Simulation

Rapport de projet

Activité d'Apprentissage S-INFO-018

LAVEND'HOMME Thomas FLORIANI Laurence

Année Académique 2020-2021
Troisième année de Bachelier en Sciences Informatiques
Première année de Master en Sciences Informatiques

Faculté des Sciences, Université de Mons

18 mai 2021

Table des matières

1 Introduction

Dans le cadre du cours de simulation, il nous est demandé d'implémenter un générateur de nombre pseudo aléatoire ainsi que les tests permettant de vérifier leur répartition uniforme.

En premier lieu, nous avons dû tester la répartition uniforme des 2.000.000 premières décimales de la constante de Néper. Afin d'y parvenir nous avons implémenté plusieurs tests : le test du χ^2, du poker et du gap

En second lieu, nous devions implémenter un générateur de nombres pseudo-aléatoire compris entre 0 et 1. Nous avons vérifié leur répartition uniforme en utilisant plusieurs tests, nous avons choisis les test du χ^2, de Kolmogorov-Smirnov et du poker

Enfin, nous avons comparé les nombres que nous avions obtenus avec ceux fournis en utilisant le module `random` de Python.

2 Répartition des décimales de la constante de Néper

2.1 Premier aperçu

Afin d'obtenir une première visualisation de la répartition des décimales de la constante de Néper, un histogramme a été construit. Les classes sont dans ce cas les chiffres de 0 à 9 donc 10 classes au total. La figure 1 montre cet histogramme et le tableau 1 contient les effectifs observés.

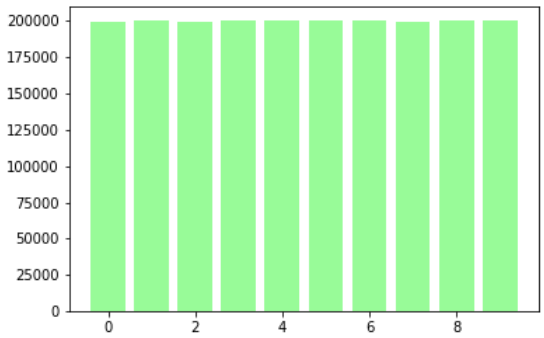

FIGURE 1 – Répartition des décimales de l'exponentielle

Une première conclusion peut être tirée à ce stade. En effet, la répartition semble bel et bien uniforme. Il faut maintenant passer aux tests pour le confirmer.

Chiffres	0	1	2	3	4
Effectifs	199093	200171	199471	200361	199923
Chiffres	5	6	7	8	9
Effectifs	200285	200395	199789	200098	200414

TABLE 1 – Table des effectifs observés dans les 2000000 premières décimales de l'exponentielle

2.2 Test du χ^2

Le test du chi carré ou chi deux permet de déterminer la nature d'une répartition, qu'elle soit continue ou discrète. Pour ce premier test du χ^2, le but est de déterminer si la répartition est bien uniforme ou non.

La première étape consiste à répartir les décimales en k classes et à construire l'histogramme de la répartition des effectifs observés. Le nombre de classes, ici, est de 10, trivial. Cette étape a déjà été effectuée au point 2.1.

Pour la seconde étape, il faut calculer l'effectif théorique auquel sera comparé l'effectif observé. Dans une distribution uniforme, chaque chiffre a une probabilité de $\frac{1}{10}$ d'apparaître. Il reste à multiplier cette dernière par le nombre total de décimales, ici de 2000000. Ce qui donne comme effectifs théoriques :

Chiffres	0	1	2	3	4
Effectifs	200000	200000	200000	200000	200000
Chiffres	5	6	7	8	9
Effectifs	200000	200000	200000	200000	200000

TABLE 2 – Table des effectifs théoriques pour $N = 2000000$

Lors de la troisième étape, il faut calculer l'écart existant entre les effectifs observés dans l'échantillon et les effectifs théoriques attendus.

Soit o_i les effectifs observés et e_i les effectifs théoriques,

$$K_r = \sum_{i=1}^{10} \frac{(o_i - e_i)^2}{e_i} = 8.65376$$

Ce nombre permet de déterminer à quel point les données observées sont différentes ou non des données théoriques.

