Un peu de probabilités avec Python

D'après un TP de Stéphane Gonnord

Buts du TP

☞ L'objet mathématique décrivant le comportement d'un dé (non pipé) à six faces s'appelle une *variable aléatoire*, suivant une loi dite *uniforme* sur l'ensemble $[\![1;6]\!]$.

Quelle est la moyenne des résultats obtenus quand on lance les dés de nombreuses fois?

- le bon sens populaire dit : « pfiou, ça va varier »;
- un bon élève de terminale doit pouvoir dire : « en moyenne, ça va tendre vers 7/2 » (notion des plus imprécises, mais c'est déjà ça);
- l'élève de MPSI/PCSI doit pouvoir dire que l'espérance après n tirages vaut toujours 7/2;
- un peu plus tard, ce même élève pourra dire (loi des grands nombres) que la moyenne *tend* vers 7/2 (en des sens subtils...);
- un peu plus tard encore, cet élève saura même décrire (grâce au théorème central limite) ce à quoi il faut s'attendre pour cette moyenne après n tirages : la loi la décrivant est proche d'une loi très simple.

On souhaite :

- faire des simulations de n lancés et évaluer les moyennes obtenues ➟ *première partie*;
- observer cette moyenne lorsque n tend vers $+\infty$ (la loi faible des grands nombres nous dit ce qu'elle fait) ➟ *deuxième partie*;
- observer la distribution des valeurs sur différents « runs » de 100 lancés (le théorème central limite décrit assez bien celle-ci) ➟ *troisième partie*;

☞ On veut simuler et représenter graphiquement une marche aléatoire dans $\mathbb{Z}\times\mathbb{Z}$ ➟ *quatrième partie.*

EXERCICE 1 *Copier le répertoire* `TP13-Probas` *contenu dans le répertoire* `Donnees` *de la classe et le coller dans son répertoire personnel. Le fichier* `cadeau.py` *contient une fonction* `chrono` *qu'on utilisera pour chronométrer certains appels de fonctions.*

Créer dans ce dossier un sous-dossier `jolis-dessins` *dans lequel seront placés les différents (jolis) pdf produits pendant ce TP.*

Lancer Spyder, créer puis sauvegarder immédiatement au bon endroit le fichier `TP13.py`.

À l'aide de F6, bien fixer la configuration d'exécution : « exécuter dans un interpréteur dédié » et « interagir avec l'interpréteur après exécution » semblent deux bons choix...

1 Premières simulations avec Python

La bibliothèque `random` fournit la fonction `randint`... qui devrait bien vous servir.

1. Écrire une fonction `tirage_de` telle que `tirage_de()` retourne une valeur parmi $[\![1;6]\!]$, chacune avec probabilité $\frac{1}{6}$.
2. Ecrire des fonctions `sommetab`, `moyennetab` et `extremumtab` retournant respectivement la somme des éléments d'un tableau de nombres, sa moyenne arithmétique et le quadruplet (`minimum`, `indice`$_{\texttt{minimum}}$,`maximum`, `indice`$_{\texttt{maximum}}$).
3. Écrire une fonction prenant un entier $N>0$ en argument, réalisant N tirages consécutifs (on parle de *run*), et renvoyant la moyenne des résultats obtenus.


```
>>> run(1000)
3.462
>>> run(1000)
3.47
```

4. Construire un tableau `t` constitué des résultats de 10^5 runs consécutifs de 10^2 tirages [1].
5. Parmi les 10^5 runs du tableau `t` :
   - Quelle est la moyenne... des moyennes?
   - Quelle est la moyenne maximale qui a été atteinte? Et la minimale?
6. Ecrire une fonction `effectif_intervalle(tab, binf, bsup)` qui détermine le nombre de valeurs d'un tableau d'entiers `tab` qui appartiennent à l'intervalle $[$`binf`$;$`bsup`$]$.
   Calculer `effectif_intervalle(t,1,3.4)` et `effectif_intervalle(t,3.6,6)`,
   puis `effectif_intervalle(t,1,3)` et `effectif_intervalle(t,4,6)`
   et enfin `effectif_intervalle(t,1,2.98)` et `effectif_intervalle(t,4.02,6)`.
   Quelle remarques peut-on faire?

   Ces questions seront reprises à la fin de la partie 3.

2 Loi (faible) des grands nombres

Si on note Y_n la variable aléatoire égale à la moyenne après n lancers aléatoires, alors l'espérance de Y_n vaut 7/2, et on a même d'après la loi (faible) des grands nombres (se démontre avec l'inégalité de Tchebichev) :

$$\forall\varepsilon>0,\qquad \mathbb{P}(|Y_n-7/2|>\varepsilon)\xrightarrow[n\to+\infty]{}0.$$

1. Écrire une fonction prenant en entrée deux entiers $a,b>0$, réalisant un run de $a\times b$ tirages, et retournant une liste constituée des moyennes obtenues (depuis le début) tous les b runs. Cette fonction retourne donc une liste de $a+1$ flottants.

   Dans l'exemple suivant, on réalise deux runs de 1000 *tirages, en notant les résultats intermédiaires tous les* 200 *tirages.*

```
>>> ab_run(5, 200)
[0, 3.425, 3.4725, 3.5383333333333336, 3.51875, 3.5089999999999995]
>>> ab_run(5, 200)
[0, 3.4, 3.48, 3.4599999999999995, 3.47875, 3.501]
```

2. Exécuter `ab_run(10, 10**k)` pour $k\in[\![3,6]\!]$.
3. Représenter l'évolution de la moyenne lors de n runs (par exemple pour $n\in\{10^5,10^6,10^7\}$. *À n fixé, on pourra relier les points d'abscisses* $100k$ *($k\in[\![0;n/100]\!]$) et d'ordonnées* `ab_run(n/100, 100)` *(en enlevant éventuellement les premiers points; vous verrez pourquoi après un premier essai).* Voici un code possible :

```
import numpy as np  #pour disposer de la structure de tableaux array
import matplotlib.pyplot as plt #pour les graphiques
import os #pour gérer proprement les chemins de fichiers avec os.path

for n in [10**5, 10**6, 10**7]:
    kinf = 10 #borne inferieure pour k
    x = ............................ #les abscisses
    y = ab_run(n//100, 100)[kinf:] #les ordonnées
```

1. Vous pouvez commencer avec 10^4 runs. Et pensez à la suite, le temps qu'*il* fasse les calculs! Quand vous passerez à la partie suivante, commentez ces calculs longs.

```
        plt.plot(x,y,color='red') #le tracé
        plt.savefig(os.path.join('jolis-dessins','evolution-moyenne-%d.pdf'%n))
        plt.clf() #efface la figure et laisse la place pour d'autres graphiques
```

4. Bonus : le théorème central limite (à venir) dit que la moyenne est « proche de l'espérance », dans une zone de largeur caractéristique $2\frac{\sigma}{\sqrt{n}}$, avec ici $\sigma = \sqrt{\frac{35}{12}}$. Visualiser ceci en ajoutant sur les graphes précédents les courbes d'équations $y = \mu \pm \frac{\sigma}{\sqrt{n}}$.

