TD 20- Proposition de solutions

Solution 1 Lois usuelles

```python
import numpy.random as rd
```

⇛ Avec le générateur rand :

Loi de Bernoulli $\mathscr{B}(p)$

```python
def Ber(p):
    if rd.rand()<p:
        return(1)
    else:
        return 0
```

Loi binomiale $\mathscr{B}(n,p)$

```python
def bin(n,p):
    s=0
    for i in range(n):
        s=s+Ber(p)
    return s
```

Loi uniforme $\mathscr{U}(\llbracket a,b \rrbracket)$

```python
def unif(a,b):
    return a+int(rd.rand()*(b-a+1))
```

Loi géométrique $\mathscr{G}(p)$

```python
def geom(p):
    r=1
    while rd.rand()>p:
        r=r+1
    return r
```

Loi hypergéométrique $\mathscr{H}(N,n,p)$

```python
def hypergeo(N,n,p):
    assert N>=n,'ENC'
    g=round(N*p) # nb d'elets gagnants
    s=0 # nb de succes
    for i in range(n):
        if rd.rand()<g/N:
            s,g=s+1,g-1
        N=N-1
    return s
```

⇛ Avec le générateur choice :

```python
Ber2=lambda p:rd.choice([0,1],p=[1-p,p])

bin2=lambda n,p:sum(rd.choice([0,1],n,p=[1-p,p]))

unif2=lambda a,b:rd.choice(range(a,b+1))

def hypergeo2(N,n,p):
    assert N>=n,'ENC'
    V=round(p*N)*[1]+round((1-p)*N)*[0]
    return sum(rd.choice(L,n,replace=False))
```

Solution 2 1. On identifie la loi de N par une loi géométrique, puis la loi de X par une loi uniforme. Script de l'expérience :

```python
def experience(p):
    N=1
    while rd.rand()>p:
        N=N+1
    X=rd.randint(1,N+1)
    if X%2==0:
        print('Perdu',N,X)
    else:
        print('Gagne',N,X)
```

2. Il suffit donc de répéter l'expérience et de dénombrer les parties gagnées :

```python
def lfgn(p,nb):
    g=0
    for i in range(nb):
        N=rd.geometric(p)
        X=rd.randint(1,N+1)
        if X%2==1:
            g=g+1
    return g/nb
```

On trouve :

```
>>> lfgn(0.4,2000)
0.7225
```

Le nombre d'impair est supérieur au nombre de pair ... sans aller plus loin on admettra donc que $\mathbf{P}(A) > \frac{1}{2}$.

Solution 3

Simulation d'une v.a.d.f.

```
def Y():
    r=rd.rand()
    if r<0.2: return 1
    elif r<0.35: return 2
    elif r<0.5: return 3
    elif r<0.9: return 4
    else: return 5
```

Ce qui se généralise en :

Simulation (loi connue)

```
def X(T):
    s,r=0,rd.rand()
    for i in range(len(T[0])):
        s=s+T[1,i]
        if r<s:
            return T[0,i]
```

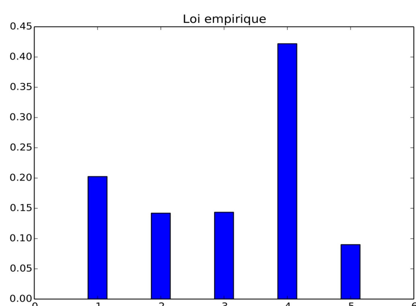
On rajoute une fonction qui détermine la loi empirique d'une v.a.d.f. dont on connaît des simulations :

Loi empirique

```
def empirique(S):
    x=[]
    for e in S:
        if not(e in x):
            x.append(e)
    x.sort()
    x,S=np.array(x),np.array(S)
    y=[sum(S==e*np.ones(np.shape(S)))/len(S)
     for e in x]
    plt.close()
    plt.bar(x,y,width=0.3,align='center')
    plt.title('Loi empirique')
    plt.savefig('empirique.pdf')
    return x,y
```

Ce qui donne :

```
S=[X(T) for i in range(2000)]
empirique(S)
```



Solution 4 • Cas d'un tirage : simulation de X

```
def tirage(n,b):
    t=0
    while rd.randint(1,n+b+1)<=n:
        n,t=n-1,t+1
    return t
```

