# TP Informatique n° 5
# Simulation de variables aléatoires discrètes

## I. L'aléatoire en informatique

### I.1. Aléatoire et déterminisme

Avant d'entamer le TP, il convient de faire un point sur la manière dont on peut produire de l'aléatoire sur machine. Commençons par détailler le vocabulaire.

**Expérience aléatoire.**
Se dit d'une expérience dont on ne peut prévoir le résultat.

**Machines déterministes.**
Les ordinateurs que nous utilisons sont des machines déterministes. Cela signifie que les algorithmes que nous écrivons sont régis par la règle suivante : pour une même entrée, l'algorithme produit toujours la même sortie.

> La notion de phénomène aléatoire est incompatible avec la prédictabilité des résultats issue du déterminisme.

### I.2. Réconcilier l'aléatoire et le déterminisme

#### I.2.a) Notion d'aléatoire ... prédictible !

Le constat précédent est sans équivoque : l'aléatoire pur ne peut être codé en machine. On va donc devoir se contenter d'une forme affaiblie de l'aléatoire conciliable avec le déterminisme des machines. En somme, il s'agit de coder **de l'aléatoire à résultats prédictibles**.
C'est exactement ce que permettent les **générateurs pseudo-aléatoires**.

#### I.2.b) Générateur pseudo-aléatoire

Un générateur pseudo-aléatoire est caractérisé par un triplet $(S, f, s)$ :

- $S$ est un ensemble fini (de cardinal grand),
- $f$ est une application $f : S \to S$,
- $s$ est un élément de $S$ ($s \in S$) appelé « graine » (*seed* en anglais).

Un tel générateur fournit une suite de nombres de $S$ notée $(x_n)$ :

- $\times$ $x_0 = s$ : le premier nombre fourni est la graine,
- $\times$ $\forall n \in \mathbb{N},\ x_{n+1} = f(x_n)$.

La suite $(x_n)$ obtenue est déterminée de manière unique par sa valeur initiale $s$.
On obtient ainsi une suite d'éléments de $S$.

Pour que ces résultats soient plus facilement exploitables, on utilise généralement une fonction $g : S \to [0, 1[$ afin de transporter les valeurs de la suite $(x_n)$ dans $[0, 1[$ : on obtient ainsi une suite $(g(x_n))$ d'éléments dans $[0, 1[$.

> Ainsi, à $s$ fixé, un générateur pseudo-aléatoire fournira toujours la même suite de réels $(g(x_n))$.

**Quels sont les avantages du pseudo-aléa ?**

- Les simulations sont reproductibles.
- En jouant sur la définition de la fonction $g$, on peut définir la répartition des valeurs fournies par le générateur. Ce dernier point permet d'envisager la simulation de variables suivant des lois de probabilités usuelles.

### I.2.c) Générateur pseudo-aléatoire en Python : la fonction `random`

La fonction `random` (de la bibliothèque `random`) implémente un générateur pseudo-aléatoire.

▶ Après avoir importé le module `random`, évaluer `random.random()`. Qu'obtient-on ? Comparer avec le résultat de votre voisin.

> On obtient un réel de $[0, 1]$. Les résultats obtenus sont différents d'un poste à l'autre, ce qui tend à faire penser à un comportement aléatoire.

▶ Évaluer la commande `random.seed(0)`. Évaluer alors plusieurs fois de suite la commande `random.random()`. Qu'obtient-on ? Comparer avec le résultat de votre voisin.

> - On obtient les résultats suivants (tronqués au millième) :
> 0.844, 0.757, 0.420, 0.258, 0.511, 0.404, 0.783, 0.303, 0.476, 0.583
> - Les résultats obtenus sont les mêmes sur tous les postes.

▶ Expliquer brièvement les résultats obtenus.

> - La deuxième série de résultat illustre la notion de générateur pseudo aléatoire. En partant tous de la même graine, la suite de nombre générée est la même pour tout le monde.
> - Le premier résultat (lorsqu'il y a différence entre les postes) démontre que la graine initiale n'est pas la même sur chaque poste. L'explication est fournie par la documentation **Python** :
>
> ```
> current system time is also used to initialize the generator when the
> module is first imported
> ```

# II. Généralités sur la simulation d'une variable aléatoire

## II.1. Simulation d'une v.a.r. : fondements mathématiques

### II.1.a) La loi (faible) des grands nombres

**Théorème 1.** *Loi (faible) des grands nombres*

*Soit $X$ une v.a.r. admettant un moment d'ordre 2.*

*Soit $(X_n)$ une suite de v.a.r. indépendantes et de même loi que $X$.*

*On a alors :* $$\forall \varepsilon > 0,\ \lim_{n\to+\infty} \mathbb{P}\left(\left|\frac{1}{n}\sum_{k=1}^{n} X_k - \mathbb{E}(X)\right| \geqslant \varepsilon\right) = 0$$

*(on dit que la suite de v.a.r. $\left(\frac{1}{n}\sum_{k=1}^{n} X_k\right)_{n\geqslant 1}$ converge en probabilité vers $\mathbb{E}(X)$.*

▶ Démontrer la loi faible des grands nombres.

