Rapport TPs Probabilités et Statistiques

Justine BACHELARD

November 2018

1 TP1

1.1 Methode de la fonction inverse

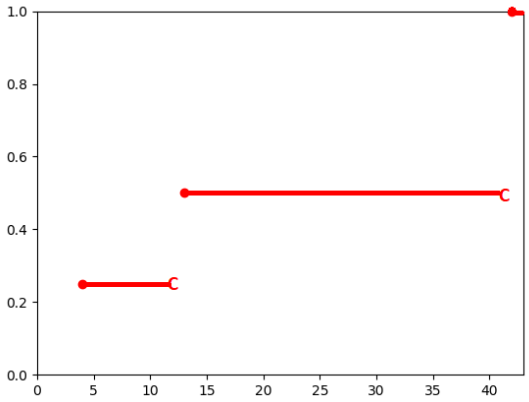
1.1.1 Decouverte

$$\psi^{-1} : \quad]0;1[\to R$$
$$t \mapsto inf(x \in R; \psi(x) \geq t)$$

1.1.2 Loi tabulée

Soient $X = \{4, 13, 42\}$ et $P(X = 4) = 0.25$, $P(X = 13) = 0.25$ et $P(X = 42) = 0.5$

Figure 1: Fonction de répartition de X



Fonction genere_loi_tabulee

```
def genere_loi_tabulee(L_e,L_p,nb):
    L_p_added = []
    L_final = []
    #On accumule les probabilités
    for i in range(0,len(L_p)):
        if i==0:
            prec=0
        else:
            prec=L_p_added[i-1]
        L_p_added.append(prec+L_p[i])
    #On choisit alors nb fois un nombre entre 0 et 1
    #(les probalités se allant jusqu'à 1)
    #Puis on cherche l'élément correspondant à cette proba.
    #Et on l'ajoute au tableau final.
    for i in range(0,nb):
        rand=random.uniform(0,1)
        for j in range (0,len(L_p_added)):
            if rand <= L_p_added[j]:
                L_final.append(L_e[j])
                break
    return L_final
```

1.1.3 Loi binomiale

Fonction genere_binomiale :

```
def genere_binomiale(n,p,nb):
    #On gère les cas d'erreur
    if (n<0 or p<=0 or p>1):
        print("erreur")
        return -1
    L_real = []
    #On réalise la loi binomiale de paramètres (n,p) nb fois.
    for i in range(0,nb):
        L_real.append(numpy.random.binomial(n, p))
    return L_real
```

1.2 Mélange

Basé sur l'exemple du tirage d'une main de jeu de cartes.
Fonction genere_arrangement :

```
def genere_arrangement(n,nb):
    #Cas d'erreur
    if (n<=0 or nb>n):
        print("erreur")
        return -1
    list = []
    main = []
    list.extend(range(0,n))
    #Nb fois
    for k in range(0,nb):
        #On échange la carte k avec une autre carte n'importe
        #où dans le paquet
        #Puis passe à la suivante
        i = random.randrange(k,n,1)
        list[i], list[k] = list[k], list[i]
        main.insert(k,list[k])
    return main
```

Une autre possible façon de faire était celle ci :

```
def genere_arrangement_wrg(n,nb):
    if (n<=0 or nb>n):
        print("erreur")
        return -1
    L_p = [1/n] * (n-1)
    return genere_loi_tabulee(range(0,n),L_p,nb)
```

Cependant, ici, on pioche sans remise, or la loi tabulée peut tirer deux fois la même carte. De plus, si l'on demande beaucoup de cartes et que l'on trie celles que l'on a déjà piochées, le programme pourrait être très long à trouver la dernière carte non piochée.

On définit un paquet de cartes comme une liste de tuple de type ("as","pique"),(2,"coeur") etc...

Fonction produit_cartesien :

```
def produit_cartesien(E1,E2):
    couples = []
    for i in E1:
        for j in E2:
            couples.append((i,j))
    return couples
```

Fonction cree_jeu :

```
def creer_jeu():
    cartes = []
    bois = ["pic","carreau","coeur","trefle"]
    val = ["as","valet","dame","roi"]
    val.extend(range(2,11))
    cartes.extend(produit_cartesien(bois,val))
    return cartes
```

Fonction genere_main :

```
#Tirer dans un jeu neuf
def genere_mains(jeu,nb_cartes,nb_mains):
    L_real = []
    #generation de nb_main mains de nb_cartes cartes
    for i in range(0,nb_mains):
        main = []
        #tirage des nb_cartes du jeu mélangé
        arrangement = genere_arrangement(len(jeu),nb_cartes)
        if arrangement != -1:
            for j in arrangement:
                main.append(jeu[j])
        else:
            print("Pb arrangement : trop de cartes demandées"+ \
            pour le jeu")
            return -1
        L_real.append(main)
    return L_real
```

