# Simulation
# Projet d'examen:
# Étude du caractère pseudo-aléatoire de $\pi$

Libert Robin BA3 Info
Umons

Juin 2018

# Table des matières

# Chapitre 1

# Introduction

## 1.1 Énoncé

Dans le cadre du projet du cours de Simulation, dispensé par Monsieur Alain BUYS, il nous a été demandé d'étudier le caractère pseudo-aléatoire des décimales de $\pi$. Pour ce faire, nous allons utiliser des tests vus au cours dans le but de savoir si les décimales de $\pi$ suivent une loi uniforme ou non.

Dans un second temps, nous devons utiliser les décimales de $\pi$ pour créer un générateur de loi uniforme [0,1[ et le comparer au générateur par défaut de Python.

## 1.2 Logiciels utilisés

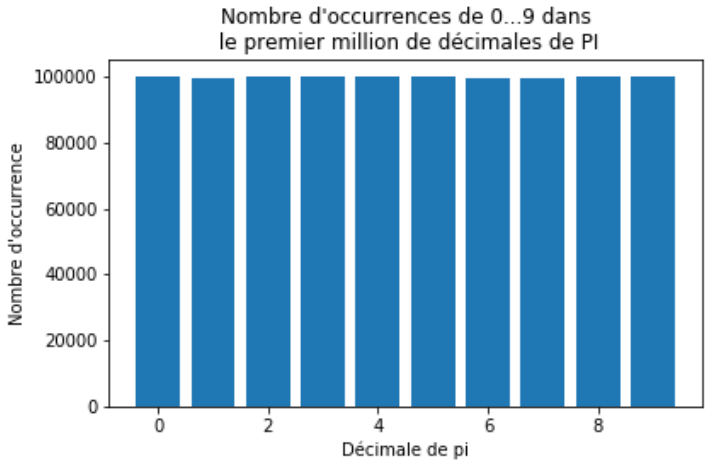
— Spyder : ide python
— Python 3 : pour la réalisation du code
— Mathplotlib : pour la réalisation des graphiques. (Bibliothèque python)
— Windows 10
— TeXnicCenter : pour la rédaction du rapport en Latex

# Chapitre 2

# Intuitions

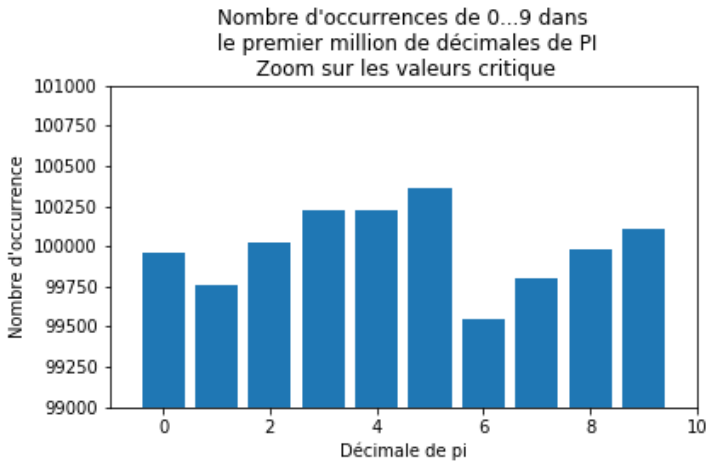
Pour nous donner une première intuition, nous avons créé un diagramme à bâtons comptant le nombre de fois que chaque digit de $\pi$ apparaît dans le premier million de décimales de $\pi$. En théorie, vu que nous considérons 10 digits allant de 0 à 9 et parcourons 1 million de décimales, une loi uniforme devrait compter 100000 occurrences de chaque digit.

Voici un aperçu de notre diagramme à bâton. Chaque digit semble bien apparaître 100000 fois et donc, les décimales de $\pi$ semblent suivre une loi normale.




Faisons un zoom sur les valeurs critiques du diagramme. Nous constatons ci-dessous que les digits n'apparaissent pas 100000 fois. Ce n'est pas pour autant que les décimales de $\pi$ ne suivent pas une loi uniforme. En théorie, ce serais le cas si nous avions et pouvions observer une infinité de décimale, mais comme ce n'est pas le cas, il est normal d'avoir une erreur. Nous allons donc devoir

vérifier avec différents tests, si cette erreur est significative ou pas en fonction du nombre de décimales que nous avons à notre disposition.




