Mémo - Python 3

Emeline LUIRARD

Préambule :

- Python possède une aide consultable en tapant `help()` dans la console. Plus généralement pour obtenir une aide concernant une fonction `bidule` du module `machin`, tapez `help(machin.bidule)`. On peut aussi ouvrir la fenêtre de l'aide interactive, dans le menu Outils.

Votre devise doit être : **coder compact et simple !** C'est comme cela que vos codes seront les plus lisibles et que vous ferez le moins d'erreurs.

1 Prise en main

1.1 Fenêtre de commande, éditeur, aide en ligne

Lorsqu'on lance le logiciel Pyzo, la fenêtre est découpée en divers cadres avec notamment :

- une fenêtre de commande (interpréteur) : où l'on peut exécuter des commandes : >>> indique que le logiciel attend vos instructions. L'interpréteur peut être utilisé comme une calculatrice : vous entrez un calcul, vous tapez sur la touche "entrée" et Python vous donne le résultat.
- une fenêtre d'éditeur de texte : où l'on peut regrouper les commandes et les sauvegarder dans un `fichier.py`.

On peut, dans la fenêtre de Python copier et coller des lignes à l'aide de la souris (y compris les exemples donnés par l'aide) ; on peut aussi naviguer dans l'historique des commandes au moyen des flèches haut et bas, cela évite de retaper les commandes si l'on exécute une fonction plusieurs fois de suite par exemple.

Python est un logiciel de calcul numérique et non de calcul formel : il effectue des calculs numériques approchés et non des calculs exacts. Par défaut, les réels sont affichés avec 15 chiffres significatifs.

N.B. *A la différence de Scilab, Python ne code pas tout en matrices. Il a ce qu'on appelle une programmation orientée objet.*

1.2 Programmation

Dès que l'on souhaite exécuter plus de deux ou trois tâches consécutives, i.e. dès que l'on sort du mode d'utilisation "supercalculatrice" de Python, on utilise l'éditeur de texte. On enregistre le fichier obtenu avec le suffixe `.py` puis on exécute le fichier dans Pyzo, au moyen du menu `Exécuter...` ou par un raccourci clavier. Pour exécuter seulement un bout de code, vous pouvez le séléctionner et cliquer sur `Exécuter la sélection`.

Pour faciliter la mise au point et la relecture du programme par vous-même et sa compréhension par le jury, il faut le commenter.

Une ligne de commentaires commence par `#`. De manière générale, Python ignore tout ce qui, sur une ligne, suit la commande `#`.

Si vous voulez afficher une variable, vous devrez utiliser `print(variable)`.

2 Syntaxe de base

2.1 Modules

```
from nom_du_module1 import *
#importe tous les élémens du module1
```

ou

```
import module2 as mod2


from module3 import sous_module as smod
```

Pour la deuxième méthode, les fonctions du module2 devront être appelées par `mod2.fonction()`. On l'utilise pour ne pas qu'il y ait d'écrasement d'autres fonctions, pour ne pas encombrer l'espace des noms. Pour les sous modules, on fera `smod.fonction()`.
Voici les principaux modules :

- `math` : pour importer les fonctions mathématiques usuelles et certaines constantes usuelles comme π (`pi`).
- `cmath` : pour les complexes : (`1j` pour i).
- `numpy` : pour utiliser le type `array` (tableau dont les éléments sont tous du même type, pratique pour les vecteurs, les matrices).
- `scipy` : outils nécessaires au calcul matriciel. Il contient un sous-module qui nous servira pour la partie aléatoire.
- `matplotlib` : pour générer des graphiques.

`numpy, scipy` et `matplotlib` sont à installer à l'aide de la commande `python3 -m pip install -user scipy` dans le terminal linux.

Le début de chaque script commencera donc comme ceci :

```
from math import *
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stat
```

2.2 Opérations usuelles

`x+=1`	remplace x par $x+1$ (valable aussi pour -,*,/)
`x**a`	x^a
`x,y=17,42`	pour affecter deux variables en même temps, toutes les expressions sont évaluées *avant* la première affectation, ex : `x,y=x+2,x`
`x,y=y,x`	du même type que le précédent, pour échanger deux variables
`a\%b`	Reste de la division euclidienne de a par b
`scipy.special.binom(n,k)`	Coefficient binomial $\binom{n}{k}$
`factorial(n)`	$n!$

2.3 Fonctions

Elles seront très utiles pour itérer un bout de code en faisant varier des paramètres. La syntaxe est la suivante

```
def nom_de_la_fonction(parametres):
        """ Details de ce que fait la fonction"""
        calculs
        return valeur_sortie
```

Dès que Python trouve un `return truc`, il renvoie `truc` et abandonne ensuite l'exécution de la fonction.


Les variables définies à l'intérieur d'une fonction sont locales : une fois le code de la fonction effectuée, elles n'existent plus. De manière générale, on évitera de donner des noms de variables identiques aux variables locales et aux variables globales.


On donnera des noms longs et explicites aux variables globales.

N.B. *On peut mettre des paramètres pas défaut, un nombre de paramètres arbitraire, etc. Si besoin, voir [CBCC16, p. 42] et/ou [Vig18, p. 14].*

2.4 Opérateurs logiques et booléens

Les opérations logiques de bases sont le "et" logique (`and`), le "ou" logique (`or`) et la négation (`not`). Les valeurs logiques (ou booléens) sont `True` et `False`. Voici divers opérateurs de comparaison :

`x==y`	x est égal à y
`x!=y`	x est différent de y
`x<=y`	$x \leq y$
`x in y`	x appartient à y (pour les listes)

Pour demander des conditions plus complexes, on peut combiner ces opérateurs de comparaison avec les booléens, par exemple, `(x==y) or (x>5)`. Si on veut transformer une liste de valeurs logiques en $0, 1$, on multiplie par 1 : par exemple, `1*[True, False]` va renvoyer `[1,0]`.

2.5 Boucles

Les syntaxes des boucles `for`, `while` et des instructions conditionnelles sont les suivantes :

```
i, a = 0,2
while (i<5) and (a<10):
        a=a**2
        i+=1
```

```
for i in range(10):
        a[i]=1/(i+1)
        a[i]=1/(i+1)
```

```
if i==j:
        a[i,j]=2
elif i<j:
        a[i,j]=1
else:
        a[i,j]=0
```


Vérifier impérativement que les boucles `while` s'arrêtent !

On veillera à bien indenter les codes dans l'éditeur : en Python, il n'y a pas de `end`, c'est l'indentation qui indique lorsque la boucle est fermée.

2.6 Listes

Une liste est une séquence d'éléments rangés dans un certain ordre, elle peut contenir des objets de types différents, par exemple, `liste=[42,'reponse']`, et sa taille peut varier au cours du temps.


En Python, on commence à compter à 0.

Pour créer une liste, connaître sa longueur et accéder à ses éléments :

`liste=[]`	crée une liste vide
`len(liste)`	renvoie la longueur de la liste
`liste[i]`	extrait l'élément à l'indice i de la liste (en comptant à partir de 0)
`liste[-1]`	extrait le dernier élément
`liste[2:4]`	extrait une sous liste

> "Il faut penser les indices comme repérant non pas les tranches de saucisson, mais les coups d'opinel qui ont permis de couper les tranches."

Les "intervalles" dans Python sont fermés à gauche et ouverts à droite : `liste[2:4]` extrait les éléments de l'indice 2 à 3 de la liste.
Pour créer des listes, Python est sympa, grâce aux listes en compréhension :
`[x**2 for x in range(10)]` retourne la liste `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`.
Enfin, on peut transformer une liste en ensemble (i.e. pas de doublons) par `set(liste)`.

