
# TP Probabilités et Statistiques en `Python`

L3 Informatique

## Première partie

# Introduction à `Python`

`Python` est un langage portable, dynamique, extensible, gratuit, qui permet une approche modulaire et orientée objet de la programmation.

## 1 Généralités

5 séances de TP sont prévues en Probabilités et Statistiques sous `Windows` :
— séance 1 (1ère partie) : découverte de `Python 3`, en lien avec les probabilités et les statistiques,
— séances 2 à 4 (2ème partie) : analyse et probabilités, en lien avec les cours,
— séance 5 : examen sur machine.

## 2 Outils

Le cas d'utilisation le plus courant est, bien entendu, la simple invocation d'un script (programme écrit en `Python`) dans l'invite de commande :
`python hello.py`

### 2.1 `ipython`

`ipython` permet de coder en lignes de commande en `Python` directement dans le terminal (la console).

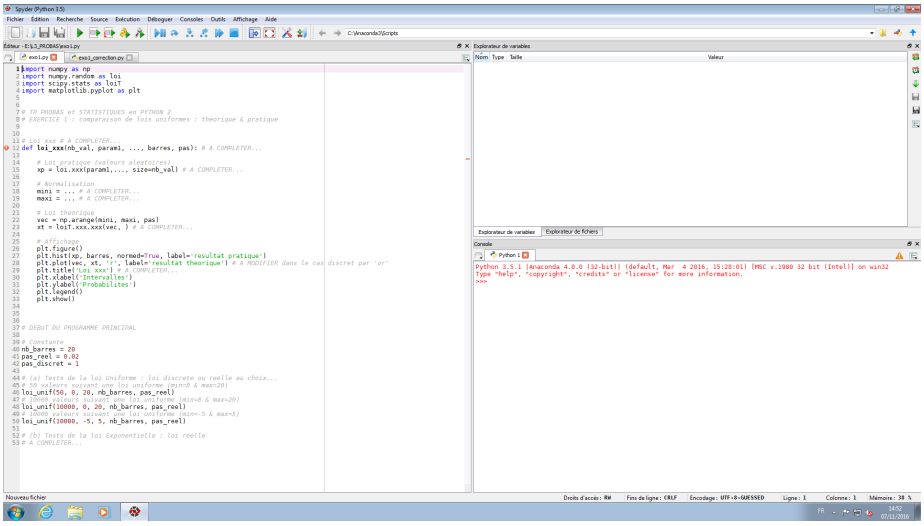
## 2.2 `spyder`

En TP, nous utiliserons l'IDE `spyder` [1] (cf. ci-dessous) :

— la partie à droite est l'**éditeur de fichier** : ici, il s'agit de l'exercice 1 que vous compléterez lors du TP2.

— la partie en bas à droite est l'**interpréteur interactif**. Nous l'utiliserons tout le long du TP1 afin de tester chaque commande.

— la partie en haut à droite est composée de différents éléments : Explorateur de variables, Explorateur de fichiers...



# 3 Matrices et vecteurs

`Numpy` est un package pour `Python` spécialisé dans la manipulation des tableaux (`array`), pour nous essentiellement les vecteurs et les matrices. Les tableaux `numpy` ne gèrent que les objets de même type. Le package propose un grand nombre de routines pour un accès rapide aux données (exemple : recherche, extraction), pour les manipulations diverses (exemple : tri), pour les calculs (exemple : calcul statistique). Les tableaux `numpy` sont plus performants (rapidité, gestion de la volumétrie) que les collections usuelles de `Python`.

## 3.1 Présentation

Il est nécessaire d'importer le package `Numpy` :

```
import numpy as np
```

1. `https://pypi.python.org/pypi/spyder`

Numpy ajoute le type `array` qui est similaire à une liste (`list`) avec la condition supplémentaire que tous les éléments sont du même type. Nous concernant ce sera donc un tableau d'entiers, de flottants voire de booléens.