En mettant les valeurs dans un tableau et calculant le pourcentage d'erreur relative, nous constatons que les résultats expérimentaux ne diffèrent jamais de plus de 0.5% des résultats théoriques.

| Digit | Théorie | Expérimentaux | Erreur relative % |
|---|---|---|---|
| 0 | 100000 | 99959 | 0.041 |
| 1 | 100000 | 99758 | 0.242 |
| 2 | 100000 | 100026 | 0.026 |
| 3 | 100000 | 100229 | 0.229 |
| 4 | 100000 | 100230 | 0.230 |
| 5 | 100000 | 100359 | 0.359 |
| 6 | 100000 | 99548 | 0.452 |
| 7 | 100000 | 99800 | 0.200 |
| 8 | 100000 | 99985 | 0.015 |
| 9 | 100000 | 100106 | 0.106 |

En résumé, les résultats préliminaires tendent à nous faire penser que les décimales de $\pi$ suivent une loi uniforme, du moins pour le premier million. Dans le chapitre suivant, nous allons tenter de prouver que le premier million de décimales de $\pi$ suivent bien une loi uniforme. Nous allons utiliser 3 tests différents, le test de $\chi^2$, le test du Gap et le test de la main de Poker.

# Chapitre 3

# Tester le caractère pseudo-aléatoire de 1 million de décimales de $\pi$

## 3.1 Test de $\chi^2$ avec 9 ddl

### 3.1.1 Rappel théorique

Dans le test de $\chi^2$, nous devons créer un histogramme de r intervalles et compter le nombre de valeurs générées dans chaque intervalle $n_i$. Dans chaque intervalle, on s'attend à avoir à peu près le même nombre de points car les probabilités $p_i$ sont égales.

Dans un premier temps, nous devons calculer $K_n = \sum_{i=1}^{r} \left( \frac{n_i - Np_i}{\sqrt{Np_i}} \right)^2$. Ensuite nous devons comparer $K_n$ et $\chi^2$ pour un certain $\alpha$, qui est la probabilité de rejeter l'hypothèse nulle $H_0$ et un certain degré de liberté qui vaut $r - 1$. Nous retenons notre hypothèse nulle si $K_n \leq \chi^2$ et nous la rejetons sinon.

r est le nombre d'événements distincts et vaut 10
$n_i$ est le nombre d'occurrences observées de l'événement i
N est la taille de l'échantillon et vaut 1000000
$p_i$ est la probabilité d'obtenir l'événement i et vaut $\frac{1}{10}$

### 3.1.2 Résultats et implémentation

Cette première fonction prend en paramètre la liste des décimales de $\pi$. Elle retourne la liste du nombre d'occurrence de chaque digit de 0 à 9 dans les décimales de $\pi$. Le nombre de fois qu'apparaît 0 dans les décimales de $\pi$ est le nombre se trouvant à l'indice 0 de la liste.

```
def countOccurences(l):
```

```
    n_occurences = [0,0,0,0,0,0,0,0,0,0]
    for n in range(len(l)):
        n_occurences[l[n]] += 1
    return n_occurences
```

Cette fonction va prendre en paramètres la liste retournée par la fonction countOccurences appliquée sur les décimales de $\pi$ et retournera la valeur de $K_n$ avec comme probabilité $\frac{1}{10}$.

```
def khi2(listReal):
    sizeEch = 0
    for e in listReal:
        sizeEch += e
    Kn = 0
    for i in range(10):
        p = sizeEch*1/10
        Kn += ((listReal[i] - p) / math.sqrt(p))**2
    return Kn
```

Voici ci-dessous les résultats que nous obtenons. Nous avons fais le test avec plusieurs valeurs de $\alpha$. Le test réussit à chaque fois, nous pouvons donc conclure que le premier million de décimales de $\pi$ suit une loi uniforme.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 5.51 | 27.88 | True |
| 0.05 | 5.51 | 16.92 | True |
| 0.1 | 5.51 | 14.68 | True |

## 3.2 Test du Gap avec 49 ddl

### 3.2.1 Rappel théorique

Ce test utilise un test de $\chi^2$, sauf que nous allons utiliser des classes différentes de celles utilisées dans le test précédent. Dans notre cas, nous devons définir ce qu'est un Gap. Nous prenons un nombre n entre [0,9], et comptons le nombre de nombres qui le sépare du prochain n et rajoutons 1. Ceci est un Gap. Par exemple, dans la suite 10554871, nous prenons 1 comme nombre n, et nous comptons le nombre de nombres qui le sépare du prochain 1. Ici, 6 nombres séparent les 1, nous avons donc un Gap de taille 6 + 1.

