Rapport TP2 : Générateurs de nombres aléatoires suivants une loi normale

Louis Moreau

18 octobre 2022

Table des matières

1 Introduction

L'objectif de ce TP est de comparer 5 méthodes pour générer des nombres aléatoires suivants une loi normale :

— Méthode inverse
— Théorème Central Limite
— Méthode de Box et Muller
— Méthode polaire de Marsaglia
— Méthode du rejet avec l'enveloppe de Laplace

Nous verrons dans un premier temps le principe de fonctionnement de chaque méthode et leur implémentation. Dans un deuxième temps, nous comparerons leur sortie à l'aide d'outils statistiques (QQ plot, scatter plot, histogramme, espérance, écart type). Dans un troisième temps, nous réaliserons des benchmarks pour comparer la performance de chaque méthode. Nous conclurons par une interprétation des résultats.

2 Fonctionnement et implémentation

2.1 Méthode inverse

Voici le code qui implémente la méthode inverse en Rust :

```
vec_loi[i] = f64::sqrt(2f64) * erf_inv(2f64 * normalize(xoshiro.next_u64()) - 1f64);
```

2.2 Théorème Central Limite

Voici le code qui implémente le théorème Central Limite en Rust :

```
let mut somme = 0f64;
for _j in 0..n {
    somme += normalize(xoshiro.next_u64());
}
vec_centrale_limite[i] = (somme - n as f64 / 2f64) / f64::sqrt(n as f64 / 12f64)
```

2.3 Méthode de Box et Muller

Voici le code qui implémente la méthode de Box et Muller en Rust :

```
let x = normalize(xoshiro.next_u64());
let y = normalize(xoshiro.next_u64());

vec_box_muller[2 * i] = f64::sqrt(-2f64 * f64::ln(x)) * f64::cos(2f64 * PI * y);
vec_box_muller[2 * i + 1] = f64::sqrt(-2f64 * f64::ln(x)) * f64::sin(2f64 * PI * y);
```

2.4 Méthode polaire de Marsaglia

Voici le code qui implémente la méthode polaire de Marsaglia en Rust :

```
let mut x: f64;
let mut y: f64;
let mut s: f64;
loop {
    x = (normalize(xoshiro.next_u64()) - 0.5f64) * 2f64;
```

```
        y = (normalize(xoshiro.next_u64()) - 0.5f64) * 2f64;
        s = x * x + y * y;
        if s > 0f64 && s < 1f64 {
            break;
        }
    }
    vec_marsaglia[2 * i] = x * (f64::sqrt((-2f64 * f64::ln(s)) / s));
    vec_marsaglia[2 * i + 1] = y * (f64::sqrt((-2f64 * f64::ln(s)) / s));
```

2.5 Méthode du rejet avec l'enveloppe de Laplace

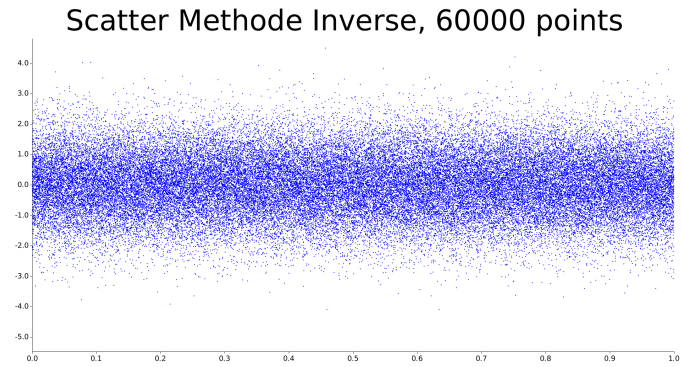
Voici le code qui implémente la méthode du rejet avec l'enveloppe de Laplace en Rust :

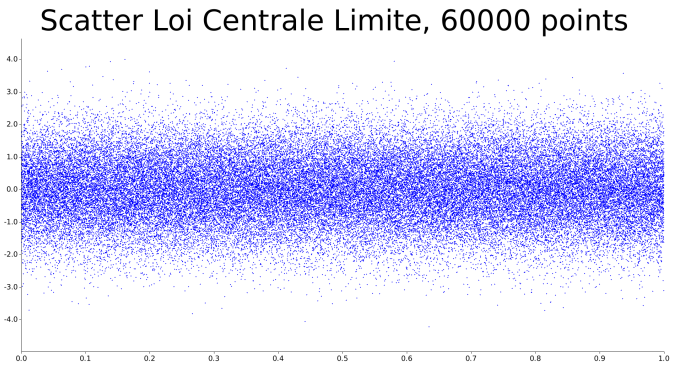
```
    loop {
        let a = normalize(xoshiro.next_u64());
        let u = normalize(xoshiro.next_u64());
        let y = f64::signum(a - 0.5f64)*f64::ln(1f64-2f64*f64::abs(a- 0.5f64));
        let f = f64::exp(-(y*y)/2f64)/ f64::sqrt(2f64 * PI);
        let g = 0.5f64* f64::exp(- f64::abs(y));
        let c = 2f64*f64::sqrt(f64::exp(1f64)/(2f64 * PI));

        if c * g * u <= f {
            if(y == 0f64) {
                println!("zero");
            }
            vec_laplace[i] = y;
            break;
        }
    }
```