1.3 Florilège de situations

1.3.1 Barb

Fonction element_from_iterable, générant une liste de n éléments de E^p où E est un itérable :

```
#n elements aléatoires de Eˆp
def element_from_iterable(E,n,p):
    if n<0 or p<0:
        print("erreur")
        return -1
    L_real = []
    L_real.extend(E)
    liste_finale = []
    for i in range(1,p):
        L_real.extend(produit_cartesien(L_real,E))
    arr = genere_arrangement(len(L_real),n)
    if arr == -1:
        return -1
```

```
    for a in arr:
        liste_finale.append(L_real[a])
    return liste_finale
```

Un élément de Barb est en réalité un arbre où l'ensemble F définit l'ensemble des feuilles possibles.
Pour créer un Barb à partir des feuilles, on peut définir trois règles et les appliquer de manière aléatoire jusqu'à ce que l'arbre se finisse.

1. On tire une feuille au hasard.
2. On renvoie () (vide), c'est la fin du barb.
3. On renvoie deux barbs, gauche et droit en générant ces derniers récursivement.

On donne les probalités souhaitées aux différentes règles, sachant que plus la troisème règle ressortira, plus l'arbre sera grand.
La probabilité du barb final est calculée par $P(arbre\_droit) * P(arbre\_gauche) * P(règle\_3)$.
Fonction genere_barb :

```
def genere_barb(feuilles):
    p_r1 = 1/4
    p_r2 = 1/4
    p_r3 = 1 - p_r1 - p_r2
    p_regles = [p_r1,p_r2,p_r3]
    r = genere_loi_tabulee([1,2,3],p_regles,1)[0]
    if r == 1:
        return [genere_loi_tabulee(feuilles,[1/len(feuilles)]*len(feuilles),1)[0],p_r1]
    elif r == 2:
        return [[],p_r2]
    else:
        a_gauche = genere_barb(feuilles)
        a_droit = genere_barb(feuilles)
        return [[a_gauche[0],a_droit[0]],p_r3*a_gauche[1]*a_droit[1]]
```

La hauteur h(x) d'un Barb étant définit par :
•$x \in F, h(x) = 1$ et $h([]) = 0$
•$A, B \in Barb, h([A, B]) = 1 + max(h(A), h(B))$
Soit une v.a X qui associe à une exécution de *genere_barb(feuilles)* l'élément barb généré. On a :
$P(X = x) \xrightarrow[h(x)\to\infty]{} 0$
Autrement dit : $\forall \xi > 0, \forall x \in Barb, \exists \eta \in N; h(x) \geq \eta \Rightarrow P(x) < \xi$

Expérimentalement, on augmente h(x) en augmentant $P(regle\_3) = 2/3$:

```
Barb : [1, []]                h=2
Proba : 0.018518518518518517

Barb : [1, [[], [[], 2]]]     h=4
Proba : 0.00022862368541380892
```

Avec ces deux simples exemples, on peut déjà voir que plus h est grand, plus la probabilité de l'arbre se rapproche de 0.

Par l'absurde, on suppose que $\exists \xi > 0, \exists x \in Barb; \forall \eta \in N; h(x) \geq \eta$ et $P(x) \geq \xi$
Pour ξ aussi petit que l'on peut (> 0) tel que, aussi bas que l'on soit dans l'arbre, on trouve P(arbre) ($P(X = x)$) $\geq \xi$.
Dans ce cas, $\sum_{i=1}^{\infty} x_i = +\infty > 1$. Or la somme des probabilités d'une v.a. X ne peut pas excéder 1. On en conclut que l'hypothèse (absurde) est fausse et celle de départ vraie.

1.3.2 Marche aléatoire

Présentation
Une marche aléatoire est défninie par L_f une liste de fonctions de E ($f_1 \circ f_2 \circ ... \circ f_n$ dans E(où E est un ensemble), où les f_i sont tirées en aléatoire uniforme dans L_f. Autrement dit, on tire un L_i au hasard et on l'applique ... n fois.
Sur une ligne :

$$L_f \left(\begin{array}{lll} aller\_a\_gauche & x \rightarrow \left| \begin{array}{lll} x-1 & si & x > 0 \\ x & sinon & \end{array} \right. \\ \\ aller\_a\_droite & x \rightarrow \left| \begin{array}{lll} x+1 & si & x < 9 \\ x & sinon & \end{array} \right. \end{array} \right)$$

