IFT3655 - Devoir 6

Mathias La Rochelle

Le samedi 2 novembre 2024

Question 1

Question 1.1

Nous savons que $h(x) \propto g(x)$. Nous avons aussi que la fonction de répartition G associée à la densité de probabilité g est

$$G(x) = \int_0^x g(t)\,dt = \int_0^x \frac{h(t)}{a}\,dt$$

Pour trouver cette CDF, nous allons procéder par cas :

Pour $x < x_1$:

$$\begin{aligned} G(x) &= \int_0^x \frac{f(x_1)}{a}\,dt \\ &= \frac{f(x_1)}{a}\,[t]_0^x \\ &= \frac{xf(x_1)}{a} \end{aligned}$$

Pour $x_1 \leq x \leq x_2$:

$$\begin{aligned} G(x) &= G(x_1) + \int_{x_1}^x \frac{16/9}{a}\,dt \\ &= G(x_1) + \frac{16}{9a}\,[t]_{x_1}^x \\ &= G(x_1) + \frac{16}{9a}(x - x_1) \end{aligned}$$

Pour $x > x_2$:

$$\begin{aligned} G(x) &= G(x_2) + \int_{x_2}^x \frac{f(x_2)}{a}\,dt \\ &= G(x_2) + \frac{f(x_2)}{a}\,[t]_{x_2}^x \\ &= G(x_2) + \frac{f(x_2)}{a}(x - x_2) \end{aligned}$$

Donc nous avons alors

$$G(x) = \begin{cases} \dfrac{f(x_1)x}{a}, \text{ si } x < x_1 \\ G(x_1) + \dfrac{16(x - x_1)}{9a}, \text{ si } x_1 \leq x \leq x_2 \\ G(x_2) + \dfrac{f(x_2)(x - x_2)}{a}, \text{ si } x_2 < x \end{cases}$$

La fonction inverse d'une CDF permet d'obtenir une valeur x sachant la probabilité p correspondante. Nous avons que $p = V$ où $V \sim \mathcal{U}(0, 1)$. Grâce à ceci, on peut écrire $G^{-1}(V) = x$ de la manière suivante :

$$G^{-1}(p) = \begin{cases} \dfrac{aV}{f(x_1)}, \text{ si } V < G(x_1) \\ x_1 + \dfrac{9a(V - G(x_1))}{16}, \text{ si } G(x_1) \leq V \leq G(x_2) \\ x_2 + \dfrac{a(V - G(x_2))}{f(x_2)}, \text{ si } G(x_2) < V \end{cases}$$

Et l'algorithme pour générer X par inversion :

```
fonction generateurParInversion()
    générer V à partir d'une loi U(0,1)
    si V < G(x_1), alors
        X = a * V / f(x_1)
    sinon si G(x_1) <= V <= G(x_2), alors
        X = x_1 + (9 * a * (V - G(x_1))) / 16
    sinon
        X = x_2 + a * (V - G(x_2)) / f(x_2)
    retourner X
```

Ici, $f(x)$ est la densité de notre variable aléatoire $X \sim \mathcal{B}(\alpha, \beta)$.

Question 1.2

On peut ré-utiliser notre algorithme à la question **1.1**.

```
fonction generateurParRejet()
    répéter
        calculer X avec generateurParInversion()
        générer V à partir d'une loi U(0,1)
    jusqu'à ce que a * V * g(X) <= f(X)
    retourner X
```

Question 2

Pour les questions qui vont suivre, les fichiers de code se retrouveront dans le `zip` fourni. Pour les questions **2.1**, **2.2**, **2.3**, **2.4** et **3**, le nom des fichiers sont `UniformGenerator`, `SequentialSearch`, `IndexSearch`, `GeneratorSpeedCalculator` et `GibbsSampling` respectivement.

Question 2.1

La classe `MRG32k3a` utilise déjà une implémentation 64 bits et le vecteur initial à 6 entiers contient déjà initialisé avec les valeurs 12345. Source : ici. C'est donc un bon outil pour vérifier la récurrence que nous avons implémentée. Dans cette dernière, on utilise le type `long` pour permettre l'implémentation de nombres entiers en 64 bits. On cast le z_n sur le type `double` pour obtenir une valeur décimale étant donné qu'on cherche une valeur entre 0 et 1. Voici ce qui se passe quand on exécute le fichier `UniformGenerator` pour $n = 4$:

output

```
Entrez un nombre de variables aléatoires :
4
L'état initial est : [12345, 12345, 12345, 12345, 12345, 12345]

Avec MRG32k3a :
u0 = 0.12701112204657714 avec l'état [12345, 12345, 3023790853, 12345, 12345, 2478282264]
Avec notre récurrence :
u0 = 0.12701112207614923 avec l'état [12345, 12345, 3023790853, 12345, 12345, 2478282264]

Avec MRG32k3a :
u1 = 0.3185275653967945 avec l'état [12345, 3023790853, 3023790853, 12345, 2478282264, 1655725443]
Avec notre récurrence :
u1 = 0.31852756547095745 avec l'état [12345, 3023790853, 3023790853, 12345, 2478282264, 1655725443]

Avec MRG32k3a :
u2 = 0.3091860155832701 avec l'état [3023790853, 3023790853, 3385359573, 2478282264, 1655725443, 2057415812]
Avec notre récurrence :
u2 = 0.30918601565525805 avec l'état [3023790853, 3023790853, 3385359573, 2478282264, 1655725443, 2057415812]

Avec MRG32k3a :
u3 = 0.8258468629271136 avec l'état [3023790853, 3385359573, 1322208174, 1655725443, 2057415812, 2070190165]
Avec notre récurrence :
u3 = 0.825846863119396 avec l'état [3023790853, 3385359573, 1322208174, 1655725443, 2057415812, 2070190165]
```

Question 2.2

Nous gérons le cas potentiel où $\lambda \leq 0$ et $\mathcal{U} \leq 0$, $1 \leq \mathcal{U}$ en utilisant une recherche séquentielle. Source : ici. On ne procède tout simplement pas au calcul si les valeurs ne respectent pas les limites. Le reste est plutôt simple. Voici ce qui se passe quand on exécute le fichier `SequentialSearch` pour $\lambda = 25$ et $\mathcal{U} = 0.8$:

output

```
Entrez la moyenne de votre distribution de Poisson :
25
Entrez votre probabilité entre 0 et 1 :
0.8
La valeur X obtenue est de 29
```

Et si aucune des valeurs entrées ne respectent les conditions :

output

```
Entrez la moyenne de votre distribution de Poisson :
-1
Entrez votre probabilité entre 0 et 1 :
1
Les valeurs entrées ne respectent pas les conditions limites.
```

Question 2.3

Nous créons un tableau de probabilités cumulatives d'une loi de Poisson pour $x = 0, 1, ..., 50$. Avec une valeur U fixe entre 0 et 1, choisie par l'utilisateur, nous retrouvons la valeur X correspondant à cette probabilité. Durant la conception de ce code, le seul problème rencontré était l'initialisation du tableau. Voici ce qui se passe quand on exécute le fichier `IndexSearch` pour $\mathcal{U} = 0.8$:

output

```
Entrez votre probabilité entre 0 et 1 :
0.8
Indice trouvé : 29
```

Et si $\mathcal{U}$ ne respecte pas les conditions :

output

```
Entrez votre probabilité entre 0 et 1 :
1.1
Exception in thread "main" java.lang.IllegalArgumentException
        at mesdevoirs.IndexSearch.mainIndexSearch.java:29
```