Rappelons tout d'abord l'inégalité de Bienaymé-Tchebychev.
Soit $Y$ une v.a.r. discrète admettant une variance $\mathbb{V}(Y)$.

$$\forall \lambda > 0, \quad \mathbb{P}(|Y - \mathbb{E}(Y)| \geqslant \lambda) \leqslant \frac{\mathbb{V}(Y)}{\lambda^2}$$

- Soit $X$ une v.a.r. admettant un moment d'ordre 2.
- Soit $(X_n)$ une suite de v.a.r. indépendantes et de même loi que $X$.
- Notons alors $Y = \dfrac{1}{n}\sum_{k=1}^{n} X_k$. Par linéarité de l'espérance et propriété de la variance (les v.a.r. étant mutuellement indépendantes) :

× $$\begin{aligned}\mathbb{E}(Y) &= \mathbb{E}\left(\frac{1}{n}\sum_{k=1}^{n} X_k\right) = \frac{1}{n}\,\mathbb{E}\left(\sum_{k=1}^{n} X_k\right) = \frac{1}{n}\sum_{k=1}^{n}\mathbb{E}(X_k) = \frac{1}{n}\sum_{k=1}^{n}\mathbb{E}(X)\\ &= \frac{1}{n}\, n \times \mathbb{E}(X) = \mathbb{E}(X)\end{aligned}$$

× $$\begin{aligned}\mathbb{V}(Y) &= \mathbb{V}\left(\frac{1}{n}\sum_{k=1}^{n} X_k\right) = \frac{1}{n^2}\,\mathbb{V}\left(\sum_{k=1}^{n} X_k\right) = \frac{1}{n^2}\sum_{k=1}^{n}\mathbb{V}(X_k) = \frac{1}{n^2}\sum_{k=1}^{n}\mathbb{V}(X)\\ &= \frac{1}{n^2}\, n \times \mathbb{V}(X) = \frac{1}{n}\,\mathbb{V}(X)\end{aligned}$$

Ainsi, pour tout $\varepsilon > 0$ : $\mathbb{P}\left(\left|\dfrac{1}{n}\sum_{k=1}^{n} X_k - \mathbb{E}(X)\right| \geqslant \varepsilon\right) \leqslant \dfrac{\mathbb{V}(X)}{\varepsilon^2}\,\dfrac{1}{n} \underset{n\to+\infty}{\longrightarrow} 0$

### II.1.b) Intérêt de ce théorème en statistique inférentielle

**Définition**
La **statistique inférentielle** est une théorie mathématique qui a pour but de retrouver les propriétés d'une loi de probabilité à partir de l'observation d'un échantillon de valeurs.

Illustrons cette théorie par un exemple simple.

**Exemple**

- On considère une population $\mathscr{P}$ dont les membres appartiennent à l'une des catégories suivantes : ***1.*** enfant, ***2.*** adolescent, ***3.*** adulte. On aimerait connaître la proportion de chaque individu. En terme probabiliste, une telle information peut être modélisée comme suit :
  × on considère l'expérience aléatoire consistant à tirer au sort un individu dans $\mathscr{P}$,
  × on note $X$ la v.a.r. donnant le numéro de la catégorie de l'individu tiré.
  Ainsi, $\mathbb{P}([X = 3])$ donne la proportion d'adultes dans la population.
- Via cette modélisation, le problème consiste alors à trouver la loi de la v.a.r. $X$. Pour ce faire :
  ***1)*** soit on est capable d'effectuer un recensement complet de la population $\mathscr{P}$. Dans ce cas, il ne nous reste plus qu'à décrire la loi de $X$ grâce aux données obtenues.
  On parle alors de **statistique descriptive** : toutes les données sont accessibles et permettent de décrire le phénomène.
  ***2)*** soit on n'est pas capable d'effectuer ce recensement (on considère par exemple qu'il serait trop long ou coûteux de le faire). Dans ce cas, on va faire appel à la **statistique inférentielle** : à partir d'un échantillon d'observations, on essaie de retrouver la loi de $X$. L'idée est de trouver un résultat approché. Dans le meilleur des cas, on pourra encadrer l'erreur d'approximation commise.