3 Théorème central limite

On va observer la moyenne Y (qui est une variable aléatoire) après $n = 100$ tirages. On ne touchera plus à cette valeur de n. Cette moyenne a une *espérance* de $\frac{7}{2}$ et un écart-type $\sigma = \sqrt{\frac{35}{12}}$. Par ailleurs, puisque $100 \simeq +\infty$, le théorème central limite dit que cette loi peut être approchée (convergence en loi) par une variable aléatoire à densité[2] suivant une « loi normale » $\mathcal{N}(7/2, \sigma^2/n)$: on va représenter la fonction de densité de cette loi.

On va lancer de nombreux runs de 100 tirages (comme à la fin de la première partie) et observer la distribution de fréquences, en comparant visuellement l'histogramme obtenu au graphe précédent. On va enfin comparer les résultats de la fin de la première partie à ce que « prédit » la théorie.

1. Une fonction de densité de la loi $\mathcal{N}(7/2, \sigma^2/n)$ est $x \mapsto \frac{\sqrt{n}}{\sigma\sqrt{2\pi}} e^{-\frac{n(x-7/2)^2}{2\sigma^2}}$. Représenter le graphe de cette fonction.

2. Pour visualiser la distribution de fréquences des valeurs expérimentales, on peut relier les points de coordonnées (v, f_v), avec v décrivant les valeurs prises par Y (donc de la forme $\frac{k}{100}$), et f_v la fréquence des apparitions de v lors de N runs, avec par exemple $N = 10^5$. Voici un code à compléter :

```
n = 100
nb_runs = 10**5

plein_de_runs = [run(n) for _ in range(nb_runs)]

x = [i/float(n) for i in range(250, 451)] #pas de moyennes trop petites ou  trop grandes
y = [plein_de_runs.count(i)/float(nb_runs) for i in x] #fréquences des moyennes

plt.plot(x, y)
```

On aura noté la très efficace méthode de liste `count`, qui fait ce qu'on pense.

3. Dans l'histogramme de la distribution expérimentale des fréquences de la moyenne Y, les ordonnées sont des fréquences et dans le graphe de la fonction de densité de la loi normale $\mathcal{N}(7/2, \sigma^2/n)$, les ordonnées sont des densités (probabilité par intervalle d'amplitude 1).

   Pour comparer les deux courbes sur un même graphique, on peut normaliser la distribution expérimentale en divisant ses ordonnées par $\frac{1}{n}$ qui est l'écart entre deux valeurs successives de la variable aléatoire discrete Y.

   De plus on peut restreindre les abscisses à l'intervalle $\left[7/2 - 6\frac{\sigma}{\sqrt{n}} ; 7/2 + 6\frac{\sigma}{\sqrt{n}}\right]$ qui contiendra près de 100 % des valeurs d'après les tables de la loi normale $\mathcal{N}(7/2, \sigma^2/n)$.

   Faites-le!

2. Les variables aléatoires à densité et le théorème de Moivre-Laplace, cas particulier du théorème central limite, sont au programme en Terminale mais pas en CPGE!

4. Si la loi de Y peut être approchée par la loi $\mathcal{N}(7/2, \sigma^2/n)$, en centrant et réduisant, la variable aléatoire $X = \frac{\sqrt{n}}{\sigma}(Y - 7/2)$, suit approximativement une loi normale centré réduite $\mathcal{N}(0,1)$, et en particulier :

$$\mathbb{P}(a \leqslant Y \leqslant b) = \mathbb{P}\left(a' \leqslant X \leqslant b'\right) \approx \frac{1}{\sqrt{2\pi}} \int_{a'}^{b'} e^{-t^2/2} dt,$$

   avec $a' = \frac{\sqrt{n}}{\sigma}(a - 7/2)$ et $b' = \frac{\sqrt{n}}{\sigma}(b - 7/2)$.

   Vérifier ceci pour $(a, b) = (3.4, 3.6)$ en évaluant cette intégrale, et en comparant aux résultats expérimentaux trouvés à la fin de la première partie. Voici un code à compléter :

```
import scipy.integrate, scipy.stats

def n01_densite(x):
        return 1/np.sqrt(2*np.pi)*np.exp(-x**2/2.)

mu, sigma = 7/2., np.sqrt(35/12.)
n = 100
a1, b1 = np.sqrt(n)/sigma*(a - mu), np.sqrt(n)/sigma*(b - mu)
integrale1 = scipy.integrate.quad(n01_densite, a1, b1) #tuple

print('Probabilite theorique P(%1.2f <= Z <= %1.2f) pour la loi N(0,1)'
'avec scipy.integrate.quad : %1.4f'%(a1, b1, integrale1[0]))

Z = scipy.stats.norm(0, 1) #definition de la loi normale
integrale2 = Z.cdf(b1) - Z.cdf(a1) #P(a1 <= Z <= b1)

print('Probabilite theorique P(%1.2f <= Z <= %1.2f) pour la loi N(0,1)'
'avec scipy.stats.norm : %1.4f'%(a1, b1, integrale2))
```

5. Bonus : déterminer la vraie distribution de probabilité de la moyenne après 100 tirages. *Il s'agit donc de sommer des lois uniformes discrètes. Ça sent la programmation dynamique; il est conseillé de considérer plutôt la distribution de la somme des tirages, histoire de rester parmi des entiers...*

4 Marche aléatoire dans le plan

Dans le plan assimilé à $\mathbb{R} \times \mathbb{R}$ on considère une particule placée initialement en $(0; 0)$ et qui se déplace aléatoirement dans l'ensemble des points à coordonnées entières assimilé à $\mathbb{Z} \times \mathbb{Z}$.

