MAP 311–Aléatoire
Python pour les probabilités

Florent Benaych-Georges

Département de Mathématiques Appliquées, École Polytechnique, Palaiseau

19 avril 2017

Slides, programmes et liens :
http://www.cmap.polytechnique.fr/~benaych/aleatoire_index.html

Python

- Première version : 1991 (Guido van Rossum, Pays-Bas). Versions 2.7 et 3 : quasiment **identiques** pour ce qui nous concerne.
- **Langage de script** (haut niveau, loin du langage machine)
- **Interprété** (*bytecode* compilé en fait)
- **Multi-usage** (calcul scientifique, web, interface graphique,...)
- Syntaxe très **lisible** et **épurée**
- **Langage Open Source**, nombreux outils standard disponibles (*batteries included*), plusieurs milliers de packages disponibles dans tous les domaines
- Nombreux interpréteurs et environnements de développement disponibles (ex : **Spyder**, dans la distribution **Anaconda**)
- Communauté d'utilisateurs très active (**StackOverFlow.com**)

Softwares écrits en Python : **BitTorrent**, **Dropbox**,. . .

Langage interprété vs compilé

- Langages **interprétés** : Python, Matlab, Scilab, Octave, R, . . .
- Langages **compilés** : C, C++, Java, . . .

Vitesse d'exécution : **interprété** $<$ **compilé**

Pourquoi, dans ce cas, considérer Python pour du calcul scientifique ?

- **Temps d'implémentation** vs **temps d'exécution** : langage lisible et épuré $\Longrightarrow$ développement et maintenance rapides
- Exécution en Python : rapide si les passages critiques sont exécutés avec un langage compilé : de nombreuses fonctions sont compilées et le code est interfaçable avec C/C++/FORTRAN

En fait : *précompilation*

source → **compilateur** → **objet** → **exécuteur** → **résultat**

: Chaîne de compilation (C, C++, Java,...)

source → **interpréteur** → **résultat**

: Chaîne d'interprétation

source → **compil.** → **bytecode** → **interp.** → **résultat**

: *Pour exécuter un programme, Python charge le fichier source .py en mémoire vive, en fait l'analyse (lexicale, syntaxique et sémantique), produit le bytecode et enfin l'exécute.*

Fonctionnement

1. Ouverture de l'environnement de développement (ex : **Spyder**). *Non indispensable mais conseillé.*
2. Au choix :
   - Commande en ligne
   - Ecriture d'un programme → exécution du programme
   - Ecriture d'une fonction → chargement de la fonction → appel à la fonction (d)

Ecriture des programmes/fonctions dans un éditeur de texte quelconque.

Console IPython ou Python (Python nécessaire pour les programmes dynamiques).

Tutoriel dans la console : ? nom_fonction (d)

Types simples (1) : `bool` (booléens)

- Deux valeurs possibles : `False, True`

```
type(False) # bool
x=(1<2) # True
type(x) # bool
```

- Opérateurs de comparaison : `==, !=, >, >=, <, <=`

```
2 > 8 # False
2 <= 8 < 15 #True
```

- Opérateurs logiques : `not, or, and`

```
(3 == 3) or (9 > 24) # True
(9 > 24) and (3 == 3) # False
```

Types simples (2) : int, long, float, complex

```python
type(2**40) # int
type(2**100) # long
#long: uniquement limite par la RAM
type(3.6) # float
type(3+2*1j) # complex
20+3 # 23
20*3 # 60
(3*1j)**2 # (-9+0j)
20 ** 3 # 8000 (puissance)
20 / 3 # 6 (division entiere en Python 2)
20 // 3 # 6 (division entiere en Python 3)
20./3 # 6.666666666666667
20 % 3 # 2 (modulo)
(2+3j) + (4-7j) # (6-4j)
(9+5j).real # 9.0
(9+5j).imag # 5.0
abs(3+4j) # 5.0 : module
```

