Rapport TP1 : Générateurs de nombres aléatoires

Louis Moreau

5 octobre 2022

Table des matières

1 Introduction

L'objectif de ce TP est de comparer 4 générateurs de nombres aléatoires :
— Un générateur congruentiel linéaire : Park & Miller
— Un générateur fibonacci retardé : Mitchell & Moore
— Un générateur à registre à décalage : Xoshiro256**
— Blum Blum Shub

Nous verrons dans un premier temps le principe de fonctionnement, l'implémentation et les faiblesses potentielles de chaque générateur de nombres aléatoires. Dans un deuxième temps, nous comparerons leur sortie à l'aide d'outils statistiques (corrélation de Pearson, QQ plot, scatter plot, histogramme). Dans un troisième temps, nous réaliserons des benchmarks pour comparer la performance de chaque générateur. Nous conclurons par des recommandations d'utilisation de chaque générateur.

2 Fonctionnement et implémentation

2.1 Park & Miller

Park & Miller est un générateur congruentiel linéaire. Comme tout générateur congruentiel linéaire, il est sous la forme :

$$x_i = (a.x_{i-1}) mod\ m$$

Avec $0 < x_i < m$. Son écriture avec ses valeurs est :

$$x_i = 16807.x_i\ mod\ (2^{32} - 1)$$

Sa période est de $2^{32} - 2$
Voici le code qui implémente le générateur de Park & Miller en Rust :

```
impl RNG32bitOutput for ParkMiller {
    fn next_u32(&mut self ) -> u32 {
        self.seed = lcg32(2u32.pow(32)-1,16807,0,self.seed);
        return self.seed;
    }
}

fn lcg32(m:u32,a:u32,c:u32,x:u32) -> u32 {
    return (a * x + c) % m;
}
```

2.2 Mitchell & Moore

Mitchell & Moore est un générateur fibonacci retardé. Comme tout fibonacci retardé, il est sous la forme :

$$x_i = (x_{i-l} + x_{i-k}) mod\ m$$

Avec $0 < x_i < m$. Son écriture avec ses valeurs est :

$$x_i = (x_{i-24} + x_{i-55}) mod\ m$$

Si m est une puissance de 2, ce générateur a l'avantage de ne nécessiter qu'une addition. Cela permet aussi d'avoir un période extrêmement longue : un multiple de $2^{55} - 1$
Voici le code qui implémente le générateur de Park & Miller en Rust :

```
    impl RNG32bitOutput for MitchellMoore {
        fn next_u32(&mut self ) -> u32 {
            let out = lagged_fibo(self.seed[23] as u64,self.seed[54] as u64,self.modulo) a
            self.seed.rotate_right(1);
            self.seed[0] = out;
            return out;
        }
    }

    fn lagged_fibo(xa:u64,xb:u64,m:u64) ->u64 {
    return(xa + xb) % m;
    }
```

2.3 Xoshiro256**

Xoshiro256** est un générateur à registre à décalage. Il est très récent (2019) et extrêmement performant comme la plupart des algorithmes à registre à décalage. Sa période est de $2^{256} - 1$.

Voici l'implémentation originale en C de l'algorithme : https://xoshiro.di.unimi.it/xoshiro256plusplus.c L'implémentation que nous utiliserons est une transcription de l'implémentation originale en Rust

2.4 Blum Blum Shub

Blum Blum Shub est un générateur de nombre aléatoire. Il se calcule avec cette équation :

$$x_i = (x_{i-1})^2 mod\ m$$

Avec $0 < x_i < m$

m est généré en multipliant 2 nombres : p et q . Ces nombres doivent être premiers et congrus à 3 modulo 4. x_0 ne doit pas être un facteur de p ou q. Pour avoir une période la plus longue possible, le PGCD de $\varphi(p-1)$ et $\varphi(q-1)$ doit être petit. Dans l'implémentation en Rust , nous utiliserons $p = 30000000091$, $q = 40000000003$, $m = 1200000003730000000273$, $x_0 = 8044134300$

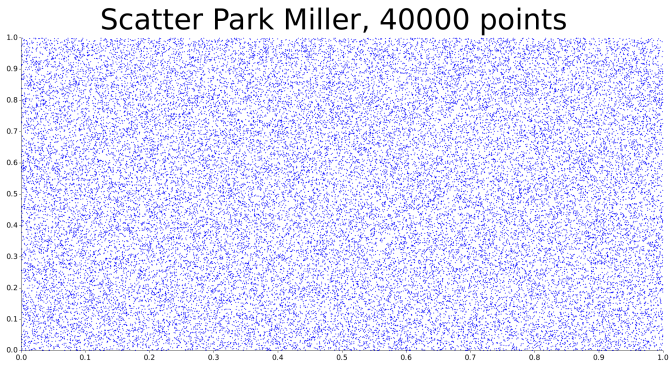
Voici le code qui implémente Blum Blum Shub en Rust :