Modification de listes :

`ma_liste[1]=4`	remplace l'élèment à l'indice 1 par 4
`liste1+liste2`	concatène deux listes
`ma_liste.append(5)`	ajoute un élément à la fin de la liste
`ma_liste.extend(L)`	ajoute en fin de liste les éléments de L
`ma_liste.insert(i,x)`	insère un élément x en position i
`ma_liste.remove(x)`	supprime la première occurence de x
`ma_liste.sort()`	trie la liste
...	

Taper `help(list)` pour obtenir toutes les "méthodes" associées. (Une méthode est une fonction qui ne s'applique qu'à un type d'objet, ici les `list`.)

Autres opérations sur les listes :

`ma_liste.count(a)`	compte le nombre de a
`ma_liste.index(a)`	premier indice de a dans la liste


On utilise assez souvent la fonction `range(start, stop[, step])` qui renvoie les entiers de *start* à *stop*. Attention, cependant, ce n'est pas une liste, c'est ce qu'on appelle un "itérable" : un curseur qui se déplace, tous ses entiers ne sont pas stockés. Si on veut une liste, on pourra utiliser `list(range(...))` ou `[range(...)]`.



Attention quand on copie des listes :

```
>>> liste=[42, 'reponse']
>>> copie=liste
>>> liste[1]=17
>>> liste, copie
[42,17], [42,17]
>>> copie[0]=0
>>> liste, copie
[0,17], [0,17]
```

Les variables `liste` et `copie` pointent vers le même objet. On dit qu'elles font "référence" au même objet. Pour faire une copie, on peut utiliser `copie=liste[:]`, mais attention si la liste contient des sous-listes !

2.7 Array : vecteurs, matrices

Un objet de type `array` est un tableau dont tous les éléments sont du même type. La différence avec les `list` : sa taille n'est pas modifiable une fois créé. Il servira à représenter les vecteurs et les matrices. Il est dans le module `numpy`. Dans la suite, je noterai M pour une matrice, V pour un vecteur, A pour un array quelconque.

```
M=np.array([[0,1],[2,3]]) # matrice
V=np.array([0,1,4,9])# vecteur ligne
```

Dans le module `scipy`, on trouve le sous-module `linalg` qui permet de faire du calcul matriciel.

On peut passer d'une liste à un array par `np.array(ma_liste)`. Inversement, pour transformer un array en liste, on fait `list(mon_array)`.

Voici quelques commandes utiles pour manipuler les `array` :

Pour créer un `array`, connaître sa longueur et accéder à ses éléments :

`np.eye(n)`	matrice identité de taille n
`np.ones((n,m))`	matrice 1 de taille $n \times m$
`np.zeros((n,m))`	matrice nulle de taille $n \times m$
`np.zeros_like(a)`	matrice nulle de même taille que a
`np.empty((n,m))`	matrice vide de taille $n \times m$
`np.arange(debut, fin(non inclus), pas)`	le tableau des entiers de *debut* à fin avec un *pas*
`np.linspace(min, max, nb de points)`	points réguliers $[min, ..., max]$
`np.diag(V)`	matrice avec en diagonale V
`np.diag(V,1)`	matrice avec en sur-diagonale V
`scipy.linalg.block_diag(D1,D2)`	matrice avec les blocs diagonaux D1 et D2
`A.shape`	renvoie (nb de lignes, nb de colonnes)
`len(V)`	renvoie la longueur du vecteur
`A.reshape([n,m])`	redimensionne le tableau A en n lignes et m colonnes. Si l'une des valeurs vaut -1, `numpy` la trouve tout seul.
`V.reshape([nbLign,1])` ou `V[:,np.newaxis]`	transforme un vecteur ligne en colonne
`V[-1]`	dernier élément du vecteur V
`V[start:end:step]`	pour extraire une sous-liste
`M[i,j]`	élément à l'indice $(i_{ligne}, j_{colonne})$

Modification et opérations sur les array :

Faire des opérations sur des `array` de tailles différentes, s'appelle du broadcasting. Cela fait les opérations terme à terme :

`A+10`	ajoute terme à terme
`M+V`	ajoute v à chaque ligne de M
`A*A`	multiplication terme à terme
`V*M`	répète v selon l'autre dimension pour avoir la même taille que M avant de multiplier terme à terme
`1/A`	inverse terme à terme
`A**n`	puissance terme à terme

La fonction puissance est à créer soi-même :

```
def puissance(mat,exp):
        m=mat
```

```
    for i in range(1,exp):
            mat=np.dot(mat,m)
    return mat
```

N.B. *De manière générale, quand on n'est pas sûr du résultat que donne un code (ex : opération terme à terme ou non), on teste sur un exemple.*

Les opérations usuelles sont les suivantes :

`A=A.astype(np.float64)`	change le type des éléments de A en flottants
`A.dot(B)`	multiplication matricielle : matrice/matrice, matrice/vecteur, vecteur/vecteur (produit scalaire)
`np.cross(U,V)`	produit vectoriel de U et V
`np.linalg.inv(M)`	inverse matriciel
`M.T`	transposée
`np.linalg.det(M)`	déterminant

`eigVal,eigVec=np.linalg.eig(M)`
 	
`eigVal,eigVec=np.linalg.eigh(M)`
	valeurs propres (avec multiplicité) et vecteurs propres à droite,
	en colonne, normalisés et dans le même ordre.
	pour les matrices symétriques, hermitiennes

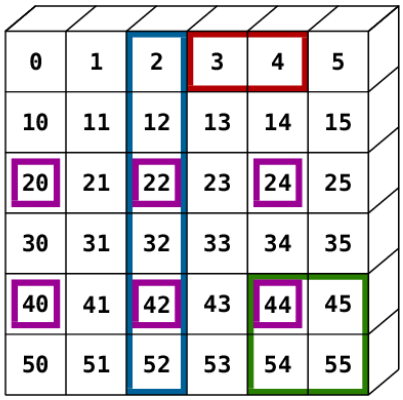
`V[0:3]=5`	remplace les 3 premiers éléments de V par des 5.
`A>0`	renvoie un `array` de même taille que A avec des booléens.
`A[A>0]`	renvoie un `array` avec les éléments de A qui vérifient la condition Ici ça fait une copie et non une référence. Pour les conditions, on utilise `&` pour "et" et `|` pour "ou"
`np.sort(A)`	renvoie un array avec les éléments de A triés

`np.sum(A)`
`np.sum(A,axis=0)`
`np.sum(A,axis=1)`
	somme des éléments de A
	pour sommer selon les colonnes,
	selon les lignes

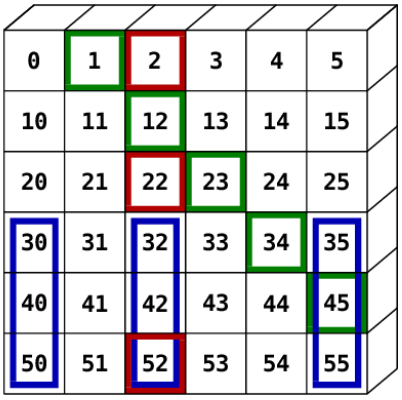
`np.cumsum(A)`	somme cumulative des éléments de A, on peut encore utiliser `axis`
`np.mean(A)` `np.cumsum(V)/(np.arange(1,n+1))`	moyenne des éléments de A, on peut encore utiliser `axis` Cela ferait un "np.cummean"
`np.std(A)`	écart-type des éléments de A, on peut encore utiliser `axis`
`np.max(A)` `np.maximum(U,V)`	renvoie le maximum d'un array, on peut encore utiliser `axis` renvoie un array `[max(U[i], V[i]), for i in range(len(U))]`

Si on cherche quelque chose en particulier, utiliser `np.lookfor('create array')`. Deux images assez explicites, pour l'extraction de listes, tirées de https://scipy-lectures.org :

