Travaux Pratiques - Probabilités

Dylan TROLES & Hugo POULIQUEN

4 mai 2016

Chapitre 1

Génération de nombres pseudo-aléatoires

1.1 Approche fréquentielle des probabilités

1.1.1 Le lancer de 5 dés

Une épreuve consiste à lancer 5 dés et à considérer la somme amenée à chaque lancer. Construire un programme qui donne :

1. Les valeurs de la somme, le nombre de sorties possibles pour chaque valeur, et la fréquence de chaque sortie.
2. Un tableau et un graphique donnant les fréquences de chaque sortie.

Pour cet exercice, nous avons fait un script python. Pour pouvoir lancer ce script, il est nécessaire d'installer la librairie matplotlib.

Listing 1.1 – Lancer_de_5_des.py

```
#!/usr/bin/python3
from random import randrange
import matplotlib.pyplot as plt

# Usage : La fonction roll_dice permet de lancer un nombre de des
# Param : nbr est le nombre de des a lancer
# Retour : La fonction retourne un tableau avec la valeur des des lances
def roll_dice(nbr):
    throw_value = []
    for i in range(nbr):
        throw_value.append(randrange(1, 7))
    return throw_value

# Usage : La fonction dice_sum effectue la somme du resultat de chaque des
# Param : - values
```

```
#         - nbr est le nombre de des lance
# Retour : La fonction retourne la somme total des des
def dice_sum(nbr, values):
    score = 0
    for i in range(nbr):
        score = score + values[i]
    return str(score)

# Usage : La fonction possibilities calcul le nombre de possibilites pour une
#         somme de des donnee apparaisse
# Param : - la somme a traiter
# Retour : La fonction retourne le nombre de possibilites de combinaison de des
# pour cette somme
def possibilities(sum_values):
    possibility = 0
    dices = [1, 2, 3, 4, 5, 6]
    for i in dices:
        for j in dices:
            for k in dices:
                for l in dices:
                    for m in dices:
                            res = i + j + k + l + m
                            if res == sum_values:
                                possibility += 1
    return possibility

# Usage : La fonction possible_sum calcul le nombre de somme possible
# Param : - Le nombre de des a traiter
# Retour : La fonction retourne le nombre de somme possible lors du lancement
#          de ce nombre de des (ici 5 des)
def possible_sum(dices_nbr):
    return str((6*dices_nbr)  - 5)

# Usage : La fonction possible_combinations calcul le nombre decombinaison pour
#         chaque somme possible
# Param : - Le nombre de des a traiter
#         - Les valeurs des des lances
# Retour : La fonction retourne le nombre de combinaisons possible
def possible_combinations(dices_nbr):
    values = []
    outputs = []
    j = dices_nbr
    for i in range(dices_nbr, (6*dices_nbr) + 1):
        values.append(possibilities(i))

    for i in values:
```

```python
            print(str(j) + ' : ' + str(i))
            j += 1
            outputs.append(j)
    return outputs, values

# Usage : La fonction frequencies calcul la frequence de chaque sortie
# Param : - Le nombre de des a traiter
#         - Les valeurs des des lances
# Retour : La fonction retourne les frequences de chaque sortie
def frequencies(dices_nbr, values):
    freqs = []
    k = dices_nbr
    for i in range(len(values)):
        frequencesres = 100*(values[i] / (6**dices_nbr))
        freqs.append(frequencesres)
        print(str(k) + ' : ' + str(frequencesres))
        k += 1
    return freqs

dices_nbr = 5 # Nombre de des

print('\nTraitement pour : ' + str(dices_nbr) + ' des comportant 6 faces de 1 a 
print('Nombre de sommes possibles : ' + possible_sum(dices_nbr))
print('\n-> Lancement des des...')
print('Somme des des lances : ' + dice_sum(dices_nbr, roll_dice(dices_nbr)))
print('\nTableau du nombre de combinaisons possibles pour chaque somme :')
outputs, values = possible_combinations(dices_nbr)
print('\nTableau des frequences de chaque somme :')
freqs = frequencies(dices_nbr, values)

# CREATION DU GRAPHIQUE
plt.title("Frequence d'apparition")
plt.plot(outputs, freqs)
plt.xlabel('Sorties possibles')
plt.ylabel('Frequence (%)')
plt.show()
```