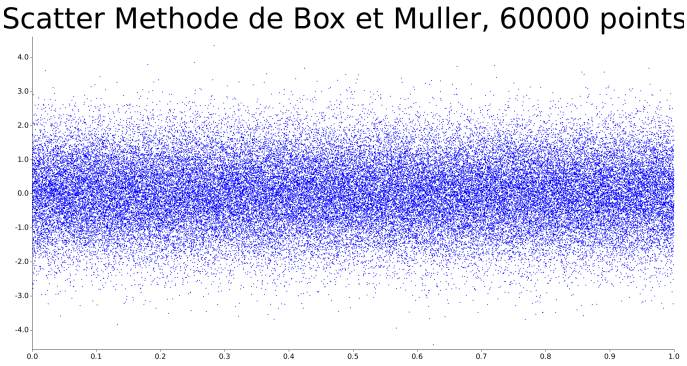
3 Analyse

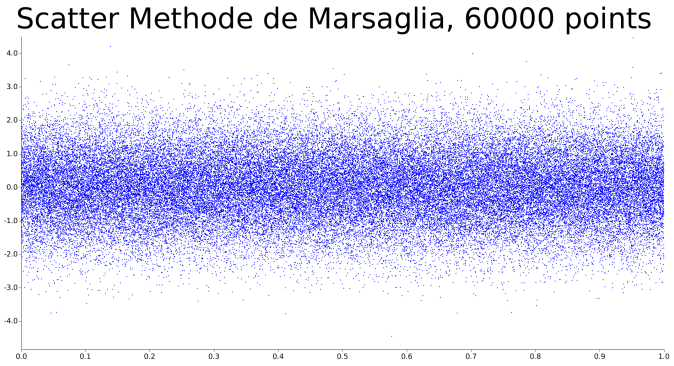
3.1 Analyse Graphique

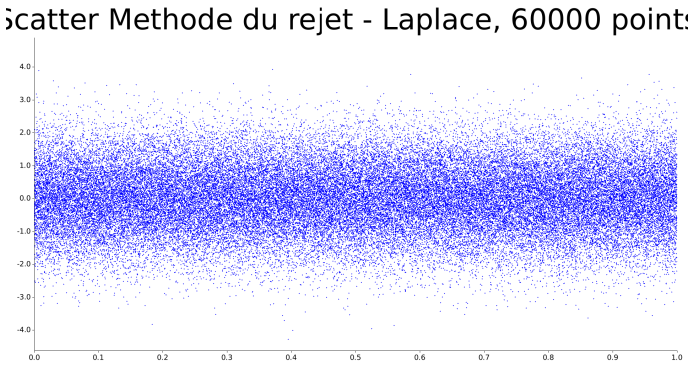
Les graphiques suivant représentent un échantillon de 10 millions de nombres à l'exception du scatter plot qui utilise un ensemble de nombres plus limité par soucis de lisibilité.