Enfin, la quatrième et dernière étape consiste à comparer la valeur précédemment calculée à la valeur du χ^2 à 9 degrés de liberté et avec un risque d'erreur de 5%. Remarque : le nombre de degrés de liberté s'obtient en soustrayant 1 au nombre de classes.

$$\chi_9^2 = 16.91898$$

En conclusion, l'hypothèse d'avoir une distribution discrète uniforme est acceptée si $K_r <= \chi_9^2$. Les résultats obtenus sont $8.65376 <= 16.91898$, ce qui signifie que la répartition des décimales de la constante de Néper est uniforme.

2.3 Test du poker

Le principe de ce test est de comparer les fréquences théoriques de l'apparition de certaines mains au poker aux fréquences effectivement observées. La taille des mains choisies est de quatre chiffres et les types de mains examinés sont :

- Les chiffres tirés sont tous différents
- La liste de chiffres comporte une paire
- La liste de chiffres comporte trois chiffres identiques
- La liste de chiffres comporte deux paires
- Les chiffres tirés sont tous identiques

Le tableau 3 présente les effectifs observés pour les types de main précédemment cités.

Type de main	**Effectif observé**
Tous différents	251785
Une paire	216258
Trois chiffres identiques	18050
Deux paires	13413
Tous identiques	494

TABLE 3 – Table des effectifs observés pour chaque type de main

Le tableau 4 contient les probabilités et les effectifs théoriques associés à chaque type de main. Remarque : La somme des probabilités est bien égal à 100%, ce qui signifie que tout les cas possibles ont bien été considérés.

Type de main	**Probabilité théorique**	**Effectif théorique**
Tous différents	$1 * 0.9 * 0.8 * 0.7 = 50.4\%$	1008000
Une paire	$6 * (1 * 0.9 * 0.8 * 0.1) = 43.2\%$	864000
Trois chiffres identiques	$4 * (1 * 0.1 * 0.1 * 0.9) = 3.6\%$	72000
Deux paires	$3 * (1 * 0.1 * 0.9 * 0.1) = 2.7\%$	54000
Tous identiques	$1 * 0.1 * 0.1 * 0.1 = 0.1\%$	2000

TABLE 4 – Table des probabilités et effectifs théoriques pour chaque type de main

Enfin, il reste à effectuer un test du χ^2 à partir des effectifs observés et des effectifs théoriques précédemment calculés. Dans ce cas, le nombre de classes correspond au nombre de types de main possibles.

$$K_r = 1.263155 \text{ et } \chi^2_4 = 9.487729$$

En conclusion, le test du χ^2 est réussi car le K_r est bien inférieur ou égal à la valeur du χ^2 à 4 degrés de liberté et avec un risque d'erreur de 5%. Ce qui signifie que la répartition observée des types de main correspond bien à la répartition théorique.

2.4 Test du gap

Le principe de ce test consiste à calculer les effectifs des longueurs de séquence de chiffres n'appartenant pas à l'intervalle choisi.

Dans un premier temps, il faut choisir deux chiffres, a et b, qui définiront l'intervalle d'appartenance. Les chiffres appartenant effectivement à cet intervalle sont dits marqués tandis que les autres sont dits non marqués. Dans le cadre du test effectué sur les décimales de la constante de Néper, a prend la valeur de 0 alors que b prend celle de 5. L'intervalle a donc la forme : $[0, 5[$

Dans un second temps, en parcourant la liste non triée, il faut calculer les distances existantes entre deux chiffres marqués. Une liste répertorie l'ensemble des longueurs possibles, il reste à calculer la fréquence pour chacune d'entre elles. Le tableau 5 présente les résultats obtenus à ce stade.

Longueur	1	2	3	4	5	6
Effectif	498509	250018	125357	62699	31187	15611
Longueur	7	8	9	10	11	12
Effectif	7818	3926	1995	956	473	223
Longueur	13	14	15	16	17	18
Effectif	124	59	29	22	8	3
Longueur	19	20	21	22	23	
Effectif	0	1	0	0	1	

TABLE 5 – Table des effectifs observés lors du test du gap

Dans un troisième temps, il faut calculer les effectifs théoriques pour chaque distance précédemment calculée.La probabilité pour un chiffre d'être marqué est :

$$p = (b - a)/n = (5 - 0)/10 = 1/2$$

A l'aide de cette probabilité, il est désormais possible de calculer la probabilité d'apparition pour chaque longueur i :

$$P_i(X = i) = (1 - p)^{i+1}$$

Pour chaque distance possible, les effectifs théoriques s'obtiennent en multipliant la probabilité correspondante par le nombre total gaps, ici 999019.