Types/conteneurs simples (3) : str (chaîne de caractères)

```
type('abc') # str
c1 = "L'eau vive"
c2 = ' est "froide" !'
c1+c2 # (concatenation)
c1*2 # "L'eau viveL'eau vive" (repetition)
c1[2] # 'e'
c1[-2] # 'v'
c1[2:5] # 'eau'
len("abc") # 3 (longueur)
"abcde".split("c") # ['ab', 'de'] (scinde)
'a-ha'.replace('a', 'o') # 'o-ho'
'-'.join(['ci', 'joint']) # 'ci-joint'
'abracadabra'.count('bra') # 2
'PETIT'.lower() # 'petit'
'grand'.upper() # 'GRAND'
```

Conteneurs simples : list

list : collection hétérogène, ordonnée et modifiable d'éléments séparés par des virgules, entourée de crochets.

```
my_list=[4,7,3.7,'E',5,7]
type(my_list) # list
len(my_list) # 6
my_list[0] # 4
range(5) # [0,1,2,3,4]
range(3,6) # [3,4,5]
range(2, 9, 2) # [2, 4, 6, 8]
my_list[1:3] # [7, 3.7]
[0,1] + [2,4] # [0,1,2,4] (concatenation)
```

Conteneur list : quelques méthodes

```
nombres = [17, 38, 10, 25, 72]
nombres.sort()
nombres # [10, 17, 25, 38, 72]
nombres.append(12)
nombres # [10, 17, 25, 38, 72, 12]
nombres.reverse()
nombres # [12, 72, 38, 25, 17, 10]
nombres.remove(38)
nombres # [12, 72, 25, 17, 10]
nombres.index(17) # 3
nombres[0] = 11
nombres[1:3] = [14, 17, 2]
nombres # [11, 14, 17, 2, 17, 10]
nombres.count(17) # 2
mean(nombres) # moyenne
std(nombres) # ecart-type
```

Par défaut, les objects en Python ne sont pas copiés.

```
x = [4, 2, 10, 9, "toto"]
y = x   # y: seulement un alias de x,
        # pas de copie effectuee ici
y[2] = "tintin" # change x(2) en "tintin"
print(x) # [4, 2, "tintin", 9, "toto"]
x = [4, 2, 10, 9, "toto"]
y = x[:]    # On demande une copie
y[2] = "tintin" # modifie y mais pas x
print(x) #  [4, 2, 10, 9, "toto"]
print(y) # [4, 2, "tintin", 9, "toto"]
```

Conteneurs simples : tuple

tuple : collection hétérogène, ordonnée, immuable.

```
t=(5,7)
type(t) # tuple
t[0] # 5
t[0]=2 # error: item assignment for tuple
```

if - [elif] - [else]

```python
if x < 0:
    print "x est negatif"
elif x % 2:
    print "x est positif et impair"
else:
    print "x n'est pas negatif et est pair"
    print "Eh oui !"
```

while

```
N = 0
x = input("Entrez un nombre positif : ")
while x > 0:
    x//=2
    N+=1
print("Approx. de log_2 du nombre : "+str(N-1))
```

for (et syntaxe des fonctions au passage)

```
for lettre in "ciao":
    print lettre,

for x in ["\n",2,'a', 3.14,"\n"]:
    print(x)
```

```
def f(n):
    print(sum([k**2 for k in xrange(n)]))
```

```
def fib(n,a=0,b=1): #0,1: val. par defaut
     '''n-th term of Fibonacci sequence...
     starting at a,b.'''#tutoriel de fib
     for i in xrange(n):
          a,b=b,a+b
     return a
```

La fonction doit être chargée (via une exécution) pour être appelée.

Bibliothèques d'usage courant en calcul scientifique

- NumPy : manipulation de tableaux numériques, fonctions mathématiques de base, simulation de variables aléatoires...
- SciPy : fonctions mathématiques plus avancées (résolution d'équations, d'équations différentielles, calcul d'intégrales...)
- Matplotlib : visualisation de données sous forme de graphiques
- scikit-learn : machine learning
- SymPy : calcul symbolique

Calcul numérique = manipulations de nombres décimaux $\neq$ calcul symbolique = manipulation d'expressions symboliques

Exemple : racines de $x^2 - x - 1 = 0$

$\hookrightarrow$ calcul symbolique : $\frac{1+\sqrt{5}}{2}, \frac{1-\sqrt{5}}{2}$