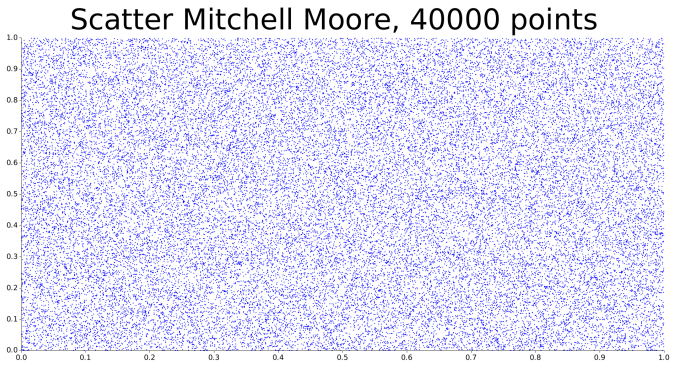
```
fn next_bit(&mut self ) -> u32 {
    self.seed = ((self.seed as u128 * self.seed as u128) % self.m) as u128;
    return self.seed as u32;
}

impl RNG32bitOutput for BlumBlumShub {
    fn next_u32(&mut self) -> u32{
        let mut out : u32 = 0;
        for _i in 0..32 {
            out = (out << 1) | (self.next_bit()as u32 & 1_u32);
        }
        return out;
    }
}
```

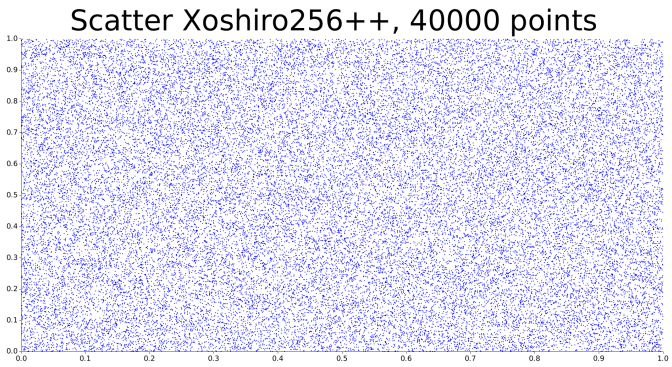
3 Analyse

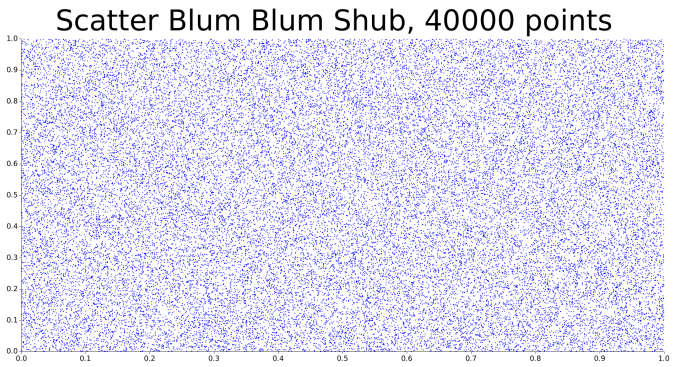
3.1 Analyse Graphique

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.

