المدرسة الوطنية للعلوم التطبيقية - آسفي
ⵜⵉⵎⵙⵙⵓⵜ ⵜⴰⵏⴰⵎⵓⵔⵜ ⵏ ⵜⵎⴰⵡⴰⵙⵜ ⵜⵉⵏⵉⵡⵉⵏ - ⴰⵙⴼⵉ
Ecole Nationale des Sciences Appliquées - Safi

Simulation de Variables Aléatoires Discrètes

Module : Probabilités et Stochastiques

Filière : G.T.R. – Première Année

Encadrant : M. El Hassan LAKHEL

Étudiant : EL MQIDDEM OTHMANE

Table des matières

Remerciements

Je tiens à exprimer ma gratitude à **Monsieur El Hassan LAKHEL** pour son encadrement précieux, sa disponibilité et ses conseils tout au long de ce module. Ses explications claires et sa rigueur pédagogique m'ont permis de mieux comprendre les concepts fondamentaux liés à la simulation de variables aléatoires.

Introduction

Dans ce projet, nous étudions la simulation de variables aléatoires discrètes à l'aide de programmes Python. L'objectif est de comprendre comment, à partir d'un générateur
uniforme U[0, 1], simuler des lois discrètes usuelles : uniforme, binomiale, géométrique,
etc. Ce rapport suit l'énoncé proposé et illustre chaque loi par des codes, démonstrations
mathématiques et figures explicatives.

Chapitre 1 — Génération Pseudo-Aléatoire

1.1 Principe Théorique

En informatique, la simulation d'une variable aléatoire repose sur la génération de nombres pseudo-aléatoires. Ces derniers ne sont pas réellement aléatoires (puisqu'ils sont produits par un algorithme déterministe), mais ils imitent très bien le comportement de variables aléatoires uniformes sur l'intervalle [0, 1[.

Cela permet, à partir de cette base uniforme, de construire par transformation des variables aléatoires suivant d'autres lois (discrètes ou continues). Le cœur de cette idée repose donc sur un générateur pseudo-aléatoire fiable et statistiquement valide.

1.2 Principe du générateur pseudo-aléatoire

partie 1.2.1 : Générateur pseudo-aléatoire en Python
Cette section vise à **manipuler la fonction** random() **et** seed() pour comprendre le comportement du générateur pseudo-aléatoire.

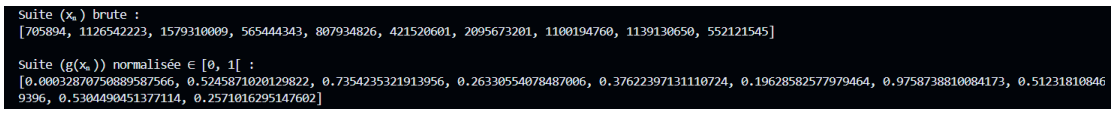
Objectifs de cette partie :

Utiliser random.random() pour générer un nombre aléatoire ∈ [0, 1[

Utiliser random.seed() pour rendre les résultats **reproductibles**

Vérifier que la suite est **déterministe si la graine est fixée**

Exemple en Python



```python
# Implémentation d'un générateur pseudo-aléatoire simple basé sur le modèle (S, f, s)

# Définir les paramètres du générateur
M = 2**31 - 1      # Grand entier (taille de S)
a = 16807          # Coefficient multiplicatif (f)
c = 0              # Incrément (si ≠ 0 : congruentiel linéaire complet)
s = 42             # Graine initiale

# Fonction f : f(x) = (a * x + c) % M
def f(x):
    return (a * x + c) % M

# Fonction g : normalisation dans [0, 1[
def g(x):
    return x / M

# Génération de la suite (xn), puis (g(xn))
def generate_sequence(seed, n):
    sequence_raw = []
    sequence_uniform = []
    x = seed
    for _ in range(n):
        x = f(x)
        sequence_raw.append(x)
        sequence_uniform.append(g(x))
    return sequence_raw, sequence_uniform
```

```python
# Point d'entrée
if __name__ == "__main__":
    n = 10
    raw, uniform = generate_sequence(seed=s, n=n)

    print("Suite (xₙ) brute :")
    print(raw)

    print("\nSuite (g(xₙ)) normalisée ∈ [0, 1[ :")
    print(uniform)
```

Explication :

J'implémente ici un **générateur congruentiel linéaire (LCG)** simple.

f(x) est la fonction de transition entre les états internes.

g(x) transforme la sortie dans **l'intervalle [0, 1[**, comme dans random.random().

Le comportement est **déterministe** : même s ⇒ même suite.

Un générateur pseudo-aléatoire peut être modélisé par un triplet (S,f,s), où S est un ensemble fini de grands entiers, fff une fonction récursive sur S, et sss une graine initiale. En informatique, la transformation g:S→[0,1[permet d'utiliser ces entiers dans des simulations probabilistes. Le code Python ci-joint montre un générateur basé sur une fonction $f(x) = (ax + c) mod\ M$, où la suite (g(xn)) obtenue est déterministe et reproductible pour une graine fixée.

la fonction random :

Objectif : Expérimenter random.random() et random.seed() pour comprendre la reproductibilité des suites pseudo-aléatoires.

```python
import random

# Q1 : Génération d'un nombre aléatoire sans seed
print("random.random() sans seed :")
print(random.random())

# Q2 : Fixer une graine (seed) et observer la suite
random.seed(0)
print("\nSuite aléatoire après random.seed(0) :")
for _ in range(5):
    print(random.random())

# Q3 : Refaire la même chose avec la même graine (seed)
random.seed(0)
print("\nReproduction de la suite avec seed(0) à nouveau :")
for _ in range(5):
    print(random.random())
```

```
random.random() sans seed :
0.5639541945955245

Suite aléatoire après random.seed(0) :
0.8444218515250481
0.7579544029403025
0.420571580830845
0.25891675029296335
0.5112747213686085

Reproduction de la suite avec seed(0) à nouveau :
0.8444218515250481
0.7579544029403025
0.420571580830845
0.25891675029296335
0.5112747213686085
PS C:\Users\elmqi\simulation-va-discretes\simulation-va-discretes\code>
```

Réponses aux questions de cette partie :

Après avoir importé le module random, évaluer random.random():

Résultat (exemple) : 0.7579544029403025
Chaque exécution **donne un résultat différent** (si on ne fixe pas la seed).

Évaluer la commande random.seed(0), puis random.random() plusieurs fois

Résultat : À chaque fois qu'on fixe random.seed(0), on obtient **la même suite** de valeurs aléatoires.

Expliquer brièvement les résultats obtenus

Réponse :

random.seed(valeur) initialise le générateur pseudo-aléatoire avec une **graine fixe**.

Cela rend la suite de valeurs **parfaitement déterministe et reproductible**, ce qui est utile pour :

le **débogage**,

la **comparaison entre étudiants**,

les **tests unitaires**.

Sans seed(), chaque appel à random.random() génère un nombre différent basé sur l'état interne du générateur (ex: temps système).