• Utilisons la fonction tirage pour simuler la partie :

```
def partie(n,b):
    X=tirage(n,b)
    Y=rd.geometric((b-1)/(n-X+b-1))-1
    return X,Y
```

• Enfin, répétons le jeux afin d'obtenir une loi empirique satisfaisante :

```
nb=200
n,b=10,5
S=[partie(n,b) for i in range(nb)]
X=[e[0] for e in S]
Y=[e[1] for e in S]
m=max(max(X),max(Y))
plt.close()
LoiX=[sum(np.array(X)==i*np.ones(nb))/nb
        for i in range(m+1)]
LoiY=[sum(np.array(Y)==i*np.ones(nb))/nb
        for i in range(m+1)]
plt.bar(range(m+1),LoiX,color='b',width=0.4
        ,align='center',label='Loi de X')
plt.bar(range(m+1),LoiY,color='w',ec='k'
        ,width=0.2,align='center',label='Loi de Y')
plt.legend()
plt.title(str(nb)+' simulations : avec n='+str(n)
        +' et b='+str(b))
plt.show()
```

Solution 5 Fonction de répartition

```
def FR(X,P):
    plt.close()
    F=[0]
    for e in P:
        F.append(F[-1]+e)
    xmin,xmax,ymin,ymax=X[0]-2,X[-1]+2,-0.2,1.2
    plt.axis([xmin,xmax,ymin,ymax],'scaled')
    plt.plot([xmin,xmax],[0,0],'k-')
    plt.plot([0,0],[ymin,ymax],'k-')
    plt.plot([xmin,X[0]],[0,0],'b-',linewidth=3)
    plt.plot([X[-1],xmax],[1,1],'b-',linewidth=3)
    plt.plot([X[-1]],[1],'bo')
    for i in range(len(X)-1):
        plt.plot([X[i],X[i+1]],[F[i+1],F[i+1]]
                 ,'b-',linewidth=3)
        plt.plot([X[i]],[F[i+1]],'bo')
    plt.yticks(F)
    plt.grid(True)
    plt.show()
```

Solution 6 Lois usuelles

1. La loi binomiale $\mathcal{B}(n,p)$:

- initialisation : $L_0=(1,0,\cdots,0)\in\mathbb{R}^{n+1}$
- pour $k=1$, cela revient à la loi de Bernoulli :

$$L_1=(1-p,p,0,\ldots,0)$$

- pour $k,j\in[\![1,n]\!]$:

$$x_{k,0}=(1-p)x_{k-1,0} \quad \text{et} \quad x_{k,j}=px_{k-1,j-1}+(1-p)x_{k-1,j}$$

```
def loi_bin(n,p):
    L=[1]+n*[0]
    for k in range(1,n+1):
        LL=1*L
        for j in range(k+1):
            L[j]=p*LL[j-1]+(1-p)*LL[j]
    return L
```

2. La loi hypergéométrique $\mathcal{H}(N,n,p)$:

- initialisation : $L_0=(1,0,\cdots,0)\in\mathbb{R}^{n+1}$
- pour $k=1$, cela revient à la loi de Bernoulli :

$$L_1=(1-p,p,0,\ldots,0)$$

- pour $k,j\in[\![1,n]\!]$, il reste $N-k+1$ boules dans l'urne :

➢ état de l'urne sachant '$x_{k-1,0}$' : pN boules gagnantes

$$x_{k,0}=\frac{(1-p)N-k+1}{N-k+1}x_{k-1,0}$$

➢ état de l'urne sachant '$x_{k-1,j-1}$' : $pN-j+1$ boules gagnantes,

➢ état de l'urne sachant '$x_{k-1,j}$' : $pN-j$ boules gagnantes :

$$x_{k,j}=\frac{pN-j+1}{N-k+1}x_{k-1,j-1}+\left(1-\frac{pN-j}{N-k+1}\right)x_{k-1,j}$$

```
def loi_hypergeo(N,n,p):
    L=[1]+n*[0]
    for k in range(1,n+1):
        LL=1*L
        for j in range(k+1):
            L[j]=(p*N-j+1)/(N-k+1)*LL[j-1]
                        +(1-(p*N-j)/(N-k+1))*LL[j]
    return L
```