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```




```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



2.8 Importation de données

Les commandes de base sont les suivantes :

```
import csv
with open('/home/dossier/blabla.csv', newline='') as f:
lignes=[ligne for ligne in csv.reader(f)]
```

`lignes` contient une liste avec les lignes du fichier, sous forme de chaînes de caractères.

3 Illustrations graphiques

3.1 Tracés en dimension deux

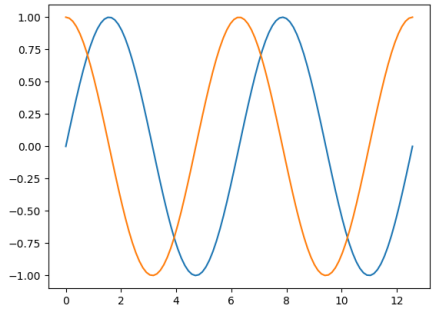
Pour tracer des graphes, nous avons besoin du sous-module `pyplot` de `matplotlib`. Ensuite, on définit la figure, qu'on appelle en général `fig`. Puis dans la figure on définit les sous-figures, selon les axes. On va utiliser la fonction `plot` qui prend en arguments des `array`.

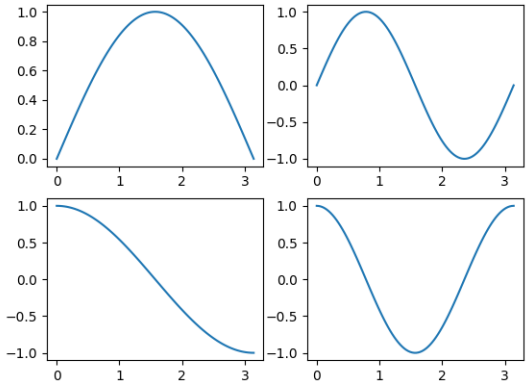
```
fig, axs=plt.subplots(n,m) # on aura des sous-figures sur n lignes et m colonnes
#sans argument quand on n'en veut qu'un.
axs[i,j].plot(x,y,"r", label="nom fonction") # trace dans le graphique (i,j),
#en couleur rouge, et associe un nom à la courbe
```

```
fig.suptitle("titre principal")
axs[0,0].set_title("titre graphe 1")
axs[0,1].set_title(r"$math latex$") # le 'r' dit à Python de ne pas interpreter
#les caracteres speciaux
#Ne pas oublier:
ax.legend() # affiche les légendes de type "label"
plt.show() # pour ouvrir la fenetre graphique
plt.clf() #efface la fenêtre graphique
```

Si on veut représenter plusieurs courbes sur le même graphique, on fait des appels à la fonction `plot` les uns à la suite des autres, ou on lui donne les différentes courbes en arguments :

```
x=np.linspace(0,4*pi,100)
y=np.sin(x)
z=np.cos(x)
fig, ax=plt.subplots()
ax.plot(x,y)
ax.plot(x,z)
#ou
ax.plot(x,y, x,z)
plt.show()
```





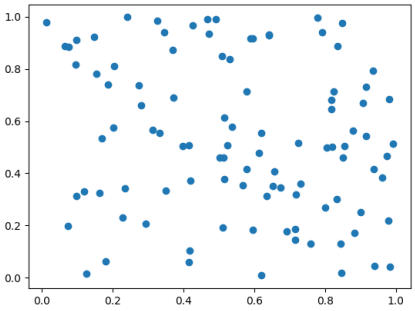
N.B. *Pour tracer une fonction en escalier, on utilisera l'argument de* `plot` *:* `drawstyle='steps-post'` *ou* `drawstyle='steps-pre'`.

Options graphiques

Lorsque on présente une illustration graphique, on doit préciser ce qu'elle représente i.e. qu'est ce qui est tracé, en fonction de quoi, quels sont les paramètres etc. Pour faciliter la compréhension et la lisibilité, on doit donc "décorer" en ajoutant un titre général, un également pour chaque axe. Lorsqu'il y a plusieurs courbes, une légende permet d'afficher la correspondance entre les couleurs/motifs et les courbes représentées.

Le troisième argument de `plot` est le style du trait. On mettra "o" pour que les points d'une courbe ne soient pas reliés ou pour avoir un nuage de points. Voici un exemple :

```
x=stats.uniform.rvs(size=(2,100))
fig, ax=plt.subplots()
ax.plot(x[0,:],x[1,:], 'o')
plt.show()
```



On change les couleurs avec : "r" pour red, "b" pour blue, "k" pour black ; le style avec "." pour un petit point, "o-" pour des gros points et une ligne, " :" pour des pointillés. Et on peut mixer le tout ! Par exemple : "ko".
D'autres commandes :

`axs[..].set_ylabel("titre ordonnees")`	Donne un titre aux ordonnées
`axs[..].set_ylim([a,b])`	limite le graphe aux ordonnées $[a, b]$.
`ax[..].set_xticks(rray)`	graduation axe des abscisses
`ax[..].set_sticklabels(["0", "1",...])`	Noms des graduations de l'axe des abscisses
`fig.tight_layout()`	Pour gérer l'espacement des boîtes graphiques

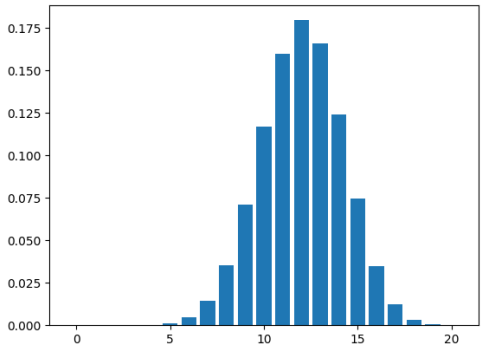
Pour écrire en LaTeX dans les titres et faire apparaître les arguments :

```
ax.plot(x,y,label=r"$\lambda :$"+str(lamb))
```

Diagramme en bâtons : histogrammes de lois discrètes

On utilise `ax.bar(x,y,width=)` à la place de `plot`. Attention, x contient les emplacements des bâtons et y contient leur hauteur (pour nous, ce sera donc les probabilités empririques). Voici un exemple :

```
x=stats.binom.pmf(np.arange(0,21),20,0.6)
fig, ax=plt.subplots()
ax.bar(np.arange(0,21),x)
plt.show()
```



Histogramme

L'histogramme d'un échantillon est un diagramme constitué de barres verticales juxtaposées, chacune de ces barres représentant le nombre de termes de l'échantillon appartenant à une classe donnée. Pour représenter un histogramme d'un échantillon d'une loi réelle, on commence donc par répartir cet échantillon en classes, chacune de ces classes correspondant aux valeurs appartenant

à un certain intervalle de $\mathbb{R}$, et on représente cette classe par un rectangle vertical dont l'aire est proportionnelle à l'effectif de la classe. On normalise souvent ces aires de façon à ce que l'aire totale soit égale à 1.
`plt.hist(X,bins=n)` répartit les données de X dans n sous intervalles de $[min(X), max(X)]$. `hist` prend la place de `plot`, on garde la même syntaxe que précédemment pour faire différents graphes par exemple. Mais pour mettre plusieurs histogrammes sur le même on fait `plt.hist([X1,X2,X3], bins=n, label=...)`.

N.B. *Avec `a=plt.hist(X,bins=n)`, on récupère en premier la hauteur des bâtons et en deuxième le découpage des sous-intervalles fait par Python.*

N.B. *On peut donner ses propres sous-intervalles dans un `array` dans l'argument `bins`. Si les intervalles sont les $([a_i, a_{i+1}])_{i\in[0,n-1]}$, alors on fera `bins=[a_0, a_1, a_2,..., a_n]`.*

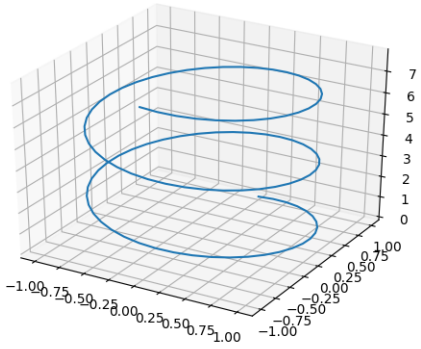
- On peut faire des histogrammes de lois discrètes avec `hist` en lui donnant pour argument `bins` les intervalles centrés en nos valeurs discrètes : Pour reprendre l'exemple de la loi binomiale, on donnera `bins=np.arange(0,22)-0.5`

- Pour superposer avec une densité, il faut demander à Python de normaliser les histogrammes, ça se fait avec l'option de `hist`, `density=True`. Puis on plote la densité.

3.2 Tracés en dimension trois

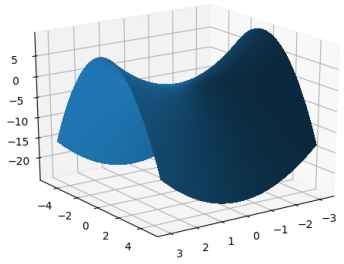
Pour tracer une courbe paramétrée en dimension 3, il nous faut procéder un peu différemment pour créer la figure. On utilisera toujours la fonction `plot` mais avec trois arguments : les projections sur chacun des axes :

