

Étude de générateurs de nombres aléatoires

Enzo DE CARVALHO

Ce rendu fait suite au premier cours de Simulation Informatique et Gestion d'Incertitudes sur les générateurs de nombres aléatoires. On cherchera dans ce rendu, en plus de les implémenter, de les étudier à travers quelques outils statistiques. Pour cela, nous utilisons ici *Python*. Le code sur lequel ce rendu s'appuie est mis à dispositions dans le dossier Random du dépôt git sur <https://git.iiens.net/de-carva2021/SIGI/>

1 Implémentations

Dans l'optique de comparer différentes implémentations de *générateurs de nombres aléatoires* (ou Random Number Generator en anglais), on crée une classe `RandomNumberGenerator` sur lequel s'appuiera chacun de nos générateurs (voir figure 1) :

1.1 *Linear Congruential Generator*

On implémente une classe LCG générique (voir figure 2), avec laquelle on peut implémenter tous les autres *LCG* vu en cours (figure 3)

1.2 Suite de Fibonacci retardée

On implémente également une suite de fibonacci retardée (voir figure 4) ; La seed permet de régler la seed du *LCG de Park et Miller* utilisé pour générer les premières valeurs nécessaire à l'initialisation de ce générateur.

Similairement à nos spécialisations de LCG, on crée une classe `MitchelMoore` par dessus la classe `LaggedFibonacci`

1.3 *Mersenne Twister*

Comme indiqué en cours, le *Mersenne Twister* est l'algorithme utilisé par le générateur de nombre aléatoire en python ; pour assurer la cohérence d'interface, on emploie la fonction `Random.random()` de la librairie python standard (voir figure 5)

1.4 *Cryptographic Network Security*

On implémente finalement le dernier algorithme sur lequel se repose *Blum Blum Shub* en figure 6

```

1 class RandomNumberGenerator:
2     def __init__(self):
3         self.max = 0 #for normalization
4         return NotImplementedError
5
6     def random(self):
7         """
8         yield a number from the RNG
9         """
10        return NotImplementedError
11
12    def seed(self):
13        """
14        set the seed of the generator
15        """
16        return NotImplementedError
17
18    def randomNorm(self):
19        """
20        return random between 0 and 1
21        """
22        return NotImplementedError
23
24    def batchRandomNormalized(self, n):
25        """
26        return n normalized values
27        """
28        return np.divide(np.array([self.random() for i in range(n)]),self.max)

```

Figure 1: Classe générique exposant l'interface de nos générateurs de nombres aléatoires

```

1 class LCG(RandomNumberGenerator):
2     """
3     Linear Congruential generator algorithm
4     """
5
6     def __init__(self, a, c, m):
7         super().__init__()
8         self.m = m
9         self.c = c
10        self.a = a
11        self.x = 0 # default seed
12        self.max = self.m + 1
13
14    def random(self):
15        self.x = (self.a * self.x + self.c) % self.m
16        return self.x
17
18    def seed(self, seed):
19        self.x = seed
20
21    def randomNorm(self):
22        return self.random() / self.max

```

Figure 2: LCG générique

```

1 class ParkMiller(LCG):
2     """
3     Park and Miller's LCG
4     """
5
6     def __init__(self):
7         super().__init__(16807, 0, (2**31) - 1)

```

Figure 3: Exemple d'un LCG implémenté à partir de notre classe LCG générique ; on fait attention de bien sucharger l'initialisateur.

```

1 class LaggedFibonacci(RandomNumberGenerator):
2     """
3     Lagged Fibonacci generator algorithm
4     """
5     def __init__(self, l, k, m):
6         self.l = l
7         self.k = k
8         self.m = m
9         self.ParkMiller = ParkMiller()
10        self.values = []
11        self.max = self.m + 1
12
13    def seed(self, seed):
14        self.ParkMiller.seed(seed)
15        self.values = [self.ParkMiller.random() for i in
16                       range(max(self.l, self.k))]
17
18    def random(self):
19        # init values if non-existents
20        if not (self.values):
21            self.values = [self.ParkMiller.random() for i in
22                           range(max(self.l, self.k))]
23        val = (self.values[-self.l] + self.values[-self.k]) % self.m
24        self.values.append(val)
25        return self.values[-1]
26
27    def randomNorm(self):
28        return self.random() / self.max