$\hookrightarrow$ calcul numérique : 1.618034, - 0.6180340

Import de bibliothèques ou de fonctions

```python
import ma_bibliotheque
# appel a une fonction de ma_bibliotheque:
ma_bibliotheque.la_fonction(...)
```

```python
import ma_bibliotheque as bibli #raccourci
# appel a une fonction de ma_bibliotheque:
bibli.la_fonction(...)
```

Moins précis (car la bibliothèque d'origine des fonctions n'est pas précisée à leur appel) :

```python
from ma_bibliotheque import la_fonction
# appel a une fonction de ma_bibliotheque:
la_fonction(...)
```

```python
from ma_bibliotheque import *
# appel a la_fonction:
la_fonction(...)
```

```
import numpy as np
b = np.array([[8,3,2,4],[5,1,6,0],[9,7,4,1]])
type(b) # numpy.ndarray
b.dtype # datatype: int
b.shape # (3,4)
c = np.array([[8,2],[5,6],[9,7]], dtype=complex)
c.dtype # datatype: complex
c[0,0] # 8+0j

#More than 2 dimensions:
d = np.array([[[8,3],[1,2]],[[5,1],[4,5]],[[9,7],[4,5]]]
d.shape # (3, 2, 2)
#Reshaping:
x = d.reshape(4,3) # tableau de taille (4,3)
d.reshape(12,1) # tableau de taille (12,1)
d.reshape(12,) # tableau unidimensionel de taille 12
np.insert(np.arange(4,9),3,17) # 4,5,6,17,7,8
```

Opérations sur les tableaux de nombres numpy

```
import numpy as np
X = np.arange(start=5,step=3,stop=16) # 5, 8,11,14
A = np.ones((2,3)) # matrix filled with ones
B = X.reshape(2,2)
C = np.zeros((3,2)) # matrix filled with zeros
D = np.eye(2) # identity matrix
np.diag([1,2]) # diagonal matrix
E = C+np.ones(C.shape) # addition: same as C+1
F = B*D # entry-wise multiplication
J=np.dot(B,D) # linear algebra product
G = F.T # transpose matrix
H = np.exp(G) #as most functions, exp is entry-wise
#(else use np.vectorize(my_function))
x = np.array([4, 2, 1, 5, 1, 10])
y=np.logical_and(x>=3, x<= 9, x!=1) # [T,F,F,T,F,F]
x[y] # [4, 5]
print(np.mean(np.random.randn(1000)>1.96))
```

Algèbre linéaire avec numpy

```
import numpy as np

A = [[2, 1, 1], [4, 3, 0]]
B = [[1, 2], [12, 0]]
C = [[1, 2], [12, 0], [-1, 2]]
D = [[1, 2, -4], [2, 0, 0], [1, 2, 3]]
E = np.bmat([[A, B], [C, D]]) # block matrix
type(E) # numpy.matrixlib.defmatrix.matrix

F = np.matrix(np.random.randn(5,5))
H = F*E # linear algebra product

B5 = np.linalg.matrix_power(B,5) # power
Bm1 = np.linalg.inv(B) # inverse
dB = np.linalg.det(B) # determinant
x = np.linalg.solve(B,[3,12])#solves B*x=[[3],[12]]
```

Analyse spectrale avec numpy

```python
import numpy as np

A = [[1, 2], [12, 3]]
x = np.linalg.eigvals(A) # eigenvalues
# eigenvalues and eigenvectors:
valp, vectp = np.linalg.eig(A)

#Hermitian matrices methods:
S = [[1, 2], [2, 3]]
y = np.linalg.eigvalsh(S) # eigenvalues
# eigenvalues and eigenvectors:
valp, vectp = np.linalg.eigh(S)

#SVD:
U,s,V=np.linalg.svd(A)
Ap = np.matrix(U)*np.diag(s)*V
print(A-Ap)
```

Copie de tableaux numpy

```
import numpy as np

x = np.array([[8,3,2],[5,1,0],[9,7,1]])
y = x
x[0,0]+=1
x[0,0]-y[0,0] # 0
z=x.copy()
x[0,0]+=1
x[0,0]-z[0,0] # 1
```

Boucles vs programmation matricielle

```
import numpy as np
from time import time

n = int(1e7)
# Methode 1. Boucle for
t1 = time()
gamma1=sum([1./i for i in xrange(1,n+1)]) - np.log(n)
t2 = time()
temps1 = t2 - t1
# Methode 2. Numpy
t1 = time()
gamma2=np.sum(1. / np.arange(1,n+1)) - np.log(n)
t2 = time()
temps2 = t2 - t1
print "Facteur de gain: ", temps1/temps2
```