```
#import est nécessaire même si Axes3D
#n'apparait pas ensuite.
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')
t = np.linspace(0, 5 * np.pi, 100)
r = np.sin(t)
x = np.cos(t)
y = t/2
ax.plot(x, y, z)
ax.legend()
```



La commande de base pour représenter une surface est `plot_surface`. La surface d'équation $z = f(x, y)$ est obtenue en entrant les matrices X et Y de coordonnées et une matrice Z de même taille telle que $Z[i, j] = f(X[i, j], Y[i, j])$. X et Y forment une grille. Voici un exemple :

```
#toujours aussi important de l'importer !
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')
X = np.arange(-pi, pi, 0.2)
Y = np.arange(-5, 5, 0.2)
#matrices avec l'ensemble des coordonnées
#de la grille
X, Y = np.meshgrid(X, Y)
Z = X**2 - Y**2
surf = ax.plot_surface(X, Y, Z, antialiased=False)
plt.show()
```



Pour toutes les options de style et de couleur, on se reportera à l'aide.

3.3 Rappels sur les illustrations de convergence

Pour illuster une convergence en loi, deux choix s'offrent à vous :

1. Tracer un histogramme et le comparer à la densité de la loi limite. Lorsque la loi limite n'est pas absolument continue (i.e. n'a pas de densité), on met en avant le diagramme en barres avec les $\mathbb{P}_X(k)$.
2. Tracer plusieurs fonctions de répartitions et illustrer que cela converge vers la fonction de répartition de la loi limite.

Attention à ne pas dire "on voit que ça converge en loi vers.." ! Comme lorsque vous êtiez au collège, vous n'aviez pas le droit de dire "le triangle est rectangle ça se voit sur le dessin". Il vous faut quantifier cette "proximité", pour cela on utilise les tests d'adéquations (du χ^2 ou de Kolmogorov-Smirnov). Si vous n'avez pas le temps de les coder pendant la préparation, parlez-en au moins à l'oral.

Pour ce qui est de la convergence presque-sûre, on trace plusieurs trajectoires.

Attention à bien garder le même aléa tout au long d'une seule trajectoire

4 Simulation des variables aléatoires

Pour tirer un réel aléatoire dans $[0,1]$, on peut utiliser la fonction `np.random.random(size=)`. Pour le reste, on utilisera le sous-module `stats` de `scipy`.

4.1 Lois usuelles

Les commandes sont les suivantes :

- `rvs` : simulation
- `pdf(x, parametre)` : probability density function i.e. densité en x
- `pmf(k, parametre)` : probability mass function i.e. densité discrête en k
- `cdf(x,parametre)` : cumulative density function i.e. fonction de répartition en x
- `ppf(q, parametre)` : percent point function i.e. q-ième quantile.

On les utilisera comme ceci : `stats.norm.rvs(parametres)`. Les paramètres sont (paramètres de forme, loc=μ, scale=σ, size=taille échantillon).

Les paramètres μ et σ sont tels que, si l'appel de `stats.xxx.rvs()` renvoie une v.a X, alors `stats.xxx.rvs(loc=mu,scale=sigma)` renvoie une v.a ayant la même loi que $\sigma X + \mu$. En tapant `stats.expon?` ou `help(stats.expon)`, on a la normalisation choisie pour la loi exponentielle par exemple.

Seuls les paramètres de forme sont obligatoires.

loi(paramètres de forme)	Lois
`norm()`	Loi normale $\mathcal{N}(0,1)$.
`gamma(a=)`	Loi gamma $\gamma(a,1)$.
`beta(a=, b=)`	Loi béta $\beta(a,b)$.
`t(df=n)`	Loi Student de n degrés de liberté.
`cauchy()`	Loi de Cauchy.
`expon(scale=1/lambda)`	Loi exponentielle $\mathcal{E}(\lambda)$.
`uniform(loc=, scale=)`	Loi uniforme $\mathcal{U}(]loc, loc+scale[)$.
`randint(low, high)`	Loi uniforme $\mathcal{U}(\{low, low+1, \cdots, high-1\})$.
`binom(n=, p=)`	Loi binomiale $\mathcal{B}(n,p)$.
`geom(p=)`	Loi géométrique $\mathcal{G}(p)$.
`poisson(mu=)`	Loi de Poisson $\mathcal{P}(\mu)$.

N.B. *Toutes ces lois, ainsi que les relations entre elles sont rappelées dans [Vig18][p. 70].*

N.B. *On peut aussi demander à Python de calculer les moments d'une variable aléatoire. Pour le moment d'ordre n :* `stats.xxx.moment(n,parametres, loc=, scale=)`.

4.2 Autres techniques de simulation

4.2.1 Inversion d'une fonction de répartition

Lorsqu'on veut simuler une variable aléatoire dont la loi n'est pas classique, on ne peut malheureusement pas utiliser le sous-module `stats`. Mais, si on connaît sa fonction de répartition F, c'est gagné !

Proposition 1. *Soit X une variable aléatoire réelle de fonction de répartition F. Pour $u \in [0,1]$, on désigne par $F^{-1}(u) := \inf\{x \in \mathbb{R},\ F(x) \geq u\}$ l'inverse généralisée de la fonction de répartition F. Si $U \sim \mathcal{U}([0,1])$ alors $F^{-1}(U)$ a pour fonction de répartition F.*

4.2.2 Méthode de rejet

Proposition 2. *Soient A, $D \in \mathcal{B}(\mathbb{R}^d)$ tels que $A \subset D$. Soit $(X_i)_{i\in\mathbb{N}}$ une suite i.i.d. de variables aléatoires définies sur un espace de probabilité $(\Omega, \mathcal{F}, \mathbb{P})$ uniformes sur D. Introduisons le temps $\tau := \inf\{i \in \mathbb{N}^*,\ X_i \in A\}$, alors X_τ est uniformément distribuée dans A, i.e. pour $B \in \mathcal{B}(\mathbb{R}^d)$, $B \subset A$, $\mathbb{P}(X_\tau \in B) = |B|/|A|$. Le nombre de tirages avant l'obtention d'une réalisation de la loi uniforme sur A par ce procédé suit une loi géométrique $\mathcal{G}(p)$, avec $p = |A|/|D|$. En particulier le nombre moyen de tirages nécessaires est $|D|/|A|$.*

De la proposition, on déduit la méthode de simulation suivante :
Soit X une variable aléatoire réelle de loi μ_X possèdant une densité continue f à support compact inclus dans $[a,b] \subset \mathbb{R}$ et telle que son graphe soit inclus dans $[a,b] \times [0,M]$, pour un certain $M \in \mathbb{R}^+$. On considère une suite $(Z_i)_{i\in\mathbb{N}} = (X_i, Y_i)_{i\in\mathbb{N}}$ de variables aléatoires uniformes dans $[a,b] \times [0,M]$ et l'on définit $\tau = \inf\{i \in \mathbb{N},\ f(X_i) > Y_i\}$. Alors X_τ a pour loi μ_X.

N.B. *Plus généralement, si f est une fonction positive, à support compact inclus dans $[a,b] \subset \mathbb{R}$ et telle que son graphe soit inclus dans $[a,b] \times [0,M]$, pour un certain $M \in \mathbb{R}^+$. Alors X_τ a pour densité* $\dfrac{f}{\int f}$.