```

Figure 4: Implémentation de la suite de fibonacci retardée

```

1 class MersenneTwister(RandomNumberGenerator):
2     """
3     MersenneTwister algorithm (native random.random())
4     """
5
6
7     def __init__(self):
8         super().__init__()
9
10    def seed(self, seed):
11        random.seed(seed)
12
13    def randomNorm(self):
14        return random.random()
15
16    def random(self):
17        return self.randomNorm()
18
19    def batchRandomNormalized(self,n):
20        return np.array([self.random() for i in range(n)])
21
22

```

Figure 5: Implémentation de *Mersenne Twister*

```

1 class CNS(RandomNumberGenerator):
2     """
3     Cryptography Network Security
4     """
5
6
7     def __init__(self, x0, M):
8         super().__init__()
9         self.M = M
10        self.x = x0 # must verify pgcd(M,x0) = 1
11        self.max = M + 1
12
13    def seed(self, seed):
14        self.x = seed
15
16    def random(self):
17        self.x = pow(self.x, 2) % self.M
18        return self.x
19
20    def randomNorm(self):
21        return self.random() / self.max

```

Figure 6: Implémentation du *CNS*

2 Observations statistiques

Une fois les algorithmes créés, on les traite et graphe à l'aide des librairies python *Seaborn*, *Matplotlib*, *Numpy* et *Scipy*, qui viennent fournies avec des fonctions utiles à l'affichage de nos données.

On génère alors pour chaque générateur, un jeu de donnée de 1 millions de valeurs normalisées.

2.1 Comparaisons

Parmi les statistiques observées, bien que la plupart des générateur ont des valeurs plutôt proches de celles théoriques, on retrouve Marsaglia et Haynes avec une assez bonne moyenne et BlumBlumShub qui a les valeurs qui s'écartent le plus grossièrement (8) ; cette disparité se voit très bien sur l'histogramme de BlumBlumShub également (7).