Question 2.4

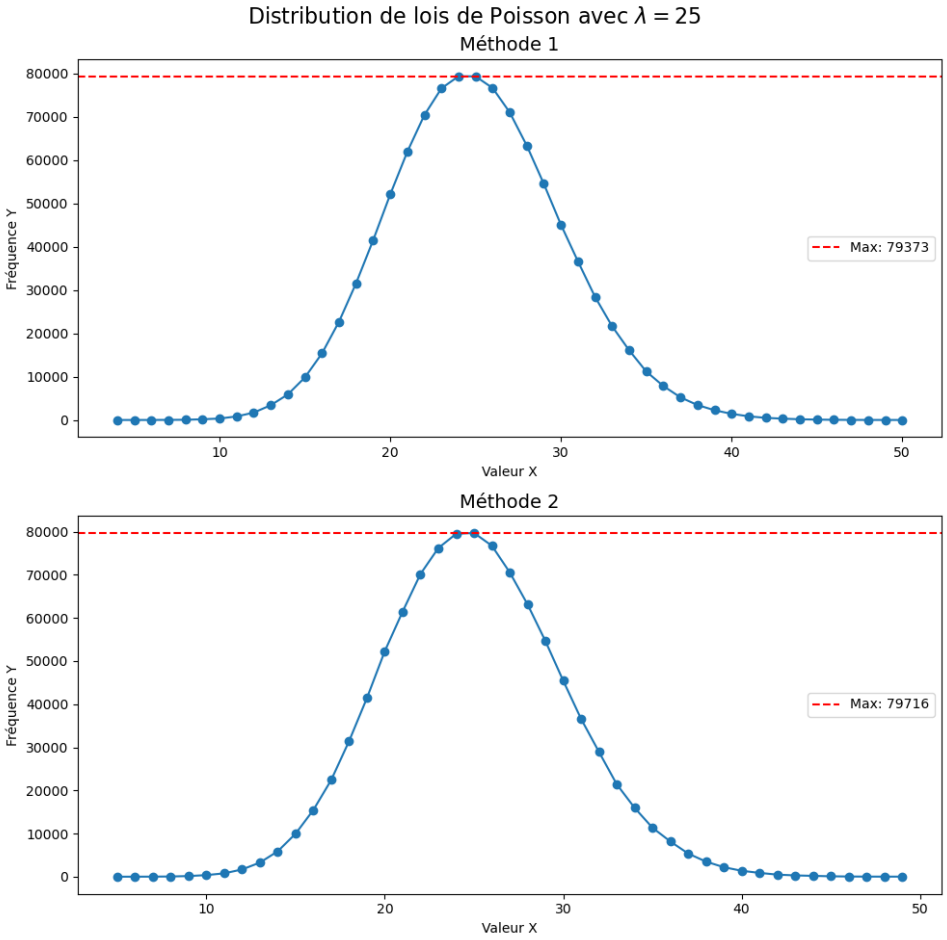
Comme on peut voir ci-dessous, les fréquences semble suivre une loi de Poisson dans les deux cas. On pourrait même dire que ça ressemble fortement à une distribution gaussienne, ce qui ne serait pas bête parce que nous avons un N assez grand. Le temps d'exécution pour la fonction en **b)** est de `700 ms` et le temps d'exécution pour la fonction en **c)** est de `61 ms`. Ces valeurs ne sont pas des moyennes mais si on le ferait, elles seraient tout de même proches de ces dernières. Cependant, elles nous disent que la méthode 2, i.e. la recherche par indice, est en moyenne **10x** plus rapide que la méthode 1, i.e. la recherche séquentielle. Cela est sûrement dû au fait qu'on calcule les CDF avant les calculs. Voici ce qui se passe quand on exécute le fichier `GeneratorSpeedCalculator` :

output

```
Méthode 1 :
Temps d'exécution : 700 ms
Nombre total des fréquences : 999996

Méthode 2 :
Temps d'exécution : 61 ms
Nombre total des fréquences : 1000001
```

Graphique explicatif :



Le code de ce graphique est :

```
1
2   import matplotlib.pyplot as plt
3
4   X1, Y1, X2, Y2 = [], [], [], []
5   with open('./Devoir 6/output1.txt') as file:
6       for line in file:
7           line_pieces = line.strip().split(",")
8           X1.append(float(line_pieces[0]))
9           Y1.append(float(line_pieces[1]))
10
11  with open('./Devoir 6/output2.txt') as file:
12      for line in file:
13          line_pieces = line.strip().split(",")
14          X2.append(float(line_pieces[0]))
```

```
            Y2.append(float(line_pieces[1]))

    max_freq_1 = max(Y1)
    max_freq_2 = max(Y2)

    fig, axs = plt.subplots(2, 1, figsize=(10, 10))
    fig.suptitle(r'Distribution de lois de Poisson avec $\lambda = 25$',
                                                                fontsize=16)
    axs[0].plot(X1, Y1, marker='o')
    axs[0].set_title('Méthode 1', fontsize=14)
    axs[0].set_xlabel('Valeur X')
    axs[0].set_ylabel('Fréquence Y')
    axs[1].plot(X2, Y2, marker='o')
    axs[1].set_title('Méthode 2', fontsize=14)
    axs[1].set_xlabel('Valeur X')
    axs[1].set_ylabel('Fréquence Y')

    axs[0].axhline(y=max_freq_1, color='red', linestyle='--',
                                            label=f'Max: {int(max_freq_1)}')
    axs[1].axhline(y=max_freq_2, color='red', linestyle='--',
                                            label=f'Max: {int(max_freq_2)}')
    axs[0].legend()
    axs[1].legend()

    plt.tight_layout()
    plt.show()