Nos classes seront le nombre de Gap d'une certaine taille pour un digit donné. Par exemple, prenons le digit 0, la première classe sera le nombre de Gap de taille 1 entre chaque 0 des décimales de $\pi$. La deuxième classe sera le nombre de Gap de taille 2 et ainsi de suite.

Chaque classe à une probabilité théorique d'avoir $\frac{1}{10} * \frac{9}{10}^{(L-1)}$ ou L est la longueur du Gap.

### 3.2.2 Résultats et implémentation

Voici comment nous avons créé la liste des classes sur lesquelles nous allons effectuer notre test de $\chi^2$. Nous lui donnons une liste de nombre en paramètre et le nombre à analyser dans cette liste.

```
def gapList(randomNumberList, n):
    gapList = [0]*1000
    gap = 1
    begin = False
    for e in randomNumberList:
        if begin == True and e == n:
            if(gap <= len(gapList)):
                gapList[gap-1] += 1
            gap = 1
        elif begin == False and e == n:
            begin = True
        elif begin == True and e != n:
            gap += 1
    acc = 0
    for e in range(-1,-len(gapList)-1,-1):
        acc += gapList[e]
        gapList[e] = acc
    return gapList
```

Ensuite, pour le test de $\chi^2$, nous faisons le même que pour le test précédent sauf que nous changeons la probabilité.

```
def khi2Gap(listReal):
    sizeEch = 0
    gapsize = 50
    for e in range(1000):
        sizeEch += listReal[e]
    Kn = 0
    for i in range(gapsize):
        p = sizeEch*((1/10)*((9/10)**i))
        Kn += ((listReal[i] - p) / math.sqrt(p))**2
    return Kn
```

Voici les résultats ci-dessous du test pour différents digits et pour différents $\alpha$. Le test est entièrement réussit.

| Numéro | $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|---|
| 0 | 0.001 | 25.82 | 85.35 | True |
| 0 | 0.05 | 25.82 | 66.34 | True |
| 0 | 0.1 | 25.82 | 62.04 | True |
| 1 | 0.001 | 24.16 | 85.35 | True |
| 1 | 0.05 | 24.16 | 66.34 | True |
| 1 | 0.1 | 24.16 | 62.04 | True |
| 2 | 0.001 | 10.64 | 85.35 | True |
| 2 | 0.05 | 10.64 | 66.34 | True |
| 2 | 0.1 | 10.64 | 62.04 | True |
| 3 | 0.001 | 10.58 | 85.35 | True |
| 3 | 0.05 | 10.58 | 66.34 | True |
| 3 | 0.1 | 10.58 | 62.04 | True |
| 4 | 0.001 | 46.08 | 85.35 | True |
| 4 | 0.05 | 46.08 | 66.34 | True |
| 4 | 0.1 | 46.08 | 62.04 | True |
| 5 | 0.001 | 31.08 | 85.35 | True |
| 5 | 0.05 | 31.08 | 66.34 | True |
| 5 | 0.1 | 31.08 | 62.04 | True |
| 6 | 0.001 | 21.29 | 85.35 | True |
| 6 | 0.05 | 21.29 | 66.34 | True |
| 6 | 0.1 | 21.29 | 62.04 | True |
| 7 | 0.001 | 17.51 | 85.35 | True |
| 7 | 0.05 | 17.51 | 66.34 | True |
| 7 | 0.1 | 17.51 | 62.04 | True |
| 8 | 0.001 | 9.99 | 85.35 | True |
| 8 | 0.05 | 9.99 | 66.34 | True |
| 8 | 0.1 | 9.99 | 62.04 | True |
| 9 | 0.001 | 10.08 | 85.35 | True |
| 9 | 0.05 | 10.08 | 66.34 | True |
| 9 | 0.1 | 10.08 | 62.04 | True |