Soit p un réel fixé dans l'intervalle $[0; 1]$ appelé paramètre de la marche aléatoire. Si la particule se situe en $(x_0; y_0)$, elle se déplace en $(x_1; y_1)$ selon les règles suivantes :

- $x_1 = x_0 + 1$ et $y_1 = y_0$ avec une probabilité de $p/2$, *la particule se déplace vers la droite*;
- $x_1 = x_0 - 1$ et $y_1 = y_0$ avec une probabilité de $p/2$, *la particule se déplace vers la gauche*;
- $x_1 = x_0$ et $y_1 = y_0 + 1$ avec une probabilité de $(1 - p)/2$, *la particule se déplace vers le haut*;
- $x_1 = x_0$ et $y_1 = y_0 - 1$ avec une probabilité de $(1 - p)/2$, *la particule se déplace vers le bas.*

Une marche aléatoire avec n déplacements élémentaires successifs est dite de longueur n.

1. Écrire une fonction `deplacement(x0, y0, p)` qui retourne les coordonnées de la nouvelle position de la particule après un déplacement élémentaire de paramètre $p \in [0; 1]$. On utilisera la fonction `random()` du module `random`.

2. Écrire une fonction `marche(n, p)` qui simule une marche aléatoire de paramètre `p` et de longueur `n` à partir de la position $(0;0)$ et qui retourne un couple `(x, y)` de listes correspondant aux abscisses et ordonnées successives de la particule.

3. Fixons $p = 0.5$ pour avoir une loi uniforme sur $[\![1;4]\!]$ lors de chaque déplacement.

   (a) Représenter graphiquement des marches aléatoires de longueur n compris dans $\{100, \ 1000, 10000\}$. Voici un exemple de code (*problème d'affichage sous Spyder avec certaines configurations*) :

```
p = 0.5
x, y = marche(1000, p)
x = np.array(x)
y = np.array(y)
plt.scatter(x, y, c=range(len(x)))
plt.colorbar() #les points seront colorés selon leur ordre d'apparition
plt.savefig(os.path.join('jolis-dessins','marche-aleatoire-%d-pas.pdf'%n))
plt.show()
plt.clf()
```

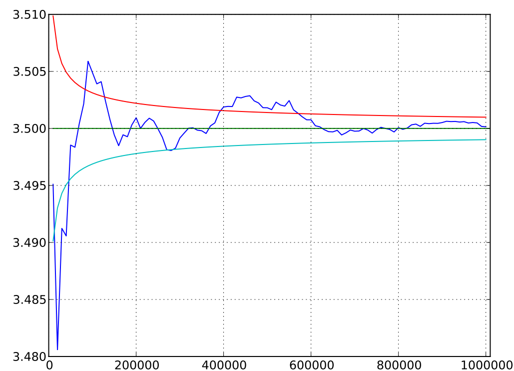
   (b) Ecrire une fonction `tempsretour(n, p)` qui simule une marche aléatoire de longueur n à partir de la position $(0;0)$ et qui retourne soit le nombre d'étapes pour le premier retour à l'origine ou $-\frac{n}{10}$ sinon. Représenter graphiquement un histogramme de la distribution des temps de retour pour 5000 marches aléatoires de longueur 1000 et calculer la fréquence de non retour à l'origine. Voici un morceau du code :

```
distrib = np.array([tempsretour(1000, 0.5) for _ in range(5000)])
plt.hist(distrib, bins=22) #bins est le nombre de classes
plt.show()
plt.clf()
```

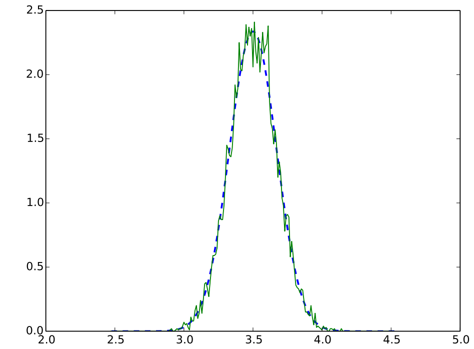
4. Reprendre les représentations graphiques de marches aléatoires en faisant varier p pour privilégier/interdire les déplacements « verticaux » puis « horizontaux ».

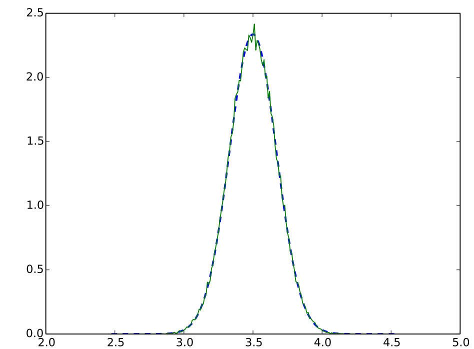
5 Gallerie

- Loi faible des grands nombres, pour 10^6 tirages.

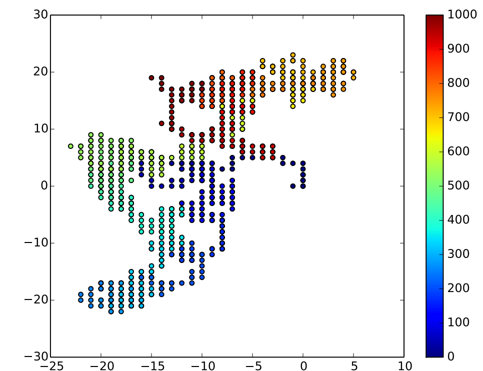


- Théorème central limite : 10^4 puis 10^5 runs de 100 tirages.





- Marche aléatoire de longueur $n = 1000$ avec $p = 0.5$.



