

Simulation Informatique et Gestion des Incertitudes

Génération de distribution Normale

Enzo DE CARVALHO

Ce rendu fait suite au second cours de Simulation Informatique et Gestion d'Incertitudes sur la loi normale. Le but est ici, à partir d'un générateur de nombre aléatoire suivant une loi uniforme, obtenir un générateur de nombre aléatoire suivant une loi normale. Pour cela, nous utilisons ici *Python*. Le code sur lequel ce rendu s'appuie est mis à dispositions dans le dossier `Normal` du dépôt git sur <https://git.iiens.net/de-carva2021/SIGI/>

1 Implémentation

Pour générer des nombres aléatoires sous une loi uniforme, nous utiliserons le générateur de nombres aléatoires de la librairie standard de python : `Random.random()` qui est, comme vu dans le cours précédent, l'implémentation d'un *Mersenne Twister*. Pour chaque échantillon, on génère 1 millions de valeurs qui, comme vu au cours précédent, est une taille acceptable d'échantillon pour bien approximer une distribution de loi uniforme.

Afin d'accélérer la génération de nombres aléatoires *et* de rendus de graphes (ces derniers étant de loin les opérations les plus couteuses en temps de calculs), on emploie le module `multiprocessing` qui permet de paralléliser des tâches sur processeurs (voir `main.py` sur le projet) (1).

2 Méthode Inverse

Comme vu en cours, la fonction inverse de la fonction de repartition F d'une v.a.r X nous permet, à partir d'une v.a.r U de loi uniforme sur $[0;1]$, de retrouver la loi de X avec $F^{-1}(U)$.

À partir de là, on peut donc utiliser la fonction inverse de la loi normale :

$$F^{-1}(x) = \sqrt{2}\text{erfinv}(2x - 1)$$

. La fonction `erfinv` est présente dans le paquet python `scipy` (2).

3 Théorème Central Limite

Avec la formule vue en cours, on génère une loi normale à partir de plusieurs loi uniformes (3).

4 Méthode de Box et Muller

Pour la méthode de Box et Muller, on prend deux échantillons de valeurs supposés indépendants, sur lesquels on applique la formule vu en cours (4)

On obtient alors deux approximations de loi normale. (5)

```

1  if __name__ == "__main__":
2      sns.set_theme(style="whitegrid", palette="rocket")
3      N = 1000000 # 10**6 random values
4      Ns = 12
5      RESOLUTION = 25
6      seed = int(datetime.now().timestamp())
7      data = more_values(N, seed, Ns)
8      #PROCESSING#####
9      multiprocessing.Process(target=job_erfinv, args=[data[0]]).start()
10     multiprocessing.Process(target=job_CentralLimitTheo, args=[data]).start()
11     multiprocessing.Process(target=job_BoxMuller, args=[data[0:2]]).start()
12     multiprocessing.Process(target=job_Marsaglia, args=[]).start()
13     multiprocessing.Process(target=job_rejectCauchy, args=[]).start()

```

Figure 1: Code python parallélisant le traitement de chaque méthode en jobs différents, permettant de pousser la machine jusqu'au maximum de ses capacités