```
def rejet():
        while True:
                X=stats.uniform.rvs(loc=a, scale= b-a)
                Y=stats.uniform.rvs(loc=0, scale= M)
                if Y<f(X):
                        return X
```

On peut généraliser la méthode de rejet de la façon suivante. Soit X une variable aléatoire réelle de loi μ_X qui possède une densité continue f_X (peut ne pas être à support compact) majorée uniformément par une densité g, i.e. il existe une constante $C \geq 1$ telle que

$$\forall x \in \mathbb{R},\ f_X(x) \leq Cg(x), \quad \text{avec} \quad \int g(y)dy = 1.$$

On suppose que l'on sait facilement simuler des variables aléatoires de loi de densité g. Soient donc $(X_i)_{i\in\mathbb{N}}$ une suite i.i.d. de variables aléatoires de loi de densité g et $(U_i)_{i\in\mathbb{N}}$ une suite i.i.d. de variables aléatoires uniformes sur l'intervalle $[0,1]$, indépendantes de $(X_i)_{i\in\mathbb{n}}$. Si on considère le temps $\tau := \inf\{i \geq 1,\ f(X_i) > Cg(X_i)U_i\}$, alors X_τ a pour loi μ_X.

4.2.3 Simulation de loi conditionnelle

Si A est un événement de probabilité strictement positive associé à une certaine expérience aléatoire et si on veut simuler sous la probabilité conditionnelle $\mathbb{P}(\cdot|A)$ une variable aléatoire dépendant de cette expérience, on effectue une suite de tirages successifs sous la probabilité initiale $\mathbb{P}$ et on conserve uniquement les tirages pour lesquels l'événement A est réalisé.

5 Chaînes de Markov

Par chaîne de Markov, on entendra ici chaîne de Markov discrête, homogène en temps, à espace d'états au plus dénombrable. Dans la suite (X_n) sera une chaîne de Markov à espace d'états E et de matrice de transtion P.

5.1 Simuler

5.1.1 A partir de sa matrice de transition

En Python, il n'y a pas (à ma connaissance) de fonction toute faite pour générer une chaîne de Markov à partir de sa matrice de transition. Mais ce n'est pas bien difficile et on retrouve la

méthode dans [Vig18][p. 98] :
On commence par créer la matrice de transition P dans un `array`. On rappelle que

$$\mathbb{P}(X_{n+1} = y | X_n = x) = P(x, y).$$

Ensuite on va utiliser la fonction `np.random.choice(a,p)` qui renvoie une variable aléatoire à valeurs dans a. p contient les probabilités associées. Ici, l'espace d'états E est en bijection avec $[\![0, Card(E) - 1]\!]$.

```
def markov_avec_P(n,P,x0):
"simule les n premiers pas d'une chaine de Markov "
"à partir de sa matrice de transition"
        X=np.zeros(n,dtype=np.int) #on indique que X contiendra seulement des entiers.
        X[0]=x0
        for k in range(n-1):
                X[k+1]=np.random.choice(a=range(len(P)), p=P[X[k],:])
                # Les états sont numérotés de 0 à len(P)-1
        return X
```

N.B. *Cette méthode est à utiliser lorsque l'espace d'états E est petit et la matrice de transition P facilement implémentable en Python. Ainsi, pour simuler une marche aléatoire sur $\mathbb{Z}$, il va falloir procéder autrement.*

5.1.2 Avec la relation de récurrence

Une chaîne de Markov peut toujours s'écrire sous la forme $X_{n+1} = f(X_n, U_{n+1})$, pour $n \geq 0$, où f est une fonction mesurable à valeurs dans E et (U_n) est une suite v.a. i.i.d. et indépendante de X_0 (donc des $(X_k)_{k<n}$).

N.B. *Lorsque la fonction f dépend du temps n, on a devant nous une chaîne de Markov non-homogène en temps. La construction de proche en proche à l'aide de la relation de récurrence s'appliquera donc aussi à ces chaînes de Markov.*

5.2 Mise en évidence des propriétés

Là où l'algèbre linéaire et les probabilités se rencontrent...

5.2.1 Etats recurrents et transients

Pour mettre en évidence, à l'aide de simulations, les états récurrents et transitoires d'une chaîne à espace d'états fini, on peut par exemple simuler une longue trajectoire de la chaîne et représenter la proportion de temps passé dans chaque état. Cette proportion sera négligeable pour les états transitoires, contrairement aux états récurrents.

5.2.2 Mesure/probabilité invariante

On rappelle qu'une mesure π est invariante pour la chaîne de Markov $(X_n)_n$ de matrice de transition P si

$$\forall y \in E, \sum_x \pi(x) P(x, y) = \pi(y),$$

ou encore $\pi P = \pi$ en prenant π sous la forme d'un vecteur ligne. π est donc un vecteur propre à droite de P associé à la valeur propre 1, c'est à dire aussi, un vecteur propre à gauche de ${}^tP$ pour la valeur propre 1. La commande `eigVal,eigVec=np.linalg.eig(P.T)` sera donc très utile afin de trouver une mesure invariante. Et la commande `np.real` permettra d'avoir le résultat sous forme de réels.

```
val_pr, vec_pr=np.linalg.eig(P.T)
val_pr=np.real(val_pr)
vec_pr=np.real(vec_pr)
pi=vec_pr[:,0] # attention le vecteur n'est pas de norme 1
pi/=np.sum(pi)
#pi contient la probabilité invariante
```

La probabilité invariante d'une chaîne de Markov irréductible se retrouve également à l'aide du théorème suivant. En effet, rappelons que sur un espace d'états fini, lorsque qu'une chaîne de Markov est irréductible, elle est automatiquement récurrente positive et admet ainsi une unique probabilité invariante.

Théorème 1. *On note X^x la chaîne qui part du point x, π son unique probabilité invariante et on introduit*

$$T_n^x(y) = \frac{1}{n}\sum_{k=0}^{n-1} \mathbb{1}_{\{X_k^x = y\}}.$$

Alors $(T_n^x(y))_n$ converge quand n tend vers l'infini vers $\pi(y)$.