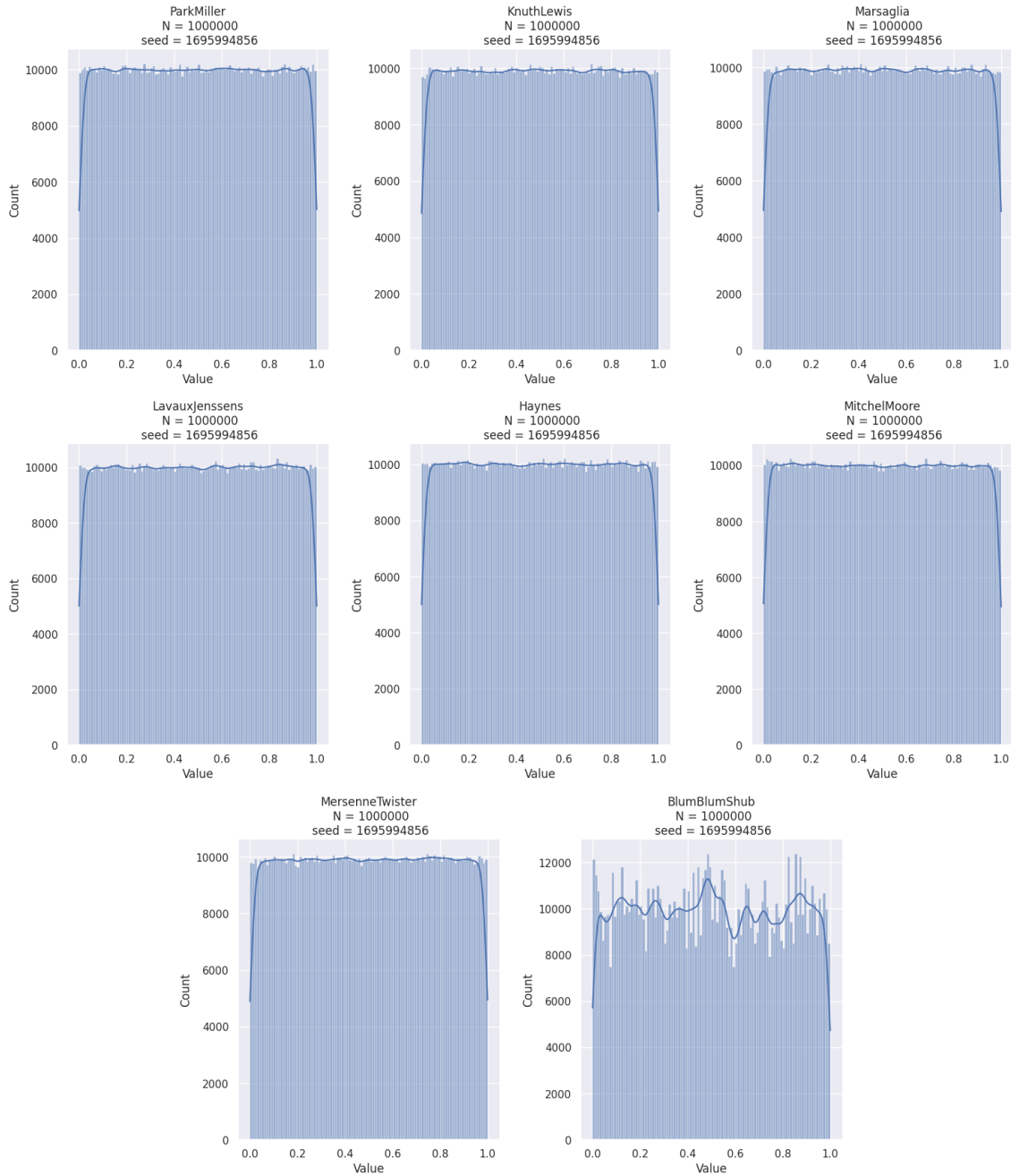


Figure 7: Histogrammes avec de résolution100 sur chacun des générateurs. Une courbe de probabilité de densité est également affichée. (en utilisant une *Kernel Density Estimation* de bande passante 0.337)

Generator	Quantile25%	Quantile50%	Quantile75%	Mean	Variance	Skew	Excess Kurtosis
Expected	1/4	1/2	3/4	1/2	1/12	0	-6/5
ParkMiller	0.250634055	0.50066613	0.75043614	0.50042309	0.083326215	-0.00205464528	-1.1997833
KnuthLewis	0.25003540	0.50003350	0.74981786	0.499900041	0.08334119	0.0010412009	-1.2005081
Marsaglia	0.250319030	0.49977684	0.74956660	0.50000410	0.083206872	0.000191009792	-1.1976268
LavauxJenssens	0.25025780	0.5003118	0.75031099	0.50017746	0.083328297	-0.00139354910	-1.1991514
Haynes	0.2496948	0.499979105	0.74991957	0.499924076	0.083313601	0.00030898176	-1.1999041
MitchelMoore	0.250210068	0.50057595	0.75067237	0.50040749	0.083351852	-0.002428923	-1.2003969
MersenneTwister	0.250282619	0.49979995	0.74960010	0.500015	0.083223536	0.00127156304	-1.198297
BlumBlumShub	0.247498412	0.49285453	0.74848385	0.496992898	0.083746446	0.013676754	-1.2001854

Figure 8: Statistiques obtenues sur les différentes générations, pour 1 000 000 de valeurs et une *seed* de 1695915794

2.2 Évolution de la densité de probabilité

Bien que je n'ai pas réussi à employer le cluster de l'école pour utiliser *MPI4py* (les paquets n'étants pas à jours, et *MPI4py* n'est que disponible pour python 2.7 sur l'école), celui-ci fut utile afin de ne pas **out of memory** - et donc quand même traiter la convergence des densités de probabilités avec une grandes quantité de données pour chacun de nos générateurs. On trace alors l'évolutions de la courbe de densité de probabilité (obtenu par *Kernel Density Estimation* , avec une bande passante de 0.337) On trace également les *QQplots* des ECDF (*Empirical Cumulative Distribution Function*). Tout cela avec $N=100$, 1 000 000 et 100 000 000 (voir figure 9 et 10).