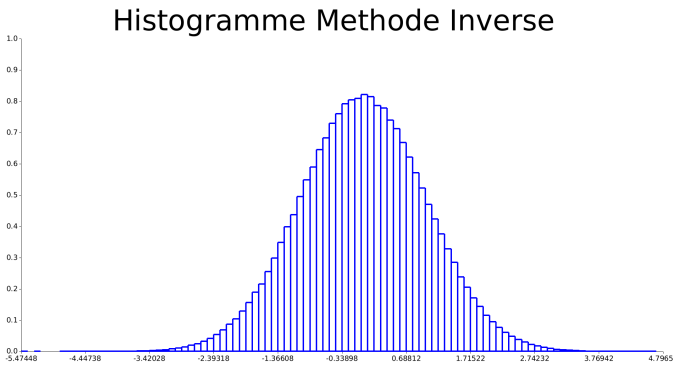


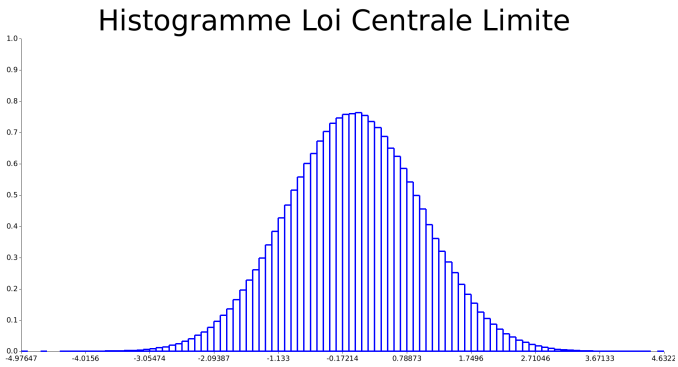


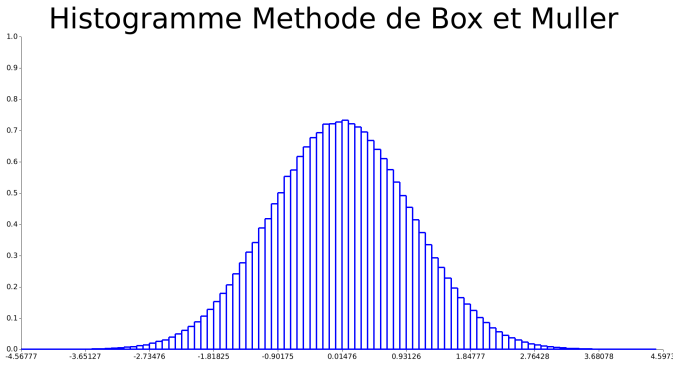


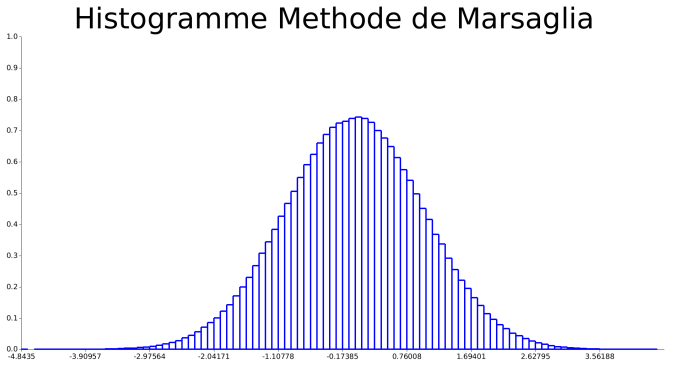


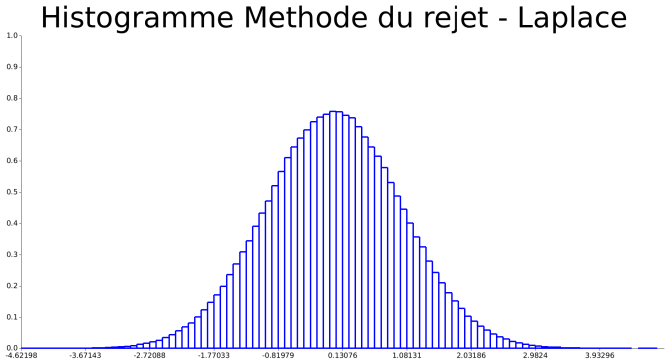
Les 5 scatter plot n'ont pas de motif visible pouvant montrer une périodicité des nombres aléatoires