Pour créer un tableau on convertit une liste via la commande `array`. Le deuxième argument est optionnel et spécifie le type des éléments du tableau.

```
a = np.array([1, 4, 5, 8], float)
```

Une matrice est un tableau (`array`) à 2 dimensions.

```
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

Représente la matrice $2 \times 3$ suivante :

$$\begin{pmatrix} 1.2 & 2.5 \\ 3.2 & 1.8 \\ 1.1 & 4.3 \end{pmatrix}$$

**Attention :** En `Python`, modifier une donnée d'une extraction d'un tableau entraîne aussi une modification du tableau initial. Si nécessaire la fonction $np.copy(a)$ ou $a.copy()$ permet de faire une copie d'un tableau $a$.

*Exemple :*

```
>>> a=np.array([1, 2, 3, 4, 5])
>>> c=np.array([1, 2, 3, 4, 5])
>>> b=a[1:3]
>>> b
array([2, 3])
>>> b[1]=0
>>> a
array([1, 2, 0, 4, 5])    # modification de "a"
>>> b=c[1:3].copy()       # methode 1
>>> b[1]=0
>>> bb=np.copy(c[1:3])    # methode 2
>>> bb[1]=0
>>> c
array([1, 2, 3, 4, 5])    # pas de modification de "c"
```

Comme pour les listes le slicing extrait les tableaux.

```
>>> a = np.arange(10)
>>> a # a[:] est equivalent
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [debut:fin+1:pas]
array([2, 5, 8])
>>> a[2:8:3] # le dernier element n'est pas inclus
array([2, 5])
>>> a[:5]    # le dernier element n'est pas inclus
array([0, 1, 2, 3, 4])
```

Dans l'instruction $[debut : fin + 1 : pas]$ les arguments peuvent être omis : par défaut l'indice $debut$ vaut 0 (le 1er élément), l'indice $fin$ est celui du dernier élément et le $pas$ vaut 1. Note : il y a donc une différence entre $[2 : 9]$ et $[2 :]$.

```
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])
```

Un pas négatif inversera l'ordre du tableau et les tableaux étant considérés cycliques la commande $a[-2]$ extrait l'avant dernier et le dernier élément.

```
>>> a[-2:]   # a[-2::1] est equivalent
array([8, 9])
```

Pour un tableau bi-dimensionnel on peut bien sûr jouer avec les deux indices.

```
>>> a=np.eye(5,5)
>>> print(a)
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
>>> a[:3,:4]=4
>>> a[::3,::4]=5
>>> a[:,3]=6
>>> print(a)
[[ 5.  4.  4.  6.  5.]
 [ 4.  4.  4.  6.  0.]
 [ 4.  4.  4.  6.  0.]
 [ 5.  0.  0.  6.  5.]
 [ 0.  0.  0.  6.  1.]]
```

Une autre méthode d'extraction consiste à écrire $a[b]$ où $b$ est un tableau d'entiers qui correspondra aux indices à extraire et $a$ un vecteur. Pour les matrices c'est $a[b, c]$ et on prend les indices de lignes dans $b$, les indices de colonnes dans $c$, successivement.

```
>>> a=np.array([2,4,6,8],float)
>>> b=np.array([0,0,1,3,2,1],int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
>>> a=a.reshape(2,2)
>>> b=np.array([0,0,1,1,0],int)
>>> c=np.array([0,1,1,1,1],int)
>>> a[b,c]
array([ 2.,  4.,  8.,  8.,  4.])
```

`Numpy/Scipy` proposent aussi le type `mat` comme matrice, exclusivement un tableau bi-dimensionnel. Comme les fonctions telles que *ones*, *eye* retournent un objet de type `array` nous n'utiliserons pas dans la suite le type `mat`. Il permet cependant de saisir les matrices à-la-Matlab et de faire le produit matriciel par le simple symbole $*$.

```
>>> a=np.mat('[1 2 4 ; 3 4 5.]')
>>> b=np.mat('[2. ; 4 ; 6]')
>>> print(a)
[[ 1.  2.  4.]
 [ 3.  4.  5.]]
>>> print(b)
[[ 2.]
 [ 4.]
 [ 6.]]
>>> print(a*b)
[[ 34.]
 [ 52.]]
```

Tout objet `array` est convertible en type `mat` et réciproquement (sous la condition que le tableau (`array`) soit uni,bi-dimensionnel).