Sur un tore discret :

$$L_f \left(\begin{array}{lll} aller\_a\_gauche & x \rightarrow \left| \begin{array}{lll} x-1 & si & x > 0 \\ 9 & sinon & \end{array} \right. \\ \\ aller\_a\_droite & x \rightarrow \left| \begin{array}{lll} x+1 & si & x < 9 \\ 0 & sinon & \end{array} \right. \end{array} \right)$$

Sur une grille avec rebond :

$$L_f \left(\begin{array}{lll} aller\_a\_gauche & x, y \rightarrow \left| \begin{array}{lll} x-1 & si & x > 0 \\ aller\_a\_droite & sinon & \end{array} \right. \\ \\ aller\_a\_droite & x, y \rightarrow \left| \begin{array}{lll} x+1 & si & x < 9 \\ aller\_a\_gauche & sinon & \end{array} \right. \\ aller\_en\_haut & x, y \rightarrow \left| \begin{array}{lll} y-1 & si & y > 0 \\ aller\_en\_bas & sinon & \end{array} \right. \\ \\ aller\_en\_bas & x, y \rightarrow \left| \begin{array}{lll} y+1 & si & y < 9 \\ aller\_en\_haut & sinon & \end{array} \right. \end{array} \right)$$

```
def marche_aleatoire(Lf,x,n):
    for i in range(n):
        r = random.randint(0,len(Lf)-1)
        x=Lf[r](x)
        print("X is now")
        print(x)
    return x

def marche_aleatoire_ligne(x,n):
    Lf=[lambda x : (x-1) if x>0 else x,lambda x: (x+1) if x<9 else
x]
    return marche_aleatoire(Lf,x,n)

def marche_aleatoire_tore(x,n):
    Lf=[lambda x : (x-1) if x>0 else 9, lambda x : (x+1) if x<9 else
0]
    return marche_aleatoire(Lf,x,n)

def marche_aleatoire_grille(pos,n):
    aller_gauche = lambda x,y : (x-1,y) if x>0 else aller_droite(x,y)
    aller_droite = lambda x,y : (x+1,y) if x<9 else aller_gauche(x,y)
    aller_haut = lambda x,y : (x,y-1) if y>0 else aller_bas(x,y)
    aller_bas = lambda x,y : (x,y+1) if y<9 else aller_haut(x,y)
    Lf=[aller_gauche,aller_droite,aller_haut,aller_bas]
    for i in range(n):
        r = random.randint(0,len(Lf)-1)
        pos=Lf[r](pos[0],pos[1])
    return pos
```

Souris et labyrinthe

Le but est de faire avancer une souris dans un labyrinthe donné. Le labyrinthe est définit sous cette forme : $\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & \rightarrow & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

$\rightarrow$ représente la souris, ainsi que sa direction. La souris peut se déplacer n'importe où, où un 1 est présent. Il y a $\frac{2}{3}$ chances qu'elle continue dans sa direction et $\frac{1}{3}$ qu'elle en change (divisées en le nombre de directions possibles).

Fonctions de marche aléatoire de la souris :

```
def afficher_lab(lab):
    for l in lab:
        for c in l:
            sys.stdout.write(str(c))
        print("")
    print("")


#Indique les directions possibles pour la souris
#-1,0 Nord, 1,0 Sud, 0,1 Est, 0,-1 Ouest
def where_can_mouse_go(lab_souris):
    directions = []
    for i in range(0,len(lab_souris)):
        for j in range(0,len(lab_souris[i])):
            if lab_souris[i][j] == "S":
                if j+1 < len(lab_souris[i]) and
                    lab_souris[i][j+1] == 1:
                    directions.append([0,1])
                if j-1 >=0 and lab_souris[i][j-1] == 1:
                    directions.append([0,-1])
                if i+1 < len(lab_souris) and
                    lab_souris[i+1][j] == 1:
                    directions.append([1,0])
                if i-1 >=0 and lab_souris[i-1][j] == 1:
                    directions.append([-1,0])
    return directions
```

```
#Fonction faisant avancer la souris n fois, selon les règles énoncées
#Par default, direction = Est
def avancer_souris_lab(lab,pos,n):
    lab_souris = lab.copy()
    if lab_souris[pos[0]][pos[1]] != 0:
        lab_souris[pos[0]][pos[1]] = "S" #La pos de la souris
    else:
        print("erreur de position")
        return -1
    #Directions possibles
    directions = [1,2,3,4]
    #Direction actuelle
    direction = [0,1]
    position = pos.copy()
    afficher_lab(lab_souris)

    for i in range(0,n):
        L_e = []
        L_p = []
        possible_directions = where_can_mouse_go(lab_souris)
        #Si elle peut emprunter sa direction actuelle
        #On partage les probabilités entre 2/3 pour celle ci
        #et 1/3 pour les autres possibilités
        if direction in possible_directions:
            L_e.append(direction)
            L_p.append((2/3) if len(possible_directions) != 1 \
                             else 1)
            possible_directions.remove(direction)
            for d in possible_directions:
                L_e.append(d)
                L_p.append((1/3)/len(possible_directions))
        #Sinon, on partage les probabilités entre toutes
        #les directions possibles
        else:
            for d in possible_directions:
                L_e.append(d)
                L_p.append(1/len(possible_directions))

        #On récupère la direction et on l'applique
        direction = genere_loi_tabulee(L_e,L_p,1)[0]
        lab_souris[position[0]+direction[0]] \
                        [position[1]+direction[1]] = "S"
        lab_souris[position[0]][position[1]] = 1
        position = [position[0]+direction[0],position[1]+direction[1]]
        afficher_lab(lab_souris)
    return position
```