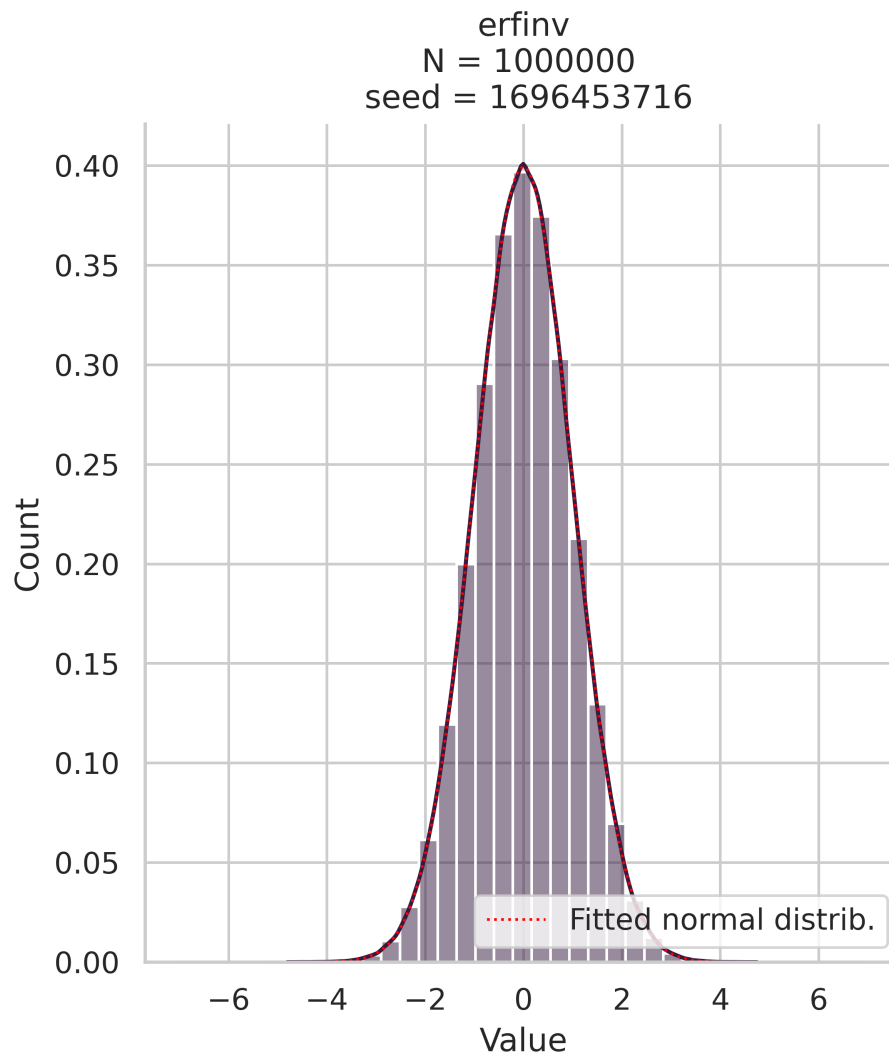


Figure 2: Histogramme de répartition de loi Normale obtenu avec la Méthode inverse. En rouge est tracée la loi Normale la plus proche obtenue avec le méthode fit de scipy.

