustrer cette *proximité de lois* en affichant, sur le même graphique, leurs histogrammes en bâtons (sans faire aucune simulation).
Indication : les fonctions `sps.poisson.pmf` et `sps.binom.pmf` donnent les probabilités associées aux différentes valeurs que peuvent prendre des variables aléatoires de loi de Poisson et de loi binomiale, la fonction `plt.stem(x, y)` affiche des barres verticales d'abs. `x` et hauteur `y`.

8 Python et Data Mining

Python n'est pas uniquement destiné uniquement à traiter, en circuit fermé, des données générées par l'utilisateur (même si cela a beaucoup d'intérêt, y compris du point de vue des sciences expérimentales). Il permet aussi le traitement de données d'origine extérieure. L'exercice suivant en est l'illustration élémentaire. On y explore un fichier de données (espérances de vie et autres statistiques pour 40 pays différents) dont on tire divers graphiques. L'analyse statistique de ces données, qui nécessiterait, idéalement, une base de données plus ample et des connaissances théoriques plus avancées, n'est pas abordée.

a) Télécharger et mettre dans un même répertoire, sur la page du cours [2], le fichier de données `LifeExpectancy.csv` et le programme `ProgammeDataMiningEleves.py`. Executer le programme (sans chercher à en comprendre toutes les lignes).

b) Dans ce programme, on convertit le fichier de données extérieur `LifeExpectancy.csv` en une liste (de listes) `mylist`. Cette liste est affichée comme un tableau bi-dimensionnel dont chaque ligne représente un pays. Les colonnes correspondent à diverses statistiques. Identifier ces statistiques en lisant la première ligne du tableau (en anglais, *physician=médecin*).

2. `http://www.cmap.polytechnique.fr/~benaych/aleatoire_index.html`

c) Dans le même programme, la partie numérique du tableau bi-dimensionnel `mylist` est converti en un tableau de type `numpy` appelé `mynparray`, qui est utilisé pour afficher un graphique de comparaison de données. En procédant de même, prolonger le programme en comparant, via ce même type de graphiques, d'autres des données contenues dans `mynparray`.

d) Afficher l'histogramme des espérances de vie dans ces 40 pays (qui correspondent à la première colonne de `mynparray`).