Exemple d'exécution :

```
 avancer_souris_lab(lab,[1,0],5)
0010        0010    0010    0010    0010    0010
S111        1S11 -> 11S1 -> 111S -> 11S1 -> 111S
0010        0010    0010    0010    0010    0010
```

Demonstration automatique

Le programme générant des théorème démontrables aléatoirement conciste à appliquer un règle dès qu'elle est applicable.
Un théorème est intéressant si sa longueur est inférieure à 10. Pour enlever ce critère, il suffit d'ajouter la condition *and len(new_th)* < 10.
Génération de théorèmes aléatoire :

```
#demontre nb théorèmes de manière aléatoire
def generer_th(nb):
    graphe = {}
    th_base = "MI"
    regles = {"I":["IU","IUIU"],"II":["U"],"UU":[""],"UIIU":["I"]}
    ths = [th_base]
    #Appliquer une règle aleat. à un th choisi aléatoirement
    while len(ths) != nb+1:
        th = genere_loi_tabulee(ths,[1/len(ths)]*len(ths),1)[0]
        for r,v in sorted(regles.items(), key=lambda x: random.random()):
            if r in th:
                value = genere_loi_tabulee(v,[1/len(v)]*len(v),1)[0]
                #Replace first occurence
                new_th = th.replace(r,value,1)
                if new_th not in ths : #AND LEN(NEW_TH) <10:
                    sys.stdout.write(th+" devient "+new_th+ \
                    " grâce à la regle "+\
                    r+" -> "+value+"\n")
                    if th not in graphe:
                        graphe[th] = []
                    graphe[th].append(new_th)
                    ths.append(new_th)
                    break
    ths.remove('MI')
    print(graphe)
    return ths
```

Essais :

```
MI devient MIUIU grâce à la regle I -> IUIU
MIUIU devient MIUUIU grâce à la regle I -> IU
MIUIU devient MIUIUUIU grâce à la regle I -> IUIU
MI devient MIU grâce à la regle I -> IU
```

```
MIUUIU devient MIIU grâce à la regle UU ->
Graphe :
{'MI': ['MIUIU', 'MIU'], 'MIUIU': ['MIUUIU', 'MIUIUUIU'], 'MIUUIU':
['MIIU']}
Liste des  théorèmes générés :
['MIUIU', 'MIUUIU', 'MIUIUUIU', 'MIU', 'MIIU']
```

Pour générer des théorèmes intéressants, on pourrait également générer $2xnb$ théorèmes à partir de *generer_th(nb)* et prendre nb théorèmes de longueur inférieure à 10. $2 * nb$ pour être sûr d'avoir assez de théorèmes intéressants, sans répétition. Mais cette fonction serait plus lourde.

Tous les théorèmes peuvent être démontrés grâce au graphe généré. Pour connaître la démonstration d'un théorème en particulier, il suffit de descendre l'arbre de la racine au théorème en question.