3. La loi de premier succès dans un tirage sans remise $\mathcal{SP}(N,p)$:

- initialisation : $L_0=(1,0,\cdots,0)\in\mathbb{R}^{n+1}$
- pour $k=1$, cela revient à la loi de Bernoulli :

$$L_1=(1-p,p,0,\ldots,0)$$

- pour $k,j\in[\![1,n]\!]$, il reste $N-k+1$ boules dans l'urne :

$$\forall j\in[\![0,N]\!]\setminus\{0,k\},\qquad x_{k,j}=x_{k-1,j}$$

$$x_{k,0}=\left(1-\frac{pN}{N-k+1}\right)x_{k-1,0} \quad \text{et} \quad x_{k,k}=\frac{pN}{N-k+1}x_{k-1,0}$$

```
def loi_PS(N,p):
    L=[1]+N*[0]
    for k in range(1,N+1):
        L[0],L[k]=(1-(p*N)/(N-k+1))*L[0]
                    ,(p*N)/(N-k+1)*L[0]
    return L
```

Solution 7 Illustration de la convergence en loi de la loi hypergéométrique vers la loi binomiale :

```
n,pN=10,0.3,[20,50,100,10000]
x,LB=range(0,n+1),loi_bin(n,p)
for i in range(4):
 plt.subplot(2,2,i+1)
 plt.ylim(0,0.5)
 plt.title('H({},{},{}) et B({},{})'.format(N[i]
        ,n,p,n,p))
 plt.bar(x,LB,width=0.5,label='loi binomiale'
        ,align='center')
 LHB=loi_hypergeo(N[i],n,p)
 plt.bar(x,LHB,width=0.2,color='w',ec='black'
        ,align='center',label='loi hypergeometrique')
 plt.xticks(x)
plt.legend(loc='best')
plt.show()
```

Solution 8 Autres simulations

1. Une permutation aléatoire sur $[\![0,n-1]\!]$ peut se formaliser comme la liste des entiers de 0 à $n-1$ permutée :

Permutation aléatoire

```
def permuation(n):
    L,P=list(range(n)),[]
    while len(L)>0:
        i=rd.randint(len(L))
        P.append(L.pop(i))
    return P
```

```
permutation=lambda n:list(rd.choice(range(n),
        size=n,replace=False))
```

2. Un tirage de l'euro-million, euroM() : 5 numéros sur $[\![1,50]\!]$ et deux étoiles sur $[\![1,11]\!]$

On peut commencer par créer une fonction assurant une combinaison de p éléments dans une liste :

Combinaison

```
def combi(p,L):
    assert p<=len(L),"pas_assez_d'elemnents"
    P,C=[],1*L
    for i in range(p):
        i=rd.randint(len(C))
        P.append(C.pop(i))
    P.sort()
    return P
euroM=lambda :(combi(5,list(range(1,51))),
                combi(2,list(range(1,12))))
```

```
>>> euroM()
([9, 15, 27, 29, 40], [4, 6])
```

Remarque : L'utilisation de choice donne directement ce type de simulations.

Solution 9 Cas où l'ordre compte : on tire une boule après l'autre et on affiche les numéros obtenus dans cet ordre.

Tirage avec ordre

```
def tirages(n)
    a=rd.randint(1,n+1)
    b=rd.randint(1,n)
    if b>=a: b=b+1
    return a,b
```

➢ Lorsque l'ordre ne compte pas : il suffit d'enlever l'ordre dans l'affichage de deux numéros, pour cela, on peut convenir d'afficher les numéros dans l'ordre croissant, ainsi l'ordre des tirages n'intervient plus :

Tirage sans ordre

```
def tiragesSO(n)
    a=rd.randint(1,n+1)
    b=rd.randint(1,n)
    if b>=a: b=b+1
    if a>b: a,b=b,a
    return a,b
```

Solution 10 Tirage dans une urne

```
def X(n):
    x=1
    numero=rd.randint(1,n+1)
    while numero>1:
        numero=rd.randint(1,numero)
        x=x+1
    return x
```

On complète successivement par :