↪ *éviter si possible boucles et tests en Python.*

Génération de variables aléatoires continues avec numpy

```
import numpy.random as npr
my_sample = npr.ma_loi(paramètres, taille_du_tableau)
```

- `npr.rand(d1,d2,...)` : tableau d1 x d2 x... de v.a.i. unif. sur $[0, 1]$
- `npr.uniform(low=a,high=b,size=n)` : v.a.i. unif. sur $[\mathsf{a}, \mathsf{b}[$ (`size=n` peut être remplacé par `size=(d1,d2,...)`, comme partout dans ce qui suit)
- `npr.randn(d1,d2,...)` : tableau d1 x d2 x... de v.a.i. $\mathcal{N}(0, 1)$
- `npr.multivariate_normal(mean=V,cov=C,size=n)` : vecteurs aléatoires indépendants de loi $\mathcal{N}(\mathtt{V}, \mathtt{C})$ rangés dans un tableau de taille n x N, où N est la taille de V et $N \times N$ celle de C
- `npr.exponential(scale=s,size=n)` : v.a.i. exponentielles de moyenne s

Beaucoup d'autres exemples sur
http://docs.scipy.org/doc/numpy/reference/routines.random.html

Génération de variables aléatoires discrètes avec numpy

```
import numpy.random as npr
my_sample = npr.ma_loi(paramètres, taille_du_tableau)
```

- `npr.randint(low=a,high=b,size=n)` : v.a. unif. sur $[\![a, b [\![$
- `npr.choice([a1,...,an],p=[p1,...,pn],size=n)` : tirages indép. dans `[a1,...,an]` de loi `[p1,...,pn]`
- `npr.permutation(mon_urne)` : permutation de `mon_urne`
- `npr.binomial(N,p,size=n)`
- `npr.geometric(p,size=n)`
- `npr.multinomial(n,tableau_des_probas,size=n)`
- `npr.poisson(alpha,size=n)`

Beaucoup d'autres exemples sur
http://docs.scipy.org/doc/numpy/reference/routines.random.html

Fonctions utiles en probabilités

- `np.mean(x)`, `np.std(x)`, `np.percentile(x)` : **moyenne**, **écart-type** et **percentile** d'un vecteur `x` (échantillon)
- `np.sum(x)` **somme** des valeurs de `x`
- `np.cumsum(x)` vecteur $[x_1, x_1 + x_2, \ldots, x_1 + \cdots + x_n]$ des **sommes cumulées** des coordonnées $x_1, \ldots, x_n$ de `x`
- `np.cov(x)` matrice $n \times n$ de **covariance** des lignes du tableau `x` de taille $n \times p$
- `scipy.stats` : bibliothèque proposant **densités**, **fonctions de répartition**, **quantiles**, etc... de lois classiques. Cf http://docs.scipy.org/doc/scipy/reference/stats.html
- `matplotlib.pylab` bibliothèque d'**affichage graphique**

Affichage graphique avec `matplotlib.pylab`

```
import matplotlib.pylab as plt
```

Pour `x`, `y` vecteurs de même dimension,

- `plt.plot(x,y)` affiche la **courbe** affine par morceaux reliant les points d'abscisses `x` et d'ordonnées `y` (nombreuses options)
- `plt.hist` trace un **histogramme** (spécifier `normed=True`). Deux options pour les colonnes : `bins=` *nombre de colonnes* ou `bins=` *abscisses des séparations des colonnes*
- `plt.bar` trace un **diagramme en bâtons**
- `plt.scatter(x,y)` affiche le nuage de **points** d'abs. `x` et d'ord. `y`
- `plt.stem(x,y)` affiche des **barres verticales** d'abs. `x` et hauteur `y`
- `plt.axis([xmin,xmax,ymin,ymax])` définit les **intervales** couverts par la figure
- `plt.axis('scaled')` impose que les **échelles** en `x` et en `y` soient les mêmes

Affichage graphique avec matplotlib.pylab

```
import matplotlib.pylab as plt
```