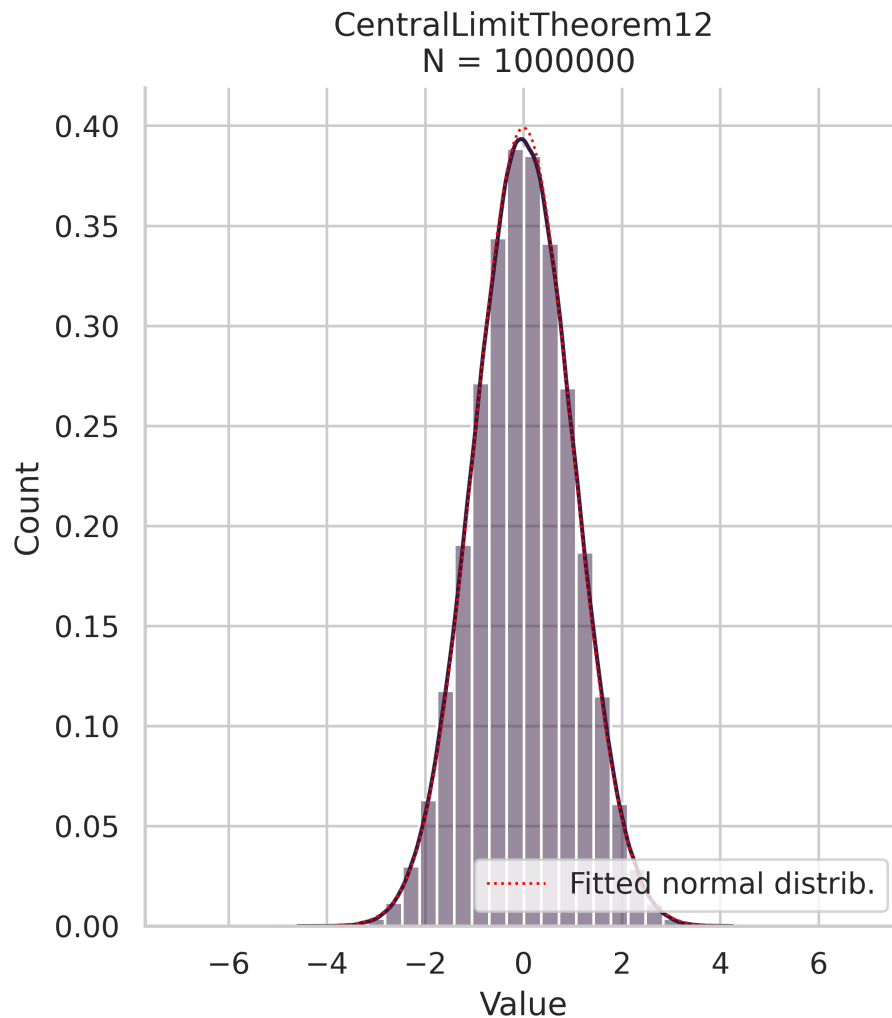


Figure 3: Approximation de la loi normale à partir de 12 lois uniformes (12 échantillons) supposées indépendantes.

Approximation de la loi normale à partir de la méthode de Box et Muller

```

1 def job_BoxMuller(uniforms):
2     name="BoxMuller"
3     print("\033[32;1mStarting BoxMuller job\033[0m")
4     if (len(uniforms) != 2):
5         raise ValueError("need two uniforms values exactly.")
6     U0 = uniforms[0]
7     U1 = uniforms[1]
8     Y0 = np.sqrt(-2*np.log(U0)) * np.cos(2*np.pi*U1)
9     Y1 = np.sqrt(-2*np.log(U1)) * np.cos(2*np.pi*U0)
10    graph.hist_distributivity_graph(N, RESOLUTION, 0
11                                   , {"BoxMuller1" : Y0})
12    graph.hist_distributivity_graph(N, RESOLUTION, 0
13                                   , {"BoxMuller2" : Y1})
14    do_stats_tests(Y0, name+"1")
15    do_stats_tests(Y1, name+"2")
16    return

```

Figure 4: Job BoxMuller pour générer l'approximation sous la méthode de Box et Muller

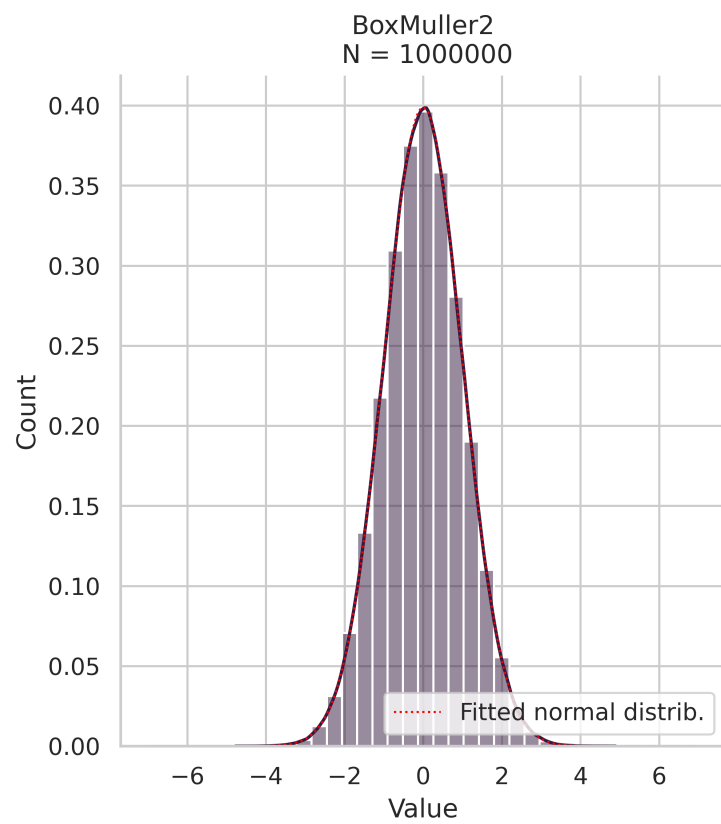
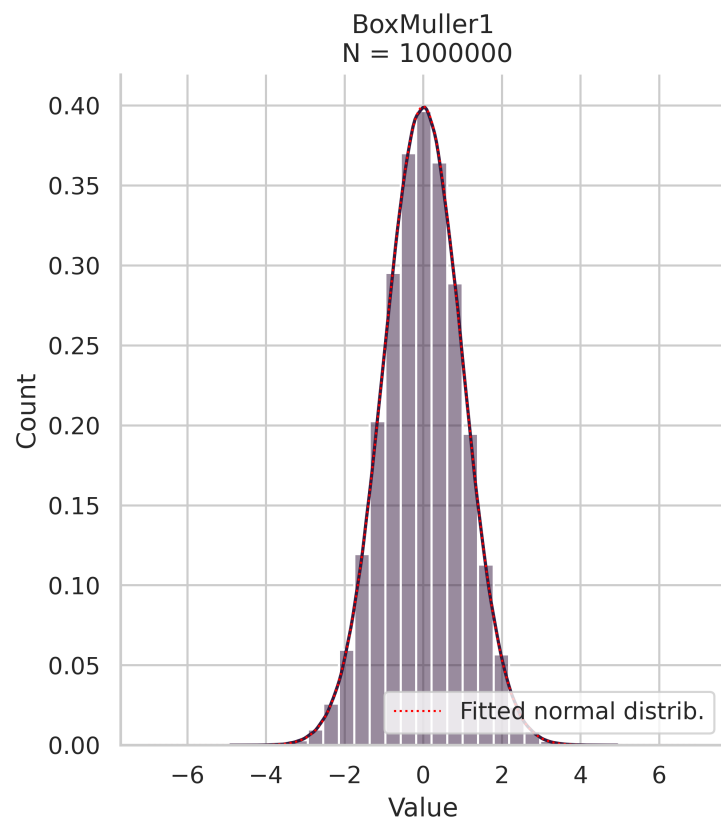