```
>>> a=np.array([1, 2, 4])
>>> np.mat(a)
matrix([[1, 2, 4]])
```

### 3.1.1 Commande de matrices particulières

$np.zeros(n)$ : vecteur nul de taille $n$
$np.zeros((n,p))$ : matrice nulle de taille $n \times p$
$np.eye(n)$ : matrice de taille $n$ avec des $1$ sur la diagonale et des $0$ ailleurs
$np.eye((n,p))$ : matrice de taille $n \times p$ avec des $1$ sur la diagonale et des $0$ ailleurs
$np.ones(n)$ : vecteur de taille $n$ rempli de $1$
$np.ones((n,p))$ : matrice de taille $n \times p$ remplie de $1$
$np.diag(v)$ : matrice diagonale dont la diagonale est le vecteur $v$
$np.diag(v,k)$ : matrice dont la diagonale décalée de $k$ est le vecteur $v$ ($k$ est un entier relatif)
$np.random.rand(n)$ : vecteur de taille $n$ à coefficients aléatoires uniformes sur $[0,1]$
$np.random.rand((n,p))$ : matrice de taille $n \times p$ à coefficients aléatoires uniformes sur $[0,1]$
$np.transpose(A)$ : ou $A.T$ renvoie la transposée du vecteur (ou de la matrice) $A$

## 3.2 Opérations sur les vecteurs/matrices

L'opérateur $+$ additionne terme à terme deux tableaux de même dimension.
La commande $2. + a$ renvoie le vecteur/matrice dont tous les éléments sont ceux de $a$ plus 2.
Multiplier un vecteur/matrice par un scalaire se fait selon le même principe : la commande $2 * a$ renvoie le vecteur/matrice de même dimension dont tous les éléments ont été multipliés par 2.

L'opérateur $*$ ne fait que multiplier terme à terme deux tableaux de même dimension :

```
>>> 4*np.ones((4,4))*np.diag([2, 3, 4., 5])
array([[  8.,   0.,   0.,   0.],
       [  0.,  12.,   0.,   0.],
       [  0.,   0.,  16.,   0.],
       [  0.,   0.,   0.,  20.]])
```

Dans le même ordre $a**3$ renvoie le vecteur/matrice de même dimension dont tous les éléments sont ceux de $a$ élévés à la puissance 3.

Devinez ce que fait $1./a$ ?

Pour le produit matriciel (le vrai) la commande $np.dot$ est là :

```
>>> np.dot(4*np.ones((4,4)),np.diag([2, 3, 4., 5]))
array([[  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.]])
```

Si $v$ est un vecteur ($array$ uni-dimensionnel) et $A$ une matrice ($array$ bi-dimensionnel) alors $np.dot(A, v)$ renvoie le produit $Av$ tandis que $np.dot(v, A)$ renvoie $v^t A$. Si le produit n'est pas possible, `Python` le signale.

```
>>> a=np.arange(4).reshape(2,2)
>>> v=np.array([-3,2])
>>> np.dot(a,v)
array([2, 0])
>>> np.dot(v,a)
array([4, 3])
>>> w=np.concatenate((v,v)) # ou w=np.concatenate((v,v), axis=0)
>>> w
array([-3,  2, -3,  2])
>>> dot(a,w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
```

$np.vdot(v, w)$ : permet de calculer le produit scalaire des vecteurs $v$ et $w$.

## 3.3 Questions

### 3.3.1 Création de matrices

Mettons en application les fonctions `Python` de création de matrice.

**Question 1 :** Construire les vecteurs et matrices suivants :

$$[1\ 4\ 2\ 9\ 14\ 3\ 16], \begin{bmatrix} 6 \\ 2 \\ 15 \\ 8 \\ 24 \\ 7 \end{bmatrix}, \begin{bmatrix} 1 & 4 & 5 & 7 \\ 5 & 7 & 3 & 11 \\ 18 & 4 & 9 & 2 \end{bmatrix}$$

**Question 2 :** Construire un vecteur incrémental commençant par $3$ jusqu'à $20$ et par pas de $0.5$.

**Question 3 :** Construire les matrices suivantes :

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{bmatrix}, \begin{bmatrix} 2 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}$$

**Question 4 :** Construire 2 matrices aléatoires, notées $B$ et $C$, de dimensions $3 \times 5$.

### 3.3.2 Opérations sur les matrices

Effectuons quelques calculs :

**Question 5 :** Calculer le produit élément à élément de $B$ et $C$ et le produit matriciel de $B$ par la transposée de $C$. Ces matrices seront respectivement notées $F$ et $G$.

**Question 6 :** Lister tous les éléments de $F$ compris entre $0.2$ et $0.6$. Lister tous les éléments de $G$ supérieur ou égal à $0.6$.

# 4 Programmation

En `Python`, les tests, boucles, fonctions, suivent ce schéma :

```
'''(test/boucle/fonction)''' :
        # Code a l'interieur
# Fin d'indentation = sortie
```

La première ligne se termine par « : » et l'indentation doit être respectée.

## 4.1 Tests

***if... else*** : pour n'exécuter des lignes de code que si une condition est vérifiée.