La fonction random.random() retourne un nombre réel pseudo-aléatoire dans l'intervalle [0,1[[0, 1[[0,1[. En fixant une graine avec random.seed(valeur), la suite générée devient déterministe et reproductible. Cela permet de refaire exactement les mêmes expériences numériques, ce qui est essentiel pour la reproductibilité scientifique et le débogage.

Simulation d'une variable aléatoire : fondements mathématiques

La loi faible des grands nombres (LFGN)

La **loi faible des grands nombres** est un résultat fondamental de la théorie des probabilités. Elle établit que, lorsqu'on répète une expérience aléatoire un grand nombre de fois, la **moyenne empirique** des résultats converge (en probabilité) vers la **valeur moyenne théorique** de l'expérience.

✅Énoncé (formel)

Soit (Xn)n∈ Nune suite de variables aléatoires :

indépendantes,de **même loi**,et de **carré intégrable** :

$E[X12] < +\infty \mathbb{E}[X_1^2] < +\infty$

Alors, pour tout ε>0 :

$$\lim n \to \infty P(| 1n\sum k = 1nXk - E[X1] |> \varepsilon) = 0$$

Ce qui signifie que la moyenne empirique

$$X^{-}n = 1n\sum Xk$$

converge en probabilité vers $E[X1]$

Démonstration

On souhaite démontrer que :

$$P(|\ X^{-}n - E[X1]\ |> \varepsilon) \to 0\ \ quand\ \ n \to \infty$$

Étape 1 — Espérance de la moyenne empirique :

Par linéarité de l'espérance :

$$E[X^{-}n] = E[1n\sum k = 1nXk] = 1n\sum k = 1nE[Xk] = E[X1]$$

Étape 2 — Variance de la moyenne empirique :

Comme les Xk sont **indépendantes et de même loi** :

$Var(X^{-}n) = Var(1n\sum k = 1nXk) = 1n2\sum k = 1nVar(Xk) = 1n2 \cdot n \cdot Var(X1) = Var(X1)n$

Étape 3 — Inégalité de Bienaymé-Tchebychev :

Pour tout ε>0, on applique l'inégalité :

$P(|\ X^{-}n - E[X1]\ |> \varepsilon) \leq Var(X1)n\varepsilon 2$

Étape 4 — Conclusion (limite) :

Comme le membre de droite tend vers 0 lorsque n→∞n \to \inftyn→∞ :

$$lim\, n \to \infty P(|\ X^{-}n - E[X1]\ |> \varepsilon) = 0$$

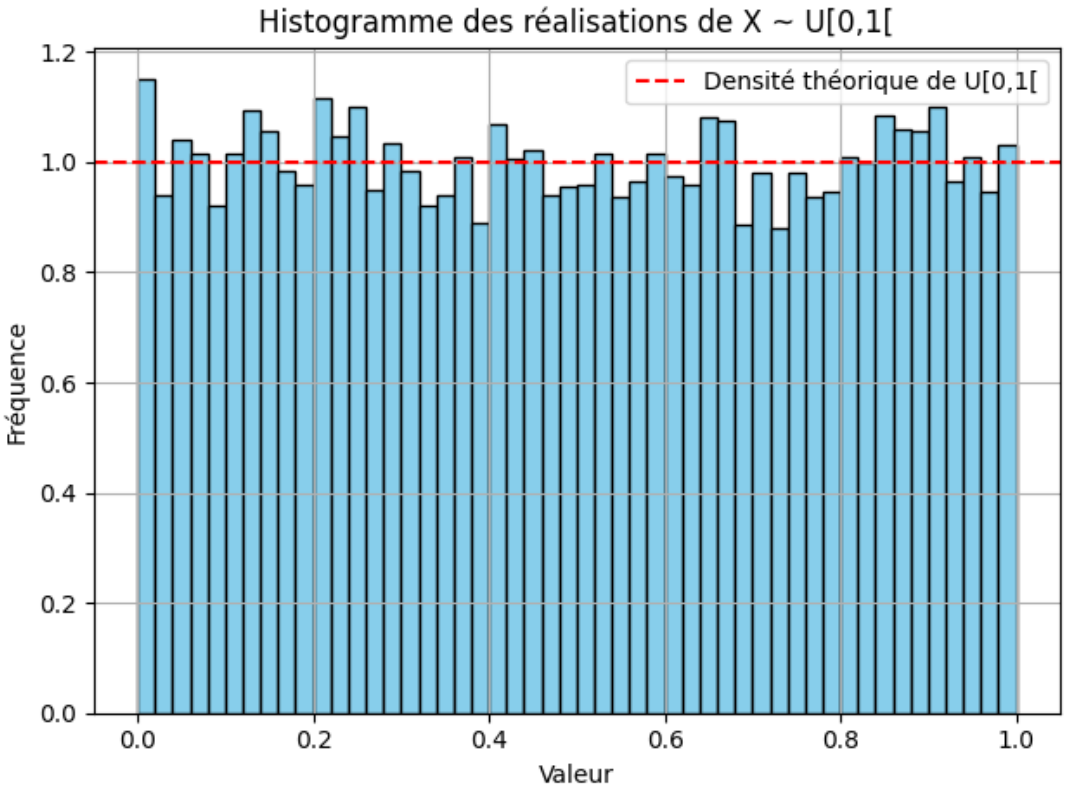
Cela démontre que X¯n **converge en probabilité** vers E[X1].

Interprétation intuitive

Cette loi confirme qu'en répétant une expérience aléatoire un grand nombre de fois, la **moyenne des résultats observés** tend à s'approcher de la **moyenne théorique**, c'est-à-dire **la valeur attendue**.