On note plusieurs choses :

- Les échantillons de petites taille (ici, 100) ne sont pas suffisante pour l'étude statistique.
- BlumBlumShub ne converge *pas* vers une distribution uniforme.

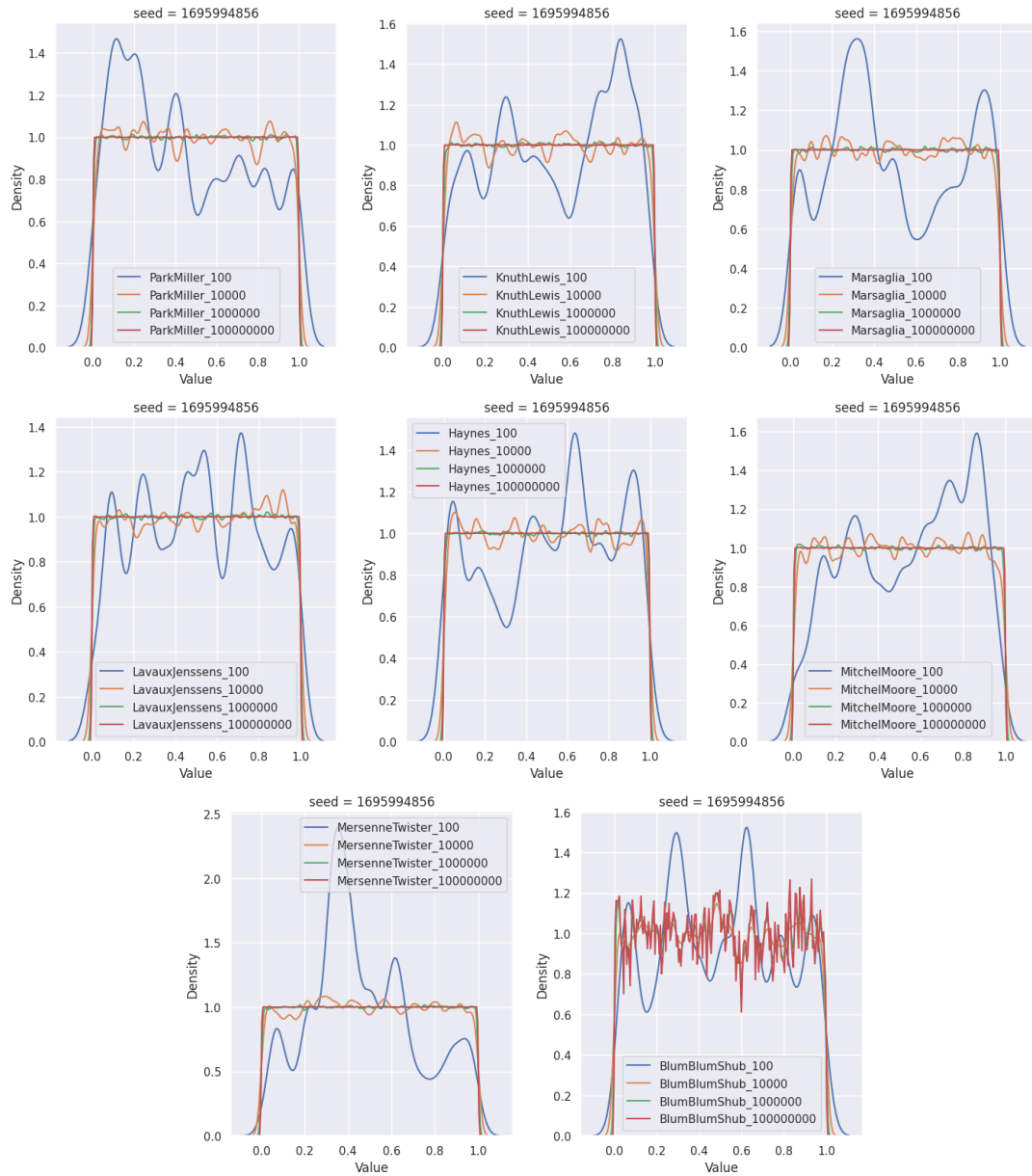


Figure 9: Evolution de la courbe de densité pour $N=100, 10000, 1000000$ puis 100000000 et une seed à 1695994856