- Dans le cas de notre population, on procède alors comme suit :
  - × on répète l'expérience consistant à tirer au sort un individu dans $\mathscr{P}$,
  - × on note $X_i$ la v.a.r. donnant le numéro de la catégorie du $i^{\text{ème}}$ individu tiré.

  On obtient ainsi une suite $(X_i)$ de v.a.r. indépendantes et de même loi que $X$ (on a notamment $\mathbb{P}([X_i = 3]) = \mathbb{P}([X = 3])$).
- Le théorème stipule que l'on peut approcher $\mathbb{E}(X)$ (la moyenne des catégories de $\mathscr{P}$) à l'aide d'un échantillon $(X_1, \dots, X_n)$. Pour ce faire, on observe la valeur $(x_1, \dots, x_n)$ de cet échantillon. Lorsque $n$ est grand, la moyenne des valeurs observées $\dfrac{1}{n}\sum_{k=1}^{n} x_k$ (c'est la moyenne statistique) devient proche de $\mathbb{E}(X)$.
- Mais cette information n'est pas très pertinente.
  Considérons, par exemple, une population de 100 individus. Si l'on sait que la moyenne des catégories est de 2.5, on obtient peu d'information sur la répartition de la population :
  - × $250 = 50 \times 3 + 50 \times 2 + 0 \times 1$.
  - × $250 = 80 \times 3 + 0 \times 2 + 10 \times 1$.
  - × $250 = 60 \times 3 + 30 \times 2 + 10 \times 1$.
  - × $250 = 66 \times 3 + 16 \times 2 + 18 \times 1$.
- La loi (faible) des grands nombres nous donne accès à une information plus précise comme la proportion d'adultes dans $\mathscr{P}$. Pour ce faire, on ne considère plus les v.a.r. $X$ et $X_i$, mais :
  - × $Z = \mathbb{1}_{[X=3]}$ la v.a.r. qui vaut 1 lorsque $X$ vaut 3 et 0 sinon. Alors :

$$\begin{aligned}
& \mathbb{E}(Z) \\
= \ & 1 \times \mathbb{P}([Z = 1]) + 0 \times \mathbb{P}([Z = 0]) \\
= \ & \mathbb{P}([Z = 1]) \\
= \ & \mathbb{P}([X = 1]) \times \cancel{\mathbb{P}_{[X=1]}([Z = 1])} + \mathbb{P}([X = 2]) \times \cancel{\mathbb{P}_{[X=2]}([Z = 1])} + \mathbb{P}([X = 3]) \times \mathbb{P}_{[X=3]}([Z = 1]) \\
= \ & \mathbb{P}([X = 3])
\end{aligned}$$

  - × $Z_i = \mathbb{1}_{[X_i=3]}$ la v.a.r. qui vaut 1 lorsque $X_i$ vaut 3 et 0 sinon.

  Le théorème stipule que la moyenne des valeurs observées $\dfrac{1}{n}\sum_{k=1}^{n} z_k$ de l'échantillon $(Z_1, \dots, Z_n)$ devient proche de $\mathbb{E}(Z) = \mathbb{P}(X = 3)$ lorsque $n$ grandit.

$\dfrac{1}{n}\sum_{k=1}^{n} X_k$ est une variable aléatoire, et $\dfrac{1}{n}\sum_{k=1}^{n} x_k$ est un nombre réel !

## II.2. Simulation d'une v.a.r. : en Python

Comme décrit en paragraphe ***I.2.c.***, la fonction `random` est un générateur de nombres **pseudo-aléatoires**. Il vérifie les propriétés suivantes.

- Si $X(\omega)$ désigne le résultat de l'appel `random()`, alors $X$ est une v.a.r. de loi uniforme sur $[0, 1[$.
- Si $X_1(\omega), \dots, X_n(\omega)$ désignent les résultats de $n$ appels successifs de `random()`, alors les variables aléatoires $X_1, \dots, X_n$ sont mutuellement indépendantes.
  *(on est donc dans le cadre d'application de la loi des grands nombres)*

En mathématiques, on travaille souvent avec des variables aléatoires, mais en informatique, on simule **une réalisation** de ces variables aléatoires ( ce qui correspond à la notion d'observation en statistique).

# III. Simulation d'une v.a.r. suivant une loi discrète usuelle

## III.0. Diagrammes en bâtons en Python

En guise d'illustration, considérons que l'on a affaire à :

× une v.a.r. $X$ pouvant prendre les valeurs $X(\Omega) = \{2, 3, 5, 7, 10\}$.