Simulation d'une v.a.r. en Python

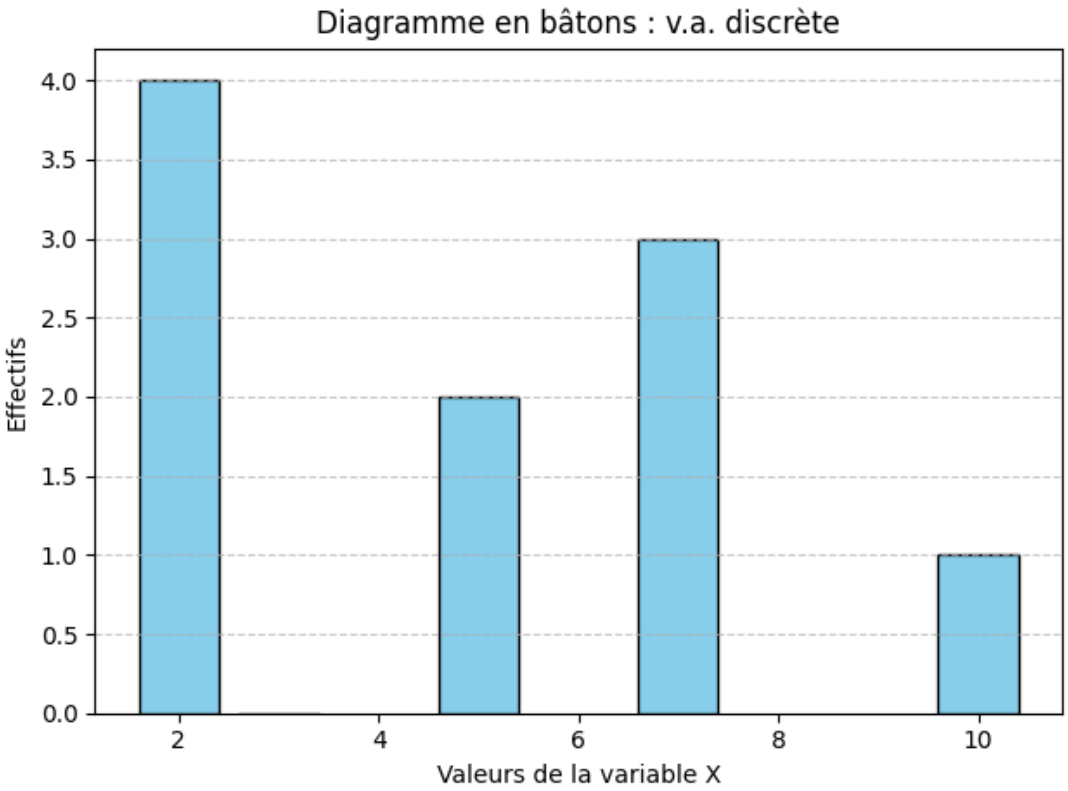
```python
import random
import matplotlib.pyplot as plt

# Simulation de 10000 réalisations d'une variable aléatoire uniforme sur [0, 1[
n = 10000
echantillons = [random.random() for _ in range(n)]

# Affichage de quelques réalisations
print("Quelques réalisations de X ~ U[0,1[ :")
print(echantillons[:10])

# Visualisation : histogramme de fréquence
plt.hist(echantillons, bins=50, color='skyblue', edgecolor='black', density=True)
plt.axhline(y=1, color='red', linestyle='--', label='Densité théorique de U[0,1[')
plt.title("Histogramme des réalisations de X ~ U[0,1[")
plt.xlabel("Valeur")
plt.ylabel("Fréquence")
```



En Python, la fonction random.random() permet de simuler une réalisation d'une variable aléatoire X ~ U[0,1[. En répétant plusieurs appels indépendants, on obtient une suite de réalisations indépendantes X1,X2,...,Xn, ce qui permet d'expérimenter la loi faible des grands nombres ou de modéliser d'autres lois par transformation. L'histogramme montre que les fréquences observées tendent à se stabiliser et à s'uniformiser, illustrant la convergence vers la densité théorique.

diagramme en bâtons avec matplotlib.pyplot.bar()

```
import matplotlib.pyplot as plt

# Valeurs possibles (classes) et leurs effectifs
absc = [2, 3, 5, 7, 10]     # valeurs prises par la variable aléatoire
ord = [4, 0, 2, 3, 1]       # effectifs associés à ces valeurs

# Tracé du diagramme en bâtons
plt.bar(absc, ord, color='skyblue', edgecolor='black')

# Personnalisation du graphe
plt.xlabel("Valeurs de la variable X")
plt.ylabel("Effectifs")
```



Pour représenter graphiquement une distribution discrète, on utilise un **diagramme en bâtons** à l'aide de la fonction plt.bar() du module matplotlib.pyplot.
Elle prend deux paramètres principaux :

absc : la liste des valeurs de la variable (abscisses),

ord : la liste des effectifs ou fréquences correspondants (ordonnées).

Ce type de graphe permet de visualiser facilement la répartition empirique d'une variable aléatoire discrète.

Loi uniforme discrète :

Simulation à l'aide de la fonction random

Que signifie qu'une v.a.r. X suit la loi uniforme discrète sur {a,...,b} ?

Cela signifie que la variable aléatoire X peut prendre **toutes les valeurs entières entre** aaa **et** bbb inclusivement, avec **la même probabilité pour chacune**.

Formellement, si $X \sim Ud(a, \ldots, b)$

$P(X = k) = 1/b - a + 1$,pour tout k∈ {a,a+1,...,b}, C'est une **loi discrète équiprobable**.

Quel est le rôle de la fonction uniforme(a, b) ?

La fonction uniforme(a, b) simule une **valeur entière au hasard dans** [a,b]avec probabilité uniforme, en :

générant un réel U∈ [0,1[

le transformant en un entier k∈ {a,...,b} par :

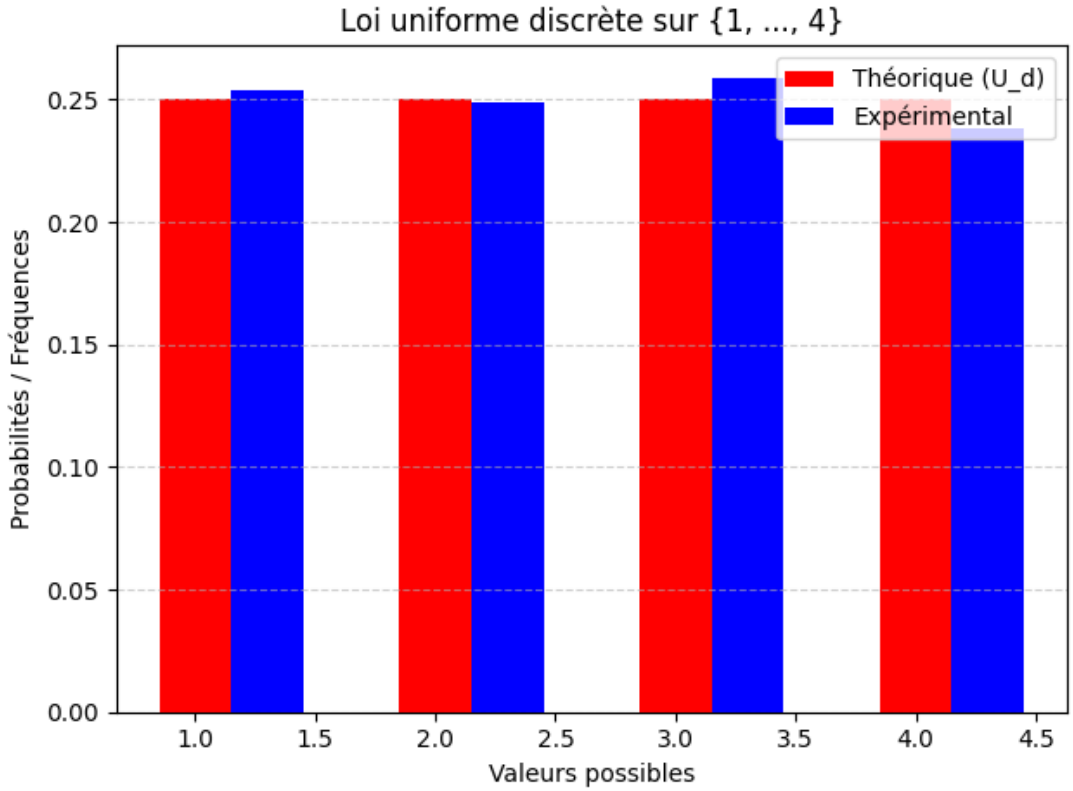
X=a+[U· (b−a+1)]