Voici le résultat généré par le script :

```
$ python3 lancer_de_5_des.py

Traitement pour : 5 des comportant 6 faces de 1 a 6
Nombre de sommes possibles : 25

-> Lancement des des...
Somme des des lances : 21
```

Tableau du nombre de combinaisons possibles pour chaque somme :

```
5 : 1
6 : 5
7 : 15
8 : 35
9 : 70
10 : 126
11 : 205
12 : 305
13 : 420
14 : 540
15 : 651
16 : 735
17 : 780
18 : 780
19 : 735
20 : 651
21 : 540
22 : 420
23 : 305
24 : 205
25 : 126
26 : 70
27 : 35
28 : 15
29 : 5
30 : 1
```

Tableau des frequences de chaque somme :

```
5 : 0.01286008230452675
6 : 0.06430041152263374
7 : 0.19290123456790123
8 : 0.45010288065843623
9 : 0.9002057613168725
10 : 1.6203703703703702
11 : 2.6363168724279835
12 : 3.9223251028806585
13 : 5.401234567901234
14 : 6.944444444444445
15 : 8.371913580246913
16 : 9.452160493827162
17 : 10.030864197530864
18 : 10.030864197530864
19 : 9.452160493827162
20 : 8.371913580246913
```

```
21  :  6.944444444444445
22  :  5.401234567901234
23  :  3.9223251028806585
24  :  2.6363168724279835
25  :  1.6203703703703702
26  :  0.9002057613168725
27  :  0.45010288065843623
28  :  0.19290123456790123
29  :  0.06430041152263374
30  :  0.012860082304526755
```

Voici le graphe généré :

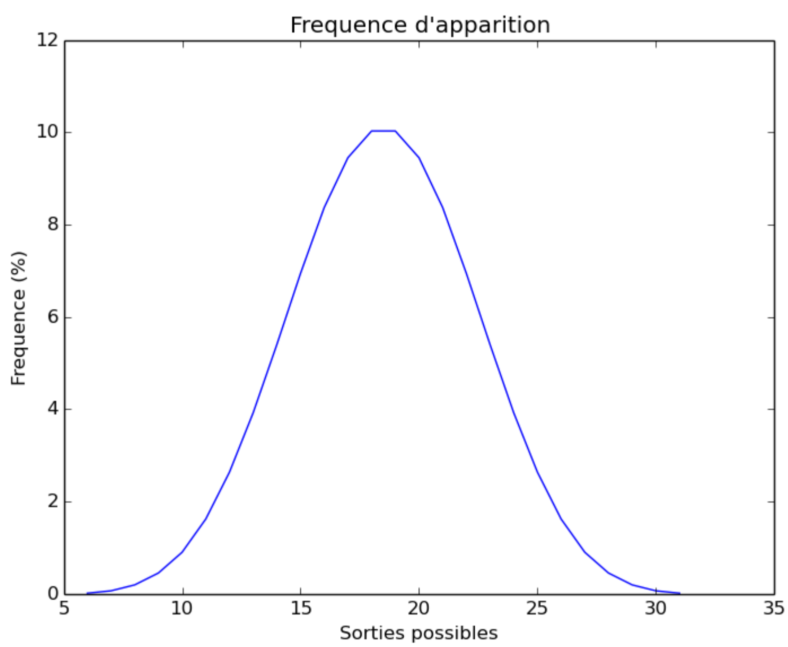

Figure 1.1 – Sorties possibles en lançant 5 dés différents en 1 fois

1.2 Nombres pseudo aléatoires

Exercice 1 :

1. Votre machine a une fonction RAND qui vous fournit de manière aléatoire un nombre de [0,1[. Donner un algorithme utilisant la fonction RAND fournissant un nombre entier entre 1 et N.