La table 6 montre les probabilités ainsi que les effectifs théoriques calculés.

Longueur	*1*	*2*	*3*	*4*	*5*	*6*
Proba [%]	50	25	12.5	6.25	3.125	1.5625
Effectif	499509.5	249754.75	124877.375	62438.6875	31219.3438	15609.6719
Longueur	*7*	*8*	*9*	*10*	*11*	*12*
Proba [%]	$7.8125e^{-1}$	$3.9062e^{-1}$	$1.9531e^{-1}$	$9.7656e^{-2}$	$4.8828e^{-2}$	$2.4414e^{-2}$
Effectif	7804.8359	3902.4180	1951.2090	975.6045	487.8022	243.9011
Longueur	*13*	*14*	*15*	*16*	*17*	*18*
Proba [%]	$1.2207e^{-2}$	$6.1035e^{-3}$	$3.0518e^{-3}$	$1.5259e^{-3}$	$7.6294e^{-4}$	$3.8147e^{-4}$
Effectif	121.9506	60.9753	30.4876	15.4382	7.6219	3.8109
Longueur	*19*	*20*	*21*	*22*	*23*	
Proba [%]	$1.9073e^{-4}$	$9.5367e^{-5}$	$4.7683e^{-5}$	$2.3842e^{-5}$	$1.1921e^{-5}$	
Effectif	1.9055	0.9527	0.4764	0.2382	0.1191	

TABLE 6 – Table des effectifs théoriques pour chaque longueur de gap

Dans un quatrième et dernier temps, il reste à effectuer un test du χ^2 avec les effectifs observés et les effectifs théoriques précédemment calculés. Dans ce cas la quantité de classes correspond au nombre de longueurs différentes observées.

$$K_r = 21.519256 \text{ et } \chi^2_{22} = 33.924438$$

En conclusion, le test du χ^2 est réussi car le K_r est bien inférieur ou égal à la valeur du χ^2 à 22 degrés de liberté et avec un risque d'erreur de 5%. Ce qui signifie que la répartition observée des longueurs de gaps correspond bien à la répartition théorique.

3 Générateurs de nombres aléatoires

3.1 Générateur 1 : Approche naïve

3.1.1 Présentation

La première méthode envisagée est la plus simple possible. Le générateur démarre à un indice de la chaîne de nombres, celui-ci est dicté par la graine. Ensuite, le générateur avance linéairement dans la séquence de décimales de la constante de Néper et regroupe les nombres par paquets de taille constante. Chaque paquet de nombres sert à définir les décimales du nombre généré.

Par exemple, avec une taille de paquets de 10 :

...11730123 | 8197068416 | 1403970198 | 3767932068 | 328237646480 ...

On obtiendra la séquence 0.8197068416 ; 0.1403970198 ; 0.3767932068 ; 0.328237646480.

3.1.2 Résultats obtenus

Le tableau 7 contient les effectifs pour chaque classe lors de la génération de 20 000 nombres.

Intervalle	**Effectif**	**Intervalle**	**Effectif**
$[0, 0.1[$	2027	$[0.5, 0.6[$	2013
$[0.1, 0.2[$	2025	$[0.6, 0.7[$	1939
$[0.2, 0.3[$	1983	$[0.7, 0.8[$	2015
$[0.3, 0.4[$	1976	$[0.8, 0.9[$	1993
$[0.4, 0.5[$	2070	$[0.9, 1[$	1950

TABLE 7 – Effectif par classe pour le premier générateur

L'histogramme 2 compare la répartition des nombres obtenus à l'aide du premier générateurs à ceux obtenus en utilisant le module `Random` de Python.

3.1.3 Problème

Il est impossible, par le fonctionnement du générateur, de générer les nombres qui ont une précision (en base décimale) qui est supérieure à la précision donnée en paramètre. Donc, même si chaque nombre a une probabilité équivalente d'apparaître, la distribution n'est pas uniforme car certains nombres ont une probabilité nulle d'être générés et d'autres une probabilité non nulle.