Figure 5: Approximations de la loi normale à partir de la méthode de Box et Muller

```

1 def Marsaglia_pair(N, seed):
2     """
3     Return a pair of two N random uniform values W1 W2
4     in [-1;1]
5     respecting :
6     for each x,y W1,W2
7         0 <= x**x + y**y < 1
8     """
9     print(
10         "creating "
11         + str(N)
12         + " random pair values for Marsaglia with random.random() of seed "
13         + str(seed)
14     )
15     W0 = np.zeros(N)
16     W1 = np.zeros(N)
17     random.seed(seed)
18     i=0
19     while i < N:
20         w0 = 2*random.random() - 1
21         w1 = 2*random.random() - 1
22         while (w0**2 + w1**2 >= 1):
23             w0 = 2*random.random() - 1
24             w1 = 2*random.random() - 1
25
26         W0[i] = w0
27         W1[i] = w1
28         i+=1
29     return [W0, W1]

```

Figure 6: Caption

5 Méthode de Marsaglia

La méthode de Marsaglia se repose sur celle de Box et Muller, mais vise à éviter l'utilisation de fonctions trigonométrique, enclins à des erreurs numériques. Pour cela néanmoins, le jeu de donnée que mange la méthode de Marsaglia nécessite de vérifier des conditions contraignantes pour la génération de nombres. Pour cela, on implémente une méthode auxiliaire pour générer nos échantillons nécessaires (6).

On obtient alors, de nouveau, une paire d'approximation de loi normale (7).

6 Méthode de rejet

On implémente ici la méthode de rejet, plus précisément, celle avec l'enveloppe de Cauchy. Pour cela, on construit d'abord une méthode auxiliaire générique qui génère des valeurs selon la méthode de rejet (8). Cette méthode étant longue à calculer (les éléments sont générés itérativement, et les appels de fonctions alourdissent ce process), on parallélise ce procédé de génération avec `multiprocessing` (9).