```
if True :
        #...
else :
        #...
```

Variante : if...elif...else.

## 4.2 Boucles

***for*** : pour parcourir des éléments.

```
for k in range(deb, fin + 1, pas) :
        #...
```

$range(deb, fin + 1, pas) \rightarrow debdeb + pas...fin$. *deb* et *pas* sont optionnels. La fonction $range()$ n'est pas obligatoire dans les boucles $for$ : elle peut être remplacée par une liste... :

```
for k in [1, 0, 5] :
        #...
```

***while*** : pour exécuter des lignes de code tant que la condition est vérifiée.

```
while True :
        #...
```

## 4.3 Compréhension de listes

Ecriture plus rapide que les boucles : *sortie* = [*calcul boucle* (optionel : *condition*)]

```
liste = [x*2 for x in l if x > 0]
```

Ce code retourne la liste $l$ dont les éléments $> 0$ ont été multipliés par 2.

### 4.4 Fonctions

Les fonctions sont souvent nécessaires (pour rendre le code plus lisible, éviter la redondance...).
Exemple de syntaxe : écriture de la fonction *nomFonction*, prenant *arg1* et *arg2* en arguments d'entrée et retournant *x* et *y* en sortie :

```
def nomFonction(arg1, arg2) :
        # Contenu de la fonction
        return x, y

# Appel de la fonction
x, y = nomFonction(2, 'toto')
```

Les fonctions qui ne retournent rien n'ont pas besoin de « return ». Comme les boucles et les conditions, la fonction se temrine à la fin de l'indentation.

### 4.5 Imports

Si la fonction précédente est dans le fichier « Fonctions.py », il faut importer le fichier pour pouvoir l'utiliser dans un autre fichier :

```
import Fonctions
x, y = Fonctions.nomFonction(arg1, arg2)
```

## 5 Probabilités et statistiques

### 5.1 Outils Classiques

Il existe plusieurs fonctions pour calculer les paramètres d'une distribution $P$ de points :

- **mean** : $np.mean(P)$ moyenne de l'ensemble $P$. Si $P$ est une matrice, on peut spécifier que la moyenne soit réalisée sur les vecteurs lignes ou colonnes : $np.mean(P, axis = 0)$, $np.mean(P, axis = 1)$.
- **std** : $np.std(P)$ écarttype de l'ensemble $P$. Si $P$ est une matrice, on retrouve le même fonctionnement que pour $mean$.
- **var** : $np.var(P)$ variance de l'ensemble $P$. Si $P$ est une matrice, on retrouve le même fonctionnement que pour $mean$.
- **cov** : $np.cov(P)$ covariance de l'ensemble $P$. Si $P$ est une matrice de taille $m \times n$, chaque colonne correspond à une variable et chaque ligne correspond à une observation. On a donc $m$ observations de chacun des $n$ variables. La commande $np.cov(P)$ permet d'obtenir la matrice de covariance de $P$.

### 5.2 Lois de probabilités

Il existe plusieurs commandes pour générer des lois de probabilités.

— **uniform** : $X = np.random.uniform(a, b, (n, m))$ génère un vecteur ou une matrice $X$ de taille $n \times m$ contenant des nombres aléatoires suivant une loi uniforme sur $[a, b]$.
$X = np.random.uniform(a, b, size = n)$ génère un vecteur $X$ de taille $n$ contenant des nombres aléatoires suivant une loi uniforme sur $[a, b]$.

— **normal** : $X = np.random.normal(\mu, \sigma, (n, m))$ génère un vecteur ou une matrice $X$ de taille $n \times m$ contenant des nombres aléatoires suivant une loi normale de moyenne $\mu$ et d'écart type $\sigma$.

— **poisson** : $X = np.random.poisson(\lambda, (n, m))$ génère un vecteur ou une matrice $X$ de taille $n \times m$ contenant des nombres aléatoires suivant une loi de Poisson de paramètre $\lambda$.

— **exponential** : $X = np.random.exponential(\mu, (n, m))$ génère un vecteur ou une matrice $X$ de taille $n \times m$ contenant des nombres aléatoires suivant une loi de exponentielle de paramètre $\mu = \frac{1}{\lambda}$.

— **geometric** : $X = np.random.geometric(p, (n, m))$ génère un vecteur ou une matrice $X$ de taille $n \times m$ contenant des nombres aléatoires suivant une loi géométrique de paramètre $p$.

Il existe respectivement dans le module *scipy.stats* des fonctions qui permettent de déterminer la loi théorique : $uniform.pdf$, $norm.pdf$, $poisson.pmf$, $expon.pdf$, $geom.pmf$.

## 6 Représentation graphique

### 6.1 Prise en main du module `matplotlib`

Les représentations graphiques se font à l'aide du package `matplotlib`[2]. Au-delà de l'affichage simple avec les valeurs par défaut, de nombreuses propriétés permettent de contrôler le graphique : épaisseur du trait, couleurs, repères, grilles, textes, etc.

Pour tracer des courbes planes, on utilise la fonction $plot(y)$, ou $plot(y, x)$ pour placer chaque point de façon déterminée sur l'axe des abscisses.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,10,0.1)

# On utilise par exemple la fonction exponentielle de Numpy
y = np.exp(-x)
```