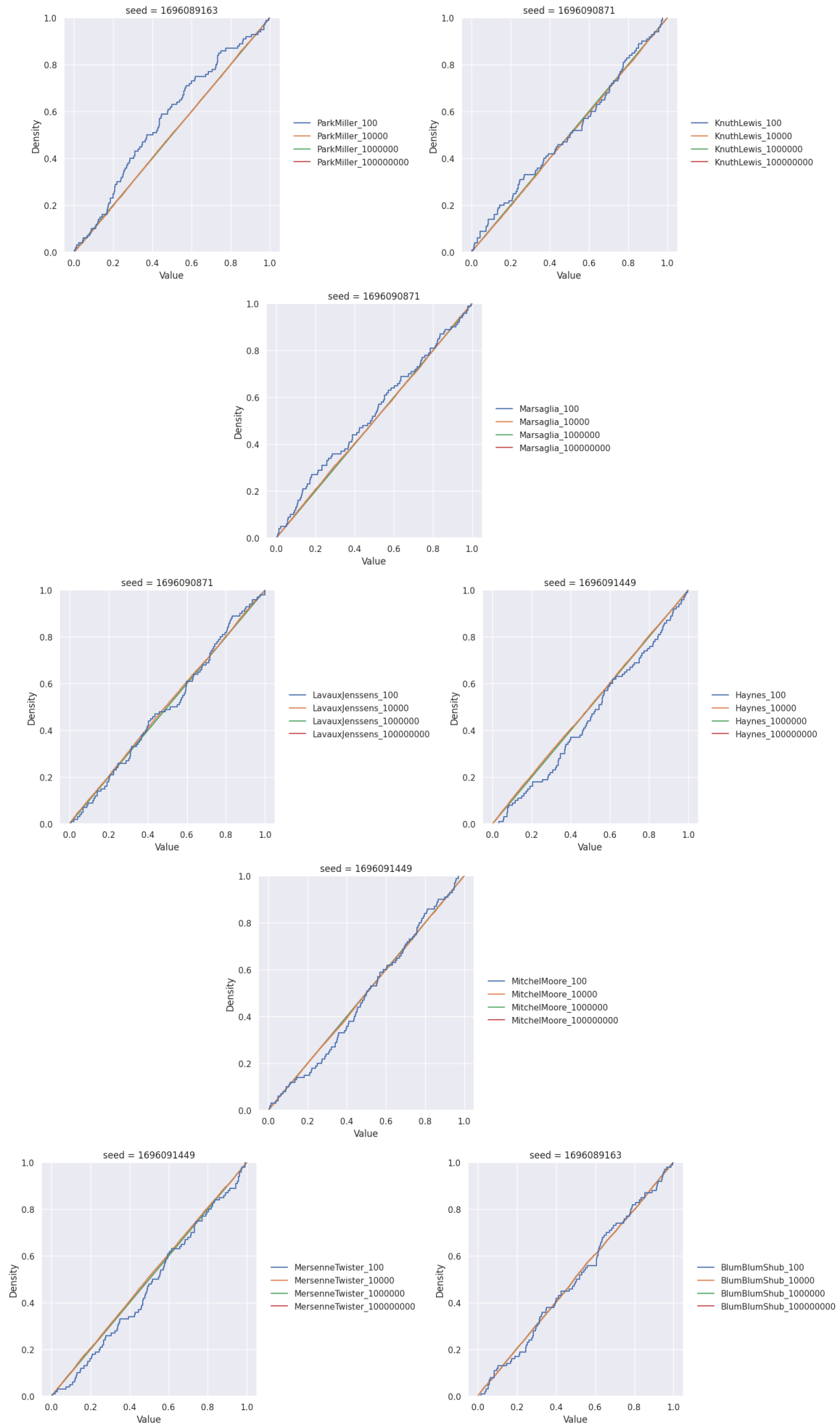


Figure 10: Evolution de la courbe ECDF au fil des itérations

```

1 def chisquared_uniform(arr, RES):
2     """
3     chisquared for uniform distrib
4     """
5     observed_freq, _ = np.histogram(arr, bins=np.linspace(0,1,RES))
6     return scipy.stats.chisquare(observed_freq)

```

Figure 11: Calcul du χ^2 . L'argument RES indique le nombre de sous-segments souhaité pour calculer la distribution dans `np.histogram`

Générateur	χ^2	p-value
ParkMiller	78.756578	0.9233523899478229
KnuthLewis	84.27800599999998	0.8369034250335825
Marsaglia	118.07660600000001	0.08177166775639845
Haynes	89.39234599999999	0.7210004674759959
MitchelMoore	105.26857999999999	0.28971011204771174
MersenneTwister	84.006746	0.842132549711396
BlumBlumShub	10499.572807999999	0.0

Figure 12: Test de χ^2 sur les différents générateurs à la seed 1695915794

2.3 Test d'uniformité : χ^2

De la même manière que nos histogrammes, (et puisque que le nombre de valeurs possibles pour chacun de nos générateur dépassent largement une valeur avec lequel il est raisonnable de faire du calcul flottant), on réduit notre étude d'erreur χ^2 sur 100 sous segments de $[0, 1]$. La distribution en ces 100 sous-segment est réalisée à l'aide de la fonction `histogram` de numpy, et le calcul