2. Pouvez-vous être satisfait de la fonction fournie par le constructeur ?

Listing 1.2 – Nombre_pseudoaleatoires_exo1.py

```
#!/usr/bin/python3
import random
import matplotlib.pyplot as plt

N = 10
nbr_loop = 10000
results = {}
sorties = []

# Boucle realisant 100 000 tirages "aleatoires" entre 1 et 10
for i in range(nbr_loop):
    result = random.randrange(0, N + 1)
    sorties.append(result)

# CREATION DU GRAPHIQUE
plt.title("Sortie␣en␣fonction␣de␣leur␣nombre␣d'apparition")
plt.hist(sorties, N)
plt.axis([0, N, 0, 0.20*nbr_loop])
plt.xlabel('Sorties')
plt.ylabel("Nombre␣d'apparitions")
plt.show()
```

Réponse : Au vu du graphe, l'aléatoire de la fonction random.randrange donnée par le constructeur n'est pas satisfaisant. Chaque barre devrait avoir une probabilité de $\frac{1}{N}$ car nous sommes dans l'ensemble $\{1, ..., N\}$. Dans notre exemple, N = 10. Pour chaque barre, $\frac{1}{N}$ devrait donc être égale à 0,10 .

Prenons la première barre, on remarque que la valeur 1 est sortie 918 fois Calculons sa probabilité : $\frac{918}{10000} = 0.0918$

Prenons la dernière barre, on remarque que la valeur 10 est sortie 1736 fois Calculons sa probabilité : $\frac{1736}{10000} = 0.1736$

Les probabilités d'apparitions de chaque valeur ne respecte donc pas 0.10. En conclusion, la fonction random.randrange n'est pas aléatoire et pour notre exemple, la valeur 10 apparait le plus souvent.

Voici le graphe généré par le script précédent :

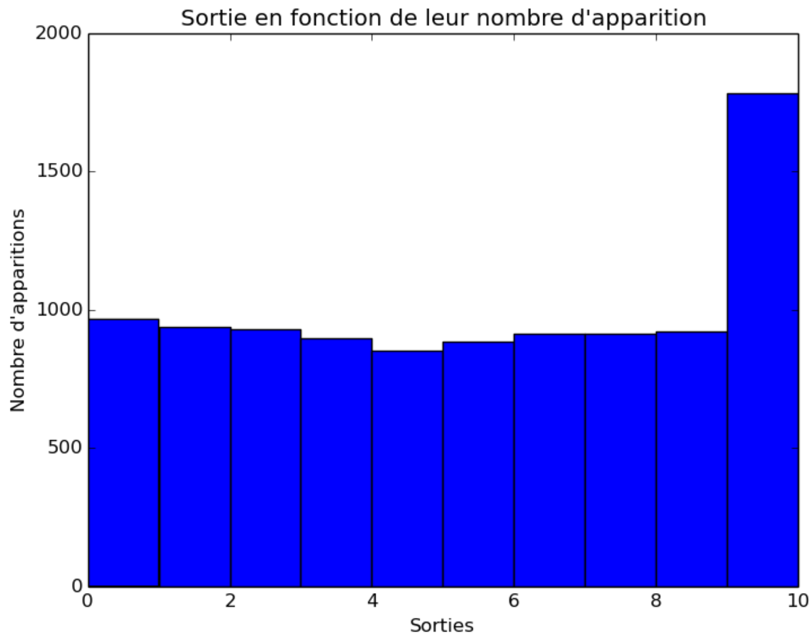

FIGURE 1.2 – Sortie en fonction de leur nombre d'apparition

Exercice 2 : Voici un programme écrit dans le langage algorithmique TestAlgo. Ce petit programme peut être amélioré.

```
algo Nombres aleatoires
var
  entier i,z;
principal
  debut
          i:=1
                tantque (i<=100)
                  z:=arrondi(3*aleatoire());
                        afficher(z)
                        i:=i+1;
                fintantque
        fin
```

Que réalise ce programme ? Qu'en pensez-vous ?