× une liste d'observations `Obs = [2, 10, 7, 5, 2, 7, 5, 7, 2, 2]`.

Cette situation se représente par un diagramme en bâtons contenant 5 **classes** (chaque valeur 2, 3, 5, 7, 10 produit une classe). Chacune de ces valeurs est portée en abscisse.

En ordonnée, on porte **l'effectif** de chaque classe, c'est à dire le nombre d'individus que comporte chaque classe. Dans notre exemple, on obtient :

× effectif de la classe 2 : 4
× effectif de la classe 3 : 0
× effectif de la classe 5 : 2
× effectif de la classe 7 : 3
× effectif de la classe 10 : 1

Ce qui correspond au tableau des effectifs : $[4, 0, 2, 3, 1]$.

### III.0.a) Création d'un diagramme en bâtons

► Écrire une fonction `position` qui prend en paramètre une liste `L` et un élément `elt` de la liste et qui renvoie la position de cet élément dans la liste.

```
def position(elt, L):
    n = len(L)
    for i in range(n):
        if L[i] == elt:
            return i
```

► Étant donnée une liste `cl` contenant les valeurs de chaque classe et une liste `Obs` contenant une liste d'observation, écrire une fonction `calcEffectif` qui renvoie le tableau des effectifs de chaque classe de l'observation.

```
def calcEffectif(cl, Obs):
    n = len(cl)
    tab = np.zeros(n)
    for elt in Obs:
        i = position(elt, cl)
        tab[i] = tab[i] + 1
    return tab
```

### III.0.b) Tracé d'un diagramme en bâtons : la fonction `bar`

- Provient du module `matplotlib.pyplot` (à faire : `import matplotlib.pyplot as plt`).
- Cette fonction s'appelle généralement avec deux arguments `absc` et `ord`.
  × `absc` : la liste des points sur lesquels les bâtons vont s'appuyer,
  × `ord` : la hauteur dans l'ordre de chaque bâton.
  Par défaut, chaque bâton est de largeur `0.8` mais on peut la changer en ajoutant comme paramètre d'appel `width = 0.2` (par exemple).
  On peut aussi modifier la couleur des bâtons en ajoutant le paramètre d'appel `color = 'b'`.

## III.1. Loi uniforme discrète

### III.1.a) Simulation à l'aide de la fonction `random`

On considère le programme suivant.

```
1  import random
2  import math
3
4  def uniforme(a,b):
5      return (a + math.floor(random.random()*((b-a)+1)))
```

**Programme 1** Simulation de la loi uniforme discrète sur $[\![a, b]\!]$

▶ Quel est le rôle de la fonction `uniforme` ? Expliquer.

L'appel `random.random()` produit un réel dans $[0, 1[$.
Par multiplication par `(b-a)+1`, ce résultat est transporté dans $[0, \mathtt{b}-\mathtt{a}+1[$.
Transformé ensuite en un entier dans $[\![0, \mathtt{b}-\mathtt{a}]\!]$ par application de la fonction `floor`.
Enfin, par ajout de `a`, on obtient un entier dans $[\![\mathtt{a}, \mathtt{b}]\!]$.
Le choix initial étant fait de manière uniforme dans $[0, 1[$, chaque entier de $[\![\mathtt{a}, \mathtt{b}]\!]$ est obtenu de manière équiprobable.

**Remarque**
La fonction `random.randomint` fournit le même résultat.

### III.1.b) Diagrammes en bâtons associés

Il s'agit maintenant de comparer :
× le diagramme en bâtons obtenu par `N` observations de la simulation de la loi uniforme,
× avec le diagramme en bâtons représentant les fréquences théoriques.
Pour plus de simplicité, on considère initialement une loi uniforme sur $[\![1, n]\!]$.

On considère le programme suivant.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Valeur des paramètres
5  N = 1000 # 1000 simulations
6  n = 4 # loi uniforme sur [[1,4]]
7
8  # Valeurs observées (résultat de la simulation)
9  Obs = [uniforme(1,n) for k in range(N)]
10
11 # Tableau des effectifs des observations
12 cl = np.linspace(1, n, n)
13 effectif = calcEffectif(cl, Obs)
14
15 # Tableau de la distribution de probabilité (valeurs théoriques)
16 P = np.zeros(n) for k in range(n):
17     P[k] = 1/n
```

**Programme 2** Étude de la loi uniforme discrète sur $[\![1, n]\!]$

▶ En ligne 9, on utilise une compréhension de liste.
Comment obtenir le même résultat avec une boucle `for` ?