Cela respecte la définition d'une **loi uniforme discrète**.

```python
import random
import math
import numpy as np
import matplotlib.pyplot as plt

# Fonction de simulation
def uniforme(a, b):
    return a + math.floor(random.random() * ((b - a) + 1))

# Fonction utilitaire : calcEffectif
def position(L, elt):
    for i in range(len(L)):
        if L[i] == elt:
            return i
    return -1

def calcEffectif(cl, Obs):
    effectifs = [0] * len(cl)
    for val in Obs:
        i = position(cl, val)
        if i != -1:
            effectifs[i] += 1
    return effectifs
```

```python
# Paramètres
N = 1000        # Nombre de simulations
a = 1
b = 4           # Uniforme sur {1, 2, 3, 4}
n = b - a + 1

#Ligne 9 équivalente avec une boucle for
Obs = []
for _ in range(N):
    Obs.append(uniforme(a, b))

# Classes
cl = list(range(a, b + 1))

#Calcul des effectifs
effectif = calcEffectif(cl, Obs)

#Distribution théorique
P = [1 / n] * n
```

```python
#Tracé du diagramme comparatif
absc = np.array(cl)
plt.bar(absc, P, color='r', width=0.3, label="Théorique (U_d)")
plt.bar(absc + 0.3, [e / N for e in effectif], color='b', width=0.3, label="Expérimental")
plt.xlabel("Valeurs possibles")
plt.ylabel("Probabilités / Fréquences")
plt.title("Loi uniforme discrète sur {1, ..., 4}")
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig("../figures/hist_uniforme_discrete.png")
plt.show()
```



Comment adapter ce programme pour une loi uniforme discrète sur {a,...,b}?

Il suffit de remplacer :

range(1, n + 1) par range(a, b + 1)

le simulateur uniforme(1, n) par uniforme(a, b)

1 / n par 1 / (b - a + 1)

Le code est donc **déjà adapté** à Ud({a,...,b}).

Une variable aléatoire X suit une loi uniforme discrète sur {a,...,b} lorsqu'elle prend toutes les valeurs entières de cet intervalle avec une même probabilité.
Le simulateur Python uniforme(a, b) implémente cette loi à partir d'un générateur pseudo-aléatoire uniforme sur [0,1[.
Le diagramme ci-dessous compare la distribution théorique à celle obtenue expérimentalement après N=1000 tirages. On constate une bonne adéquation entre les deux.

Loi binomiale

Pour quel type d'expérience définit-on une variable aléatoire (v.a.r.) suivant une loi binomiale ?

Une variable aléatoire suit une loi binomiale lorsqu'on répète **n fois une expérience de Bernoulli** (expérience à deux issues : succès ou échec) **indépendantes et identiques**, où la probabilité de succès est p à chaque répétition.

Que signifie qu'une v.a.r. X suit la loi binomiale de paramè tres ?

Cela signifie que X est le nombre de succès obtenus en nnn répétitions indépendantes d'une expérience de Bernoulli de paramètre p.

X peut prendre des valeurs entières entre 0 et n.

La probabilité que X=k est donnée par :

$$P(X = k) = (nk)pk(1 - p)n - k, k = 0,1, \dots, n$$

Simulation à l'aide de la fonction random

```python
import random
import numpy as np
import matplotlib.pyplot as plt
from math import comb

# 1. Fonctions de simulation

def Bernoulli(p):
    """Simule une variable aléatoire Bernoulli B(p)"""
    return 1 if random.random() < p else 0

def binomiale(n, p):
    """Simule une variable aléatoire binomiale B(n, p)"""
    return sum(Bernoulli(p) for _ in range(n))


# 2. Paramètres

N = 1000   # nombre de simulations
n = 40     # nombre d'essais
p = 0.3    # probabilité de succès
```

```python
# 3. Simulation des variables binomiales

Obs = []
for _ in range(N):
    Obs.append(binomiale(n, p))

# 4. Calcul des effectifs observés

effectif = np.array([Obs.count(k) for k in range(n+1)])

# 5. Calcul de la distribution théorique

P = np.zeros(n+1)
for k in range(n+1):
    comb_val = comb(n, k)          # ligne 18 complétée ici
    P[k] = comb_val * (p**k) * ((1-p)**(n-k))
```

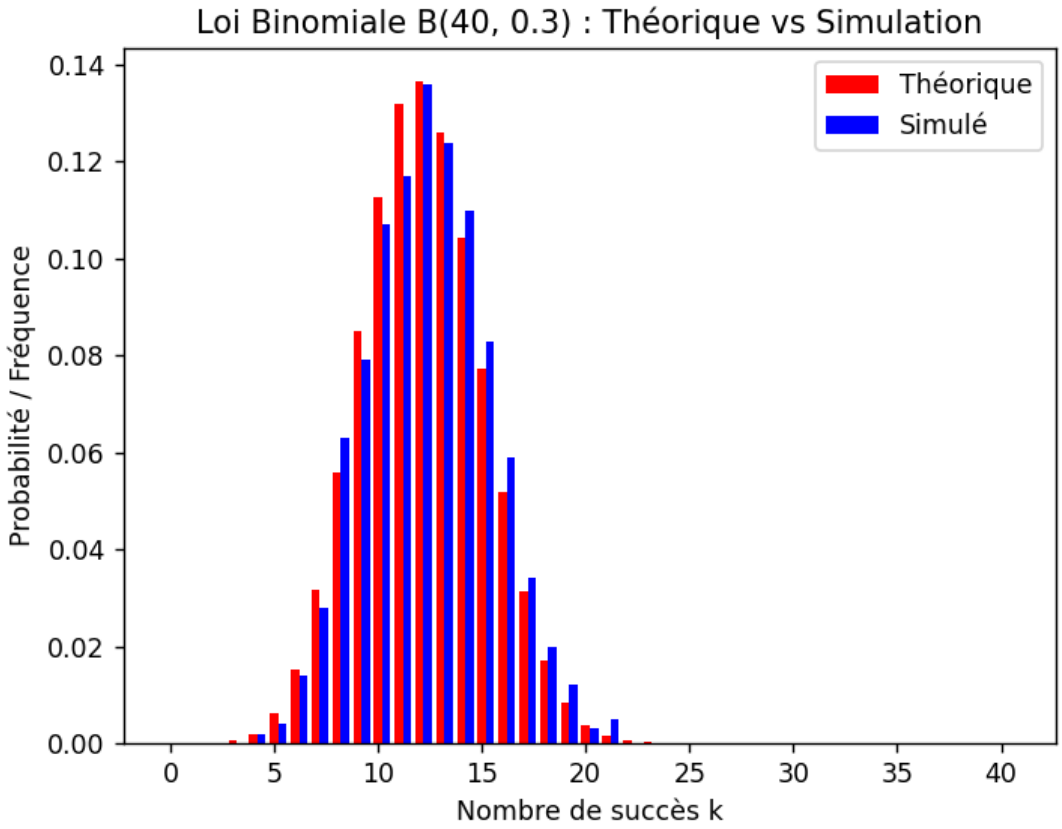
```python
# 6. Tracé des diagrammes en bâtons

absc = np.arange(n+1)  # abscisses : valeurs 0,1,...,n

plt.bar(absc, P, color='r', width=0.4, label='Théorique')
plt.bar(absc + 0.4, effectif / N, color='b', width=0.4, label='Simulé')

plt.xlabel('Nombre de succès k')
plt.ylabel('Probabilité / Fréquence')
plt.title('Loi Binomiale B({}, {}) : Théorique vs Simulation'.format(n, p))
plt.legend()
plt.show()
```

Une v.a.r suit une loi binomiale si elle compte le nombre de succès dans nnn répétitions d'une expérience de Bernoulli.