- 2 : comptabilise le premier tirage
- 3 : premier tirage
- 4 : on continue tant qu'il y a des boules dans l'urne : numero>1
- 5 : nouveau tirage sachant que le boule sont numérotés de 1 à numero−1
- 6 : on comptabilise un tirage supplémentaire

Solution 11 1. Le script est :

```
def Y(n,p1,p2):
    X=0
    for i in range(n):
        if rd.rand()<p1: X=X+1
    Y=0
    for i in range(X):
        if rd.rand()<p2: Y=Y+1
    return Y
```

2. Déterminons le nombre moyen d'appels de la fonction rand() : dans la première boucle for, il y a n appels ; dans la deuxième il y a en X.
Donc le nombre cherché est : $n+\mathbf{E}(X) = n+np_1 = n(p_1+1)$.
3. Un autre script est :

Avec l'instruction binomial

```
Y2=lambda n,p1,p2:rd.binomial(rd.binomial(n,p1),p2)
```

Solution 12 Tirages dans des urnes

```
def urnes(n):
    N,X=0,1
    for i in range(n):
        if rd.randint(1,n+1)==1:
            N,X=N+1,0
    return X,N
```

On peut aussi utiliser binomial :

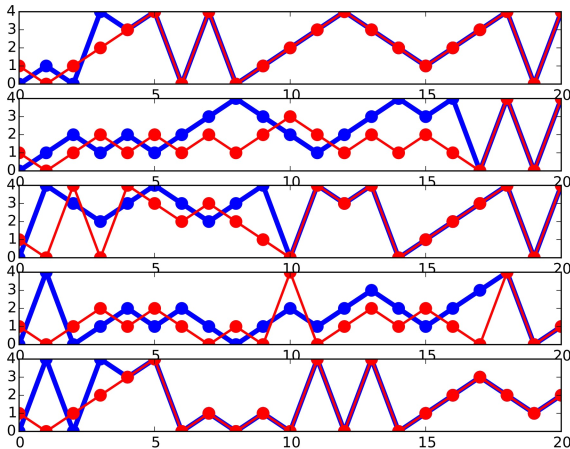
```
def urnes(n):
    N=rd.binomial(n;1/n)
    if N==0: return 1,N
    else: return 0,N
```

Solution 13 Schéma de Markov
➜ La simulation :

```
def dep(n):
    L1,L2,i=[0],[1],0
    a=lambda :rd.choice([-1,1])
    while i<n and L1[-1]!=L2[-1]:
        L1.append((L1[-1]+a())%5)
        L2.append((L2[-1]+a())%5)
        i=i+1
    while i<n:
        L1.append((L1[-1]+a())%5)
        L2.append(L1[-1])
        i=i+1
    return L1,L2
```

➜ La représentation graphique :

```
n=20
x=range(n+1)
nb=5 #nb de simulations
for i in range(1,nb+1):
    plt.subplot(nb,1,i)
    L1,L2=dep(n)
    plt.plot(x,L1,'b-',lw=4)
    plt.plot(x,L1,'bo',ms=10,mec='b')
    plt.plot(x,L2,'r-',lw=2)
    plt.plot(x,L2,'ro',ms=10,mec='r')
    plt.yticks([0,1,2,3,4])
plt.show()
```



Solution 14 1. La variable Pn retourne une approximation de la probabilité qu'aucune paire de piles ne sortent lors des n premiers lancers.
2. Notons F_i l'évènement "obtenir Face lors du ième lancer".
La famille $(F_1, \overline{F_1} \cap F_2, \overline{F_1} \cap \overline{F_2})$ est un système complet d'évènements. La formule des probabilités totales donne

$$P_n = \mathbf{P}(F_1)P_{n-1} + \mathbf{P}(\overline{F_1} \cap F_2)P_{n-2} + \mathbf{P}(\overline{F_1} \cap \overline{F_2}) \times 0$$

Ce qui donne $P_n = \frac{1}{2}P_{n-1} + \frac{1}{4}P_{n-2}$ avec $P_1 = 1$ et $P_2 = \frac{3}{4}$.
La suite (P_n) est de type récurrente linéaire d'ordre deux :