Corrigé du TP 13 Probabilités

Chenevois-Jouhet-Junier

1 Imports de bibliothèques et réglage du répertoire de travail

```
import random, time, os
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate, scipy.stats

REPERTOIRE = 'jolis-dessins' #nom du répertoire de stockage des jolis dessins
```

- Toutes les figures seront rangées dans un répertoire `jolis-dessins` situé dans le répertoire de travail `TP13` contenant le script. Avec `Pyzo`, il faut régler le répertoire de travail (point de départ du chemin relatif des ressources externes) en exécutant le script avec l'option `Run as script` ou la séquence de touches `CTRL + SHIFT + E`.
- On peut vérifier la valeur du répertoire de travail avec la fonction `pwd` si l'interpréteur est Ipython.
- Une fonction de chronométrage est contenue dans `cadeau.py`.

```
def chrono(f):
    """Remplace la fonction f par la fonction fbis
    en affichant le temps d'exécution de la fonction f"""

    def fbis(*args, **kargs):
        debut = time.time()
        res = f(*args, **kargs)
        fin = time.time()
        print("temps : %.6f secondes"%(fin-debut))
        return res

    return fbis
```


2 Premieres simulations avec Python questions 1 et 2

- Question 1

```
def tirage_de():
    """simule un lancer de dé équilibré"""
    return random.randint(1, 6)

"""
>>> [tirage_de() for _ in range(10)]
[1, 4, 4, 6, 1, 5, 2, 6, 4, 3]
"""
```

- Question 2

```
def sommetab(t):
    """somme des éléments d'un tableau de nombres"""
    s = 0
    for e in t:
        s += e
    return s

def moyennetab(t):
    """moyenne des éléments d'un tableau"""
    return sommetab(t)/float(len(t))


def extremumtab(t):
    """retourne le quaduplet (minimum, indice(minimum), maximum,
    indice(maximum))  d'un tableau de nombres
    """
    mini, indexmini, maxi, indexmaxi = t[0], 0, t[0], 0
    for k in range(1, len(t)):
        e = t[k]
        if e< mini:
            mini, indexmini = e, k
        elif e > maxi:
            maxi, indexmaxi = e, k
    return mini, indexmini, maxi, indexmaxi
```

3 Premieres simulations avec Python questions 3 et 4

- Question 3

```
def run(n):
    """moyenne des résultats  sur n lancers de dés consécutifs"""
    #return sum([tirage_de() for _ in range(n)])/float(n)
    return moyennetab([tirage_de() for _ in range(n)])

"""
```

```
>>> run(1000)
3.534
>>> run(10000)
3.5077
"""
```

- Question 4

```
def tab1(m, n):
    """tableau des résultats de m runs de n tirages"""
    return [run(n) for _ in range(m)]
```

Chronométrage pour les résultats de 10^4 runs de 10^2 tirages, ce qui représente 10^6 lancers de dés.

```
"""
>>> t = chrono(tab1)(10**4, 10**2)
temps : 1.668359 secondes
>>> t[0]
3.58
>>> len(t)
10000
"""

t = tab1(10**5, 10**2)
```

Chronométrage pour les résultats de 10^5 runs de 10^2 tirages, ce qui représente 10^7 lancers de dés.

```
"""
>>> t = chrono(tab1)(10**5, 10**2)
temps : 16.776327 secondes
"""
```

4 Premieres simulations avec Python question 5

- Moyenne de toutes les simulations :

```
"""
>>> moyennetab(t)
3.5000690999999993
"""
```

Moyennes minimale et maximale qui ont été atteintes :

```
"""
>>> extremumtab(t)
(2.74, 82526, 4.21, 23506)
"""
```


5 Premieres simulations avec Python question 6

```
def effectif_intervalle(tab, binf, bsup):
    """Retourne le nombre de moyennes d'un tableau de m moyennes
    de n tirages, qui se trouvent entre binf et bsup"""
    cpt = 0
    for e in tab:
        if binf <= e and e <= bsup:
            cpt += 1
    return cpt
```

On observe une symétrie de la distribution par rapport à $3,5$ et une diminution des effectifs dans les plages $[1;3,5-\alpha]$ et $[3,5+\alpha;6]$ lorsque α croit.

```
"""
>>> effectif_intervalle(t,1,3.4)
29067
>>> effectif_intervalle(t,3.6,6)
28974
>>> effectif_intervalle(t,1,3)
182
>>> effectif_intervalle(t,4,6)
186
>>> effectif_intervalle(t,1,2.98)
131
>>> effectif_intervalle(t,4.02,6)
130
"""
```

6 Loi faible des grands nombres, questions 1 et 2

- Question 1

```
def ab_run(a, b):
    """Réalise un run de axb tirages et retourne une liste
    des moyennes obtenues depuis le debut par incrémentation de b tirages"""
    tmoy  = [0]
    s = 0
    for i in range(1, a+1):
        tmoy.append((tmoy[-1]*(i-1) + run(b))/i)
    return tmoy

def ab_run2(a,b):
```

```
    tmoy, s  = [0], 0
    for i in range(1, a*b+1):
        s += tirage_de()
        if i%b == 0:
            tmoy.append(float(s)/i)
    return tmoy

"""
>>> ab_run(5, 200)
[0, 3.305, 3.4050000000000002, 3.395, 3.415, 3.4490000000000003]
"""
```

- Question 2

```
"""
>>> for k in range(3, 7):
...     print(ab_run(10, 10**k), end='\n\n')
...
[0, 3.487, 3.504, 3.4923333333333333, 3.48275, 3.4996, 3.4915,
 3.488142857142857, 3.49375, 3.4905555555555554, 3.4906]

[0, 3.4749, 3.492, 3.4809333333333333, 3.488425, 3.48846,
 3.4897833333333335, 3.4929571428571426, 3.4890125,
 3.4895666666666667, 3.4906]

[0, 3.49901, 3.501515, 3.501386666666667, 3.4998625, 3.500774,
 3.500583333333333, 3.5006471428571143, 3.50068375,
  3.501212222222222, 3.501309]

[0, 3.501357, 3.501925, 3.500897, 3.5006585, 3.5001964,
 3.5001498333333334, 3.499802857142857, 3.4997615,
  3.4996607777777777, 3.4994792]
"""
```

7 Loi faible des grands nombres, questions 3 et 4

- Si on se fixe un $\varepsilon > 0$, la *loi faible des grands nombres* nous dit que la moyenne des réalisations de la variable aléatoire `tirage_de()` (ou X pour simplifier) est de plus en plus probablement proche de son espérance $\frac{7}{2}$ lorsque n tend vers l'infini.

$$\forall\varepsilon > 0, \lim_{n\to+\infty} \mathbb{P}\left(\left|\frac{1}{n}\sum_{k=1}^{n} x_k \ - \ \mathrm{E(X)}\right| \geqslant \varepsilon\right) = 0$$