de la *p-value* et du χ^2 est faite à l'aide de la fonction `chisquare` de scipy (voir figure 11). La p-value nous permet ici d'estimer la qualité de nos résultats, avec une p-value la plus proche de 1 la meilleure ; ainsi parmi les résultats les plus notables, on remarque BlumBlumShub a une *p-value* de 0, et que ParkMiller a la meilleure *p-value* (12).

2.4 Étude de l'évolution de la moyenne selon les *seed*

Afin d'avoir un regard plus global sur nos générateurs, et les valeurs obtenues en fonctions des *seed*, on trace, pour chaque générateurs, l'évolution de la moyenne selon la quantité de valeurs aléatoires, pour 500 *seeds* différentes (chaque ligne rouge sur un graphe représente alors l'évolution de la moyenne selon une seed) (??). De ce fait, on peut observer la distributions des valeurs obtenues selon les seeds, et noter quelques biais :

- Bien que BlumBlumShub converge grossièrement vers une moyenne fausse, on peut noter que sa convergence vers sa moyenne est plus rapide que les autres générateurs (ie, on atteint le comportement attendu plus rapidement).
- Certains générateurs n'ont pas un comportements de valeurs aléatoire uniforme satisfaisant avant un très grands nombres de données¹. Notamment, le plus "lent" semble être le Mersenne twister et LavauxJenssens.