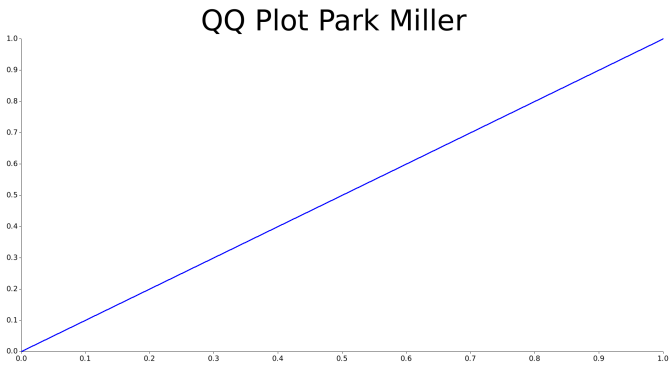


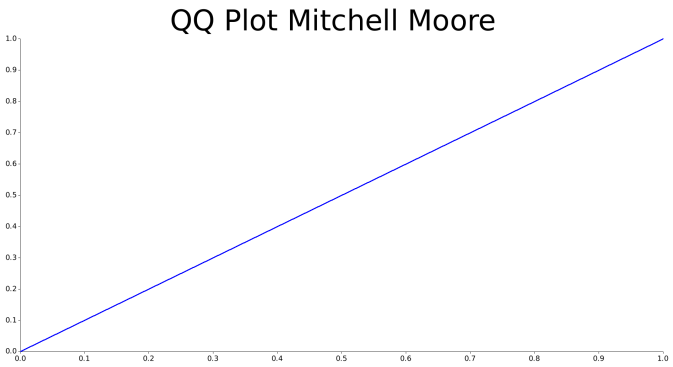


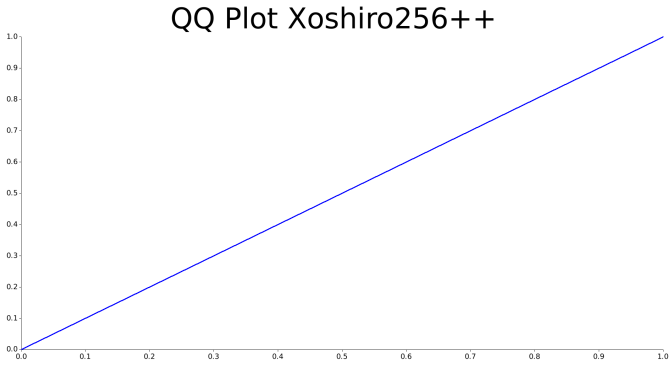


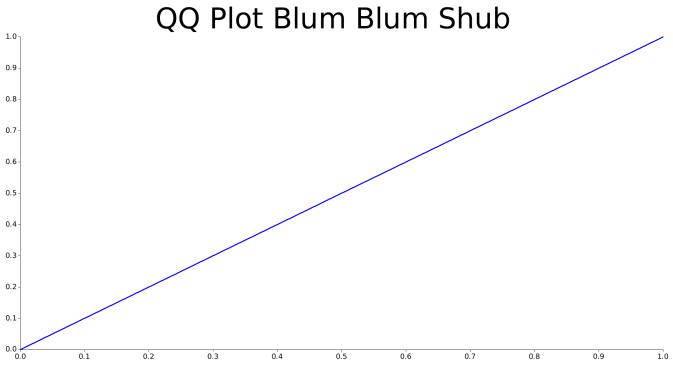


Les 4 scatter plots ne présentent pas de motif reconnaissable visuellement. On peut donc conclure que les générateurs ne sont pas périodiques sur l'échantillon.