```
X=markov_avec_P(t,P,x_0)
pi_approche=np.zeros(3)
for i in range(len(P)):
        pi_approche[i]=np.mean(X==i)
```

Lorsque la chaîne est, en plus, apériodique, on sait qu'elle converge en loi vers la mesure invariante : `pi_approche2=puissance(P,500)[0,:]`

6 La méthode de Monte-Carlo

La méthode de Monte-Carlo est une méthode de calcul approché d'intégrales, basée sur la loi des grands nombres. Elle permet de calculer des valeurs approchées d'intégrales, d'espérances, de probabilités, en utilisant des réalisations i.i.d. d'une loi que l'on sait simuler.
Ainsi, si $f : [a,b]^d \to \mathbb{R}$ est une fonction intégrable par rapport à la mesure de Lebesgue sur $[a,b]^d$ et que l'on souhaite évaluer l'intégrale

$$I(f) := \int_{[a,b]^d} f(x)\,\mathrm{d}x,$$

on commence par se donner une suite $(X_n)_{n\geq 1}$ de variables aléatoires i.i.d. de loi uniforme sur $[a,b]^d$. D'après la LGN, l'estimateur

$$I_n(f) := \frac{1}{n}\left(f(X_1) + \ldots + f(X_n)\right)$$

convergence presque sûrement et dans $\mathbb{L}^1$ vers la limite $\mathbb{E}[f(X_1)] = \int_{[a,b]^d} f(x) \frac{1}{(b-a)^d} \, \mathrm{d}x = \frac{1}{(b-a)^d} I(f)$. Pour n grand, la somme $(b-a)^d I_n(f)$ fournit ainsi une approximation de $I(f)$.

Quand on intégre sur un intervalle de mesure différente de 1, on n'oublie pas de renormaliser :

$$(b-a)^d \mathbb{E}[f(X_1)] = \int_{[a,b]^d} f(x) \, \mathrm{d}x.$$

La vitesse de convergence de cette méthode est donnée par le TCL : elle est de l'ordre de $\sqrt{n}$. Elle est lente par rapport à des méthodes déterministes. Cependant cette vitesse ne dépend pas de la régularité de l'intégrande f et dépend plus faiblement de la dimension d que les méthodes déterministes.

Pour contrôler l'erreur commise dans l'approximation, on fournira un intervalle de confiance. Le TCL en donne un si on sait majorer la variance de $f(X_1)$.

N.B. *En fonction de l'intégrale à calculer, on peut utiliser d'autres lois que la loi uniforme. Exemple :* $\int_{\mathbb{R}} \sin(x) e^{-x^2} \, \mathrm{d}x$.

6.1 Performance algorithmique

D'un point de vue pratique, si l'on veut utiliser la méthode Monte-Carlo pour estimer une intégrale/espérance du type $\mathbb{E}[f(X)]$, où X est un vecteur aléatoire dans $\mathbb{R}^d$ et $f : \mathbb{R}^d \to \mathbb{R}$ est une fonction mesurable raisonnable, il est nécessaire de pouvoir simuler informatiquement un $n-$échantillon $(X_1, \ldots, X_n)$ de même loi que X, ce qui nécessite un nombre d'opération de l'ordre de $O(nd)$. Le calcul de la moyenne empirique des $f(X_i)_{1 \leq i \leq n}$ est du même ordre. Au final, la mise en oeuvre de la méthode de Monte-Carlo implique un nombre de l'ordre de $O(nd)$ opérations. Si l'on note σ^2 la variance $\sigma^2 := \mathrm{var}(X)$ que l'on supposera finie, l'erreur d'approximation commise dans la méthode de Monte-Carlo est

$$\varepsilon_n := \mathbb{E}[f(X)] - \frac{1}{n} \sum_{i=1}^{n} f(X_i).$$

D'après le théorème central limite, si $(x_1, \ldots, x_n)$ est une réalisation de $(X_1, \ldots, X_n)$, l'intervalle ci-dessous est un intervalle de confiance asymptotique de niveau 95% pour $\mathbb{E}[f(X)]$:

$$\left[\frac{1}{n} \sum_{i=1}^{n} f(x_i) - \frac{1.96 \, \sigma}{\sqrt{n}}, \; \frac{1}{n} \sum_{i=1}^{n} f(x_i) + \frac{1.96 \, \sigma}{\sqrt{n}} \right].$$

La vitesse de convergence de l'algorithme est donc de l'ordre de $\sqrt{n}/\sigma$ pour un coût de $O(nd)$ opérations. Cela signifie que pour un algorithme comptant n opérations élémentaires, la précision est de l'ordre de $\sigma\sqrt{d/n}$ ce qui implique que

1. la méthode de Monte-Carlo est sans intérêt (si ce n'est pédagogique) pour le calcul d'intégrales en petite dimension ou pour le calcul d'intégrales de fonctions régulières.
2. il est crucial de minimiser la variance σ. Si l'intégrale $\mathbb{E}[f(X)]$ admet plusieurs représentations de type $\mathbb{E}[g(Y)]$, on aura tout intérêt à choisir celle qui est associée à la variance minimale.

   Ainsi, on peut considérer d'autres lois que la loi uniforme sur un pavé...

La variance σ^2 est naturellement donnée par $\sigma^2 = \mathbb{E}[f^2(X)] - \mathbb{E}[f(X)]^2$. L'objectif étant de déterminer une estimation de $\mathbb{E}[f(X)]$, quantité supposée inconnue, il paraît vraisemblable que le calcul explicite de la variance soit, dans les cas pertinents, impossible. La stratégie est alors de remplacer la variance théorique par son estimateur empirique

$$\sigma_n^2 := \frac{1}{n}\sum_{k=1}^{n}\left(f(X_k) - \frac{1}{n}\sum_{i=1}^{n} f(X_i)\right)^2.$$

D'après le lemme de Slutsky, si $(x_1, \ldots, x_n)$ est une réalisation de $(X_1, \ldots, X_n)$, alors

$$\left[\frac{1}{n}\sum_{i=1}^{n} f(x_i) - \frac{1.96\,\sigma_n}{\sqrt{n}},\ \frac{1}{n}\sum_{i=1}^{n} f(x_i) + \frac{1.96\,\sigma_n}{\sqrt{n}}\right].$$

est encore un intervalle asymptotique au niveau 95% de $\mathbb{E}[f(X)]$.

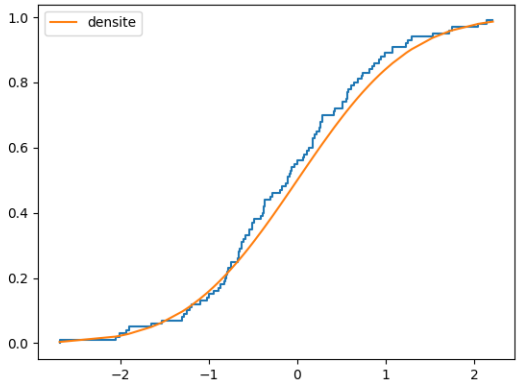
7 Fonction de répartition empirique

La fonction de répartion empirique d'un n-échantillon $(X_1, \ldots, X_n)$ est la fonction croissante $F_n : \mathbb{R} \to [0, 1]$ définie par

$$F_n(y) = \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{X_k \leq y}.$$

Si on dispose d'un échantillon de taille n, on peut tracer le graphe de la fonction de répartition empirique de cet échantillon en ordonnant préalablement cet échantillon. L'exemple suivant tire un échantillon de taille 100 d'une loi normale, trace le graphe de sa fonction de répartition empirique et la compare à la fonction de répartition de la loi normale.