- Il n'empêche que même pour n assez grand, cette moyenne peut s'écarter notablement de l'espérance , mais avec une probabilité très faible et controlée par le nombre de réalisations.
- On dit que la moyenne des réalisations de la variable aléatoire *converge en probabilité* vers son espérance.

```
def evolution_moyenne(n):
```



```
    """Graphique de l'évolution de la moyenne lors de n lancers de dés
    obtenu en reliant les points d'abscisses 100k avec k
    entre 0 et n//k et d'ordonnées ab_run(n//100,100)."""
    mu = 7/2.
    sigma = np.sqrt(35/12.)

    def fbinf(n):
        return mu - sigma/np.sqrt(n)

    def fbsup(n):
        return mu + sigma/np.sqrt(n)

    #on ne représente pas les moyennes sur les 1000 premiers tirages
    #pour lesquelles la moyenne peut etre trop éloignée de 7/2
    kinf = 10
    x = [100*k for k in range(kinf,n//100+1)]
    y = ab_run(n//100, 100)[kinf:]
    plt.plot(x,y,color='red')
    plt.plot(x, fbinf(x), color='blue', linestyle='--')
    plt.plot(x, fbsup(x), color='blue', linestyle='--')
    plt.title("Evolution des moyennes pour %d runs"%n)
    plt.xlabel("n")
    plt.ylabel("Moyenne")
    plt.title("Loi faible des grands nombres")
    plt.savefig(os.path.join('jolis-dessins','evolution-moyenne-%d.png'%n), dpi=80)
    #plt.clf() #efface la figure et laisse la place pour d'autres graphiques

def central_limite():
    for n in [10**5, 10**6, 10**7]:
        evolution_moyenne(n)
```

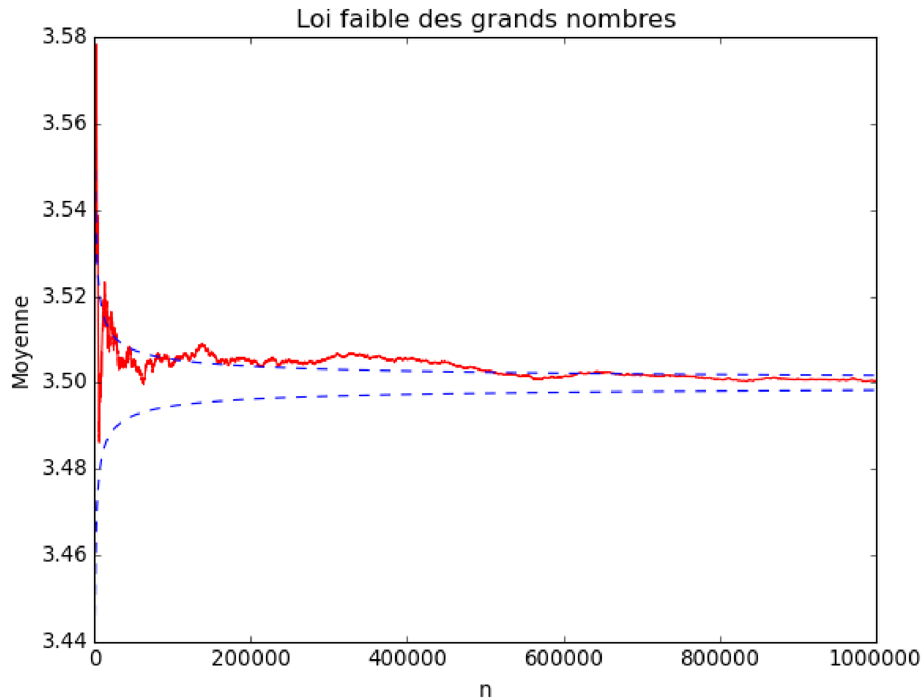


Figure 1:

8 Théorème central limite Question 1

- On rappelle *le théorème de Moivre-Laplace* (cas particulier du *théorème central limite* pour les lois binomiales) énoncé en classe de terminale :

  Soit $p \in [0;1]$. On suppose que pour tout entier naturel n non nul, la variable aléatoire X_n suit la loi binomiale $\mathcal{B}(n\,;\,p)$. Soit $Z_n = \frac{X_n - np}{\sqrt{np(1-p)}}$, la variable centrée et réduite associée à X_n. Pour tous réels a et b avec $a \leqslant b$ on a :

  $$\lim_{n\to+\infty} \mathrm{P}\left(a \leqslant Z_n \leqslant b\right) = \int_a^b \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{t^2}{2}}\,\mathrm{d}t$$

- Plus généralement, *le théorème central limite* peut s'énoncer ainsi :

  Soit $(X_n)_{n\in\mathbb{N}}$ une suite de variables aléatoires indépendantes et identiquement distribuées admettant une espérance μ et une variance σ^2. Soit $S_n = \sum_{k=1}^{n} X_k$ la somme des variables aléatoires X_k et soit $S_n^* = \frac{S_n - n\mu}{\sqrt{n}\sigma}$, la variable centrée et réduite associée à S_n. Pour tous réels a et b avec $a \leqslant b$ on a :

  $$\lim_{n\to+\infty} \mathrm{P}\left(a \leqslant S_n^* \leqslant b\right) = \int_a^b \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{t^2}{2}}\,\mathrm{d}t$$

  On dit que la suite (S_n^*) *converge en loi* vers une variable aléatoire qui suit une loi normale centrée réduite. D'après les propriétés de stabilité des lois normales par transformation affine, on peut même dire que la suite des moyennes $\left(\frac{S_n}{n} = \frac{\sqrt{n}\sigma S_n^* + n\mu}{n}\right)$ converge en loi vers une variable aléatoire suivant une loi normale d'espérance μ et de variance $\frac{\sigma^2}{n}$.

- Dans ce TP, les variables aléatoires X_n suivent une loi discrète uniforme sur $[a;b] = [1;6]$ d'espérance $\frac{a+b}{2} = \frac{1+6}{2} = \frac{7}{2}$ et d'écart-type $\sqrt{\frac{n^2-1}{12}} = \sqrt{\frac{35}{12}}$.