## 3.3 Test du Poker avec 4 ddl

### 3.3.1 Rappel théorique

### 3.3.2 Résultats et implémentation

Cette fonction sert à créer la liste des classes que l'on a utilisé pour faire notre test de $\chi^2$

```
def pokerList(randomNumberList):
    pokerList = [0]*5
    c = 0
    nOccurences = [0]*10
```

```
    n=0
    for e in randomNumberList:
        if c < 5:
            nOccurences[e] += 1
        if c == 4:
            for i in nOccurences:
                if i != 0:
                    n += 1
            pokerList[n-1] += 1
            n = 0
            c = -1
            nOccurences = [0]*10
        c += 1
    return pokerList
```

Pour notre test de $\chi^2$, nous faisons la même chose que pour les 2 tests précédents sauf que cette fois-ci la probabilité est un peu plus compliquée à calculer. En effet, nous avons besoin du nombre de Stirling, donc nous avons fait une fonction à part nommée stirling.

```
def khi2Poker(listReal):
    sizeEch = 0
    for e in range(5):
        sizeEch += listReal[e]
    Kn = 0
    for i in range(5):
        p = stirling(5,i+1)
        c = 0
        while -(i+1) != c:
            p = p * (10+c)
            c -= 1
        p = p / 10**5
        Kn += ((listReal[i] - (sizeEch*p)) / math.sqrt(sizeEch * p))**2
    return Kn
```

Pendant nos recherches pour pour mieux comprendre le nombre de stirling, nous sommes tombé sur cet algorithme l'implémentant. Après avoir vu cela, nous n'avons pas trouvé d'autre façon de faire donc nous avons décidé d'utiliser cet algorithme dont les références sont en commentaire du code.

```
"""
# Stirling Algorithm
# Cod3d by EXTR3ME
# https://extr3metech.wordpress.com
"""
def stirling(n,k):
    n1=n
```

```
    k1=k
    if n<=0:
        return 1

    elif k<=0:
        return 0

    elif (n==0 and k==0):
        return -1

    elif n!=0 and n==k:
        return 1

    elif n<k:
        return 0

    else:
        temp1=stirling(n1-1,k1)
        temp1=k1*temp1
        return (k1*(stirling(n1-1,k1)))+stirling(n1-1,k1-1)
```

Voici les résultats pour le test du poker, nous pouvons observer que le test est entièrement réussis pour les 3 valeurs de $\alpha$ choisie.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 4.61 | 18.47 | True |
| 0.05 | 4.61 | 9.49 | True |
| 0.1 | 4.61 | 7.78 | True |

## 3.4 Conclusion

Tous les tests ayant réussis, il semblerait que le premier million de décimales du nombre $\pi$ suit une loi uniforme.

# Chapitre 4

# Comparer notre générateur de loi uniforme à celui de python

## 4.1 Fonctionnement de notre générateur de loi uniforme

Quand un utilisateur veut utiliser notre générateur de loi uniforme pour générer un nombre pseudo-aléatoire, le générateur retiens le nombre de millisecondes écoulé $S$ depuis ma date de naissance. Ce nombre $S$ nous donne un indice $I$ dans la liste des décimales de $\pi$ que nous avons en mémoire. Nous utilisons également la méthode Middle Square pour générer un nombre pseudo aléatoire qui nous servira de décalage $D$. La méthode Middle Square prend en entrée $S$. Notre fonction randomIntList(n) prend un paramètre $N$ qui est le nombre de digits, compris entre 0 et 9, que nous désirons. Pour chaque $N$ désiré, nous ajoutons un décalage $D$ à $I$ et avec une opération de modulo, pour ne pas sortir de la liste des décimales de $\pi$, nous prenons le digit se trouvant à l'indice $I$ dans la liste des décimales de $\pi$ et nous ajoutons ce digit à notre liste de nombres pseudo aléatoires. Tous les 1000 digits généré, nous recalculons $S$ et $D$ de la même manière que précédemment pour ne pas avoir de cycles dans notre génération de nombres. Notre méthode randomIntList(n) nous renvois donc simplement une liste de $N$ digits aléatoires compris entre 0 et 9.

Lors de la conception de notre générateur de loi uniforme, nous aurions pu simplement utiliser EPOCH ou Middle Square pour générer un seed et avec une opération de modulo, utiliser ce seed et parcourir les décimales de $\pi$ pour créer notre générateur. Nous aurions obtenu une loi uniforme car nous avons montré précédemment que $\pi$ suit une loi uniforme. Le problème est que pour générer un grand nombre avec un unique seed, si nous parcourons toutes les décimales de $\pi$ en notre possession une fois, nous risquons d'avoir des répétitions dans notre génération de nombres.