La fonction `Bernoulli(p)` simule un succès ou un échec.

La fonction `binomiale(n,p)` simule la somme de nnn expériences Bernoulli.

On calcule la distribution théorique via la formule $(nk)pk(1-p)n-k$ On compare par un graphique les fréquences simulées et la loi théorique.



Un histogramme rouge représentant la loi binomiale théorique B(40,0.3)B(40, 0.3)B(40,0.3).

Un histogramme bleu représentant les fréquences obtenues par simulation (1000 répétitions).

Ces deux histogrammes devraient être proches, ce qui valide la simulation.

Loi discrète quelconque

Théorie et explications

Loi discrète et fonction de répartition

- Soit X une variable aléatoire discrète qui prend les valeurs $x_1, x_2, \ldots, x_n$ avec probabilités respectives $p_1, p_2, \ldots, p_n$.
- La fonction de répartition $F_X(x) = P(X \leq x)$ est une fonction en escalier, avec :

$$F_X(x) = \begin{cases} 0 & \text{si } x < x_1 \\ p_1 & \text{si } x_1 \leq x < x_2 \\ p_1 + p_2 & \text{si } x_2 \leq x < x_3 \\ \cdots & \\ p_1 + p_2 + \cdots + p_i & \text{si } x_i \leq x < x_{i+1} \\ 1 & \text{si } x \geq x_n \end{cases}$$

- On définit les sommes cumulées $r_k = \sum_{i=1}^{k} p_i$, avec $r_0 = 0$ et $r_n = 1$.

Loi uniforme discrète U{0,…,5}

- Valeurs prises par X (les x_i) :

$$x_i = 0, 1, 2, 3, 4, 5$$

- Probabilités associées (les p_i) :

$$p_i = \frac{1}{6} \quad \text{pour } i = 0, \ldots, 5$$

- Sommes cumulées (les $r_k = \sum_{i=1}^{k} p_i$) :

$$r_0 = 0, \quad r_1 = \frac{1}{6}, \quad r_2 = \frac{2}{6}, \quad r_3 = \frac{3}{6}, \quad r_4 = \frac{4}{6}, \quad r_5 = \frac{5}{6}, \quad r_6 = 1$$

Tracé de la fonction de répartition FX(x) = P(X ≤ x)F_X(x)