Réponse :

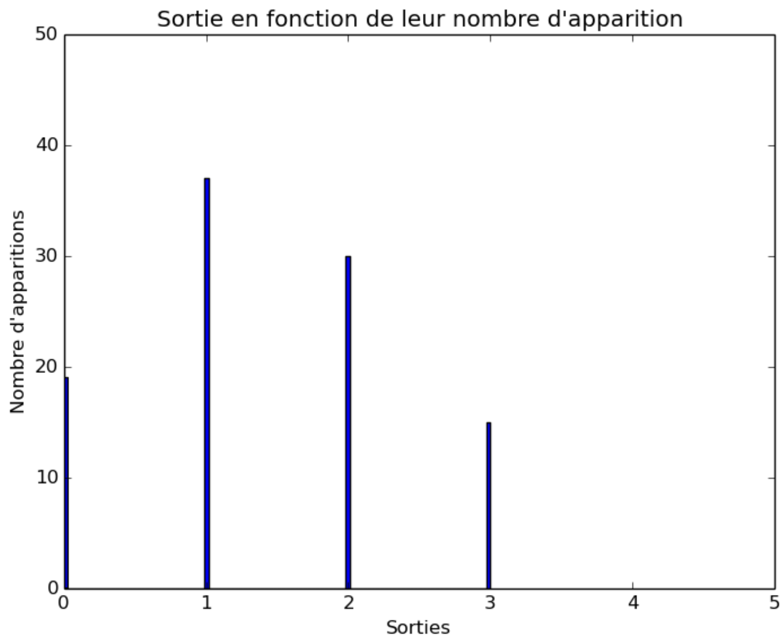

FIGURE 1.3 – Sortie en fonction de leur nombre d'apparition

Ce programme donne un nombre entier entre 0 et 3.

Au vu du graphe nous pensons que cet algorithme se rapproche d'un algorithme aléatoire mais n'en est pas un. Pour le prouver calculons la probabilité théorique de chaque valeur :

$\frac{1}{N+1} = 0.25$ avec N = 3

Voici les fréquences observées à partir du graphe dans l'expérimentation

0 : 0,15

1 : 0,32

2 : 0,33

3 : 0,20

Cette fois, les probabilités obtenues se rapprochent un peu plus de la probabilité théorique. L'aléatoire n'est toujours pas uniforme.

1.3 Génération de nombres aléatoires

1.3.1 Générateurs congruentiels linéaires

Exercice 3 : Donner la suite des nombres aléatoires et en rechercher les périodes pour :

1. b = 3, X_0 = 7, a = 5, N = 16

```
        -- b = 3, X0 = 7, a = 5, N = 16--
n  = 0 1 2 3 4  5  6 7  8  9 10 11 12 13 14 15 16
Xn = 7 6 1 8 11 10 5 12 15 14 9 0  3  2  13 4  7
La periode est de : 16
```

2. b = 6, X_0 = 7, a = 5, N = 16

```
        -- b = 6, X0 = 7, a = 5, N = 16--
n  = 0 1 2 3 4  5 6  7  8 9  10 11 12 13 14 15 16
Xn = 7 9 3 5 15 1 11 13 7 9  3  5  15 1  11 13 7
La periode est de : 8
```

3. b = 0, X_0 = 1, a = 2, N = 17

```
        -- b = 0, X0 = 1, a = 2, N = 17--
n  = 0 1 2 3 4  5  6  7 8 9 10 11 12 13 14 15 16 17
Xn = 1 2 4 8 16 15 13 9 1 2 4  8  16 15 13 9  1  2
La periode est de : 8
```

4. b = 6, X_0 = 1, a = 2, N = 17

```
        -- b = 6, X0 = 1, a = 2, N = 17--
n  = 0 1 2 3  4 5  6 7 8 9 10 11 12 13 14 15 16 17
Xn = 1 8 5 16 4 14 0 6 1 8 5  16 4  14 0  6  1  8
La periode est de : 8
```

5. b = 0, X_0 = 3, a = 13, N = 2^7

```
        -- b = 0, X0 = 3, a = 13, N = 128--
La suite est trop importante pour etre affichee

La periode est de : 32
```

6. b = 0, X_0 = 4567, a = 9749, N = 2^{17}

```
        -- b = 0, X0 = 4567, a = 9749, N = 131072--
La suite est trop importante pour etre affichee

La periode est de : 32768
```

Ces résultats sont issus du script suivant :