```
for k in range(N):
    Obs = Obs + [uniforme(1, n)]
```

▶ Il reste alors à tracer les diagrammes en bâtons correspondants.
Compléter le programme suivant.

```
# Tracé des deux diagrammes
absc = np.linspace(1,n,n)
plt.bar(absc, P, color = 'r', width = 0.2)
plt.bar(absc + 0.2, effectif / N, color = 'b', width = 0.2)
plt.show()
```

**Programme 3** Tracé des diagrammes en bâtons correspondants

*(on pourra recopier ces lignes à la fin du programme précédent)*

▶ Comment peut-on adapter le programme précédent afin d'obtenir le diagramme associé à la simulation d'une loi uniforme discrète sur $[\![a, b]\!]$ (et plus seulement $[\![1, n]\!]$) ?

Il faut commencer par introduire `a` et `b` puis remplacer :

× `uniforme(1,n)` par `uniforme(a,b)`

× `np.linspace(1,n,n)` par `np.linspace(a,b,b-a+1)`

## III.2. Loi binomiale

### III.2.a) Simulation à l'aide de la fonction `random`

▶ Écrire une fonction `Bernoulli(p)` simulant une variable aléatoire de loi $\mathcal{B}(p)$.

```
def Bernoulli(p):
    if random.random() < p:
        return 1
    else:
        return 0
```

▶ Écrire une fonction `binomiale(n,p)` simulant une variable aléatoire de loi $\mathcal{B}(n, p)$.

```
def binomiale(n,p):
    S = 0
    for i in range(n):
        S = S + Bernoulli(p)
    return S
```

**Remarque**
La fonction `np.random.binomial(n, p)` fournit le même résultat.

### III.2.b) Diagrammes en bâtons associés

Il s'agit maintenant de comparer :

× le diagramme en bâtons construit à partir de 1000 simulations indépendantes,

× avec le diagramme en bâtons correspondant à la loi $\mathcal{B}(40, 0.3)$ (fréquences théoriques).

On considère le programme suivant.

```
1  # Valeur des paramètres
2  N = 1000
3  n = 40
4  p = 0.3
5
6  # Valeurs observées (résultat de la simulation)
7  Obs = []
8  for k in range(N):
9      Obs = Obs + [binomiale(n, p)]
10
11 # Tableau des effectifs des observations
12 cl = np.linspace(0, n, n+1)
13 effectif = calcEffectif(cl, Obs)
14
15 # Tableau de la distribution de probabilité (valeurs théoriques)
16 P = np.zeros(n+1)
17 for k in range(n+1):
18     comb = math.factorial(n)/(math.factorial(k) * math.factorial(n-k))
19     P[k] = comb * (p**k) * ((1-p)**(n-k))
```

**Programme 4** Étude de la loi $\mathcal{B}(40, 0.3)$

▶ En ligne 7, 8, 9, on utilise une boucle `for`.
Comment obtenir le même résultat avec une compréhension de liste ?

```
[binomiale(n, p) for k in range(N)]
```

▶ Que signifie $X \hookrightarrow \mathcal{B}(n, p)$ ? Dans quel type d'expérience cette loi est-elle utilisée.

***a)*** $X(\Omega) = [\![0, n]\!]$

***b)*** $\forall k \in X(\Omega),\ \mathbb{P}(X = k) = \binom{n}{k} p^k\ q^{n-k}$

On considère une expérience aléatoire qui consiste en une succession de n épreuves indépendantes, chacune d'entre elles ayant deux issues : succès obtenu avec probabilité $p$ et échec obtenu avec probabilité $q = 1 - p$. Autrement dit, l'expérience consiste à effectuer $n$ épreuves de Bernoulli identiques (i.e. de même paramètre) et indépendantes (le résultat de l'une ne dépend pas du résultat des autres).

Alors la v.a.r. $X$ donnant le nombre de succès de l'expérience vérifie : $X \hookrightarrow \mathcal{B}(n, p)$.

▶ Compléter la ligne 18 du programme précédent.

```
comb = math.factorial(n)/(math.factorial(k) * math.factorial(n-k))
```

▶ Il reste alors à tracer les diagrammes en bâtons correspondants.
Compléter le programme suivant.