Pour $x < 0$, $F_X(x) = 0$

Pour $0 \leq x < 1$, $F_X(x) = \frac{1}{6}$

Pour $1 \leq x < 2$, $F_X(x) = \frac{2}{6}$

Pour $2 \leq x < 3$, $F_X(x) = \frac{3}{6}$

Pour $3 \leq x < 4$, $F_X(x) = \frac{4}{6}$

Pour $4 \leq x < 5$, $F_X(x) = \frac{5}{6}$

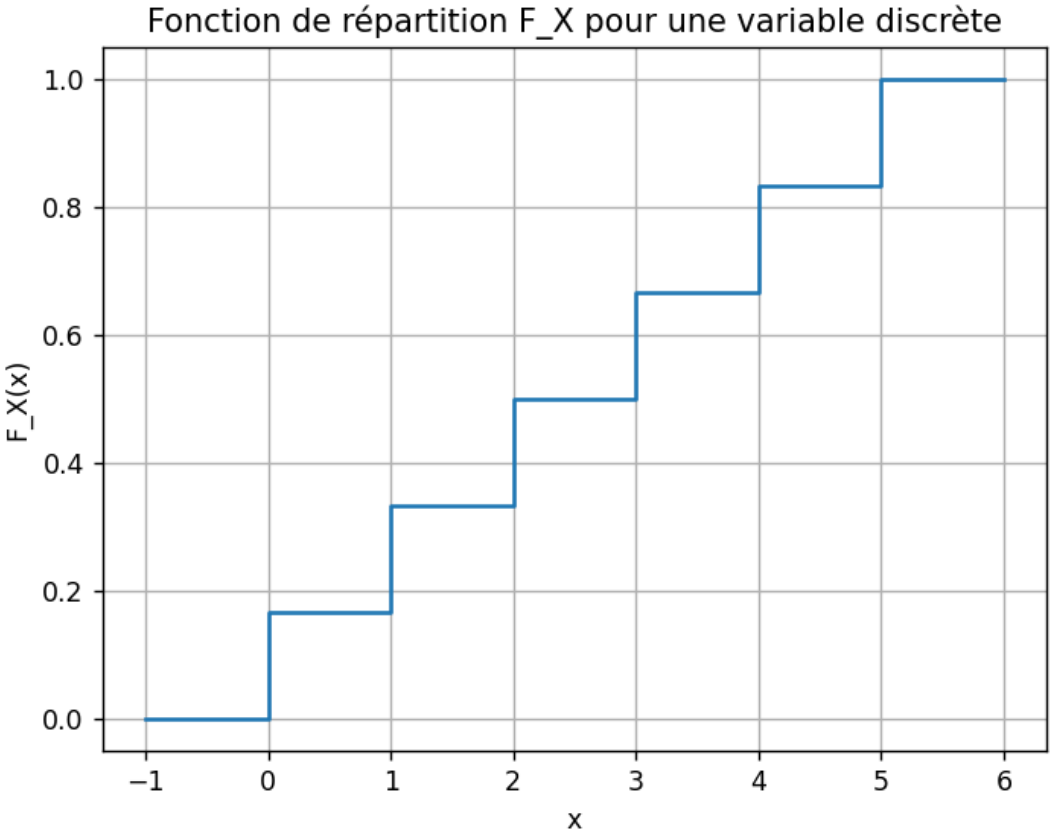
Pour $x \geq 5$, $F_X(x) = 1$

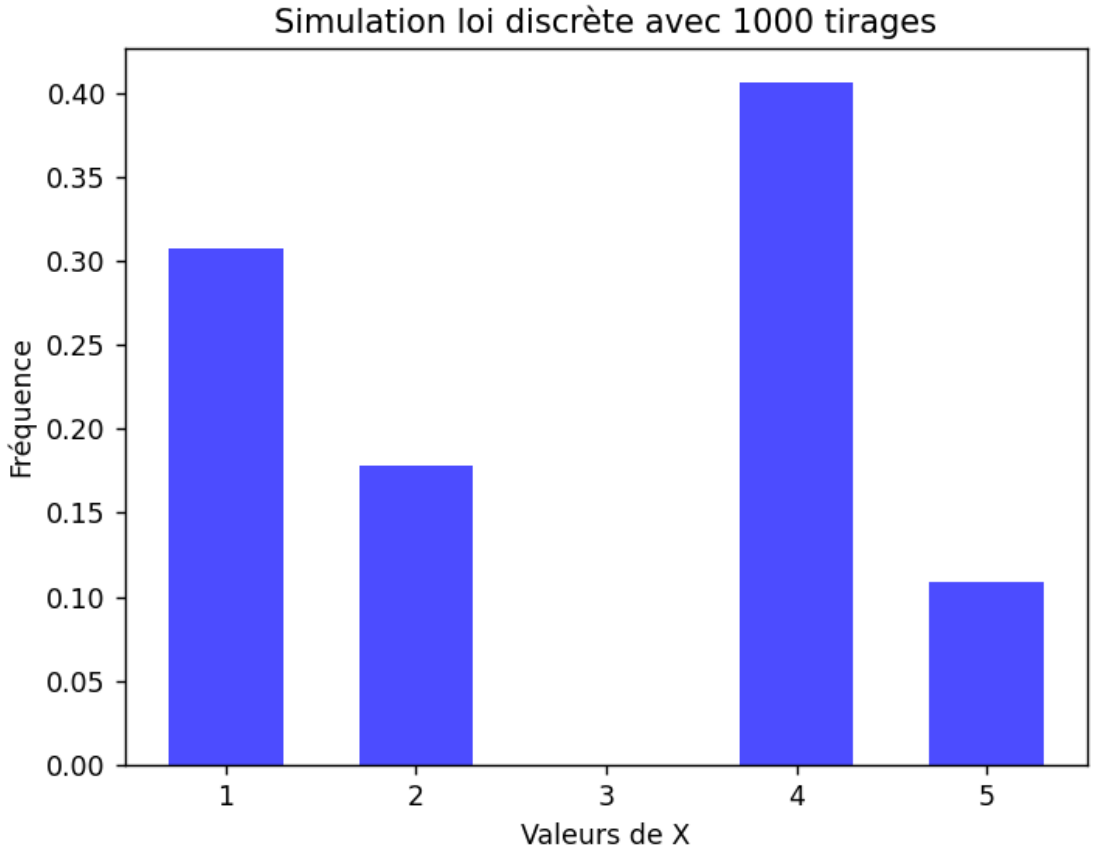
```python
import random
import matplotlib.pyplot as plt
import numpy as np

def sommeCumulee(P):
    """Calcule la somme cumulée des probabilités P."""
    r = []
    cumul = 0
    for p in P:
        cumul += p
        r.append(cumul)
    return r

def discreteQ(P, X):
    """Simule une variable aléatoire discrète selon la méthode d'inversion."""
    r = sommeCumulee(P)
    U = random.random()
    for k in range(len(r)):
        if U < r[k]:
            return X[k]
    return X[-1]
```

```python
def trace_fonction_repartition(X, P):
    """Trace la fonction de répartition d'une variable discrète."""
    r = [0] + sommeCumulee(P)  # r0 = 0
    x_min = min(X) - 1
    x_max = max(X) + 1

    # Préparer les points pour la fonction en escalier
    xs = []
    ys = []

    xs.append(x_min)
    ys.append(0)

    for i in range(len(X)):
        xs.append(X[i])
        ys.append(r[i])
        xs.append(X[i])
        ys.append(r[i+1])

    xs.append(x_max)
    ys.append(1)
```

```
    plt.step(xs, ys, where='post')
    plt.xlabel('x')
    plt.ylabel('F_X(x)')
    plt.title('Fonction de répartition F_X pour une variable discrète')
    plt.grid(True)
    plt.show()

def simulation_et_tracé(P, X, N=1000):
    """Simule N tirages et trace le diagramme en bâtons des fréquences."""
    simulations = [discreteQ(P, X) for _ in range(N)]
    effectifs = [simulations.count(xi) for xi in X]
    frequences = np.array(effectifs) / N

    plt.bar(X, frequences, width=0.6, color='blue', alpha=0.7)
    plt.xlabel('Valeurs de X')
    plt.ylabel('Fréquence')
    plt.title(f'Simulation loi discrète avec {N} tirages')
    plt.show()

# --- Exemple pour loi uniforme discrète U{0,...,5} ---
X_uniforme = [0, 1, 2, 3, 4, 5]
P_uniforme = [1/6] * 6
```

```
# Tracé fonction de répartition
trace_fonction_repartition(X_uniforme, P_uniforme)

# --- Exemple pour loi discrète arbitraire ---
P_arbitraire = [0.3, 0.2, 0.4, 0.1]
X_arbitraire = [1, 2, 4, 5]

# Simulation + tracé diagramme en bâtons
simulation_et_tracé(P_arbitraire, X_arbitraire)
```


Fonction de répartition F_X pour une variable discrète
F_X(x)
x
1.0
0.8
0.6
0.4
0.2
0.0
−1
0
1
2
3
4
5
6


Simulation loi discrète avec 1000 tirages
Fréquence
Valeurs de X
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1
2
3
4
5

Réponses aux questions sur la loi géométrique

Pour quel type d'expérience définit-on une variable aléatoire suivant une loi géométrique ?

Une variable aléatoire suit une loi géométrique lorsqu'elle modélise le nombre d'essais successifs nécessaires pour obtenir le **premier succès** dans une suite d'expériences indépendantes de Bernoulli, chacune ayant une probabilité de succès p.

Que signifie qu'une variable aléatoire X suit la loi géométrique de paramètre p ?

Cela signifie que pour k=1,2,3,…, la probabilité que X prenne la valeur k (le premier succès au k-ème essai) est :

$P(X=k)=(1-p)^{k-1}p$

Autrement dit, on a k−1 échecs (avec probabilité 1−p) avant le premier succès (avec probabilité p).

Fonction geom(p) simulant une variable géométrique via Bernoulli

La fonction Bernoulli simule un essai unique (succès avec probabilité ppp, échec sinon). Pour simuler la variable géométrique, on compte le nombre d'essais jusqu'au premier succès.