2. `http://matplotlib.org/`

```
# Affichage des points relies entre eux
plt.figure()
plt.plot(y, x)
plt.xlabel('axe des abcisses')
plt.ylabel('axe desordonne')
plt.show() # Affiche la courbe

# Affichage de chaque point avec des ronds rouges
plt.plot(y, x, 'ro')
plt.show() # Affiche la courbe des points
```

Pour tracer des surfaces 3D, il existe des fonctions comme $mplot3d$.
La fonction $hist$ permet de tracer des histogrammes.

```
nombres_gaussiens = np.random.randn(1000)

# Affichage de l'histogramme
plt.figure()
plt.hist(nombres_gaussiens)
plt.show()
```

Les points peuvent être représentés par différents symboles : $+, x, ., o.$.
Les couleurs peuvent être modifiés selon leurs abréviations : $g$ (vert), $r$ (rouge), $k$ (noir), $b$ (bleu), $c$ (cyan), $m$ (magenta), $y$ (jaune), $w$ (blanc).

On peut superposer plusieurs courbes sur un même graphe en appelant plusieurs fois la fonction $plot$ avant d'afficher la courbe avec la fonction $show$.

## 6.2 Représentation des lois statistiques

Les lois statistiques peuvent être différentiées par leur forme. Pour cela un affichage peut être nécessaire.
**Question 1 :** Représenter par une courbe la loi normale théorique $\mathcal{N}(2, 1)$ sur l'intervalle $[-1, 5]$.

**Question 2 :** Représenter sur un même graphe, sur l'intervalle $[0, 10]$, la loi normale théorique $\mathcal{N}(5, 3)$ et la loi exponentielle théorique de paramètre $\mu = 1$.

**Question 3 :** Dans une nouvelle figure, afficher l'histogramme d'une distribution aléatoire de $n = 1000$ points d'une loi normale $\mathcal{N}(5, 1)$. Comparez avec la loi théorique de la question précédente.

## Deuxième partie

# Analyse et probabilités

## Exercice 1 : lois de probabilités

### Objectifs :

— voir l'influence du nombre de valeurs et des paramètres des lois,
— comparer les résultats pratiques et théoriques en affichant les histogrammes de probabilités et les courbes théoriques.

1. Récupérez le fichier « exo1.py » qui se trouve sous Moodle.
2. Editez la fonction « loi_xxx », analysez chacune des parties et **comprendre son fonctionnement**.
3. En suivant le modèle de la fonction précédente (et en la complétant !), **créez 5 fonctions** afin de traiter les 5 lois suivantes... Pour chacune des lois, effectuez les tests suivants et **commentez les résultats**.