```
# Tracé des deux diagrammes
absc = np.linspace(0, n, n+1)
# Diagramme de la distribution théorique
plt.bar(absc, P, color = 'r', width = 0.4)
# Diagramme des fréquences observées
plt.bar(absc + 0.5, effectif / N, color = 'b', width = 0.4)
plt.show()
```

**Programme 5** Tracé des deux diagrammes en bâtons

## III.3. Loi discrète quelconque

### III.3.a) Notation et fonction de répartition

On considère une v.a.r. finie $X$. On note alors :

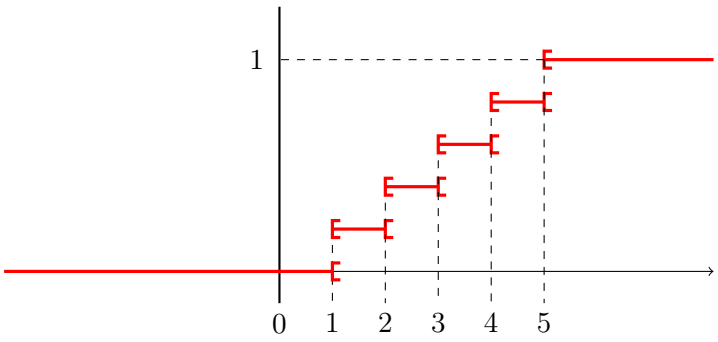
× $X(\Omega) = \{x_1, \dots, x_n\}$ le support de $X$,

× $\forall k \in [\![1, n]\!],\ p_k = \mathbb{P}(X_k = x_k)$,

× $\forall k \in [\![1, n]\!],\ r_k = \sum_{j=1}^{k} p_j$.

Ainsi, $r_0 = 0$, $r_n = 1$, et $(r_k)$ est strictement croissante.

▶ On considère $X$ une v.a.r. discrète telle que $X \hookrightarrow \mathcal{U}([\![1, 5]\!])$.
Tracer la fonction de répartition $F_X$.




▶ Donner les valeurs de $(x_i)$, $(p_k)$ et $(r_k)$ pour la variable $X$ précédente.

- $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$, $x_5 = 5$.
- $p_1 = \dots = p_5 = \dfrac{1}{5}$.
- $r_0 = 0$, $r_1 = \dfrac{1}{5}$, $r_2 = \dfrac{2}{5}$, $r_3 = \dfrac{3}{5}$, $r_4 = \dfrac{4}{5}$, $r_5 = 1$.

Les valeurs $(r_k)$, valeurs cumulées des probabilités, peuvent se lire sur le graphe de $F_X$ puisque : $r_k = F(x_k)$. Ce constat se retrouve dans la dénomination anglaise des fonctions de répartition : elles sont nommées **cumulative distribution function**.

### III.3.b) Méthode d'inversion

La proposition suivante va nous permettre de simuler toute loi discrète à partir de la simulation d'une loi uniforme sur $[0,1[$.

**Proposition 1.** *Méthode d'inversion*

*Soit $U$ une v.a.r. de loi uniforme sur $[0,1[$.*

*Soit $X$ une v.a.r. finie (on utilise les notations $(x_k)$, $(p_k)$ et $(r_k)$ comme ci-dessus).*

*Soit $Y$ une v.a.r. définie comme suit.*

- $Y(\Omega) = \{x_1, \dots, x_n\}$
- $\forall \omega \in \Omega,\ Y(\omega) = x_k$ *où* $k$ *est l'unique entier de* $[\![1,n]\!]$ *tel que :* $r_{k-1} \leqslant U(\omega) < r_k$

*Dans ce cas, la v.a.r. $Y$ a même loi que la v.a.r. $X$.*

▶ Écrire une fonction `sommeCumulee` prenant en paramètres la liste $[\mathtt{p_1}, \dots, \mathtt{p_n}]$ et renvoie la liste $[\mathtt{r_1}, \dots, \mathtt{r_n}]$ définie comme ci-dessus.

```
def sommeCumulee(P):
    """
    Prend en paramètre une liste P et \
    renvoie la liste des sommes cumulées
    """
    R = []
    r = 0
    for p in P:
        r = r + p
        R.append(r)
    return R
```

**Remarque**
La fonction `np.cumsum(P)` fournit le même résultat.

▶ Écrire une fonction `discreteQ(X, P)` prenant en paramètres les listes $[p_1, \dots, p_n]$ et $[x_1, \dots, x_n]$ et simulant une variable aléatoire à l'aide de la proposition précédente.

```
def discreteQ(X, P):
    R = sommeCumulee(P)
    u = random.random()
    i = 0
    while (u >= R[i]):
        i = i + 1
    return X[i]
```

▶ Tracer le diagramme en bâtons obtenu en réalisant 1000 simulations avec $P = [0.3, 0.2, 0.4, 0.1]$ et $X = [1, 2, 4, 5]$.