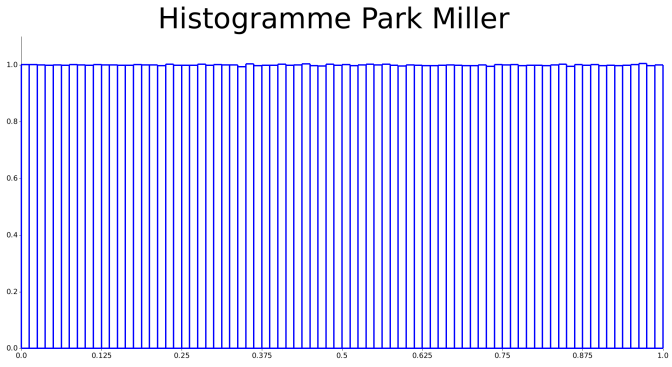


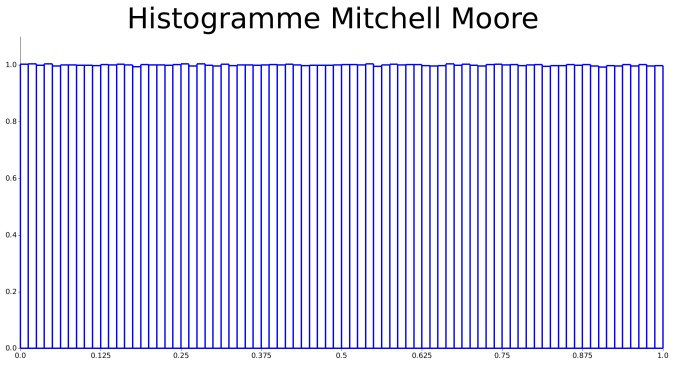


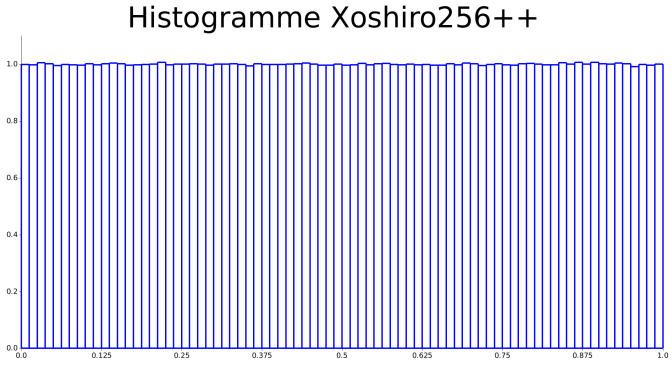


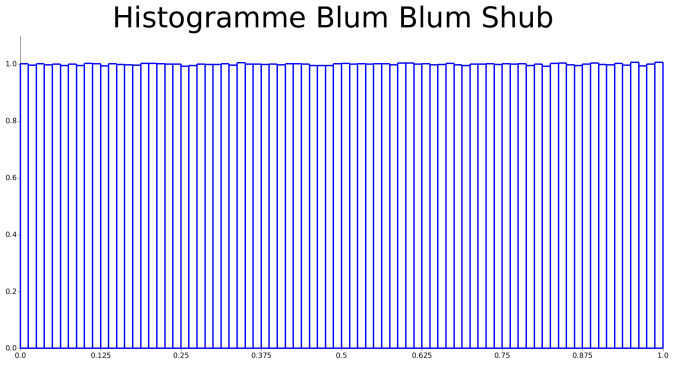


Les 4 QQ plots présentent des droites rectilignes sans anomalie.









Les 4 histogrammes sont relativement plats, cela correspond à ce que l'on attend d'une variable uniforme aléatoire.

3.2 Corrélation de Pearson

	Park Miller	Mitchell & Moore	Xoshiro256++	Blum Blum Shub
Corrélation	**0.00003**	**-0.00030**	**-0.00025**	**0.00043**

Les 4 générateurs n'ont pas de corrélation de Pearson significative entre 2 séries de 10 millions de variables générées avec des seeds proches.

3.3 Conclusion analyse

A partir de la lecture de ses 3 types de graphiques et de la corrélation de Pearson, on peut conclure que les 4 générateurs testés n'ont pas de défaut majeur.

4 Performance

Les tests de chaque générateur sont effectués les uns à la suite des autres sur un même ordinateur. Ici, un ordinateur avec un processeur AMD Ryzen 5800X est utilisé. Le test consiste à chronométrer le temps pris par chaque générateur pour générer 10 millions de nombres aléatoires de 32bit à la suite. On dérive alors le temps pris par itération en divisant le temps obtenu par 10 millions.

	Park & Miller	Mitchell & Moore	Xoshiro256**	Blum Blum Shub
Pour 10 000 000	**9.75ms**	**85.7ms**	**9.62ms**	**2571.84 ms**
Pour 1	**1ns**	**9ns**	**1ns**	**257ns**

5 Conclusion

A partir des connaissances que l'ont possède et des résultats des tests, on peut recommander deux des quatre générateurs. Pour la génération de nombre aléatoire pour un usage non cryptographique, Xorhsiro256** apparaît comme le vainqueur ici, il est le plus performant, génère des nombres avec un bon aléatoire et a une période très longue. Pour la génération de nombre aléatoire pour un usage cryptographique, Blum Blum Shub est le seul adapté à ce cas. Bien que plus lent que les 4 autres, et notamment par rapport à Xorhsiro256** avec un facteur de 100, il est cryptographiquement sûr du notamment au fait que la factorisation de m soit difficile et que seulement 1bit est sorti par itération.

6 Programme

Le programme est accessible en suivant ce lien : https://github.com/Louis-Moreau/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 /random_number_generato
. Ensuite, une seule commande suffit :

```
cargo run --release
```

6.3 Fonctionnement du programme

Pour chaque générateur aléatoire, le programme suit ces étapes :
— Initialisation du générateur
— Début du test de performance
— Génération de 10 millions de valeurs
— Fin du test de performance
— Génération du scatter plot
— Génération du QQ plot
— Génération de l'histogramme
— Analyse avec corrélation de Pearson