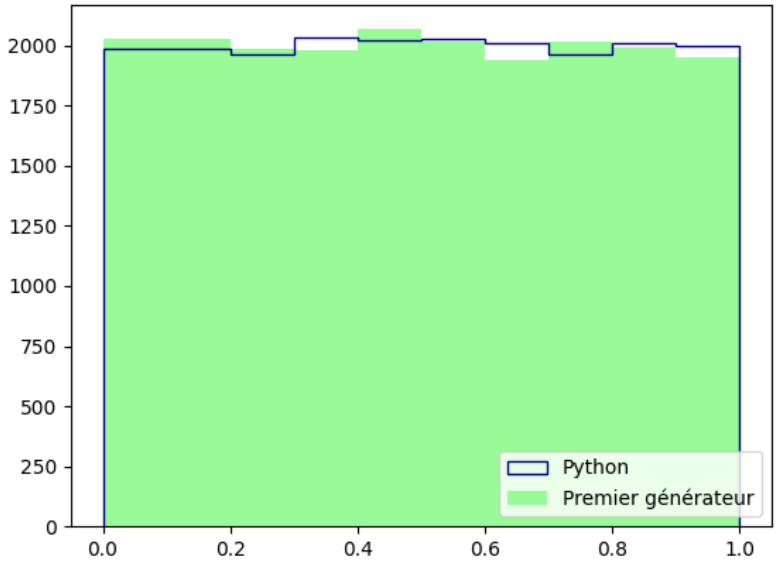

FIGURE 2 – Comparatif générateur 1 et Python

3.2 Générateur 2 : Approche racine

3.2.1 Présentation

Pour pouvoir générer n'importe quel nombre flottant à double précision possible, une idée est de définir un point dans un espace à plusieurs dimensions, et ensuite prendre la distance par rapport au centre des axes comme nombre généré. Par choix arbitraire, trois dimensions ont été choisies dans le cadre de ce travail.

Pour se faire, il faut donc générer trois nombres entre 0 et 1 avec la même méthode que précédemment. Ensuite, la distance par rapport au centre des axes peut être calculée par la racine de la somme des carrés de chaque nombre.

Les nombres à générer sont compris dans l'intervalle $[0, 1]$. Or, les points se trouvant dans le cube à trois dimensions de coté 1 se trouvent à une distance comprise dans l'intervalle $[0, \sqrt{3}]$ du centre les axes. Il est donc nécessaire de diviser le nombre calculé précédemment par un facteur $\sqrt{3}$.

3.2.2 Résultats obtenus

Le tableau 8 contient les effectifs pour chaque classe lors de la génération de 20 000 nombres.

Intervalle	Effectif	Intervalle	Effectif
$[0, 0.1[$	56	$[0.5, 0.6[$	4857
$[0.1, 0.2[$	384	$[0.6, 0.7[$	4519
$[0.2, 0.3[$	1071	$[0.7, 0.8[$	2843
$[0.3, 0.4[$	2081	$[0.8, 0.9[$	328
$[0.4, 0.5[$	3268	$[0.9, 1[$	84

TABLE 8 – Effectif par classe pour le premier générateur

3.2.3 Exemple :

Avec une méthode similaire au générateur 1, générer 3 nombres. Dans cet exemple, 0.8197068416 ; 0.1403970198 et 0.328237646480.

Le nombre généré par le générateur sera donc la représentation flottante de :

$$\frac{\sqrt[3]{0.8197068416^2 + 0.1403970198^2 + 0.328237646480^2}}{\sqrt{3}}$$

L'histogramme 3 compare la répartition des nombres obtenus à l'aide du premier générateurs à ceux obtenus en utilisant le module `Random` de Python.