Le taquin

Fonction réalisant un mélange du taquin : (on rappelle qu'un melange de la grille est un mélange de taquin si la permutation de cette dernière est dite paire)

```
#Fonction inndiquant si le mélange réalisé est soluble ou non
def is_soluble(taquin_list):
    n = 0
    a = taquin_list.index('V')
    pos = a
    taquin_list[len(taquin_list)-1], taquin_list[a] = taquin_list[a],
taquin_list[len(taquin_list)-1]

    n = n+1
    for i in range(len(taquin_list)-1,0,-1):
        a = taquin_list.index(i)
        taquin_list[i-1], taquin_list[a] = taquin_list[a], taquin_list[i-1]
        n= n+1
    if n % 2 == (len(taquin_list)-(pos+1)) % 2 :
        return True
    return False
```

```
#Fonction de mélange
def melange_taquin(n,p):
    m = 0
    #On créer la grille
    taquin_tab=[0]*n
    for i in range(n):
        taquin_tab[i] = [0] * p
    is_soluble_b = False
    #On la mélange jusqu'à ce que le jeu soit soluble
    while is_soluble_b != True:
        #possible si permutation paire
        list_taquin = genere_arrangement(n*p,n*p)
        list_taquin = [x+1 for x in list_taquin]
        a = list_taquin.index(n*p)
        list_taquin[a] = "V"
        is_soluble_b = is_soluble(list_taquin.copy())

    for i in range(0,n):
        for j in range(0,p):
            taquin_tab[i][j] = list_taquin[m]
            m = m + 1
    print(taquin_tab)
```

Fonction renvoyant les taquins atteignables à partir de la position pos :

```
def atteignables_taquin(taquin_tab,pos):
    list = []
    if pos[1]+1 < len(taquin_tab[0]) -1:
        list.append(taquin_tab[pos[0]][pos[1]+1])
    if pos[1]-1 >= 0:
        list.append(taquin_tab[pos[0]][pos[1]-1])
    if pos[0]+1 < len(taquin_tab) -1:
        list.append(taquin_tab[pos[0]+1][pos[1]])
    if pos[0]-1 >= 0:
        list.append(taquin_tab[pos[0]-1][pos[1]])
    return list
```

Phrases auto-référentes

Fonction renvoyant une liste du nombre de 0,1,... dans une phrase composée de chiffres :

```
def nombre_chiffre(prefixe):
    list = [0]*10
    for c in prefixe:
        list[int(c)] = list[int(c)] + 1
    return list
```

```
print(nombre_chiffre("1123345678889"))
MOTS :
[0, 2, 1, 2, 1, 1, 1, 1, 3, 1]
```

Le problème posé par une marche aléatoire dans l'espace des phrases jusqu'à obtention d'une phrase correct est un paradoxe. Par exemple, s'il y a 0 chiffre 0 dans une phrase, la phrase finale obtenue sera 0 0. Cependant, il y a maintenant un 0 de plus qui disparait lorsque l'on met à jour la phrase -¿ 1 0. Cela créer une boucle qui ne pourra pas être résolue.

2 TP2

2.1 Introduction

2.1.1 Approche graphique

Soit X, v.a de support $0, 1$, où $P(X = 1) = 0.1$. F_n est la fréquence de réalisation du 1 dans n réalisations de X.

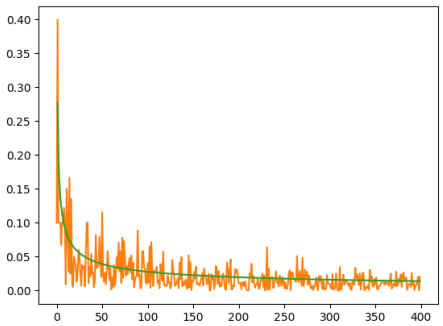
```
def f(n):
    freq = 0
    for i in range(0,n):
        res = genere_loi_tabulee([0,1],[0.9,0.1],1)[0]
        if res == 1:
            freq = freq + 1
    return freq/n

#N = 3   :   [0,0,1] R[0]=1/3   [1,1,1] R[1]=1   [0,1,1] R[2]=2/3
#-> Irrégulier, les points ne sont pas relatifs aux précédents
#On voit les points abérants, il n'influence pas la suite de la courbe
```

Représentation les courbes $|f(n) - P(X = 1)|$ et son equation approximative : $y = \frac{1}{(13*n)^{\frac{1}{2}}}$

```
def genere_n_f(n_essais_max):
    l = []
    th = numpy.arange(1,n_essais_max,1)
    y = numpy.arange(0,1,0.1)
    for i in range(1,n_essais_max+1):
        l.append(abs(f(i)-0.1))
    plt.plot(l,'C1',label='f(n)-P(X=1)')
    plt.plot(th,numpy.divide(1,numpy.power(13*th,0,5)),'C2',label='approx')
```

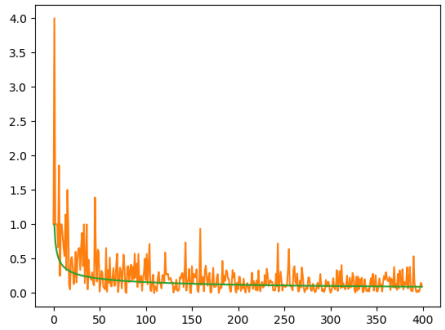
Figure 2: Représentation de 400 réalisation de $|f(n) - P(X = 1)|$