```
def histoQ(X, P):
    N = 10000 # 10000 simulations

    Obs = [discreteQ(X, P) for k in range(N)]
    cl = X
    plt.bar(cl, Obs/N, color = 'b', width = 0.8)
```

## III.4. Loi géométrique

### III.4.a) Simulation à l'aide de la fonction `bernoulli`

▶ En utilisant la fonction `Bernoulli`, écrire une fonction `geom(p)` qui simule une variable aléatoire de loi $\mathcal{G}(p)$.

```
def geom(p):
    rang = 0
    while Bernoulli(p) == 0 :
        rang = rang + 1
    return rang
```

**Remarque**
La fonction `np.random.geometric(p)` fournit le même résultat.

### III.4.b) Illustration de la loi des grands nombres

▶ Comment illustrer la loi des grands nombres dans le cas de la loi géométrique ? On pourra utiliser la fonction `np.random.geometric` du module `numpy`.

```
N = 10000
p = 0.3
Obs = [np.random.geometric(p) for k in range(N)]

# La moyenne des observés (empirique) proche de E(X) = 1/p
moyE = np.sum(Obs)/N
print(moyE)
```

### III.4.c) Simulation à l'aide de la méthode d'inversion

On considère la fonction suivante.

```
1 def geoinv(p):
2     return(math.ceil(math.log(1-random.random())/math.log(1-p)))
```

▶ En s'inspirant de la méthode d'inversion, justifier que la fonction précédente permet de simuler une variable aléatoire de loi $\mathcal{G}(p)$.

Soit $X$ une v.a.r. telle que $X \hookrightarrow \mathcal{G}(p)$. Notons $q = 1-p$.
La fonction de répartition de $X$ est définie par : $F_X(k) = 1 - q^{\lfloor k \rfloor}$.
On a, d'après le théorème précédent :

$$\begin{aligned} X(w) = k &\Leftrightarrow r_{k-1} \leqslant U(w) < r_k \\ &\Leftrightarrow F(x_{k-1}) \leqslant U(w) < F(x_k) \\ &\Leftrightarrow 1 - q^{k-1} \leqslant U(w) < 1 - q^k \\ &\Leftrightarrow q^k < 1 - U(w) \leqslant q^{k-1} \\ &\Leftrightarrow k \ln(q) < \ln(1-U(w)) \leqslant (k-1)\ln q \\ &\Leftrightarrow k > \frac{\ln(1-U(w))}{\ln q} \geqslant (k-1) \end{aligned}$$

Or : $\ln(q) = \ln(1-p)$. Et ainsi : $\dfrac{\ln(1-U(w))}{\ln(1-p)} < k \leqslant \dfrac{\ln(1-U(w))}{\ln(1-p)} + 1.$
On en conclut que : $k = \left\lceil \dfrac{\ln(1-U(w))}{\ln(1-p)} \right\rceil$

## III.5. Loi de Poisson

### III.5.a) Simulation exacte de la loi de Poisson

▶ En s'inspirant de la méthode d'inversion, écrire une fonction `poisson(mu)` qui simule une variable aléatoire de loi $\mathcal{P}(\mu)$.

```
1 def poisson(mu):
2     x = random.random()
3     F = 0
4     i = 0
5     while F<x:
6         i = i+1
7         F = F + math.exp(-mu)*mu**i/math.factorial(i)
8     return (i-1)
```

**Remarque**
La fonction `np.random.poisson` conduit au même résultat.

En Python, `lambda` est un mot-clé, il est interdit de l'utiliser comme nom de variable.

### III.5.b) Vérification de la simulation à l'aide d'un intervalle de confiance

▶ Soit $X$ une variable aléatoire de loi $\mathcal{P}(\mu)$. À l'aide de l'inégalité de Bienaymé-Tchebychev, déterminer un intervalle $I$ tel que $\mathbb{P}(X \in I) \geqslant 0,95$.

Vérifier le résultat trouvé à l'aide d'une simulation.

- D'après l'inégalité de Bienaymé-Tchebychev : $\mathbb{P}(|X - \mathbb{E}(X)| \geqslant \varepsilon) \leqslant \dfrac{\mathbb{V}(X)}{\varepsilon^2}$.
  Ainsi : $1 - \mathbb{P}(|X - \mathbb{E}(X)| \geqslant \varepsilon) \geqslant 1 - \dfrac{\mathbb{V}(X)}{\varepsilon^2}$ soit $\mathbb{P}(|X - \mathbb{E}(X)| < \varepsilon) \geqslant 1 - \dfrac{\mathbb{V}(X)}{\varepsilon^2}$.
- Or $\mathbb{E}(X) = \mu$, $\mathbb{V}(X) = \mu$ donc $1 - \dfrac{\mathbb{V}(X)}{\varepsilon^2} = 1 - \dfrac{\mu}{\varepsilon^2}$. Enfin :

$$|X - \mu| < \varepsilon \quad \Leftrightarrow \quad -\varepsilon < X - \mu < \varepsilon \quad \Leftrightarrow \quad X - \varepsilon < \mu < X + \varepsilon$$

- Pour connaître $\mu$ avec un niveau de confiance $1 - \alpha = 0.95$ (*i.e.* $\alpha = 0.05$), il faut choisir $\varepsilon$ tel que : $1 - \mu \geqslant 1 - \alpha \quad \Leftrightarrow \quad \alpha \geqslant \dfrac{\mu}{\varepsilon^2} \quad \Leftrightarrow \quad \varepsilon \geqslant \sqrt{\dfrac{\mu}{\alpha}}$.
- En prenant $\alpha = 0.005$ et $\varepsilon = \sqrt{\dfrac{\mu}{\alpha}}$, on obtient un intervalle de confiance $I = ]X - \varepsilon, X + \varepsilon[$ permettant de tester la simulation précédente.
  D'un point de vue théorique : $\mathbb{P}(\mu \in I) \geqslant 0.95$. Si on teste $n = 10000$ fois notre simulation, $\mu$ devrait donc appartenir à $I$ plus de 9500 fois. Vérifions-le :