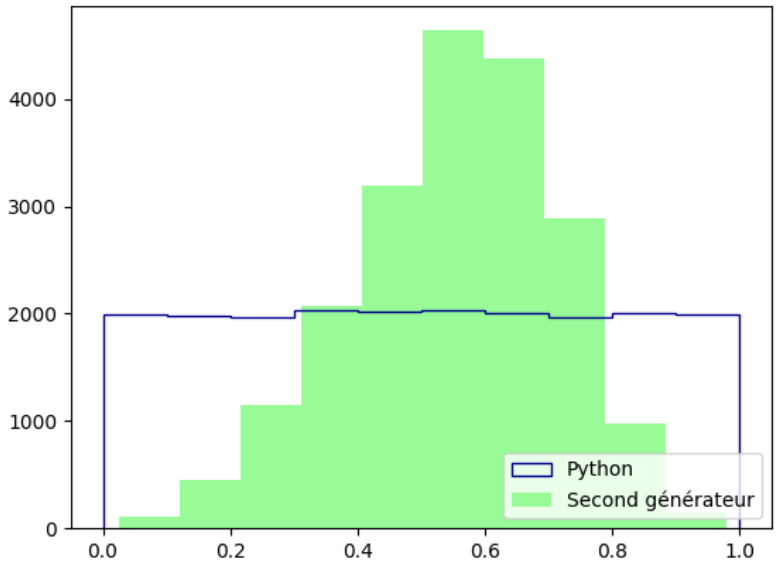

FIGURE 3 – Comparatif générateur 2 et Python

3.2.4 Problème

Bien que ce générateur puisse théoriquement générer tous les nombres flottants à double précision, ce générateur a un énorme problème. Comme le montre expérimentalement la Figure 3, la répartition n'est pas uniforme. Ceci peut être expliqué par le fait qu'il y ait moins de combinaisons pour former les nombres proches de 0 et de 1 que les nombres proches de 0.6. En effet, il n'y a qu'une seule manière de générer 0 : $\frac{\sqrt[3]{0^2+0^2+0^2}}{\sqrt{3}}$ et 1 : $\frac{\sqrt[3]{1^2+1^2+1^2}}{\sqrt{3}}$ alors qu'il y a plusieurs de combinaisons pour générer 0.5 ou 0.6.

Ce générateur est donc à éviter comme générateur de nombres uniforme dans l'intervalle $[0, 1]$ et donc ne sera pas étudié plus en profondeur.

3.3 Générateur 3 : Approche binaire

3.3.1 Présentation

Un autre moyen pour générer tous les nombres possible est le suivant : un nombre flottant n'est plus considéré comme une suite finie de nombres mais comme une suite de 64 bits utilisant l'encodage IEEE754.

Dans un nombre à double précision IEEE754, il y a 1 bit de signe, 11 bits d'exposant, et 52 bits de mantisse.

Pour générer des nombres entre 0 et 1, le bit de signe est fixe ; les 11 bits d'exposant ne sont que partiellement fixe (il n'y a pas de valeurs élevées d'exposants et peu de probabilité d'avoir des valeurs faibles) ; et les 52 bits de mantisse peuvent valoir n'importe quelle valeur.

Par simplification, on va supposer qu'il faut générer les 52 bits de mantisse.

Un nombre entier N de taille 52 bits est généré à partir des chiffres de e. Les chiffres de e est une série de décimale. Chaque chiffre contient une *entropie de Shannon* de $\log_2(10) \simeq 3.3219$. Il est donc possible de générer 3 bits pour chaque chiffre de e. Par simplicité, le reste de l'information (les nombres plus grands que $2^3 - 1 = 7$) n'est pas utilisée.

Une fois le nombre entier aléatoire N calculé, il reste à le transposer sur un nombre à virgule flottante compris entre 0 et 1. Pour ce faire, N est divisé par 2^{52}. Comme le bit de poids fort a une probabilité de 0.5 de valoir 1 et une probabilité 0.5 de valoir 0, N a la même probabilité 0.5 de se retrouver plus grand que 2^{51} et plus petit que 2^{51}. Donc, le nombre généré a la même probabilité d'être plus grand que 0.5 et plus petit que 0.5. La même logique peut être appliquée pour chaque bit de N.

3.3.2 Algorithme

Algorithm 1: Generator 3

Input: *precision* : Le nombre de bits à générer
numbers : Un tableau des décimales de e
index : La position courante dans *numbers*, initialement assigné selon la graine
Result: Un nombre flottant à double précision, compris entre 0 et 1.