- l'équation caractéristique est : $r^2 - \frac{1}{2}r - \frac{1}{4} = 0$
- le discriminant est : $\Delta = \frac{1}{4} + 1 = \frac{5}{4}$
- les solutions sont : $r_1 = \frac{\frac{1}{2} - \frac{\sqrt{5}}{2}}{2} = \frac{1-\sqrt{5}}{4}$ et $r_2 = \frac{1+\sqrt{5}}{4}$
- il existe $\alpha, \beta \in \mathbb{R}$ tel que $P_n = \alpha r_1^{n-1} + \beta r_2^{n-1}$
- (optionnel) recherche de α et β :

$$\begin{cases} P_1 = 1 = \alpha + \beta \\ P_2 = \frac{3}{4} = \alpha\frac{1-\sqrt{5}}{4} + \beta\frac{1+\sqrt{5}}{4} \end{cases} \Leftrightarrow \begin{cases} \alpha = \frac{\sqrt{5}-2}{2\sqrt{5}} = \frac{5-2\sqrt{5}}{10} \\ \beta = \frac{5+2\sqrt{5}}{10} \end{cases}$$

Comme $r_1, r_2 \in]-1;1[$ alors $r_1^{n-1} \underset{n\to+\infty}{\longrightarrow} 0$ et $r_2^{n-1} \underset{n\to+\infty}{\longrightarrow} 0$ alors $P_n \underset{n\to+\infty}{\longrightarrow} 0$.
3. Considérons le lancer d'un dé et la non obtention d'une paire de 6. La ligne 2 deviendrait :

```
u=rd.binomial(1,1/6,size=(n,Nexp))
```

Solution 15 Cette situation relève d'un espace probabilisé infini. Il n'est pas impossible que l'expérience ne s'arrête pas, mais on peut montrer que cet évènement est négligeable. Ainsi, on ne s'inquiète pas de savoir si le programme va s'arrêter en temps fini.

A une étape donnée, il suffit de connaitre le résultat du précédent tirage pour pouvoir conclure, ainsi on ne gardera pas en mémoire tous les tirages.

Les boules ont toutes la même probabilité de sortir ; on modélise le tirage par une loi de Bernoulli de paramètre $\frac{1}{4}$:
`rand()<1/4` ou `randint(4)<1` ou `binomial(1,1/4)==1`
1. Rang de la première fois où une deuxième boule blanche consécutive est obtenue :

```
def paire():
    nb=0
    n=0
    while nb<2:
        n=n+1
        if rd.rand()<1/4:
            nb=nb+1
        else:
            nb=0
    return n
```

2. Notons Y la variable aléatoire avec $T(\Omega) = [\![2, +\infty[\![$.
Le système complet d'évènements $(N_1, B_1 \cap N_2, B_1 \cap B_2)$ et la formule des probabilités totales donnent :

$$\forall n \geq 3, \qquad y_n = \mathbf{P}(Y = n) = \frac{3}{4}y_{n-1} + \frac{3}{16}y_{n-2}$$

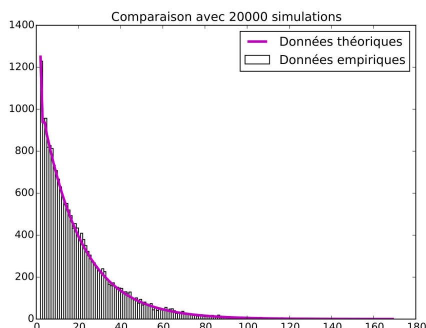
De plus, $y_2 = \frac{1}{16}$ et $y_3 = \mathbf{P}(N_1 \cap B_2 \cap B_3) = \frac{3}{32}$. L'étude cette suivante récurrente linéaire d'ordre deux donne :

$$\forall n \geq 2, \qquad y_n = \frac{7-\sqrt{21}}{224}\left(\frac{3-\sqrt{21}}{8}\right)^{n-2} + \frac{7+\sqrt{21}}{224}\left(\frac{3+\sqrt{21}}{8}\right)^{n-2}$$