```
n=100
#un échantillon
X=stats.norm.rvs(loc=0,scale=1, size=n)
fig,ax=plt.subplots()
#on le classe:
X_sort=np.sort(X)
ax.plot(X_sort,np.arange(n)/n,drawstyle='steps-pre')
ax.plot(X_sort,stats.norm.cdf(X_sort), label='densite')
ax.legend()
plt.show()
```



8 Tests statistiques : tests du χ^2 et test de Kolmogorov-Smirnov.

8.1 Rappels sur les tests

Pour construire un test, il faut tout d'abord définir deux hypothèses :

— L'hypothèse nulle H_0, celle qu'on pense être vraie en général. Elle est la plus précise possible.

— L'hypothèse alternative H_1. On prend souvent le complémentaire de H_0.

Ensuite, on construit une statistique D, qui est une fonction de notre échantillon qui va vérifier :

— Sous H_0, D suit (asymptotiquement) une loi de fonction de répartition connue

— Sous H_1, D est (asymptotiquement) grand avec une grande probabilité.

Il va ensuite falloir définir une région de rejet pour construire la règle de décision :

$$D \in R \Rightarrow \text{on rejette } H_0$$
$$D \notin R \Rightarrow \text{on ne rejette pas } H_0$$

Pour trouver cette région, on définit le niveau du test $\alpha = \mathbb{P}_{H_0}(H_0 \text{ rejeté})$. Il doit être le plus petit possible. Alors $\alpha = \mathbb{P}_{H_0}(H_0 \text{ rejeté}) = \mathbb{P}_{H_0}(D \in R)$.

8.2 Les tests du χ^2

8.2.1 Test d'adéquation à une loi de probabilité sur un ensemble fini

Ce test a pour but de décider si un vecteur d'observations est ou non une réalisation d'un échantillon de variables aléatoires de loi donnée. Cette dernière loi doit être à valeurs dans un ensemble **fini**. Soit $(x_1, \cdots, x_n)$ une réalisation d'un vecteur aléatoire $(X_1, \cdots, X_n)$ i.i.d. de loi inconnue $p = (p_1, \cdots, p_k)$, une probabilité sur $[\![1, k]\!]$. On note, pour $i \in [\![1, k]\!]$, $N_i(n) = Card\{j \in [\![1, n]\!], X_j = i\}$.

On suppose par ailleurs donnée une loi $p^0 = (p_1^0, \ldots, p_k^0)$. On souhaite tester l'hypothèse nulle $H_0 = \{p = p^0\}$ contre l'hypothèse alternative $H_1 = \{p \neq p^0\}$. On définit alors la statistique

$$D_n = D_n(p^0, X_1, \cdots, X_n) := n \times \sum_{i=1}^{k} \frac{(N_i(n)/n - p_i^0)^2}{p_i^0} = \sum_{i=1}^{k} \frac{(N_i(n) - np_i^0)^2}{np_i^0},$$

dont le comportement asymptotique est le suivant :

Théorème 2.

$$D_n = \sum_{i=1}^{k} \frac{(N_i(n) - np_i^0)^2}{np_i^0} \begin{cases} \overset{\mathcal{L}}{\Longrightarrow} \chi^2(k-1) & \text{sous } H_0, \\ \underset{n\to+\infty}{\overset{p.s.}{\longrightarrow}} +\infty & \text{sous } H_1. \end{cases}$$

La commande `D,p_valeur=scipy.stats.chisquare(f_obs, f_exp)` calcule la statistique D_n et calcule une p-valeur à partir du nombre d'occurrences observées `f_obs` et attendues `f_exp`.

Étant donné un niveau α (souvent $\alpha = 5\%$) et un réel η_α tel que $\mathbb{P}(\chi^2 > \eta_\alpha) = \alpha$, la zone de rejet $W_n = \{D_n > \eta_\alpha\}$ fournit alors un test de niveau asymptotique α pour $H_0 = \{p = p^0\}$ contre $H_1 = \{p \neq p^0\}$.

Etant donnés une mesure de probabilité p^0 de support fini A, un vecteur de données $(x_i)_{1\leq i\leq n} \in A^n$ et un seuil $0 < \alpha < 1$, la fonction suivante donne le résultat du test du χ^2 d'adéquation de niveau α.

```
def adequation_chi_deux(A,p,X,alpha):
        n=len(X)
        f=np.zeros_like(A)
        for i in range(len(A)):
                f[i]=(X==A[i]).sum() # pour compter le nombre d'occurences
        D,p_valeur=stats.chisquare(f,p*n)
        t= stats.chi2.ppf(q=1-alpha, df=len(A)-1)
        if D>t:
                return ("On rejette H0")
        else:
                return("On ne rejette pas H0")
```

N.B. *En pratique, on considère que l'approximation en loi par $\chi^2(k-1)$ est valide sous H_0 si $n \times \min_{1\leq j\leq k} p_j^0 \geq 5$. Si cette condition n'est pas satisfaite, on peut regrouper les classes à trop faibles effectifs afin d'atteindre ce seuil.*

N.B. *Lorsque l'on a affaire à des lois sur $\mathbb{N}$, $\mathbb{R}$,..., on peut tout de même utiliser le test du χ^2 en découpant l'espace en un nombre fini de classes.*

8.2.2 Test d'adéquation à une famille de lois

On peut aussi se demander si la loi de l'échantillon appartient ou non à une famille de lois $(p(\theta))_{\theta\in\Theta}$, $\Theta \subset \mathbb{R}^d$. On note $\hat{\theta}_n$ l'estimateur du maximum de vraisemblance de θ. On va alors tester l'hypothèse nulle $H_0 = \{p \in \{p(\theta), \theta \in \Theta\}\}$ contre l'hypothèse alternative $H_1 = \{p \notin \{p(\theta), \theta \in \Theta\}\}$. On définit la statistique

$$D'_n = D'_n(p, X_1, \cdots, X_n) := \sum_{i=1}^{k} \frac{(N_i(n) - np_i(\hat{\theta}_n)^2}{np_i(\hat{\theta}_n)},$$

dont le comportement asymptotique est le suivant :

Théorème 3.

$$D'_n = \sum_{i=1}^{k} \frac{(N_i(n) - np_i(\hat{\theta}_n)^2}{np_i(\hat{\theta}_n)} \begin{cases} \overset{\mathcal{L}}{\Longrightarrow} \chi^2(k-d-1) & \text{sous } H_0, \\ \underset{n\to+\infty}{\overset{p.s.}{\longrightarrow}} +\infty & \text{sous } H_1. \end{cases}$$

8.2.3 Test d'indépendance

On dispose d'un échantillon d'une loi $Z = (X, Y)$ et l'on souhaite déterminer si les variables X et Y sont indépendantes. Considérons donc n données $(z_1, \dots, z_n) = ((x_1, y_1), \dots, (x_n, y_n))$ dont on suppose qu'elles sont les réalisations indépendantes et identiquement distribuées de variables aléatoires $(Z_1, \dots, Z_n) = ((X_1, Y_1), \dots, (X_n, Y_n))$ à valeurs dans des ensembles finis $[\![1, r]\!]$, $[\![1, s]\!]$. On note $p = (p_{ij}, 1 \leq i \leq r, 1 \leq j \leq s)$ la loi du couple $Z = (X, Y)$, c'est-à-dire :

$$p_{ij} = \mathbb{P}(Z = (i, j)) = \mathbb{P}(X = i, Y = j).$$

On introduit

$$N_{ij} = Card\{k, X_k = i, Y_k = j\}, \quad N_{i.} = N_{i1} + \dots + N_{is}, \quad N_{.j} = N_{1j} + \dots + N_{rj}.$$

Alors, avec l'hypothèse $H_0 = \{X$ et Y sont indépendants$\}$ et $H_1 = H_0^C$,

Théorème 4.

$$D_n = \sum_{i=1}^{r} \sum_{j=1}^{s} \frac{(N_{ij} - \frac{Ni.N_{.j}}{n})^2}{\frac{Ni.N_{.j}}{n}} \begin{cases} \overset{\mathcal{L}}{\Longrightarrow} \chi^2((r-1)(s-1)) & sous\ H_0, \\ \underset{n \to +\infty}{\overset{p.s.}{\longrightarrow}} +\infty & sous\ H_1. \end{cases}$$

La commande `D, p_valeur, dlib, expected=scipy.stats.chi2_contingency(f_obs)` calcule la statistique D_n et la p-valeur, à partir d'un tableau à deux entrées contenant les nombres d'occurrences observées pour chaque coordonnée. Cela retourne également le degré de liberté et le nombre d'occurrences attendues.

À nouveau, étant donnés un niveau α et un réel η_α tel que $\mathbb{P}(\chi^2 \geq \eta_\alpha) = \alpha$, la zone de rejet $W_n = \{D_n > \eta_\alpha\}$ fournit un test de niveau asymptotique α de $H_0 = \{X$ et Y indépendantes$\}$ contre $H_1 = \{X$ et Y non indépendantes$\}$.
Etant donnés un vecteur $(x_i, y_i)_{1 \leq i \leq n}$ et un seuil $0 < \alpha < 1$. Le programme suivant donne le résultat du test du χ^2 d'indépendance de niveau α.