   (a) **Loi uniforme sur un intervalle** $[a, b]$
   — 50 valeurs suivant une loi uniforme : $a = 0$ et $b = 20$,
   — 10000 valeurs suivant une loi uniforme : $a = 0$ et $b = 20$,
   — 10000 valeurs suivant une loi uniforme : $a = -5$ et $b = -5$.

   (b) **Loi exponentielle de paramètre** $\lambda$
   — 50 valeurs suivant une loi exponentielle : $\lambda = 0.02$,
   — 10000 valeurs suivant une loi exponentielle : $\lambda = 0.02$,
   — 10000 valeurs suivant une loi exponentielle : $\lambda = 0.8$.

   (c) **Loi géométrique de paramètre** $p$
   — 50 valeurs suivant une loi géométrique : $p = 0.07$,
   — 10000 valeurs suivant une loi géométrique : $p = 0.07$,
   — 10000 valeurs suivant une loi géométrique : $p = 0.2$.

   (d) **Loi de Poisson de paramètre** $\lambda$
   — 50 valeurs suivant une loi de Poisson : $\lambda = 5$,
   — 10000 valeurs suivant une loi de Poisson : $\lambda = 5$,
   — 10000 valeurs suivant une loi de Poisson : $\lambda = 0.5$,
   — 10000 valeurs suivant une loi de Poisson : $\lambda = 50$.

   (e) **Loi Normale d'espérance** $\mu$ **et d'écart-type** $\sigma$
   — 50 valeurs suivant une loi Normale : $\mu = 0$ et $\sigma = 1$,
   — 10000 valeurs suivant une loi Normale : $\mu = 0$ et $\sigma = 1$,
   — 10000 valeurs suivant une loi Normale : $\mu = 5$ et $\sigma = 0.5$,
   — 10000 valeurs suivant une loi Normale : $\mu = 50$ et $\sigma = 500$.

## Exercice 2 : espérance et variance

1. Prenez 3 échantillons $X_1$, $X_2$ et $X_3$ de loi uniforme dans l'intervalle $[10, 20]$. La taille des échantillons est respectivement de 1000, 10000 et 100000 valeurs.
2. Calculez l'espérance et la variance de ces 3 échantillons.
3. Comparez avec les valeurs théoriques et **commentez les résultats**.
4. Effectuez les traitements précédents (cf. questions 2 et 3) pour :
   (a) 3 échantillons de loi normale de paramètres $\mu = 0$ et $\sigma = 1$,
   (b) 3 échantillons d'une loi exponentielle avec $\lambda = 0.5$.

## Exercice 3 : matrice de covariance

1. Prenez un échantillon $X$ de loi normale de taille 1000 et de paramètres $(0, 1)$.
2. Prenez un échantillon $Y$ de loi uniforme de taille 1000 dans l'intervalle $[10, 20]$.
3. Prenez un échantillon $Z$ de loi uniforme de taille 1000 dans l'intervalle $[0, 1]$.
4. Calculez les matrices de covariances de :
   (a) $X$ et $Y$,
   (b) $X$ et $Z$,
   (c) $Y$ et $Z$,
5. **Commentez le contenu de chacune des matrices de covariances.**
6. Pourquoi les valeurs de covariances sont faibles ?

## Exercice 4 : coefficient de corrélation

On considère les variables aléatoires suivantes (définies à partir des variables $X$ et $Y$ de l'exercice précédent) :
— $X$ et $X + Y$,
— $X$ et $X * Y$,
— $2 * X + Y$ et $3 * X + Y$,

1. Calculez les coefficients de corrélation de ces variables,
2. **Commentez les résultats.**

# Exercice 5 : ligne de régression

## Rappel

La ligne de régression est définie par :

$$y = \frac{cov(X,Y)}{var(X)}(x - m_X) + m_Y$$

1. Prenez 2 échantillons $X$ et $Y$ de loi uniforme de taille 20 sur l'intervalle $[0, 9]$.
2. Affichez dans une figure, sous la forme d'une croix noire, les 20 points définis à travers les échantillons $X$ et $Y$ précédents. Un point possède des coordonnées $(x, y)$.
   Exemple : Point 1 $(x_1, y_1)$, Point 2 $(x_2, y_2)$, ... Point 20 $(x_20, y_20)$.
3. Calculez les moyennes de $X$ et de $Y$ ($m_X$ et $m_Y$), la variance de $X$ ($var(X)$) et la covariance ($cov(X, Y)$).
4. Tracez en rouge la ligne de régression de $Y$ sur $X$, sachant que $x$ prend ses valeurs entre 0 et 9.
5. Ecrivez maintenant une fonction « ligne_regression » qui prend en entrée les bornes et la taille de la loi uniforme et renvoie les valeurs de $y$ et $x$.
6. Appelez cette fonction puis affichez les points et la droite de régression.
7. Calculez la droite de régression pour un intervalle $[0, 9]$ et des tailles de 10, 100 et 1000. **Que remarquez-vous ?**