Représentation les courbes $\frac{|f(n)-P(X=1)|}{P(X=1)}$ et son equation approximative : $y = \frac{1}{n^{\frac{2}{5}}}$

```
#génere n_essais_max fn
def genere_n_f(n_essais_max):
    l = []
    th = numpy.arange(1,n_essais_max,1)
    y = numpy.arange(0,1,0.1)
    for i in range(1,n_essais_max+1):
        l.append(abs(f(i)-0.1)/0.1)
    plt.plot(l,'C1',label='f(n)-P(X=1)/P(X=1)')
    plt.plot(th,numpy.divide(1,numpy.power(th,0.4)),'C2',label='approx')
    plt.show()
```

Figure 3: Représentation de 400 réalisation de $\frac{|f(n)-P(X=1)|}{P(X=1)}$



2.1.2 Création des outils

L'égalité de Bienaymé Tchebychev affirme que si U est une v.a où $\exists E(U)$ et $\exists \sigma(U)$ alors :
$\forall \alpha \in R_+^* P(|U - E(U)| \geq \alpha) \leq \frac{\sigma^2}{\alpha^2}$.
On cherche n tel qu'on veut estimer (P=1) à $10^(- 3)$ près, avec une fiabilité d'au moins 99%.
On choisit $U = \frac{1}{n}\sum_{i=1}^{n} x_i$ où x_i est un évènement de Bernouilli associant au $i_e me$ lancé 1 ssi la réponse est 1.
On a bien alors $E(U) = P(X = 1)$ car $E(U) = \frac{1}{n}\sum_{i=1}^{n} E(x_i) = \frac{1}{n}\sum_{i=1}^{n} 1 * P(Xi = 1) + 0 * P(Xi = 0) = \frac{1}{n}\sum_{i=1}^{n} P(X = 1) = P(X = 1)$.
Pour sa variance : $\sigma^2 = \frac{np(1-p)}{n^2} = \frac{p(1-p)}{n}$
On remplace : $P(|f(n) - P(X = 1)| \geq \alpha) \leq \frac{p(1-p)}{n*\alpha^2}$.
On fixe ensuite la précision (α) et la fiabilité. On veut que $P(|f(n) - P(X = 1)| \geq \alpha) \leq risque \quad de \quad rater \iff \frac{p(1-p)}{n*\alpha^2} \leq risque$.
Normalement, on ne connait pas P(X=1). Donc il faut que ce soit vrai pour

toute valeur de p, en l'occurence, au moins pour la valeur de p où p(1-p) atteind son maximum sur [0;1]. Avec un tracé, il est facilement trouvable que pour p=0.5, p(1-p) est à son maximum de 0.25.
Il faut donc que $risque \geq \frac{0.25}{n*\alpha^2} \Leftarrow n \geq \frac{0.25}{risque*\alpha^2}$.

```
def verif_rep(risque,alpha,N,nb_tests):
    L = (f(N) for i in range(nb_tests))
    L = (abs(u-0.1) for u in L)
    L = (1 for u in L if u >= alpha)
    return sum(L)/nb_tests
```

Intervalle de prédiction :
I vérifiant $P(X \in I) \geq 1 - \alpha$, où $(1 - \alpha)$ est la fiabbilité. La précision η est la plus petite valeur vérifiant $P(|\Theta - \theta| \leq \eta) \geq 1 - \alpha$, où Θ est le paramètre de l'estimation par valeur ponctuelle θ.
La fiabilité correspond à la fréquence où l'évènement voulu arrive, et la précision à son écartement de la valeur désirée. Par exemple, "Dans 95% des cas (fiabilité), la flèche tombe à moins de 3m de la cible (précision)".

Fonction renvoyant la valeur minimale de n :
$\frac{1}{2}$ de la largeur de I $= \frac{c}{n+c^2} * \sqrt{nf*(1-f)+\frac{c^2}{4}}$
On ne connait pas f, alors on cherche pour le pire des cas, obtenu pour f=0.5 :
$\frac{c}{n+c^2} * \sqrt{n*0.25+\frac{c^2}{4}}$
La fonction consiste à calculer c (c = *norm.ppf(1-alpha/2,loc = 0,scale = 1)*), puis à résoudre $\frac{1}{2}$ de la largeur de I $= \eta$, de là on pourra en ressortir n.

```
def min_n(precision,fiabilite):
    alpha = 1 - fiabilite
    c = norm.ppf(1-(alpha/2),loc = 0,scale = 1)
    nb_it = 30
    n = 0
    freq = 0.25
    def func(x):
        return ((c/(x*c**2))*numpy.sqrt(x*freq*(x-freq)+(c**2/4)))-precision
    while(n < nb_it and n*freq < 5 and n*(1-freq) < 5):
        n = fsolve(func,nb_it+10)[0]
        nb_it = max(nb_it,n+1)
        freq=f(math.ceil(n))
    return (nb_it,freq)

print(min_n(0.95,0.01))
RuntimeWarning: The iteration is not making good progress, as measured
by the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
(41.0, 0.025)
```