Listing 1.3 – Nombre_pseudo-aleatoires_exo3.py

```
#!/usr/bin/python3

# Usage : La fonction congru generator permet de generer des congruences lineair
# Param : les parametres sont ceux donnes par l'exercice pour le calcul des cong
# Sortie: La sortie est une suite de nombres aleatoires avec la periode
def Congru_generator(b, X, a, N, display = 1):
    first = True
    p = 0
    n_res = []
    Xn_res = []
    print('\n--␣b␣=␣' + str(b) +',␣X0␣=␣' + str(X) + ',␣a␣=␣' + str(a) \
        + ',␣N␣=␣' + str(N) + '--')

    X = first_element = X
    Xn_res.append(first_element)  # Traitement de X0
    n_res.append(0)
    X = ((a*X)+b) % N
    Xn_res.append(X)  # Traitement de X1
    n_res.append(1)
    for j in range(2, N + 1): # Boucle pour generer chaque nombre
        X = ((a*X)+b) % N
        if(X == first_element and first is True): # Recherche de la periode
            p = j
            first = False
        n_res.append(j)
        Xn_res.append(X)
    if(display != 0): # Option pour ne pas afficher des logs en permanence
        print('n␣␣=', end="␣")
        for k in n_res: # Affichage du resultat en ligne
            print (str(k), end="␣")
        print('\nXn␣=', end="␣")
        for l in Xn_res:
            print (str(l), end="␣")
    else:
        print('La␣suite␣est␣trop␣importante␣pour␣etre␣affichee␣dans␣la␣console')
    print('\nLa␣periode␣est␣de␣:␣' + str(p))

Congru_generator(3, 7, 5, 16)
Congru_generator(6, 7, 5, 16)
Congru_generator(0, 1, 2, 17)
Congru_generator(6, 1, 2, 17)
Congru_generator(0, 3, 13, 2**7, 0)
Congru_generator(0, 4567, 9749, 2**17, 0)
```

Exercice 4 :

1. Écrire une fonction Scilab qui pour argument de sortie une suite $(U_n)_n \in$ **N** de réels compris entre 0 et 1, générée par un générateur congruentiel linéaires, et pour argument d'entrée X_0, a, b et N, les paramètres usuels d'un tel générateur.
   Réponse :

Listing 1.4 – Exercice4.sce

```
function [Liste]=generateur_un(b,x,a,n)
        Liste = list(); //Creation de liste
        X = x;
        U = X/n; //Calcul de U0
        Liste($+1) = U; //U0 tete de liste
        for i=1:(n-1)
                y=a*X+b;  //Calcul Xn suivant l'expression
        X=modulo(y,n);  //Enonce
        U=X/n;  //Calcul Un
        Liste($+1) = U; //Ajout Un dans liste
        i=i+1;
    end
endfunction
```

2. En utilisant la fonction écrite en Scilab, étudiez les générateurs qui suivent : on regardera, en particulier, et si c'est possible, la période.
   — Pour a = 25, b = 16 et N = 256 = 2^8 et pour X_0 successifs : $X_0 = 12$; $X_0 = 11$; $X_0 = 0$
   — **Turbo-Pascal** a = 129, b = 907633385 et N = 2^{32}
   — **Unix** a = 1103515245, b = 12345 et N = 2^{32}
   — **Matlab** a = 19807, b = 0 et N = 2^{31}-1

1.4 Autres méthodes

1.4.1 Méthode de VON NEUMANN

Comment fonctionne cette méthode ?

— On se donne un nombre A de N chiffres
— On l'élève au carré
— On choisit comme nombre suivant le nombre de N chiffres formé par la tranche du milieu du carré obtenu
— On divise ensuite par 10^N

1. Montrer que l'on obtient bien une suite de nombres compris entre 0 et 1 (1 exclu)
   Réponse : A est un nombre de N chiffres.
   $10^{N-1} \leq A < 10^N$
   On élève au carré : $10^{2N-2} \leq A^2 < 10^{2N}$

B est le nouveau nombre formé par la tranche du milieu du carré obtenu :
$10^{N-1} \leq \mathrm{B} < 10^N$
On divise par 10^N : $0{,}1 \leq \frac{B}{10^N} < 1$