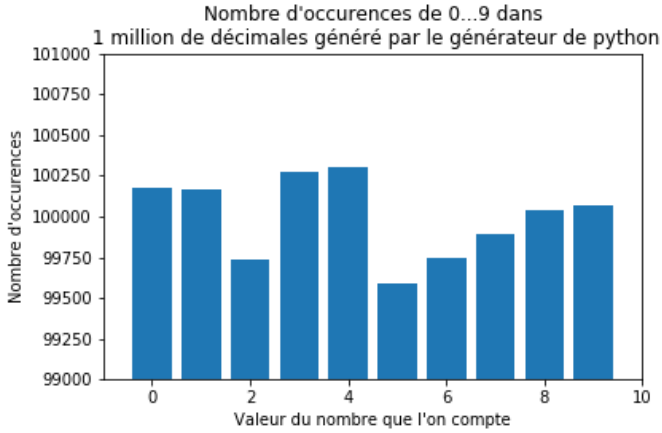
```
def middleSquare(n):
    if(n<=3):#car n > 4 pour que cette fonction fonctionne
        n+=4
    new = n*n
    s = str(new)
    l = len(s)
    indexBegin = int(math.ceil(l/4))
    indexEnd = int(math.ceil(l*(2/3))+1)
    s1 = s[indexBegin:indexEnd]
    answer = int(s1)
    return answer

def randomIntList(n):
    index = seed()
    decalage = 0
    randomNumber = []
    ms = middleSquare(index)
    for e in range(n):
        if(e % 1000 == 0):
            index = seed()
            decalage = 0
            ms = middleSquare(index)
        randomNumber.append(pi_decimals[(index+decalage)%999999])
        decalage += ms+1#plus un si ms vaut 0
    return randomNumber
```

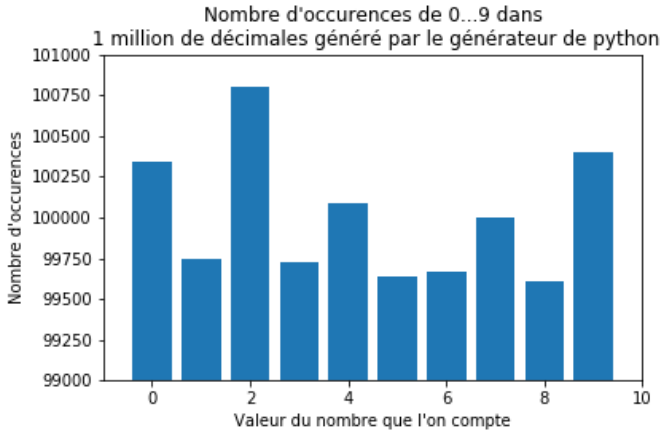
Dans l'énoncé de ce projet, il nous est demandé de fournir un générateur de loi uniforme avec des nombres compris dans l'intervalle $[0, 1[$. Pour avoir un tel nombre, l'utilisateur devra utiliser la fonction myRandom() se trouvant dans le fichier piRandom.py. Cette fonction va se servir de la fonction précédente, randomIntList(17), en lui demandant de générer une liste de 17 digits. Nous avons choisis 17 digits à cause de la précision des float en python. Ensuite, nous assemblons les nombres de cette liste et mettons un 0 devant le nombre obtenu pour avoir un nombre pseudo aléatoire se trouvant dans l'intervalle $[0, 1[$.

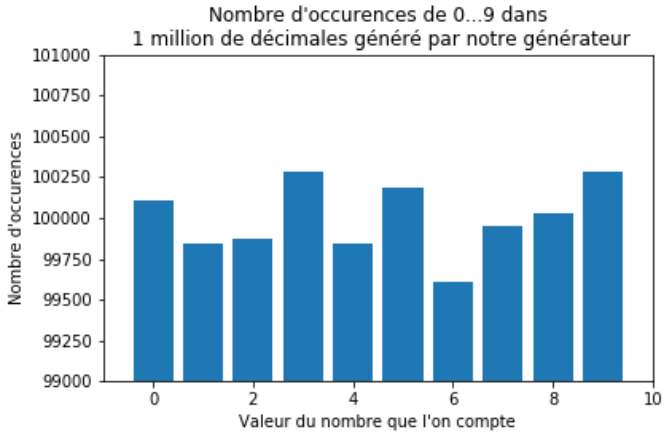
```
def myRandom():
    r = randomIntList(17)#because of float precision
    strvalue = ""
    for e in r:
        strvalue += str(e)
    floatvalue = float(strvalue)
    value = floatvalue / 100000000000000000.
    return value
```