- `plt.show()` **affiche** les fenêtres créées dans le script
- `plt.figure()` crée une **nouvelle fenêtre** graphique
- `plt.title("mon titre")` donne un **titre** à une figure
- `plt.legend(loc='best')` affiche la **légende** d'un graphique (en position optimale)
- `plt.subplot` **subdivise** la fenêtre graphique de façon à y afficher plusieurs graphiques

Représentation d'un échantillon de loi discrète

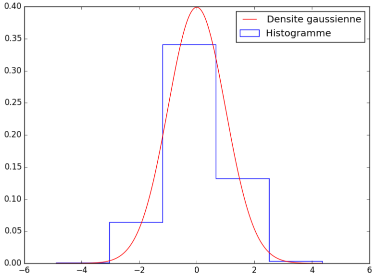
```
import numpy as np
import scipy.stats as sps
import matplotlib.pyplot as plt
plt.close("all")

n, p, N = 20, 0.3, int(1e4)
B = np.random.binomial(n, p, N)
f = sps.binom.pmf(np.arange(n+1), n, p)
plt.hist(B,bins=n+1,normed=1,range=(-.5,n+.5),\
    color="white",label="loi empirique")
plt.stem(np.arange(n+1),f,"r",label="loi theorique")
plt.legend()
plt.show()
```

Histogramme d'un échantillon de loi continue

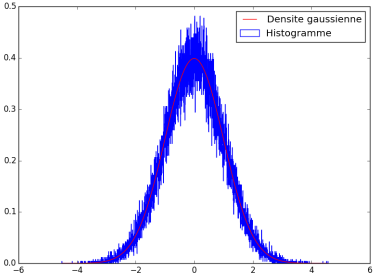
```
import numpy as np
import scipy.stats as sps
import matplotlib.pyplot as plt
plt.close("all")

E = np.random.randn(int(1e5))#echantillon

x = np.linspace(-4,4,1000)
f_x = sps.norm.pdf(x) #Densite gaussienne
plt.plot(x,f_x,"r",label="Theory")
#Affichage histo:
plt.hist(E,bins=50,normed=1,label="Data")
plt.legend(loc='best')
plt.show()
```

Cf. `RepLoiContinue.py` pour un programme plus détaillé.

Représentation d'un échantillon de loi continue : combien de colonnes dans l'histogramme ?



(a) Trop peu de colonnes



(b) Trop de colonnes

Idéal $\approx N^{1/3}$ colonnes (N : taille de l'échantillon)

Cf. `SliderHistograms.py` (à executer dans une console Python car dynamique) pour une approche empirique.

Théorème (*Loi des Grands Nombres*, Kolmogorov, 1929)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi :*

$$\mathbb{E}[|X_1|] < \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \xrightarrow[n\to\infty]{p.s.} \mathbb{E}[X_1],$$

$$\mathbb{E}[|X_1|] = \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \quad \textit{diverge p.s.}.$$

Théorème (*Loi des Grands Nombres*, Kolmogorov, 1929)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi :*

$$\mathbb{E}[|X_1|] < \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \xrightarrow[n\to\infty]{p.s.} \mathbb{E}[X_1],$$
$$\mathbb{E}[|X_1|] = \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \quad \textit{diverge p.s.}.$$

```python
import matplotlib.pyplot as plt
import numpy as np
plt.close("all")
n=int(1e3)
S=np.cumsum(np.random.rand(n))/np.arange(1,n+1)
plt.plot(range(1,n+1),S,'r',label="S_n")
plt.plot((1,n),(.5,.5),"b--",label="Esperance")
plt.ylabel('S_n'),plt.xlabel("n")
plt.legend(loc='best'),plt.title("LGN")
plt.show()
```

Théorème (*Loi des Grands Nombres*, Kolmogorov, 1929)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi :*

$$\mathbb{E}[|X_1|] < \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \xrightarrow[n\to\infty]{p.s.} \mathbb{E}[X_1],$$

$$\mathbb{E}[|X_1|] = \infty \quad \Longrightarrow \quad S_n := \frac{X_1 + \cdots\cdots + X_n}{n} \quad \textit{diverge p.s.}.$$

Soient s, U v.a. indépendantes, U uniforme sur $[0,1]$ et $s = \pm 1$ avec probas 0.5,0.5. Alors la v.a. $X := sU^{-1/\alpha}$, appelée ici α-variable aléatoire, a pour densité $(\alpha/2)\mathbb{1}_{|x|\geq 1}|x|^{-\alpha-1}$
$\hookrightarrow$ **espérance finie** si $\alpha > 1$.