  D'après le théorème central limite, la suite des moyennes $\left(\frac{\sum_{k=1}^{n} X_k}{n}\right)$ *converge en loi* vers une variable aléatoire qui suit une loi normale d'espérance $\mu = \frac{7}{2}$ et de variance $\frac{\sigma^2}{n} = \frac{35}{12n}$.

  Si on considère que $n = 100$ est assez grand, on peut comparer la variable aléatoire moyenne $\frac{\sum_{k=1}^{100} X_k}{100}$ et la variable aléatoire à densité de loi normale d'espérance $\mu = \frac{7}{2}$ et de variance $\frac{\sigma^2}{n} = \frac{35}{12\times 100} \approx 0,03$.

```
def normaldensite(mu, sigma, n, x):
    """fonction de densite de la loi normale N(mu,sigma**2/n)
    approchant la moyenne Y de n réalisations indépendantes d'une
    variable  aléatoire d'espérance mu et d'écart-type sigma"""
    return np.sqrt(n/(2*np.pi))/sigma*np.exp(-n*(x-float(mu))**2/(2*sigma**2))


def graphique_loi_normale(mu, sigma, n, binf, bsup, alone=True):
    """Représentation de la fonction de densite de la loi normale
    N(mu,sigma) sur un intervalle [binf, bsup] centré sur mu"""
    x = np.linspace(binf, bsup, 1000)
    plt.title(r"Densité d'une loi normale $\mathcal{N}\left(\mu=%1.2f;\sigma^2=%1.2f\right)$"%(
    plt.plot(x, normaldensite(mu,sigma,n,x),linestyle='--',linewidth=2, label='Densité théoriq
    #sauvegarde sur disque et fermeture de la figure si alone == True
    if alone:
        plt.savefig(os.path.join('jolis-dessins',
        'fonction-densite-normale-mu-%1.2f-sigma-%1.2f-n-%d.png'%(mu,sigma,n)), dpi=80)
        #plt.clf()

n = 100
mu = 7/2. #esperance
sigma = np.sqrt(35/12.)  #ecart-type
#On a vu en terminale qu'avec une amplitude de 6 écart-types
#on a déjà 99,7 % des valeurs.
binf, bsup = mu-3*sigma/np.sqrt(n), mu+3*sigma/np.sqrt(n)
```

```
"""
In [37]: binf, bsup
Out[37]: (2.9876524617020204, 4.0123475382979796)

In [38]: graphique_loi_normale(7/2., np.sqrt(35/12.), 100, binf, bsup, alone=True)
"""
```

9 Théorème central limite Questions 2 et 3

- Question 2

```
def graphique_distribution(n, nb_runs, binf, bsup, dilatation=1, alone=True):
    """Graphique de la distribution des valeurs expérimentales,
    dilatation est un facteur de multiplication des ordonnées.
    """
    #nb_runs moyennes sur n lancers
    plein_de_runs = [run(n) for _ in range(nb_runs)]
    #une plage de moyennes qui nous intéresse
    x = np.array([i/float(n) for i in range(int(binf*n), int(bsup*n)+1)])
    #fréquences de ces moyennes
    y = np.array([plein_de_runs.count(i)/float(nb_runs) for i in x])
    plt.plot(x, y*dilatation, label='Densité empirique')
    #sauvegarde sur disque et fermeture de la figure si alone == True
    if alone:
        plt.savefig(os.path.join('jolis-dessins',
    'distribution-experimentale-n-%i-nbruns-%i.png'%(n, nb_runs)), dpi=80)
        #plt.clf()
```

- Question 3

```
def comparaison_graphique_distribution_loinormale(mu, sigma, n, nb_runs):
    """
    On va se restreindre à mu-6*sigma/np.sqrt(n), mu+6*sigma/np.sqrt(n),
    ce sera suffisant pour contenir près de 100 % des valeurs
    """
    binf, bsup = mu-6*sigma/np.sqrt(n), mu+6*sigma/np.sqrt(n)
    dilatation = n  #diviser par 1/n c'est multiplier par n
    graphique_loi_normale(mu, sigma, n, binf, bsup, alone=False)
    graphique_distribution(n, nb_runs, binf, bsup, dilatation, alone=False)
    plt.title(r"Convergence en loi de la moyenne d'une loi uniforme discrète sur $[ 1; 6]$" + '
    plt.legend(loc = 'lower right')
    plt.savefig(os.path.join('jolis-dessins',
    'comparaison-distribution-densite-n-%i-nbruns-%i.png'%(n, nb_runs)), dpi=80)
    #plt.clf()
```

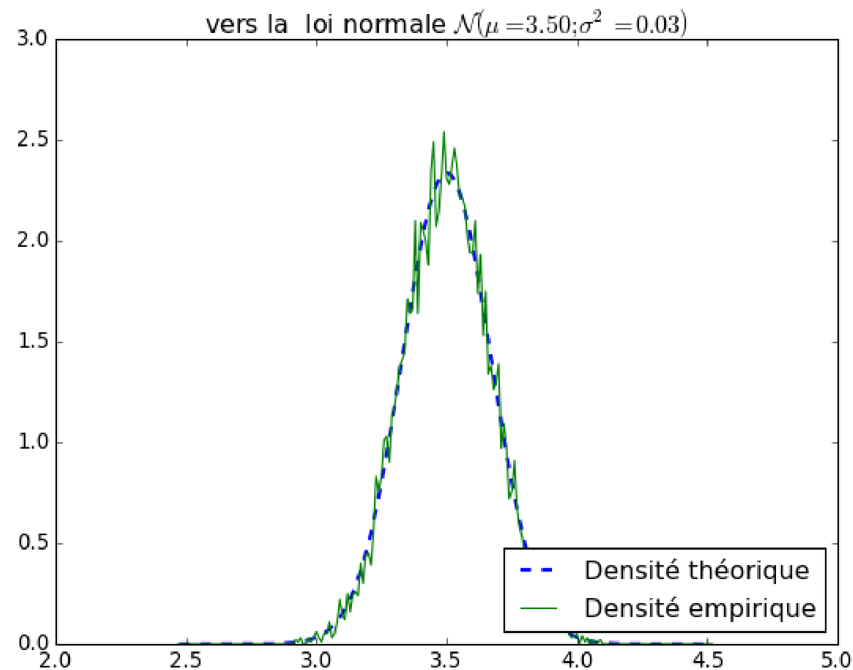



Figure 2:

10 Théorème central limite Question 4

```
def comparaison_numerique_distribution_loinormale(mu, sigma, n, nb_runs, a, b):
    """Soit Yn la moyenne de n réalisations indépendantes d'une variable
    aléatoire d'espérance mu et d'écart-type sigma (loi uniforme  sur
    [1;6] dans ce TP. D'apres le Theoreme Central Limite, la variable
    centrée réduite (Yn-mu)/sigma converge en loi vers une variable
    aléatoire  à densité Z suivant la loi normale N(0,1).
    On compare donc P(a <= Y <= b) et P (a1 <= Z <= b1).
    """

    def n01_densite(x):
```

```
        return 1/np.sqrt(2*np.pi)*np.exp(-x**2/2.)

    a1, b1 = np.sqrt(n)/sigma*(a - mu), np.sqrt(n)/sigma*(b - mu)
    tab = [run(n) for _ in range(nb_runs)]
    freq = effectif_intervalle(tab, a, b)/float(nb_runs)
    print('Frequence experimentale de moyennes entre %1.2f et %1.2f : %1.4f'%(a, b, freq))
    integrale1 = scipy.integrate.quad(n01_densite, a1, b1) #tuple
    print('Probabilite theorique P(%1.2f <= Z <= %1.2f) pour la loi N(0,1)'
    'avec scipy.integrate.quad : %1.4f'%(a1, b1, integrale1[0]))
    Z = scipy.stats.norm(0, 1) #definition de la loi normale
    integrale2 = Z.cdf(b1) - Z.cdf(a1) #P (a1 <= Z <= b1)
    print('Probabilite theorique P(%1.2f <= Z <= %1.2f) pour la loi N(0,1)'
    'avec scipy.stats.norm : %1.4f'%(a1, b1, integrale2))

"""
>>> comparaison_numerique_distribution_loinormale(7/2.,np.sqrt(35/12.),
100,10**5,3.4, 3.6)
Frequence experimentale de moyennes entre 3.40 et 3.60 : 0.4610
Probabilite theorique P(-0.59 <= Z <= 0.59) pour la loi N(0,1)avec
 scipy.integrate.quad : 0.4418
Probabilite theorique P(-0.59 <= Z <= 0.59) pour la loi N(0,1)avec
 scipy.stats.norm : 0.4418
"""
```

11 Théorème central limite Question 5

C'est un exemple de programmation dynamique. Pour une version en `javascript`, voir ici.

```
def distribution_somme_des(n):
    """Vraie distribution d'effectifs de la somme de n dés équilibrés
    à 6 faces. Programmation dynamique avec tableau memorisant les
    différentes sommes deja atteintes et le nombre de fois ou elles ont
    ete atteintes. Boucle sur le nombre de lancers, puis les des puis
    les sommes precedentes"""
    somme = [[i, 1] for i in range(1, 7)]
    for k in range(2, n+1):
        newsomme = [[s[0]+1, s[1]] for s in somme] #face 1
        for de in range(2, 7): #les autres faces
            for j in range(de-1, len(newsomme)):
                newsomme[j][1] += somme[j-(de-1)][1]
            newsomme.append([somme[-1][0]+de,somme[-1][1]])
        somme = newsomme
    return somme

def distribution_somme_des2(n):
    """Version 2, boucle sur le nombre de lancers, puis les sommes
    precedentes puis les des."""
```



```
    somme = [[i, 1] for i in range(1, 7)]
    for k in range(2, n+1):
        newsomme = [[somme[0][0]+de, somme[0][1]] for de in range(1, 7)]
        for s in somme[1:]:
            for de in range(1, 6):
                #somme déjà atteinte avec un dé de moins et une valeur de de + 1
                newsomme[-6+de][1] += s[1]
            #nouvelle somme atteinte avec un 6
            newsomme.append([s[0]+6, s[1]])
        somme = newsomme
    return somme

def distribution_somme_des3(nbdes):
    """Version raccourcie"""
    t = [(k, 1) for k in range(1, 7)]
    n = 6 #taille de la distibution
    for i in range(1, nbdes):
        t = [ (nbdes + j , sum(t[j - k][1] for k in range(max(0, j - n + 1),
         min(j, 5) + 1))) for j in range(n + 5)]
        n += 5
    return t
"""
Exemples de distributions d'effectifs pour la somme de  2 ou 3 des.
On retouve le paradoxe du duc de toscane pour 3 des :
P(S=9)=25/6**3 et P(S=10)=27/6**3

>>> distribution_somme_des(2)
[[2, 1], [3, 2], [4, 3], [5, 4], [6, 5], [7, 6], [8, 5], [9, 4],
[10, 3], [11, 2], [12, 1]]
>>> distribution_somme_des(3)
[[3, 1], [4, 3], [5, 6], [6, 10], [7, 15], [8, 21], [9, 25], [10, 27],
 [11, 27], [12, 25], [13, 21], [14, 15], [15, 10], [16, 6], [17, 3],
 [18, 1]]
"""
```

12 Marche aléatoire Question 1

```
def deplacement(x0, y0, p):
    """deplacement élémentaire"""
    de = random.random()
    if de <= p/2.:
        x, y = x0 + 1, y0
    elif de <= p:
        x, y = x0 - 1, y0
    elif de <= p + (1 - p)/2.:
        x, y = x0, y0 + 1
    else:
```

```
        x, y = x0, y0 - 1
    return x, y

"""Test de la fonction deplacement pour p=1/2 (loi uniforme)
>>> les_deplacements = [deplacement(0,0,1/2) for _ in range(10000)]
>>> les_effectifs = [les_deplacements.count(d) for d in [(0,1),(0,-1),(1,0),(-1,0)]]
>>> les_effectifs
[2470, 2524, 2505, 2501]
"""
```

13 Marche aléatoire Question 2

```
def marche(n, p):
    """Retourne le couple (x,y) des listes d'abscisses et d'ordonnées
    des positions successives de la particule au cours d'une marche de
    longueur n"""
    x, y = [0], [0]
    for k in range(n):
        x1, y1 = deplacement(x[-1], y[-1], p)
        x.append(x1)
        y.append(y1)
    return x, y

"""
>>> x, y = marche(1000, 1/2.) #Nord/Sud/Ouest/Est équiprobables
>>> x[-1], y[-1]
(40, -22)
>>> x, y = marche(1000, 0) #P(Nord)=P(Sud)=1/2 et  P(Ouest)=P(Est)=0
>>> x[-1], y[-1]
(0, 4)
>>> x, y = marche(1000, 1) #P(Nord)=P(Sud)=0 et  P(Ouest)=P(Est)=1/2
>>> x[-1], y[-1]
(24, 0)
"""
```