La résolution n'a pas pu être faite, je suppose donc que mon code est partiellement faux.
Afin de trouver une fonction estimant la probabilité d'obtenir "oui", il faut trouver n minimum en généralisant la fonction ci-dessus, puis effectuer n fois la fonction en question pour trouver une probabilité fiable.

```
def min_n_f(precision,fiabilite,function):
    alpha = 1 - fiabilite
    c = norm.ppf(1-(alpha/2),loc = 0,scale = 1)
    nb_it = 30
    n = 0
    freq = 0.25
    def func(x):
        return ((c/(x*c**2))*numpy.sqrt(x*freq*(x-freq)+(c**2/4)))-precision
    while(n < nb_it and n*freq < 5 and n*(1-freq) < 5):
        n = fsolve(func,nb_it+10)[0]

        nb_it = max(nb_it,n+1)
        freq = 0
        for i in range(math.ceil(n)):
            if function() == "oui":
                freq = freq+1
        freq=freq/math.ceil(n)
    return (nb_it,freq)

def estim_p(precision,fiabilite,func):
    (n,freq) = min_n_f(precision,fiabilite,func)
    p = 0
    for i in range(int(n)):
        if func() == "oui":
            p = p+1
    return (p/n,int(n))


print(estim_p(0.95,0.01,bn.quota()))
(0.5365853658536586,41)
```

2.2 Jeu de cartes

2.2.1 Estimations de probabilité

Afin d'estimer ces probabilités, nous pouvons nous servir de la fonction codée *estim_p(precision,fiabilite,func)* en remplaçant *func* par la fonction ci-dessous (étant possiblement fausse, les estimations récupérées ne seront certainement pas très fiables).

Obtenir une main contenant 4 as

```
def main_4_as():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    nb_as = 0
    for c in main:
        if c[1] == 'as':
            nb_as = nb_as + 1
    if nb_as == 4:
        return "oui"
    return "non"
```

Obtenir un carré

```
def main_carre():
    tous_carres_possibles = []
    jeu = tp1.creer_jeu()
    main = tp1.genere_mains(jeu,5,1)[0]

    #On calcule tous les carrés possibles
    vals = ["as","valet","dame","roi"]
    vals.extend(range(2,11))
    for val in vals:
        memes_cartes = []
        for c in jeu :
            if val == c[1]:
                memes_cartes.append(c)
        for c in jeu:
            if c not in memes_cartes:
                tous_carres_possibles.append(tp1.produit_cartesien([memes_cartes],[c]))

    #On vérifie si la main est dans la liste des carrés possibles
    # if main in tous_carres_possibles ne suffit pas, car
    # elle prend en compte l'ordre des cartes
    for main_ca in tous_carres_possibles:
        same = True
        for carte in main_ca:
            if carte not in main:
                same=False
                break
        if same:
            return "oui"
    return "non"
```

Obtenir au moins 2 as

```
def main_2_as_min():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    nb_as = 0
    for c in main:
        if c[1] == 'as':
            nb_as = nb_as + 1
    if nb_as >= 2:
        return "oui"
    return "non"
```

Obtenir au moins 2 as, dont l'as de coeur

```
def main_2_as_min_avc_ascoeur():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    nb_as = 0
    for c in main:
        if c[1] == 'as':
            nb_as = nb_as + 1
    if nb_as >= 2 and ('coeur','as') in main:
        return "oui"
    return "non"
```

Obtenir au plus 2 as

```
def main_2_as_max():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    nb_as = 0
    for c in main:
        if c[1] == 'as':
            nb_as = nb_as + 1
    if nb_as <= 2:
        return "oui"
    return "non"
```

Obtenir uniquement du coeur

```
def main_uniquement_coeur():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    for c in main:
        if c[0] != 'coeur':
            return "non"
    return "oui"
```

Obtenir au moins 2 coeurs et 1 as

```
def main_2_coeur_1_as_min():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    nb_as = 0
    nb_coeur = 0
    for c in main:
        if c[0] == 'coeur':
            nb_coeur = nb_coeur + 1
        if c[1] == 'as':
            nb_as = nb_as + 1
    if nb_as >= 1 and nb_coeur >=2:
        return "oui"
    return "non"
```