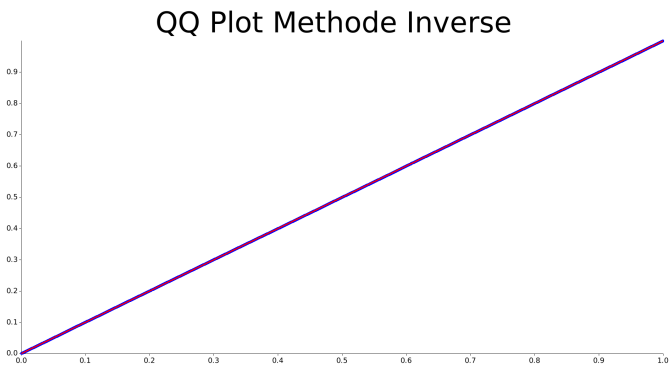


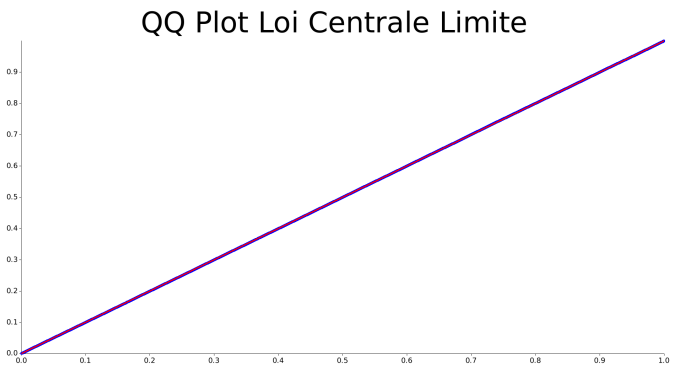


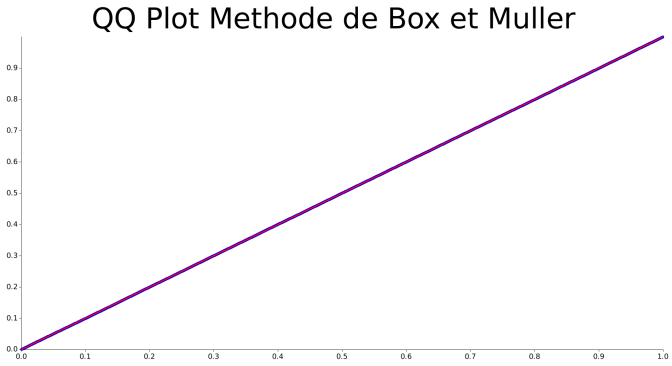


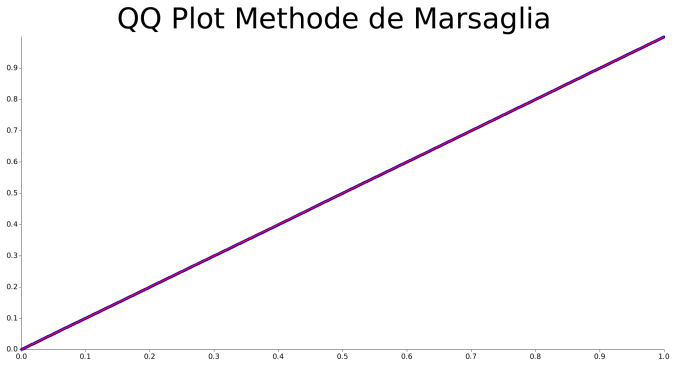


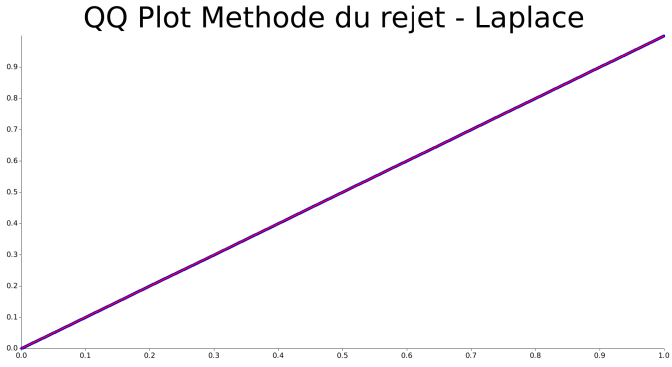
Les 5 histogrammes ont la forme d'une courbe de Gauss sans anomalie visible











Les 5 QQ plots suivent la droite en rouge correspondant à une loi normale parfaite.

3.2 Analyse numérique

	Inverse	**Centrale Limite**	**Box et Muller**	**Marsaglia**	**Rejet-Laplace**
Espérance	**-0.00204**	**-0.00035**	**0.00104**	**-0.00008**	**0.00035**
Ecart type	**1.00153**	**1.00144**	**0.99880**	**1.00034**	**0.99894**

Chacune des méthodes de génération a une espérance proche de 0 ($\pm$ 0,002) et un écart type proche de 1 ($\pm$ 0,002) conformément à la loi normale attendue.

3.3 Conclusion analyse

Grâce à la lecture graphique et l'analyse des espérances et écarts-types, on peut conclure que ces 5 méthodes sont de bon générateurs de nombres aléatoires suivant une loi normale.

4 Benchmark

Ici, un ordinateur avec un processeur AMD Ryzen 5800X est utilisé. Le test consiste à chronométrer le temps pris pour chaque méthode pour générer 10 millions de nombres aléatoires de 64 bits à la suite. On dérive alors le temps pris par itération en divisant le temps obtenu par 10 millions.

	Inverse	**Centrale Limite**	**Box et Muller**	**Marsaglia**	**Rejet-Laplace**
Temps Total	**20.7ms**	**86.1ms**	**13.9ms**	**7.7ms**	**41.6ms**
Par itération	**20ns**	**86ns**	**13ns**	**7ns**	**41ns**

5 Conclusion

Toutes les méthodes présentées ici sont des bonnes méthodes pour générer des nombres aléatoires suivant une loi normale. Cependant, si la performance est un critère important, il apparaît que la méthode polaire de Marsaglia est la plus adaptée.

6 Programme

Le programme est accessible en suivant ce lien : https://github.com/Louis-Moreau/tp2_prng_ensiie

6.1 Installation de Rust

Pour installer Rust : https://www.rust-lang.org/tools/install

6.2 Utilisation du programme

Pour compiler et lancer le programme, il est nécessaire d'être placé dans le fichier /lois_proba . Ensuite, une seule commande suffit :

```
cargo run --release
```