```
N = 10000
mu = 3
alpha = 0.05
eps = math.sqrt(mu / alpha)
X = [poisson(mu) for k in range(N)]
somme = 0
for x in X:
    if (x - eps < mu) and (mu < x + eps):
        somme = somme + 1
```

- Via cette simulation, on trouve $\mu$ près de 9995 fois dans $I$. C'est bien mieux qu'escompté. La raison est simple : l'inégalité de Bienaymé-Tchebychev est assez grossière. De ce fait, elle n'est pas adaptée à l'écriture d'intervalles de confiance.

### III.5.c) Approximation de la loi de Poisson

**Proposition 2.**

*Soit $(p_n)$ une suite de réels de $]0,1[$ telle que : $np_n \underset{n\to+\infty}{\longrightarrow} \mu > 0$.*

*On considère une suite de variables aléatoires $(X_n)$ telle que : $X_n \hookrightarrow \mathcal{B}(n, p_n)$.*

*Alors, pour tout $k \in \mathbb{N}$ :* $\boxed{\lim_{n\to+\infty} \mathbb{P}(X_n = k) = e^{-\mu}\,\dfrac{\mu^k}{k!}}$

*On dit que la suite $(X_n)$ converge en loi vers une variable aléatoire de loi $\mathcal{P}(\mu)$.*

▶ À l'aide de cette approximation, écrire une fonction `poissonRare(mu, n)` qui simule une variable aléatoire dont la loi est **proche** de la loi de Poisson $\mathcal{P}(\mu)$. On pourra choisir $p_n = \frac{\mu}{n}$.

```
def poissonRare(mu, n):
    return binomiale(n, mu/n)
```

### III.5.d) Diagrammes en bâtons associés

▶ Compléter le programme suivant permettant de tracer sur un même diagramme la distribution théorique de la loi de poisson $\mathcal{P}(1)$ ainsi que deux distributions approchées.
Les diagrammes obtenus seront représentées pour les abscisses $[\![0, 15]\!]$.

```
# Valeur des paramètres
N = 10000
m = 15
mu = 1
n1 = 50
n2 = 200

# Distribution théorique
P = [math.exp(-mu)*(mu**k)/math.factorial(k) for k in range(m+1)]

# 1er Tableau des effectifs des observations
Obs1 = [poissonRare(mu, n1) for k in range(N)]
cl = np.linspace(0, m, m+1)
effectif1 = calcEffectif(cl, Obs1)

# 2eme Tableau des effectifs des observations
Obs2 = [poissonRare(mu, n2) for k in range(N)]
effectif2 = calcEffectif(cl, Obs2)

# Diagramme en bâtons associés
absc = np.linspace(0, m, m+1)
plt.bar(absc, P, color = 'r', width = 0.2)
plt.bar(absc + 0.2, effectif1 / N, color = 'b', width = 0.2)
plt.bar(absc + 0.4, effectif2 / N, color = 'y', width = 0.2)

plt.show()
```

**Remarque**
On aurait aussi pu écrire directement la distribution approchée :

```
PApp1 = [ math.factorial(n1)/(math.factorial(k)*math.factorial(n1-k)) * \
(mu/n1)**k * (1-mu/n)**(n1-k) for k in range(m+1)]
```

et dessiner le digramme correspondant.