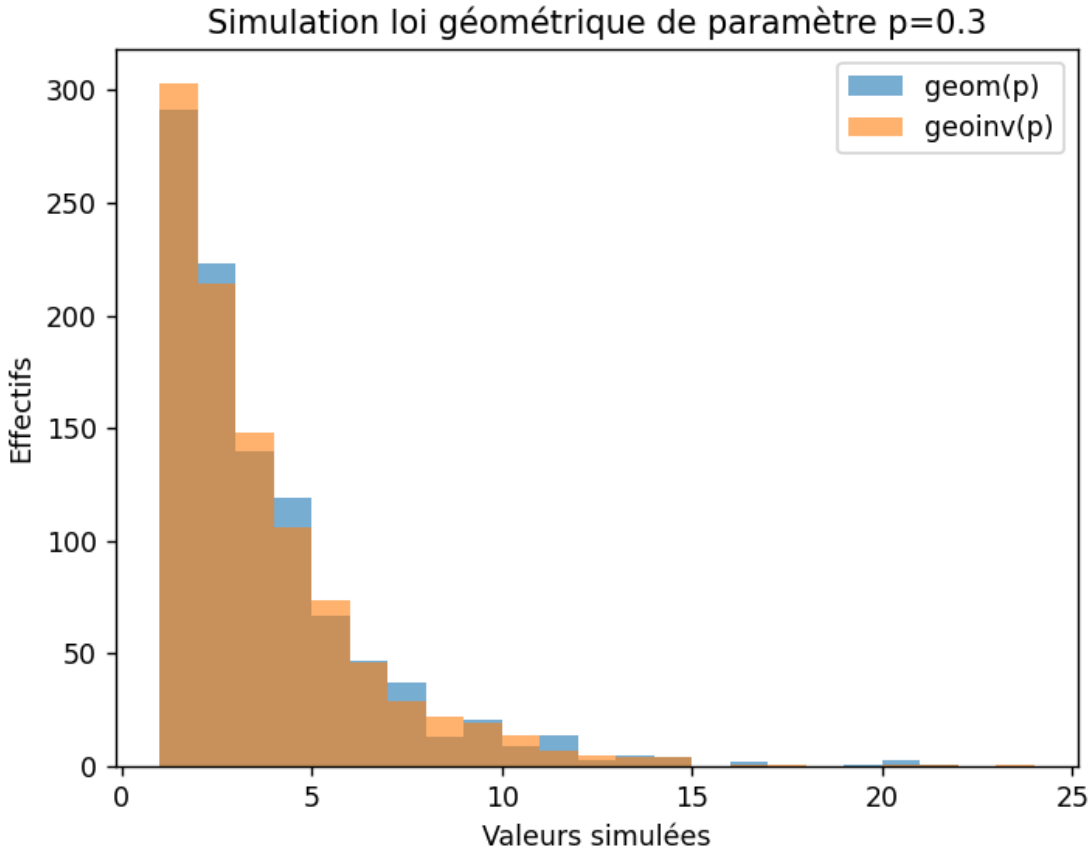
Illustration de la loi des grands nombres

On peut générer un grand nombre de variables aléatoires géométriques (par exemple via np.random.geometric(p)), calculer la moyenne empirique, et montrer qu'elle tend vers la valeur théorique de l'espérance :
$E(X)=1/p$

```python
import random
import math
import matplotlib.pyplot as plt
import numpy as np

def Bernoulli(p):
    """Simule un essai de Bernoulli avec succès p."""
    return 1 if random.random() < p else 0

def geom(p):
    """Simule une variable aléatoire suivant la loi géométrique de paramètre p en utilisant Bernoulli."""
    count = 1
    while Bernoulli(p) == 0:
        count += 1
    return count
```

```python
def geoinv(p):
    """Simule une variable aléatoire suivant la loi géométrique par méthode d'inversion."""
    U = random.random()
    return math.ceil(math.log(1 - U) / math.log(1 - p))

def loi_des_grands_nombres(p, N=10000):
    """Illustre la loi des grands nombres pour la loi géométrique."""
    samples = np.random.geometric(p, N)
    moyennes_cumulees = np.cumsum(samples) / np.arange(1, N+1)

    plt.plot(moyennes_cumulees, label='Moyenne empirique')
    plt.axhline(y=1/p, color='r', linestyle='--', label=f'Espérance théorique 1/p = {1/p}')
    plt.xlabel('Nombre de tirages')
    plt.ylabel('Moyenne cumulative')
    plt.title('Illustration de la loi des grands nombres (loi géométrique)')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```python
# --- Exemple d'utilisation ---
p = 0.3
# Simulation avec la fonction geom(p)
simulations_geom = [geom(p) for _ in range(1000)]
# Simulation avec la méthode d'inversion
simulations_geoinv = [geoinv(p) for _ in range(1000)]
# Histogramme des deux simulations
plt.hist(simulations_geom, bins=range(1, max(simulations_geom)+2), alpha=0.6, label='geom(p)')
plt.hist(simulations_geoinv, bins=range(1, max(simulations_geoinv)+2), alpha=0.6, label='geoinv(p)')
plt.xlabel('Valeurs simulées')
plt.ylabel('Effectifs')
plt.title(f'Simulation loi géométrique de paramètre p={p}')
plt.legend()
plt.show()
# Illustration de la loi des grands nombres
loi_des_grands_nombres(p)
```


Simulation loi géométrique de paramètre p=0.3
geom(p)
geoinv(p)
Effectifs
Valeurs simulées
0
50
100
150
200
250
300
0
5
10
15
20
25

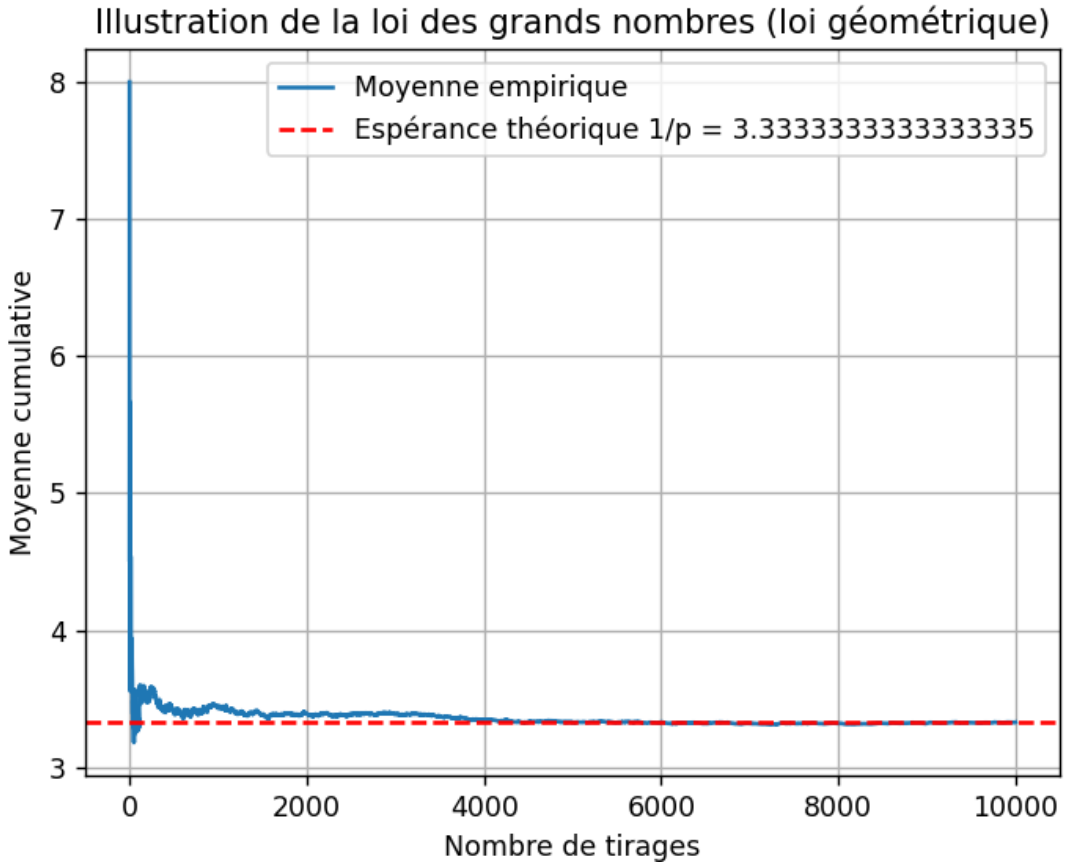


Illustration de la loi des grands nombres (loi géométrique)
Moyenne empirique
Espérance théorique 1/p = 3.3333333333333335
Moyenne cumulative
Nombre de tirages
3
4
5
6
7
8
0
2000
4000
6000
8000
10000