Le script `SliderLGN.py` (à executer dans une console Python car dynamique) teste ce théorème pour les α-variables aléatoires pour différentes valeurs de n et de α :
$\hookrightarrow$ convergence ou non de la moyenne empirique $\frac{X_1+\cdots\cdots+X_n}{n}$ (transition à $\alpha = 1$)

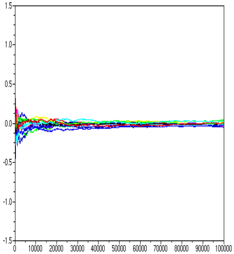
$(X_i)_{i\geq 1}$ v.a. indépendantes de même loi d'espérance $\mathbb{E}[X_1] = \mu$:

$$S_n = \frac{X_1 + \cdots\cdots + X_n}{n} \longrightarrow \mu \quad \text{si } n \to \infty.$$

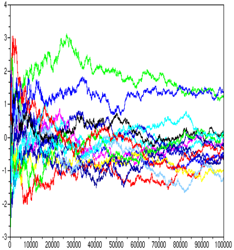
Quelle correction apporter à l'approximation $S_n \approx \mu$?

$\hookrightarrow$ on cherche $\beta > 0$ tel que $n^\beta (S_n - \mu) \longrightarrow \ell \neq 0$, i.e.

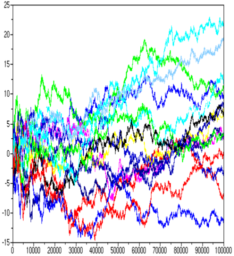
$$S_n \approx \mu + \frac{\ell}{n^\beta}$$



(c) $\beta = 0.2$



(d) $\beta = 0.5$



(e) $\beta = 0.7$

Script `SliderPrelimTCL.py` (à executer dans une console Python car dynamique)

Théorème (*Central Limite*, Laplace, 1812, Lindeberg, 1920)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi d'espérance* μ *et d'écart-type* σ *:*

$$n^{1/2}\left(\frac{X_1+\cdots\cdots+X_n}{n}-\mu\right)/\sigma \xrightarrow[n\to\infty]{loi} \mathcal{N}(0,1).$$

Théorème (*Central Limite*, Laplace, 1812, Lindeberg, 1920)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi d'espérance μ et d'écart-type σ :*

$$n^{1/2}\left(\frac{X_1+\cdots\cdots+X_n}{n}-\mu\right)/\sigma \xrightarrow[n\to\infty]{loi} \mathcal{N}(0,1).$$

Moyenne empirique des $X_i \approx$ moyenne théorique,

avec une erreur aléatoire gaussienne d'ordre $1/\sqrt{n}$

Théorème (*Central Limite*, Laplace, 1812, Lindeberg, 1920)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi d'espérance* μ *et d'écart-type* σ *:*

$$n^{1/2}\left(\frac{X_1+\cdots\cdots+X_n}{n}-\mu\right)/\sigma \xrightarrow[n\to\infty]{loi} \mathcal{N}(0,1).$$

```
import matplotlib.pyplot as plt
import scipy.stats as sps
import numpy as np
n,m,sigma=int(1e3),int(1e4),3**(-.5)
X=2*np.random.rand(m,n)-1
S=np.sum(X,axis=1)/(np.sqrt(n)*sigma)
M=max(np.abs(S))
x=np.linspace(-M,M,1000)
y=sps.norm.pdf(x)
plt.plot(x,y,'r',label="densite")
plt.hist(S,bins=round(m**(1./3)*M*.5),\
    normed=1,histtype='step',label="Histogramme")
plt.legend(loc='best'),plt.title("TCL"),plt.show()
```

Théorème (*Central Limite*, Laplace, 1812, Lindeberg, 1920)