Afin d'obtenir les paramètre à passer dans notre fonction de rejet, on prend l'inverse

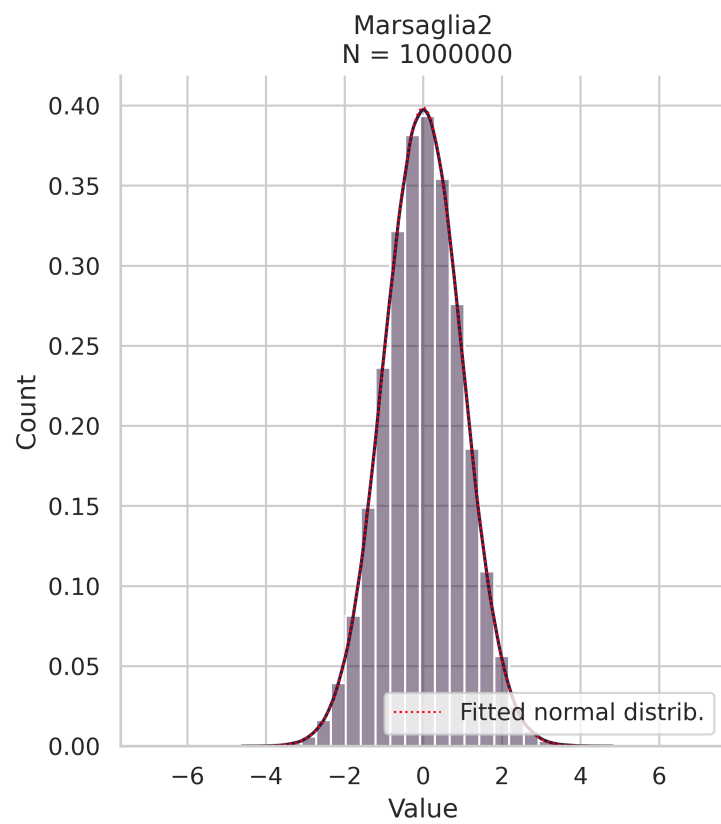
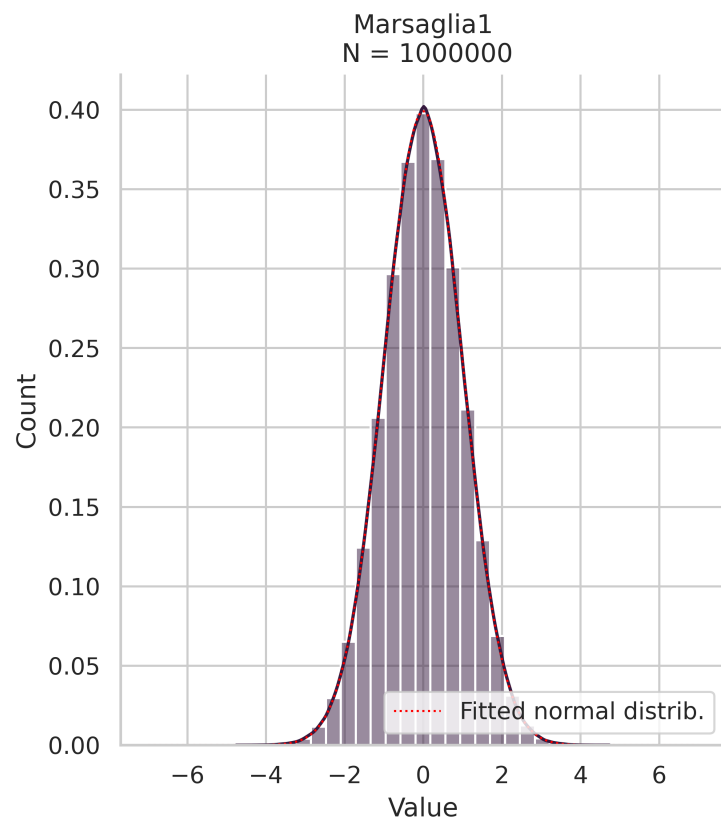


Figure 7: Approximations de la loi normale à partir de la méthode de Marsaglia

```

1 def reject_val(N, seed, Inv, c, g, f):
2     """
3     Return N random values
4     following reject method on c*g(X)*rand < f(X)
5     """
6     print(
7         "creating "
8         + str(N)
9         + " random pair values with reject method with random.random() of seed "
10        + str(seed)
11    )
12    i=0
13    random.seed(seed)
14    res = np.zeros(N)
15    while i < N:
16        u = Inv(random.random())
17        while (c*g(u)*random.random() > f(u)):
18            u = Inv(random.random())
19        res[i] = u
20        i += 1
21    return res

```

Figure 8: Génération de valeurs avec une méthode de rejet.

de la fonction de répartition de la loi de cauchy (centrée):

$$F_{cauchy}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan x$$

qui est alors

$$F_{cauchy}^{-1}(x) = \tan(\pi(x - \frac{1}{2}))$$

7 Tests

scipy inclut déjà une myriade de test statistique que l'on peut donc réaliser sur nos résultats.