```

Question 3

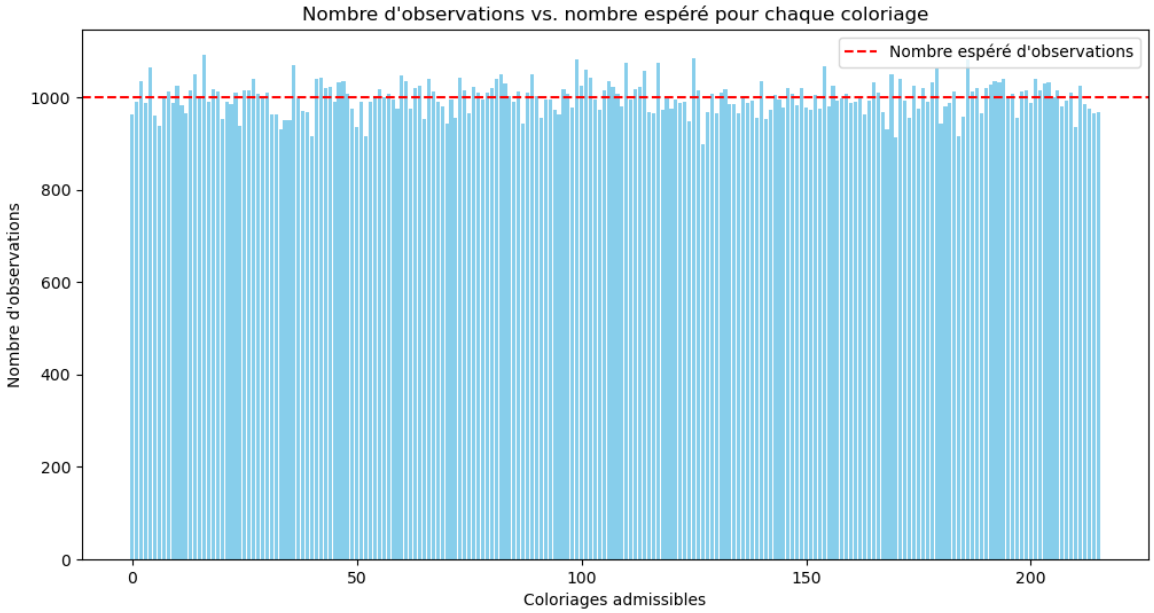
Tout d'abord, nous calculons la statistique du test de Pearson (χ^2) et obtenons une valeur de `262.564` ce qui semble élevé par rapport à la valeur critique attendue pour un degré de liberté de 215. Ensuite, en calculant la valeur p, nous trouvons environ `0.0148` ce qui est inférieur au seuil habituel $\alpha = 0.05$. Cela confirme qu'il y a de bonnes raisons à croire que la distribution obtenue ne suit pas une loi uniforme, i.e., qu'il y a une différence statistique signtificative entre les données observées et celles qui proviendraient d'une distribution uniforme. Cela laisse croire que chaque coloriage admissible n'a pas la même probabilité d'être tiré qu'un autre. Voici ce qui se passe quand on exécute le fichier `GibbsSampling` :

output

```
Le coloriage 3,4,3,1,2, a été visité 963
Le coloriage 1,3,1,2,3, a été visité 992
Le coloriage 4,2,1,3,4, a été visité 1036
Le coloriage 4,2,4,1,4, a été visité 989
...
Le coloriage 2,1,3,4,2, a été visité 985
Le coloriage 1,4,2,3,1, a été visité 977
Le coloriage 2,3,2,1,2, a été visité 966
Le coloriage 1,4,1,3,4, a été visité 969

Nombre total de coloriage admissible : 216000
```

Graphique explicatif :



Le code des valeurs obtenues et du graphique est :

```python

import matplotlib.pyplot as plt
import scipy.stats as stats

observations = []
with open('./Devoir 6/output3.txt') as file:
    for line in file:
        observations.append(int(line.strip()))

expected_observations = 1000

chi_squared = 0
for observation in observations:
    chi_squared += ((observation - expected_observations) ** 2) /
                                                    expected_observations

print("Khi-carré :", chi_squared)

dof = len(observations) - 1
p_value = 1 - stats.chi2.cdf(chi_squared, dof)

print("Valeur-p:", round(p_value, 4))

```

```
    plt.figure(figsize=(12, 6))
    plt.bar(range(216), observations, color='skyblue')
    plt.axhline(y=expected_observations, color='r', linestyle='--',
                                        label='Nombre espéré d\'observations')
    plt.xlabel('Coloriages admissibles')
    plt.ylabel('Nombre d\'observations')
    plt.title('Nombre d\'observations vs. nombre espéré pour chaque coloriage')
    plt.legend()
    plt.show()

```