$(X_i)_{i\geq 1}$ *v.a. indépendantes de même loi d'espérance* μ *et d'écart-type* σ *:*

$$n^{1/2}\left(\frac{X_1+\cdots\cdots+X_n}{n}-\mu\right)/\sigma \xrightarrow[n\to\infty]{loi} \mathcal{N}(0,1).$$

Soient s, U v.a. indépendantes, U uniforme sur $[0,1]$ et $s=\pm 1$ avec probas 0.5,0.5. Alors la v.a. $X := sU^{-1/\alpha}$, appelée ici α-variable aléatoire, a pour densité $(\alpha/2)\mathbb{1}_{|x|\geq 1}|x|^{-\alpha-1}$
$\hookrightarrow$ **espérance finie** si $\alpha > 1$ et **écart-type fini** si $\alpha > 2$.

Le script `SliderTCL.py` (à executer dans une console Python car dynamique) teste ce théorème pour les α-variables aléatoires pour différentes valeurs de n et de α : histogramme de l'erreur en fonction de α (transition théorique à $\alpha = 2$)

Lecture et écriture dans un fichier externe

```
x=range(5)
#Creation et ecriture:
mon_flux=open("my_data.txt","w") #w=write
mon_flux.write(str(x))
mon_flux.close()

#Ecriture a la fin d'un fichier existant:
mon_flux=open("my_data.txt","a") #a=append
mon_flux.write("\n"+str(x+1))
mon_flux.close()

#Lecture:
mon_flux=open("my_data.txt","r") #r=read
y=mon_flux.read()
print(y)
```

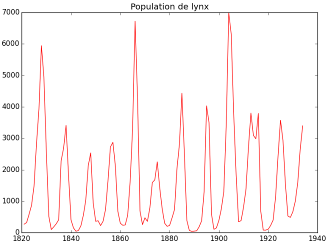
Voir aussi `np.save`, `np.load`, `np.savetxt`, `csv.reader`...

Lecture et écriture dans un fichier externe : numpy data files

```
import numpy as np

x=np.random.rand(5,3)

np.save("my_npy_file.npy",x) #creation et ecriture

x2=np.load("my_npy_file.npy") #lecture
```

Lecture dans un fichier externe : exemple

```
import numpy as np
import matplotlib.pyplot as plt
plt.close("all")
f=open("PopLynxRegionCanada_1821_1934.dat","r")
ytxt=f.readlines() # list de str
y=[int(row) for row in ytxt] # convertit str en int
plt.plot(range(1821,1935),y,"r")
plt.title("Population de lynx")
plt.tight_layout() #pratique pour l'export
plt.show()
```



Python pour l'analyse : exemples

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
def f(x):
    return np.exp(x)+x
#zeros of f, computed starting at -0.2:
a=sp.optimize.fsolve(f,-.2)
print(f(a))
#integral of f from 0 to 1:
b=sp.integrate.quad(f,0,1)
def g(y,t):
    return y
T=np.arange(start=0,stop=1,step=.001)
#solution, at T, of y'=g(y,t) starting at 1 at T[0]:
y=sp.integrate.odeint(g,1,T)
plt.plot(T,np.log(y),"r")
plt.show()
```

Bibliographie : les tutoriels/sites officiels

- Python : https://docs.python.org/2/tutorial/
- NumPy : http://docs.scipy.org/doc/numpy/reference/
- ScipyStats : http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html
- Matplotlib : http://matplotlib.org/users/pyplot_tutorial.html
- NumPy user guide (pdf) : https://docs.scipy.org/doc/numpy-1.8.0/numpy-user-1.8.0.pdf
- Matplotlib user guide (pdf) : http://matplotlib.org/Matplotlib.pdf
- scikit-learn : http://scikit-learn.org/stable/
- SymPy : http://www.sympy.org/fr/index.html
- Anaconda (distribution contenant l'interface de développement **Spyder**) : https://www.continuum.io/downloads

Bibliographie : quelques cours

- Un cours de l'X : http://www.cmap.polytechnique.fr/~gaiffas/intro_python.html
- Un très bon cours du lycée Saint Louis : http://mathprepa.fr/python-project-euler-mpsi/
- Un cours de l'INRIA : http://www.labri.fr/perso/nrougier/teaching/index.html
- Un cours d'Orsay : http://www.iut-orsay.u-psud.fr/fr/specialites/mesures_physiques/mphy_pedagogie.html