2. Donner un algorithme fournissant ces nombres
Réponse :

Listing 1.5 – VonNeumann.py

```
#!/usr/bin/python3
import matplotlib.pyplot as plt
import math

def VonNeumann(N, A, display):
        if not N:
            # Calcul de la longueur du nombre
            N = len(str(A))
        square = A * A
        # Calcul de la longueur du carre du nombre
        square_len = len(str(square))
        part = square_len * 0.25
        square = str(square)
        # On sauvegarde la tranche du milieu du carre
        middle_slice = square[int(part):int(part+N)]
        # On divise cette tranche par 10^N
        res = float(middle_slice) / float(10**N)

        if(display == 1):
            print(  "----\nA␣:␣" + str(A)+ "\nN␣:␣" + \
                    str(N) + "\nCarre␣de␣A␣:␣" + \
                    str(square) + "\nTranche:␣" + \
                    str(middle_slice) + "Resultat␣:␣" + \
                    str(res)
            )
        return res

VonNeumann(4, 5678, 1)
VonNeumann(6, 123456, 1)

outputs = []
loop_nbrs = 100
N = 10
for i in range(loop_nbrs):
    outputs.append(math.trunc(VonNeumann(None, i, 0)*N+1))

plt.title("Sortie␣en␣fonction␣de␣leur␣nombre␣d'apparition")
plt.hist(outputs, loop_nbrs)
plt.axis([0, N , 0, 0.2*loop_nbrs])
```

```
plt.xlabel('Sorties')
plt.ylabel("Nombre d'apparitions")
plt.show()
```

3. Faire une étude pour N = 4, et A = 5678
   Réponse :

```
A=5678
N=4
Carre de A=32239684
Tranche=2396
Resultat=0.2396
```

4. Faire une étude pour N = 6, et A de votre choix
   Réponse :

```
A=123456
N=6
Carre de A=15241383936
Tranche=241383
Resultat=0.241383
```

5. Que penser de cette méthode ?
   Réponse :
   Pour 100 lancés, on obtient les résultats de la figure suivante. Ces résultats montrent que l'aléatoire de VonNeumann n'uniformise pas les numéros de sortie. Si l'on compare avec la fonction random de Python (cf Ex1), VonNeumann est nettement plus aléatoire. La différence entre les apparitions de chaque valeur est moins importante que dans l'analyse de l'exercice 1.
   Cette méthode est plus aléatoire que la fonction random de Python, mais n'est pas uniforme car toutes les valeurs n'ont pas le même nombre d'apparitions.

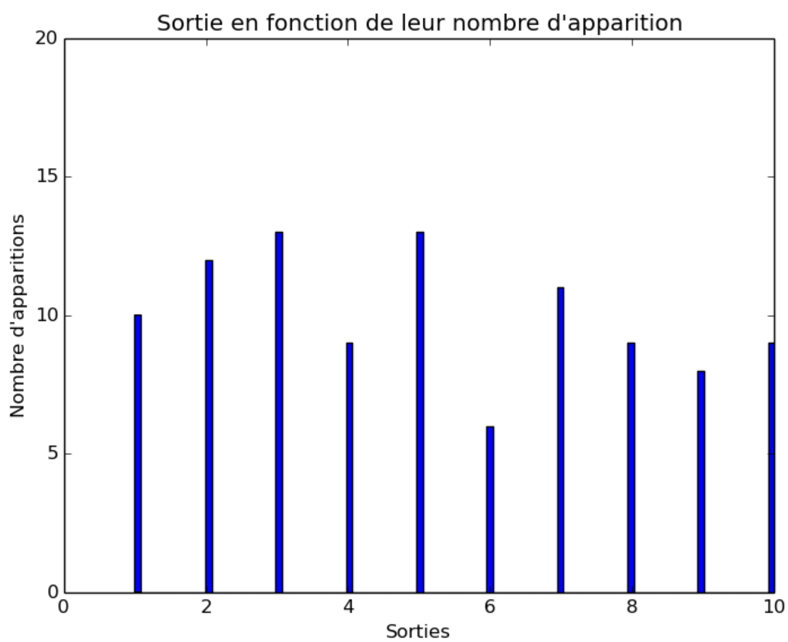

Figure 1.4 – Sortie en fonction de leur nombre d'apparition