Function *random* **:**

```
nbBits ← 0
N ← 0
while nbBits < precision do
    rn = numbers[index]
    if rn > 7 then
        continue
    end
    three_bits ← (rn&1, (rn&2)//2, (rn&4)//4)
    foreach i ∈ 0..2 do
        if nbBits > precision then
            break
        end
        N ← (N << 1)|three_bits[i]
        nbBits ← nbBits + 1
    end
    index ← (index + 1)  mod len(numbers)
end
return N / 2^precision
```

3.3.3 Résultats obtenus

Le tableau 7 contient les effectifs pour chaque classe lors de la génération de 20 000 nombres.

Intervalle	**Effectif**	**Intervalle**	**Effectif**
$[0, 0.1[$	2033	$[0.5, 0.6[$	2038
$[0.1, 0.2[$	2067	$[0.6, 0.7[$	1967
$[0.2, 0.3[$	2033	$[0.7, 0.8[$	1909
$[0.3, 0.4[$	1935	$[0.8, 0.9[$	1992
$[0.4, 0.5[$	2082	$[0.9, 1[$	1935

TABLE 9 – Effectif par classe pour le premier générateur

L'histogramme 4 compare la répartition des nombres obtenus à l'aide du premier géné-

rateur à ceux obtenus en utilisant le module `Random` de Python.

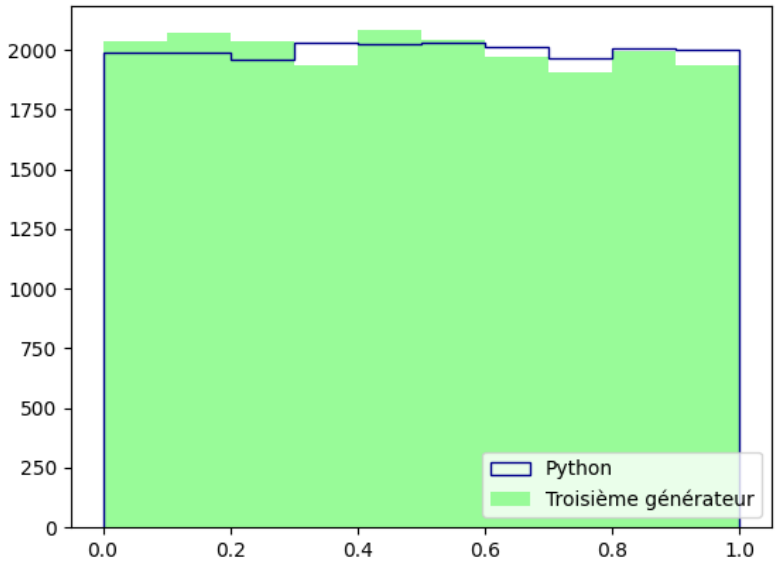

FIGURE 4 – Comparatif générateur 3 et Python

3.3.4 Problème

Pour une raison obscure qui n'a pas été trouvée par les auteurs de ce rapport, les résultats de l'implémentation en Python 3.9 ne sont pas corrects sous Microsoft Windows avec des tailles de mantisse élevée.

Une version améliorée de cet algorithme générerait directement le nombre flottant en encodage binaire, en calculant une mantisse et un exposant, en prenant en compte le fait que toutes les mantisses ne sont pas valides.

4 Vérification des générateurs de nombres aléatoires

Il reste à effectuer des tests de répartition sur les nombres précédemment générés. Cependant seuls les résultats pour le générateur 3 seront présentés en détail.

4.1 Test du χ^2

Il n'est pas possible d'appliquer directement le test du χ^2, décrit à la section 2.2, car il s'agit d'une distribution continue entre 0 et 1. C'est pourquoi la première étape consiste

à séparer les nombres en 10 classes, ce qui a déjà été fait à la section 3. La suite du test se fait pareillement. Les valeurs de χ^2 sont avec un risque d'erreur de 5%.

Générateur	K_r	χ^2	**Réussite**
1	6.8915	16.9190	Oui
2	14565.5585	16.9190	Non
3	16.3595	16.9190	Oui
Python (Mersenne Twister)	12.5750	16.9190	Oui

TABLE 10 – Test du χ^2 pour les générateurs

Le test du χ^2 est réussi pour le générateur 1 et 3 et a échoué pour le second générateur. Ce qui confirme les premières constatations lors de la visualisation des résultats.