```
def matrice_adequation(X,Y,A,B):
        r=len(A)
        s=len(B)
        M=np.ones((s,r))
        for i in range(r):
                for j in range(s):
                        M[i,j]=((X==A[i])&(Y==B[j])).sum()
        return M

def adequation_chi_deux(X,Y,alpha,A,B):
        n=len(X)
        r=len(A)
        s=len(B)
        f_obs=matrice_adequation(X,Y,A,B)
        D, p_valeur, dlib, expected=stats.chi2_contingency(f_obs)
        t= stats.chi2.ppf(q=1-alpha, df=(r-1)*(s-1))
        if D>t:
                return ("On rejette l'indépendance")
        else:
                return("On ne rejette pas l'indépendance")
```

8.2.4 Test d'homogénéité

On dispose de ℓ échantillons différents $E_1, \cdots, E_\ell$ à valeurs dans $[\![1, k]\!]$. On se demande si ces échantillons ont la même loi. On note

$$O_{ij} = Card\{x \in E_j, x = i\} \;\; O_{i.} = O_{i1} + \cdots + O_{i\ell}, \;\; O_{.j} = O_{1j} + \cdots + O_{kj},$$

et $n = \sum_{i=1}^{k} \sum_{j=1}^{\ell} O_{ij}$. Alors, avec l'hypothèse $H_0 = \{$les échantillons sont issus de la même loi$\}$ et $H_1 = H_0^C$,

Théorème 5.

$$D_n = \sum_{i=1}^{k} \sum_{j=1}^{\ell} \frac{(O_{ij} - \frac{Oi.O_{.j}}{n})^2}{\frac{Oi.O_{.j}}{n}} \begin{cases} \overset{\mathcal{L}}{\Longrightarrow} \chi^2((k-1)(\ell-1)) & sous\ H_0, \\ \underset{n\to+\infty}{\overset{p.s.}{\longrightarrow}} +\infty & sous\ H_1. \end{cases}$$

La commande `D, p_valeur=scipy.stats.friedmanchisquare(echant1, echant2, echant3,...)` calcule la statistique D_n et p-valeur, à partir des échantillons.

8.3 Tests non paramétriques

Dans toute cette section μ désigne une mesure de probabilité sur $\mathbb{R}$ et F la fonction de répartition associée. On considère $(X_1, \ldots, X_n)$ un $n-$échantillon de loi μ et on note $(X_{(1)}, \ldots, X_{(n)})$ la statistique d'ordre associée.

8.3.1 Theorèmes de Glivenko-Cantelli et Kolmogorov-Smirnov

La fonction de répartition empirique F_n associée à l'échantillon $(X_1, \ldots, X_n)$ est définie pour tout $x \in \mathbb{R}$ par

$$F_n(x) := \frac{Card\{1 \le k \le n, X_k \le x\}}{n},$$

ou encore

$$F_n(x) = \frac{k}{n} \;\text{ si }\; X_{(k)} \le x < X_{(k+1)}.$$

Le théorème de Glivenko–Cantelli assure que $||F_n - F||_\infty$ tend presque sûrement vers zéro lorsque n tend vers l'infini, d'autre part, le théorème de Kolmogorov-Smirnov assure que, si F est continue, $\sqrt{n}\,||F_n - F||_\infty$ converge en loi lorsque n tend vers l'infini vers une variable K dont la loi est appelée loi de Kolmogorov :

$$\lim_{n\to+\infty} \mathbb{P}\left(\sqrt{n}\,||F_n - F||_\infty \le x\right) = \mathbb{P}(K \le x) = 1 - 2\sum_{k=1}^{+\infty} (-1)^{k-1} e^{-2k^2x^2}.$$

8.3.2 Test d'adéquation de Kolmogorov–Smirnov

Grâce au théorème de Kolmogorov–Smirnov, on peut facilement mettre en oeuvre un test pour déterminer si un vecteur de données est ou non une réalisation d'un échantillon de loi prescrite. Si la loi en question a pour fonction de répartition F, on calcule la fonction de répartition empirique F_n associée aux données ainsi que la statistique

$$D_n = \sqrt{n}\,||F_n - F||_\infty\,.$$

À un seuil $0 < \alpha < 1$, on associe le nombre c_α tel que $\mathbb{P}(K \ge c_\alpha) = \alpha$. On a par exemple

α	0.10	0.05	0.025	0.01	0.005	0.001
c_α	1.22	1.36	1.48	1.63	1.73	1.95

La zone de rejet $W_n = \{D_n > c_\alpha\}$ fournit alors un test de niveau asymptotique α de l'hypothèse $H_0 = \{$les données sont des réalisation i.i.d. de loi de fonction de répartition $F\}$ et $H_1 = H_0^C$,

Théorème 6.

$$D_n = \sqrt{n}\,||F_n - F||_\infty \begin{cases} \xRightarrow{\mathcal{L}} \mu_{KS} & \text{sous } H_0, \\ \xrightarrow[n\to+\infty]{p.s.} +\infty & \text{sous } H_1. \end{cases}$$

La commande `D, p_valeur=scipy.stats.kstest(echant, loi)` calcule la statistique D_n et la p-valeur, à partir de l'échantillon. `loi` peut être le nom d'une loi dans `scipy.stats` ou alors le nom d'une fonction, que vous aurez créée, donnant la fonction de répartition F. Par exemple `D, p_valeur=stats.kstest(Y, 'expon')`

8.3.3 Comparaison d'échantillon, test de Smirnov

Soient $(X_1, \dots, X_n)$ et $(Y_1, \dots, Y_m)$ deux échantillons et F_n et G_m les fonctions de répartitions empiriques associées.
On cherche à tester l'hypothèse $H_0 = \{$Les deux échantillons proviennent d'une même loi continue$\}$ contre l'hypothèse alternative $H_1 = H_0^C$. Pour cela, on considère la statistique

$$D_{n,m} := \sqrt{\frac{mn}{m+n}} \times \sup_{x\in\mathbb{R}} |F_n(x) - G_m(x)|\,.$$

Théorème 7.

$$D_{n,m} := \sqrt{\frac{mn}{m+n}} \times \sup_{x\in\mathbb{R}} |F_n(x) - G_m(x)| \begin{cases} \xRightarrow{\mathcal{L}} \mu_{KS} & \text{sous } H_0, \\ \xrightarrow[n\to+\infty]{p.s.} +\infty & \text{sous } H_1. \end{cases}$$

La zone de rejet associée au test est du type $\{D_{m,n} \geq c_\alpha\}$ où $\mathbb{P}(K \geq c_\alpha) = \alpha$.
La commande `D, p_valeur=scipy.stats.ks_2samp(echant1, echant2)` calcule la statistique $D_{n,m}$ et la p-valeur, à partir des deux échantillons.

Références

[CBCC16] Alexandre Casamayou-Boucau, Pascal Chauvin, and Guillaume Connan. *Programmation en Python pour les mathématiques - 2e éd.* Dunod, Paris, 2e édition edition, January 2016.

[Vig18] Vincent Vigon. *python proba stat.* Independently published, October 2018.