7.1 Test de Shapiro-Wilk

Le test de *Shapiro-Wilk* est un test de normalité d'un échantillons, Une statistique W proche de 1 donne une meilleure confiance que nos données sont proches d'une distribution normale. La p-value est également données, et permet de valider ou non notre test. Une p-value faible (disons < 0.05) indique qu'on a assez de preuve pour rejeter l'hypothèse nulle (ie. que nos données semblent suivre une distribution normale.).

D'après les valeurs obtenues avec le test fourni par scipy `scipy.stats.shapiro`, nos valeurs sont largement validées par ce test (10).

```

1 def job_rejectCauchy():
2     name = "rejectCauchy"
3     print("\033[32;1mStarting Cauchy reject job\033[0m")
4     number = 10
5     with multiprocessing.Pool(number) as pool:
6         res = pool.starmap(reject_val
7                             ,[(int(N/10), seed+i, GINVcauchy, cCauchy, gcauchy
8                                , scipy.stats.norm.pdf) for i in range(number)])
9         pool.close()
10        pool.join()
11        fres = np.array(res).flatten()
12        graph.hist_distributivity_graph(N, RESOLUTION, 0
13                                         , {name : fres})
14        do_stats_tests(fres, name)
15    ^^I^^Ireturn

```

Figure 9: Job lui même parallélisé pour la méthode de rejet avec l'enveloppe de Cauchy.

name	W	pvalue
CentralLimitTheorem	1.0000817775726318	1.0
erfinv	1.000215768814087	1.0
BoxMuller1	1.0002559423446655	1.0
BoxMuller2	1.0002448558807373	1.0
Marsaglia1	1.0001895427703857	1.0
Marsaglia2	1.0003535747528076	1.0
rejectCauchy	1.0002989768981934	1.0

Figure 10: Statistique du test de Shapiro-Wilk sur nos différents résultats.

name	stat	pvalue
CentralLimitTheorem	474.5453785864751	8.990427301795029e-104
erfinv	1.7805072225645828	0.41055161902561177
BoxMuller1	0.9613047979479669	0.6183798298122378
BoxMuller2	5.088435295233699	0.07853446958192158
Marsaglia1	1.099736705924237	0.577025769113791
Marsaglia2	8.254907410433615	0.016123882754928123
rejectCauchy	1.0535064870909125	0.5905191323914063

Figure 11: Statistique du test d'Agostino.

name	stat
CentralLimitTheorem	17.204236145480536
erfinv	0.5175240325042978
BoxMuller1	0.15566400391981006
BoxMuller2	0.41687416681088507
Marsaglia1	0.3586901732487604
Marsaglia2	0.637765143532306
rejectCauchy	0.3175980248488486

Figure 12: Statistique du test d'Anderson Darling.

7.2 Test D'Agostino

Le test d'Agostino se base sur le Kurtosis et la *skewness* de nos données pour renvoyer une statistique. Plus la statistique est proche de 0, plus notre distribution se rapproche d'une distribution normale. La p-value se lit de la même manière que pour le test précédent.

D'après les valeurs obtenues avec le test fourni par scipy `scipy.stats.normaltest`, La méthode Central Limite est la plus mauvaise (ce qui se reflète également visuellement ; on peut voir les courbes diverger (3)), et les test de BoxMuller et Marsaglia ne donnent pas de résultats équivalents (le second échantillon est bien meilleur que le premier) (11).

7.3 Test de Anderson-Darling

Le test de Anderson-Darling se lit comme le test d'Agostino, me se base plutôt sur la déviation des données par rapport à une distribution normale.

D'après les valeurs obtenues avec le test fourni par scipy `scipy.stats.anderson`, La méthode Central Limite est encore une fois la plus mauvaise (12).