14 Marche aléatoire Question 3 a)

```
def trace_marche(n, p):
    x, y = marche(n, p)
    x = np.array(x)
    y = np.array(y)
    plt.scatter(x, y, c=range(len(x)))
    plt.colorbar() #points colorés selon leur ordre d'apparition
    plt.title('marche-aleatoire-n%d-p%1.2f.png'%(n, p))
```



```
    plt.savefig(os.path.join('jolis-dessins',
    'marche-aleatoire-n%d-p%1.2f.png'%(n, p)), dpi=80)
    plt.show()
    #plt.clf()
```

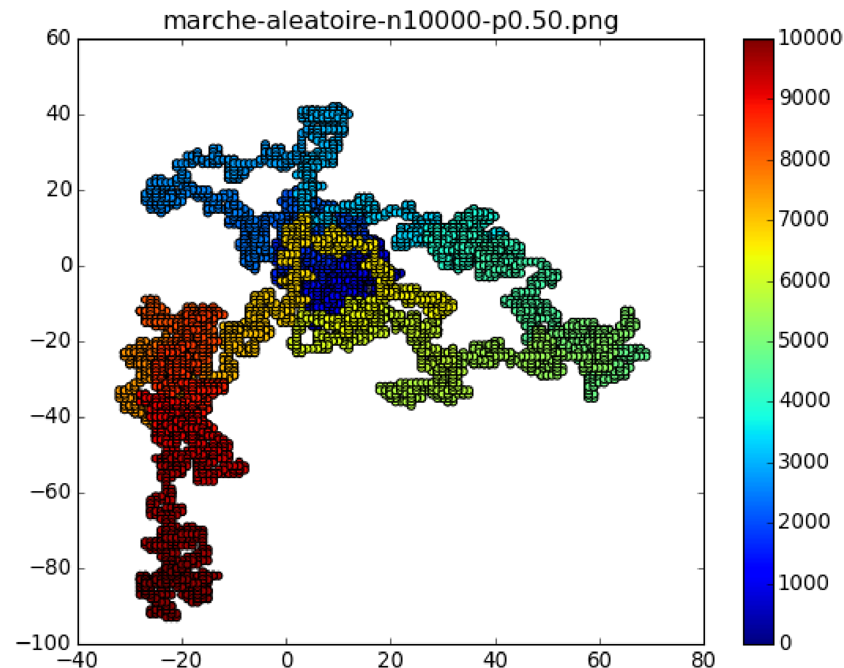


Figure 3:

On donne quelques graphiques, d'abord avec un paramètre $p = 0,5$ (loi uniforme sur l'ensemble des quatre déplacements possibles) puis un paramètre $p = 1$ (déplacement horizontal uniquement).

```
>>> for n in [500, 1000, 10000]:
...     trace_marche(n, 0.5)
```

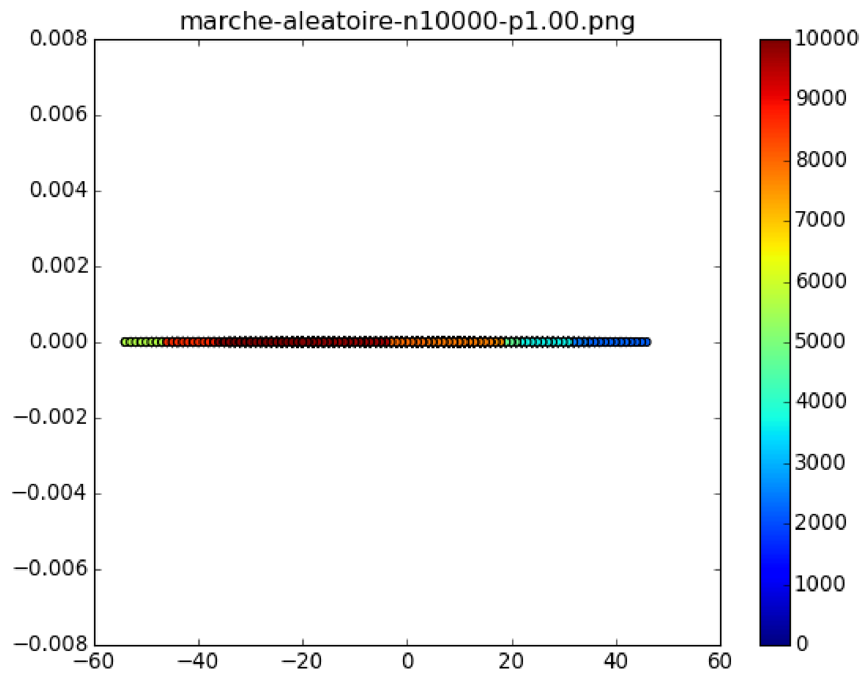


Figure 4:

15 Marche aléatoire Question 3 b)

```
def tempsretour(n, p):
    x, y = [0], [0]
    for k in range(1, n+1):
        x1, y1 = deplacement(x[-1], y[-1], p)
        x.append(x1)
        y.append(y1)
        if x1 == 0 and y1 == 0:
            return k
    return -0.1*n
```



```
def histo_retour(n, p):
    distrib = np.array([tempsretour(n, 1/2.) for _ in range(5*n)])
    effectifs, bornes, rectangles = plt.hist(distrib, bins=22) #bins est le nombre de classes
    plt.savefig(os.path.join('jolis-dessins',
    'histo-retour-origine-n%d-p%1.2f-N%d.pdf'%(n, p, 5*n)))
    plt.show()
    #plt.clf()
    return effectifs, bornes, rectangles

"""
>>> effectifs, bornes, rectangles = histo_retour(1000, 0.5)
>>> effectifs
array([ 1609.,     0.,  2753.,    198.,    102.,     ...     ,6.])
>>> bornes
array([ -100.,    -50.,      0.,     50.,    100., ...   ,  950.,   1000.])
>>> effectifs[0]/5000
0.32179999999999997  #frequence de non-retours à l'origine
"""
```