Obtenir uniquement des figures

```
def main_uniquement_figures():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    for c in main:
        if isinstance(c[1], int) or c[1] == 'as':
            return "non"
    return "oui"
```

Obtenir une grande suite

```
def main_grande_suite():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    #Definir les valeurs possibles dans l'ordre
    #en remplaçant les figures et l'as par leur numéro.
    #L'as pouvant être en première et derniere position,
    #il est a début et à la fin
    vals = []
    vals.extend(range(1,14))
    vals.append(1)

    #On transforme toutes les valeurs de la main en nombres
    main = [(x[0],1) if x[1]=='as' else \
        (x[0],11) if x[1]=='valet' else \
        (x[0],12) if x[1]=='dame' else \
        (x[0],13) if x[1]=='roi' else x for x in main]
    #On les trie dans l'ordre croissant, comme vals
    main.sort(key=lambda x: x[1])
    #On regarde si l'ordre de la main et vals sont les mêmes
    #Si c'est le cas pour toute la main, c'est une suite
    j = vals.index(main[0][1])
    for i in range(1,5):
```

```
        j = j+1
        if main[i][1] != vals[j]:
            return "non"
    return "oui"
```

Obtenir une grande suite de valeur dans le même bois

```
def main_grande_suite_bois():
    main = tp1.genere_mains(tp1.creer_jeu(),5,1)[0]
    vals = []
    vals.extend(range(1,14))
    vals.append(1)
    main = [(x[0],1) if x[1]=='as' else \
        (x[0],11) if x[1]=='valet' else \
        (x[0],12) if x[1]=='dame' else \
        (x[0],13) if x[1]=='roi' else x for x in main]
    main.sort(key=lambda x: x[1])
    j = vals.index(main[0][1])
    for i in range(1,5):
        j = j+1
        #Si les nombres ne se suivent pas,
        #Ou si le bois et différent du précédent
        if main[i][1] != vals[j] or main[i][0] != main[i-1][0]:
            return "non"
    return "oui"
```

3 TP3

3.1 Premiers Pas

3.1.1 Deviner l'impossible ?

Le générateur 'impossible' renvoie un entier dans [1..N], N étant inconnu. Le but est de générer plusieurs estimateurs de N et determiner le plus efficace. Définissons θ comme le renvoie maximum de *impossible* sur 10000 essais.

```
def n_realisations(n):
    return [etu.impossible() for i in range(n)]

#Estimation theta
N = max((etu.impossible() for i in range(10000)))

Le N max est :
1102611
```

biais = $E(estimateur(n)) - N$
EQM = ecart moyen entre ce que je veux estimer et ce que j'estime
EQM = $E((estima(n) - N)^2)$

Estimateur $\phi = 2 * moyenne \quad réalisations - 1$

```
esti = []
for i in range(10):
    reals = n_realisations(20)
    esti.append(2*np.mean(reals)-1)
print("ESTIMATION 1")
print(esti)
print(np.mean([abs((esti[i] - N)) for i in range(10)]))
biais = np.mean([abs((esti[i] - N))/N for i in range(10)])
eqm = ((np.mean(esti) - N)*(np.mean(esti) - N))/N

#esti
[1314095.1, 1195764.1, 934138.6, 986661.3, 1205886.5, 1077408.0, 1321586.3,
1216566.8, 1359463.4, 1373024.5]
#biais
157773.48000000004
#biais relatif à N et EQM
BIAIS : 0.1430907908591516 EQM : 8331.974997865605
```

Estimateur $\psi = 2 * médianne \quad réalisations$

```
esti = []
for i in range(10):
    reals = n_realisations(20)
    esti.append(2*np.median(reals))
```

```
print("ESTIMATION 2")
print(esti)
biais = np.mean([abs((esti[i] - N))/N for i in range(10)])

[1054765.0, 645356.0, 1189462.0, 1172052.0, 1097296.0, 1070621.0, 1089049.0,
1108082.0, 1102166.0, 1020170.0]
#biais relatif
BIAIS : 0.07261101149906904
```

$Estimateur\ \zeta = max(moyenne \quad réalisations)$

```
esti = []
for i in range(10):
    reals = n_realisations(20)
    esti.append(2*np.mean(reals)-1)
print("ESTIMATION 3")
esti = max(esti)
print(esti)
biais = (abs(np.mean(esti) - N))/N

#esti
661075.94
#biais relatif à N
BIAIS : 0.40044506794045354
```