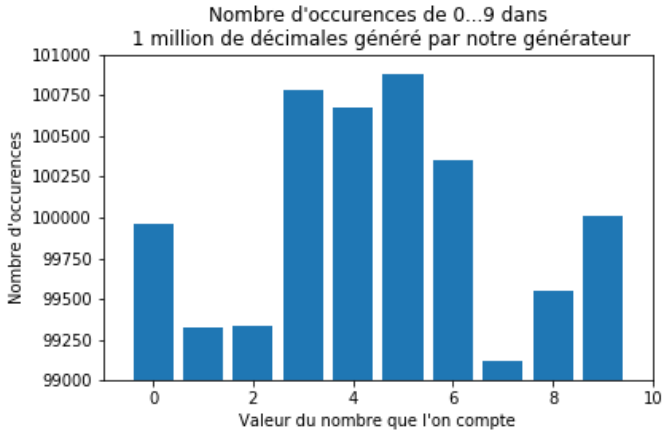
## 4.2 Intuition

Les 4 graphiques ci-dessous représentent le nombre d'occurrences de chaque digit allant de 0 à 9 pour 1 million de digits généré à l'aide du générateur de python (images 1 et 2) et à l'aide de notre propre générateur (images 3 et 4). Avec un million de digits généré, il arrive que l'on soit très proche d'une lois uniforme pour les 2 générateurs, c'est à dire moins de 0.5% d'écart avec les valeurs théoriques attendues. La plupart du temps, nous sommes en dessous de 1% d'écart avec les valeurs théoriques attendue. Lors de nos tests, il est arrivé que notre générateur soit légèrement au dessus de 1% d'écart.






Pour faire les tests qui vont suivre, nous avons généré 1 million de décimales avec notre générateur et 1 million de décimales avec le générateur de python. Ces tests nous permettent d'avoir une idée globale du caractère pseudo-aléatoire de ces suites de nombres. Cependant, si nous générons à nouveau un million de décimales avec chaque générateur, les résultats obtenus ne seront pas les mêmes à cause du caractère pseudo-aléatoire de ces suites de nombres.

## 4.3 Test de $\chi^2$ avec 9 ddl

Résultats du test de $\chi^2$ pour le générateur de python.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 7.56 | 27.88 | True |
| 0.05 | 7.56 | 16.92 | True |
| 0.1 | 7.56 | 14.68 | True |

Résultats du test de $\chi^2$ pour notre générateur.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 8.26 | 27.88 | True |
| 0.05 | 8.26 | 16.92 | True |
| 0.1 | 8.26 | 14.68 | True |

Nous constatons que les 2 échantillons que nous avons généré réussissent ce test. L'échantillon généré à l'aide du générateur python colle un peu plus à une loi uniforme que le notre dans ce cas.

## 4.4 Test du Gap avec 49 ddl

Résultats du test du gap pour le générateur de python.

| Numéro | $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|---|
| 0 | 0.001 | 26.00 | 85.35 | True |
| 0 | 0.05 | 26.00 | 66.34 | True |
| 0 | 0.1 | 26.00 | 62.04 | True |
| 1 | 0.001 | 14.12 | 85.35 | True |
| 1 | 0.05 | 14.12 | 66.34 | True |
| 1 | 0.1 | 14.12 | 62.04 | True |
| 2 | 0.001 | 29.99 | 85.35 | True |
| 2 | 0.05 | 29.99 | 66.34 | True |
| 2 | 0.1 | 29.99 | 62.04 | True |
| 3 | 0.001 | 8.11 | 85.35 | True |
| 3 | 0.05 | 8.11 | 66.34 | True |
| 3 | 0.1 | 8.11 | 62.04 | True |
| 4 | 0.001 | 17.95 | 85.35 | True |
| 4 | 0.05 | 17.95 | 66.34 | True |
| 4 | 0.1 | 17.95 | 62.04 | True |
| 5 | 0.001 | 13.60 | 85.35 | True |
| 5 | 0.05 | 13.60 | 66.34 | True |
| 5 | 0.1 | 13.60 | 62.04 | True |
| 6 | 0.001 | 5.83 | 85.35 | True |
| 6 | 0.05 | 5.83 | 66.34 | True |
| 6 | 0.1 | 5.83 | 62.04 | True |
| 7 | 0.001 | 21.82 | 85.35 | True |
| 7 | 0.05 | 21.82 | 66.34 | True |
| 7 | 0.1 | 21.82 | 62.04 | True |
| 8 | 0.001 | 15.87 | 85.35 | True |
| 8 | 0.05 | 15.87 | 66.34 | True |
| 8 | 0.1 | 15.87 | 62.04 | True |
| 9 | 0.001 | 37.21 | 85.35 | True |
| 9 | 0.05 | 37.21 | 66.34 | True |
| 9 | 0.1 | 37.21 | 62.04 | True |