# Exercice 6 : test du $\chi^2$

1. Prenez un échantillon $X$ de loi uniforme de taille 10 sur l'intervalle $[0, 1]$.
2. Calculez et affichez l'histogramme pour un pas $0.2$.
3. Nous voulons comparer cette suite statistique à la loi théorique uniforme sur $[0, 1]$ en utilisant le critère du $\chi^2$.
   (a) Ecrivez une fonction « chi2 » qui, à partir des probabilités statistiques et théoriques et de la taille de l'échantillon, calcule la valeur du $\chi^2$.
   (b) Plutôt que d'utiliser la table du $\chi^2$ (comme en cours), calculez la probabilité $p$ de conformité des lois théoriques et statistiques. Utilisez la fonction *chi2.cdf* du module *scipy.stats*. **Que concluez-vous ?**

# Exercice 7 : chaîne de Markov

## Rappel :

Une chaîne de Markov est définie par sa matrice de transition $P$. Nous considérerons ici uniquement les chaînes irréductibles. Si $P$ est une matrice $n * n$, les probabilités « limites » peuvent être trouvées à l'aide du système d'équations vu en cours :

$$(p(1), \ldots, p(n)) \, .P = (p(1), \ldots, p(n)) \qquad \text{et} \qquad p[1] + \ldots + p[n] = 1$$

1. Prenez cette matrice P :

$$\begin{bmatrix} 0.5 & 0.25 & 0.25 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0.2 & 0.8 & 0 \end{bmatrix}$$

   et ces probabilités initiales : $(1, 0, 0)$.
2. Afin d'effectuer des multiplications matricielles plus facilement, nous transformerons les matrices « array » au format « mat » avec *np.mat*. Calculez les probabilités obtenues (en multipliant par les puissances de $P$ correspondantes) après $n$ étapes ($n = 1, 2, 4, 8 \ldots$).
3. **À partir de quelle étape, converge-t-on vers les probabilités limites ?** Vérifiez vos résultats en effectuant le calcul sur papier.

# Exercice 8 : partition d'intervalles

1. Créez une fonction qui, prenant en entrée des nombres aléatoires suivant une loi uniforme $X$ dans l'intervalle $[0, 1]$ et des probabilités $p_1$, $p_2$ et $p_3$, renvoie un vecteur contenant uniquement les valeurs $0$, $1$ et $2$ telle que :
$$P(X = 0) = \frac{1}{4}, \; P(X = 1) = \frac{1}{2} \text{ et } P(X = 2) = \frac{1}{4}$$
2. Testez pour des tailles de 1000, 10000 et 100000 valeurs.
3. Vérifiez que les proportions obtenues correspondent bien aux probabilités fixées dès le départ, à savoir : $\frac{1}{4}$, $\frac{1}{2}$ et $\frac{1}{4}$.
4. Ecrivez une fonction qui calcule la probabilité d'obtenir une séquence $s$ de $k$ nombres : $s = [s_{1,2}, \ldots, s_k]$ avec $s_i \in 0, 1, 2, i = 1, \ldots k$. Cette fonction prend en entrée une séquence $s$ à tester et des probabilités $p_1$, $p_2$ et $p_3$ (par exemple, celles de la question précédente : $\frac{1}{4}$, $\frac{1}{2}$, $\frac{1}{4}$).
5. Testez votre fonction avec les 2 séquences suivantes :
   — $s_1 = [0, 1, 2]$. *Solution :* $P(s_1) = 0.0313$,
   — $s_2 = [0, 1, 2, 2]$. *Solution :* $P(s_2) = 0.0078$.
6. Quelle est la séquence de $5$ nombres la plus probable et quelle est sa probabilité ?