4.2 Test de Kolmogorov-Smirnov

Ce test ne peut être utilisé que dans le cas d'une distribution continue. Il compare la fonction de répartition théorique, dans ce cas uniforme, avec la fonction de répartition de l'échantillon empirique. L'idée principale est de calculer la distance maximale entre ces deux fonctions. Si cette distance dépasse un certain seuil, l'hypothèse sera rejetée.

Dans un premier temps, la fonction de répartition empirique est calculée à partir des nombres obtenus. Ensuite, la fonction de répartition théorique est également construite de la manière suivante. Pour i allant de 1 à 9, la valeur théorique est i/n, avec n, la taille de la liste de nombres à tester.

Dans un second temps, la distance maximale existante entre ces deux fonctions est calculée :

$$D = \max(|i/n - o_i|)$$

Enfin, il reste à comparer cette distance à la valeur du seuil :

$$K = 1.358/\sqrt{n} \text{ Avec n : le nombre d'échantillon}$$

Comme pour le test du χ^2, les générateurs 1 et 3 réussissent ainsi que le Mersenne Twister par défaut de python. Tandis que le test pour le générateur 2 échoue.

4.3 Test du gap

Le principe de ce test est le même que pour celui décrit à la sous-section 2.4 à quelques différences près. Le test est utilisé avec un intervalle de $[0.0, 0.5[$ pour marquer un nombre.

Générateur	D	K	Réussite
1	0.01788	0.03037	Oui
2	0.2380	0.03037	Non
3	0.01984	0.03037	Oui
Python (Mersenne Twister)	0.01514	0.03037	Oui

TABLE 11 – Test de Kolmogorov-Smirnov pour les générateurs

Le calcul de la probabilité d'appartenir à l'intervalle est également légèrement différent :

$$p = b - a = 0.5 - 0 = 1/2$$

Les autres calculs s'effectuent de manière semblable. Le tableau 13 contient les effectifs théoriques et empiriques pour chaque longueur de gaps rencontrée.

Longueur	Observés	Théoriques	Longueur	Observés	Théoriques
1	5114	5077	9	20	19.8320
2	2552	2538.5	10	2	9.9160
3	1276	1269.25	11	5	4.9580
4	652	634.625	12	1	2.4790
5	275	317.3125	13	1	1.2395
6	152	158.6563	14	0	0.6198
7	61	79.3281	15	1	0.3099
8	42	39.6641			

TABLE 12 – Effectifs empiriques et théoriques obtenus lors du test du gap sur les nombres générés par le générateur 3

En fin de compte, le test du χ^2 est effectué avec les effectifs observés et les effectifs théoriques précédemment calculés :

Générateur	K_r	χ^2	Réussite
1	6.8289	22.3620	Oui
2	6426.9045	33.9244	Non
3	20.5532	23.6848	Oui
Python (Mersenne Twister)	12.8851	18.3070	Oui

TABLE 13 – Test du gap pour les générateurs

En conclusion, le test du gap est réussi pour les générateurs 1 et 3. Et de nouveau a échoué pour le second générateur.

5 Conclusion

Lors de la première partie de ce projet, nous avons pu tester la répartition uniforme des 2.000.000 premières décimales de la constante de Néper. Les tests du χ^2, du poker, du gap et Kolmogorov-Smirnov ont réussi. C'est pourquoi, nous avons tendance à dire que la répartition est bien uniforme. Or, ce n'est pas pour autant que nous pouvons affirmer qu'il s'agit de chiffres aléatoires, car il y a pas moins aléatoire qu'une constante.

Ensuite, nous avons exploré diverses techniques pour implémenter un générateur de nombres pseudo-aléatoires à partir de la constante de Néper. Les approches naïves et binaires fournissent des résultats similaires lors de leur comparaison au générateur de Python ainsi qu'au moment de les tester. Par contre, l'approche racine donne les plus mauvais résultats.

Enfin, il ne s'avère pas si simple de générer de l'aléatoire à partir de quelque chose déterministe. De plus, même en multipliant les tests sur les nombres générés, nous ne pouvons jamais être sûrs et certains d'avoir effectivement des nombres aléatoires.