Remarque : Simuler $Y-1$ par $\mathcal{G}\left(\frac{1}{16}\right)$ reviendrait à considérer les couples de tirages indépendants alors qu'ils ont un tirage en commun. Le modèle de la loi géométrique ne peut être retenu même s'il s'agit d'un temps d' attente.
3. Comparaison des données théorique et empiriques obtenue après n simulations :

```
def y(n):
 return((7-np.sqrt(21))/7/32
        *((3-np.sqrt(21))/8)**(n-2)
        +(7+np.sqrt(21))/7/32
        *((3+np.sqrt(21))/8)**(n-2))
```

```
def compare(n):
    L=[paire() for i in range(n)]
    M=max(L)
    plt.hist(L,bins=range(2,M+1),color='w',
        label='Donnees empiriques')
    plt.title('Comparaison avec '+str(n)
        +' simulations')
    G=[n*y(k) for k in range(2,M+1)]
    plt.plot(range(2,M+1),G,'m-',lw=3,
        label='Donnees theoriques')
    plt.legend()
```



Solution 16 Comme le tirage est sans remise, il est normal de diminuer le nombre de boules rouges lorsqu'une telle boule sort. En revanche, si c'est une boule bleue ou verte qui sort alors la partie s'arrête ; il est donc inutile de gérer le nombre de boules bleues et vertes.

```
def urneVBR(b,v,r):
    y=0
    x='R'
    while x=='R':
        y=y+1
        alea=rd.randint(1,n+b+r+1)
        if alea<=b:
            x='B'
        elif alea<=n+b:
            x='V'
        else:
            r=r-1
    if x=='V':
        return y,'PERDU'
    else:
        return y,'GAGNE'
```

Par exemple :

```
y,r=urneVBR(10,10,10)
print(r,'en',y,'tirage(s).')
```

Solution 17 Calcul d'une intégrale

```
def Monte_Carlo(f,a,b,n):
    n=int((b-a)*n)
    return((b-a)/n*sum([f(rd.rand()*(b-a)+a)
        for i in range(n)]))
```

Ce qui donne :

```
>>> Monte_Carlo(lambda t:4/(1+t*t),0,1,1000)
3.1484371679294014
```

⇛ Évaluation de la vitesse de convergence : effectuant pour plusieurs valeurs de n un moyenne de N simulations, on trace la courbe :

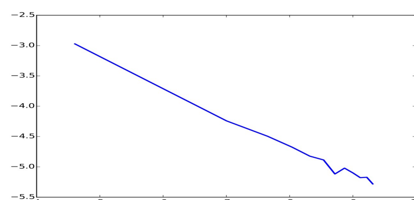
$$\left(\ln(n), \ln\left(\frac{1}{n}\sum_{k=1}^{n}|g(X_k)-\ell|\right)\right)$$

```
N=100 # nb de simulations par valeur de n
x=range(100,12000,1000)
y=[np.sum([abs(Monte_Carlo(f,0,1,n) -np.pi)
        for k in range(N)])/N for n in x]

import matplotlib.pyplot as plt
plt.close()
plt.plot(np.log(x),np.log(y),lw=2)
plt.show()
```

On trouve un pente de coefficient directeur $-\dfrac{1}{2}$.

La vitesse convergence semble donc dépendre de n en $\mathcal{O}\left(\dfrac{1}{\sqrt{n}}\right)$.



Remarque : Le **théorème centrale limite** donne un résultat analogue en introduisant des intervalles de confiance (vu en terminale).

Notons $\tilde{g_n} = \dfrac{1}{n}\sum_{k=1}^{n} g(X_k)$ alors $\ell = \mathbf{E}(\tilde{g_n})$ et

$$\frac{\sqrt{n}(\tilde{g_n}-\ell)}{\sigma_g} \hookrightarrow \mathcal{N}(0;1)$$

L'erreur pour un intervalle de confiance de risque β donne :

$$\mathbf{P}\left(|\tilde{g_n}-\ell| \leq t_{1-\frac{\beta}{2}}\frac{\sigma_g}{\sqrt{n}}\right) \geq 1-\beta$$

où $\Phi\left(t_{1-\frac{\beta}{2}}\right) = 1-\dfrac{\beta}{2}$ et Φ la fonction de répartition de la loi normale centrée réduite.

Solution 18 Fléchettes

```
flechettes=lambda n:4*np.sum([norm(rand(2))<1
        for i in range(n)])/n
```

Ce qui donne :

```
>>> flechettes(10000)
3.1444000000000001
```