Approximation de la loi de Poisson

La loi de Poisson peut être vue comme la limite d'une loi binomiale $B(n, p_n)$ quand $n \to \infty$ et $p_n = \frac{\lambda}{n}$, de sorte que $np_n = \lambda$ reste constant.

Formellement, pour tout $k \in \mathbb{N}$:

$$\lim_{n\to\infty} P(X_n = k) = \lim_{n\to\infty} \binom{n}{k} p_n^k (1-p_n)^{n-k} = e^{-\lambda}\frac{\lambda^k}{k!}$$

La fonction de masse de la loi de Poisson est donc :

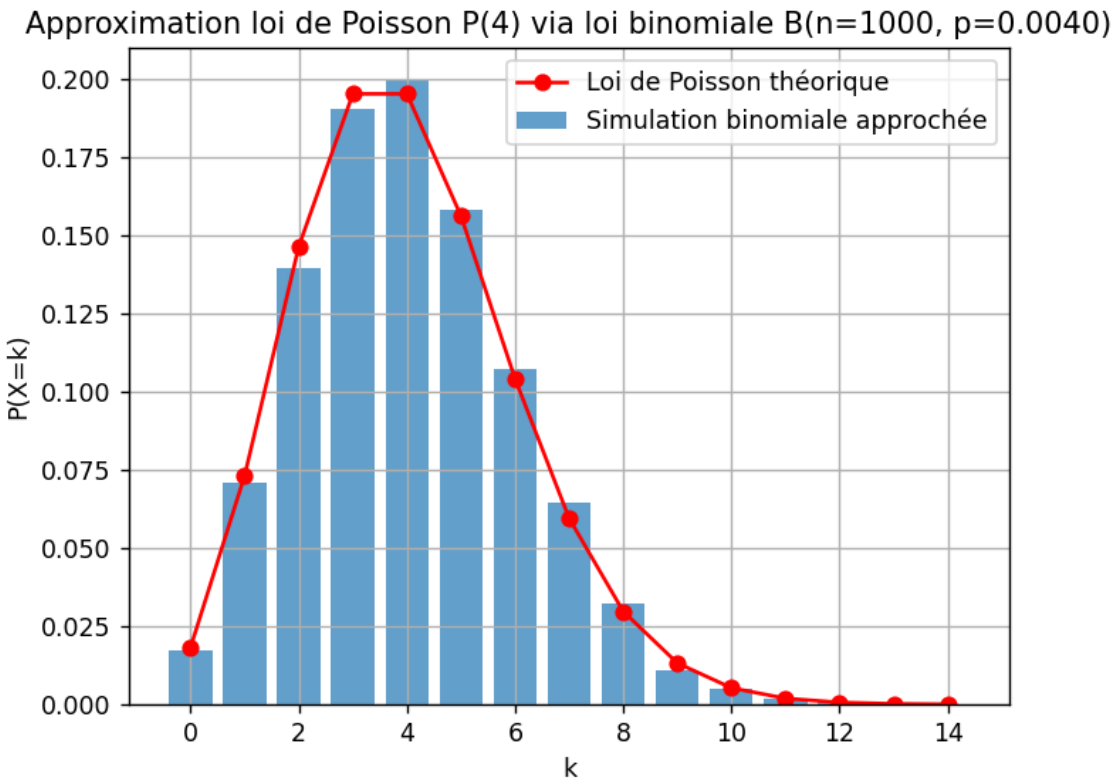
$$P(X = k) = e^{-\lambda}\frac{\lambda^k}{k!}, \quad k = 0, 1, 2, \ldots$$

Fonction python pour simuler la loi de Poisson par approximation binomiale

```python
import random
import math
import matplotlib.pyplot as plt
from collections import Counter

def Bernoulli(p):
    """Simule un essai Bernoulli avec succès p."""
    return 1 if random.random() < p else 0

def poissonRare(lmbda, n=1000):
    """
    Simule une variable aléatoire approchant la loi de Poisson P(lambda)
    par la loi binomiale B(n, p) avec p = lambda/n.
    """
    p = lmbda / n
    # Somme de n essais Bernoulli indépendants de paramètre p
    somme = 0
    for _ in range(n):
        somme += Bernoulli(p)
    return somme
```

```python
# --- Exemple d'utilisation ---

lmbda = 4  # paramètre lambda de la loi de Poisson
n = 1000   # grand n pour approximation

# Simulation de 10000 variables suivant la loi approchée
simulations = [poissonRare(lmbda, n) for _ in range(10000)]

# Comptage des occurrences
freq = Counter(simulations)

# Préparation des données pour affichage
x_vals = sorted(freq.keys())
y_vals = [freq[k]/10000 for k in x_vals]

# Affichage du diagramme en bâtons de la distribution simulée
plt.bar(x_vals, y_vals, alpha=0.7, label='Simulation binomiale approchée')
```

```python
# Affichage de la vraie loi de Poisson (formule)
poisson_pmf = [math.exp(-lmbda) * (lmbda**k) / math.factorial(k) for k in x_vals]
plt.plot(x_vals, poisson_pmf, 'ro-', label='Loi de Poisson théorique')

plt.xlabel('k')
plt.ylabel('P(X=k)')
plt.title(f'Approximation loi de Poisson P({lmbda}) via loi binomiale B(n={n}, p={lmbda/n:.4f})')
plt.legend()
plt.grid(True)
plt.show()
```



Conclusion

Ce projet a permis d'explorer de manière concrète la simulation de variables aléatoires discrètes en s'appuyant sur les outils offerts par Python. À travers la manipulation de générateurs pseudo-aléatoires et la modélisation de lois usuelles comme l'uniforme, la binomiale, la géométrique ou encore la loi de Poisson, nous avons pu illustrer des concepts fondamentaux de la théorie des probabilités.
Les résultats numériques, appuyés par des représentations graphiques et des comparaisons avec les distributions théoriques, valident la pertinence des méthodes employées.
Ce travail démontre également l'importance de la reproductibilité via l'usage de graines (seed) et la puissance de la simulation pour explorer et comprendre des phénomènes aléatoires complexes. Il constitue ainsi une base solide pour des applications futures en statistiques, apprentissage automatique ou modélisation stochastique.

GITHUB LINK : **https://github.com/oeisthename/simulation-va-discretes**