¹Pour une déontologie plus poussée, il faut comparer ces "cones" graphique à plusieurs jeu de données de *vrai* aléatoire uniforme

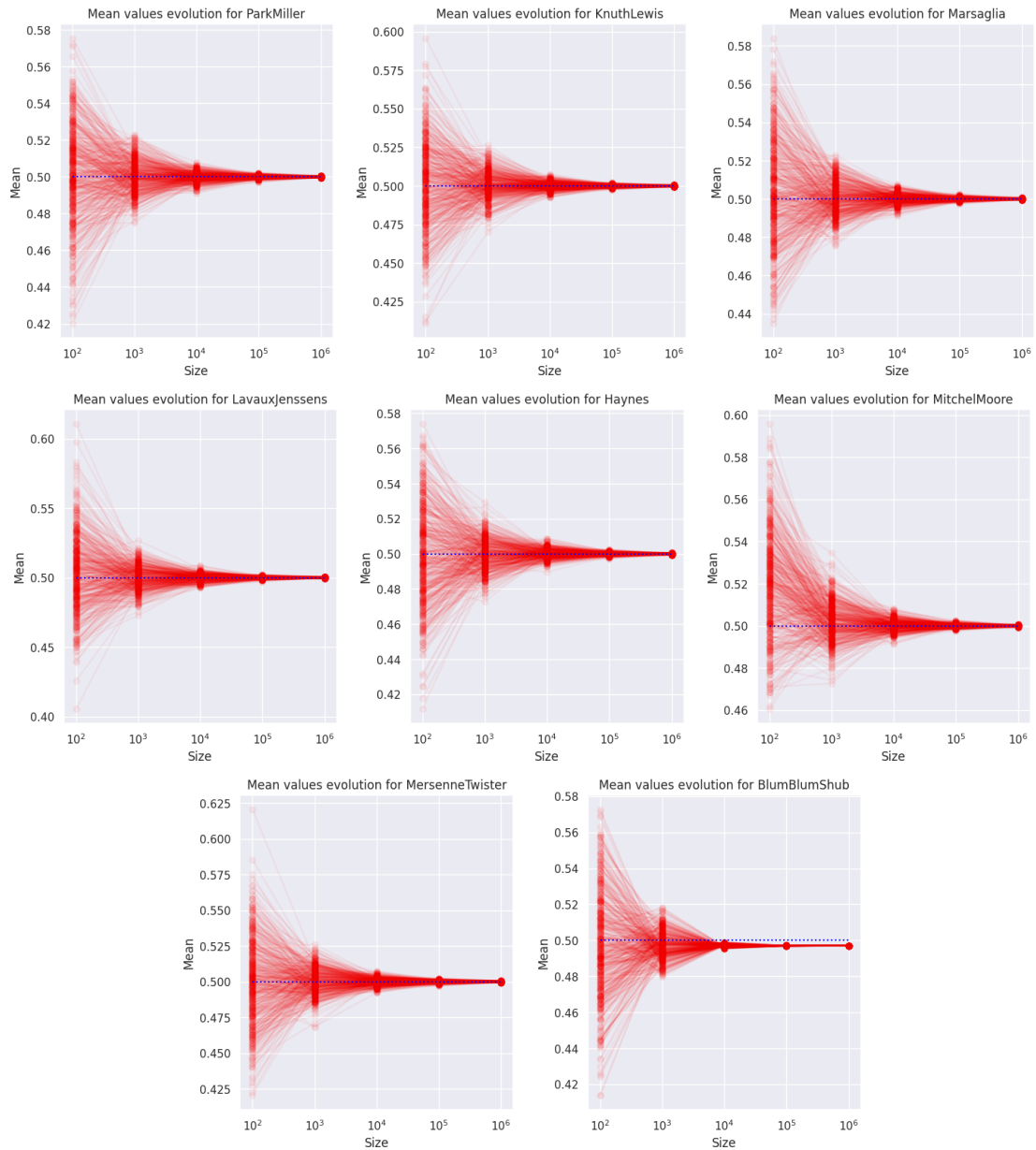


Figure 13: Évolution des moyennes selon le nombres de valeurs, pour 500 seeds

- MitchelMoore a un biais, où la moyenne tends à être penchées vers les valeurs positives pour un faible nombre de valeurs générées. Une manière de l'interpréter et que les 1000 premières valeurs générées avec MitchelMoore ont grossièrement plus de chances d'être positives.