Résultats du test du gap pour notre générateur.

| Numéro | $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|---|
| 0 | 0.001 | 14.47 | 85.35 | True |
| 0 | 0.05 | 14.47 | 66.34 | True |
| 0 | 0.1 | 14.47 | 62.04 | True |
| 1 | 0.001 | 12.19 | 85.35 | True |
| 1 | 0.05 | 12.19 | 66.34 | True |
| 1 | 0.1 | 12.19 | 62.04 | True |
| 2 | 0.001 | 14.92 | 85.35 | True |
| 2 | 0.05 | 14.92 | 66.34 | True |
| 2 | 0.1 | 14.92 | 62.04 | True |
| 3 | 0.001 | 16.98 | 85.35 | True |
| 3 | 0.05 | 16.98 | 66.34 | True |
| 3 | 0.1 | 16.98 | 62.04 | True |
| 4 | 0.001 | 17.50 | 85.35 | True |
| 4 | 0.05 | 17.50 | 66.34 | True |
| 4 | 0.1 | 17.50 | 62.04 | True |
| 5 | 0.001 | 28.07 | 85.35 | True |
| 5 | 0.05 | 28.07 | 66.34 | True |
| 5 | 0.1 | 28.07 | 62.04 | True |
| 6 | 0.001 | 53.69 | 85.35 | True |
| 6 | 0.05 | 53.69 | 66.34 | True |
| 6 | 0.1 | 53.69 | 62.04 | True |
| 7 | 0.001 | 45.92 | 85.35 | True |
| 7 | 0.05 | 45.92 | 66.34 | True |
| 7 | 0.1 | 45.92 | 62.04 | True |
| 8 | 0.001 | 12.33 | 85.35 | True |
| 8 | 0.05 | 12.33 | 66.34 | True |
| 8 | 0.1 | 12.33 | 62.04 | True |
| 9 | 0.001 | 31.57 | 85.35 | True |
| 9 | 0.05 | 31.57 | 66.34 | True |
| 9 | 0.1 | 31.57 | 62.04 | True |

Nous avons eu un soucis dans notre algorithme nous permettant d'effectuer le test du gap, donc nous avons du résoudre le problème. Pour faire les 2 tests ci-dessus, nous avons dû générer à nouveau un million de digits entre 0 et 9 avec chacun des générateurs. Donc les données qui ont été testé pour le test du gap ne sont pas les mêmes que celles testé pour le test de $\chi^2$ et le test du poker.

Nous constatons que les tests ont réussis entièrement pour les deux générateurs.

## 4.5 Test du Poker avec 4 ddl

Résultats du test du gap pour le générateur de python.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 1.80 | 18.47 | True |
| 0.05 | 1.80 | 9.49 | True |
| 0.1 | 1.80 | 7.78 | True |

Résultats du test du gap pour notre générateur.

| $\alpha$ | $K_n$ | $\chi^2$ | $K_n \leq \chi^2$ |
|---|---|---|---|
| 0.001 | 5.64 | 18.47 | True |
| 0.05 | 5.64 | 9.49 | True |
| 0.1 | 5.64 | 7.78 | True |

Les 2 tests ont été entièrement réussis par les 2 échantillons générés avec un gros avantage, dans ce cas précis, pour les nombres générés par le générateur de python.

## 4.6 Conclusion

Notre générateur est capable de générer une suite de nombres pseudo-aléatoire en se basant sur les décimales de $\pi$. Nous avons créé 2 échantillons de 1 millions de digits, un échantillon créé par notre générateur et un échantillon créé par le générateur de python.

Lors de nos observations, nous constatons que le générateur de python produit en moyenne des lois uniformes avec un écart type plus petit que ce que notre générateur produit, ce qui est préférable.