ISIMA - Deuxième Année

Laboratoire #2 : Génération de Variates Aléatoires

Nom : Benkhali
Groupe : ISIMA
Professeur : Dr. David Hill
Date : 6 octobre 2024

Table des matières

Introduction **2**
Partie 1 : Implémentation du Mersenne Twister 3
Recherche et Téléchargement 3
Compilation et Test 3
Commentaires 3
Partie 2 : Génération de Nombres Uniformes 4
But de la Partie 4
Implémentation 4
Résultats 4
Visualisation 4
Commentaires 4
Partie 3 : Reproduction de Distributions Empiriques Discrètes 5
But de la Partie 5
Implémentation 5
Résultats 5
Visualisation 5
Commentaires 5
Partie 4 : Reproduction de Distributions Continues 6
But de la Partie 6
Implémentation 6
Visualisation 6
Commentaires 6
Partie 5 : Simulation de Lois Non-Réversibles 7
But de la Partie 7
Implémentation 7
Visualisation 7
Commentaires 8

Ouverture sur la Théorie de la Percolation **9**

Conclusion **10**

Introduction

Ce rapport présente la génération de variates aléatoires en utilisant le générateur Mersenne Twister. Nous suivrons les étapes demandées dans le laboratoire, allant de la recherche sur le générateur à l'implémentation de différentes méthodes de génération, en passant par la visualisation des résultats obtenus.

Partie 1 : Implémentation du Mersenne Twister

Recherche et Téléchargement

Nous avons trouvé l'implémentation C du Mersenne Twister sur la page de Matsumoto. Le code source a été téléchargé et décompressé à l'aide de la commande suivante :

```
tar zxvf mt19937-2002.tgz
```

Compilation et Test

Pour compiler l'implémentation, nous avons utilisé :

```
gcc mt19937-2002.c -o mt_test
```

Après compilation, nous avons exécuté le programme pour vérifier la reproductibilité des résultats, en utilisant les fonctions `genrand_int32` et `genrand_real1`. Voici un exemple de code de test :

```
#include "mt19937ar.h"
#include <stdio.h>

int main() {
    init_genrand(5489); // Initialisation du g n rateur avec une graine
    printf("10 nombres pseudo-al atoires g n r s :\n");
    for (int i = 0; i < 10; i++) {
        printf("genrand_int32: %u, genrand_real1: %f\n", genrand_int32(), g
    }
    return 0;
}
```

Commentaires

Les sorties obtenues étaient conformes aux résultats attendus, confirmant la bonne implémentation du générateur Mersenne Twister.

Partie 2 : Génération de Nombres Uniformes

But de la Partie

Nous allons générer des nombres uniformes entre $[-98, 57.7]$ à l'aide de la fonction 'uniform'.

Implémentation

La fonction 'uniform' est définie comme suit :

```
double uniform(double a, double b) {
    return a + (b - a) * genrand_real1();
}
```

Résultats

Voici quelques exemples de nombres générés :
— -59.30, -63.35, 50.89, 55.31, -89.75

Visualisation

L'histogramme montrant la répartition des nombres générés est affiché ci-dessous :

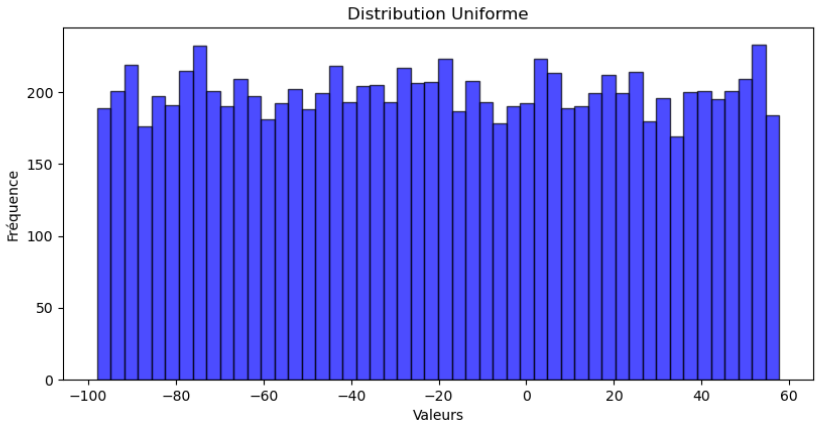

FIGURE 1 – Distribution uniforme des nombres entre -98 et 57.7

Commentaires

L'histogramme montre une distribution uniforme, ce qui valide le bon fonctionnement de la fonction 'uniform'.

Partie 3 : Reproduction de Distributions Empiriques Discrètes

But de la Partie

Nous allons simuler une distribution discrète avec trois classes : A, B et C. Les proportions respectives de ces classes sont 35%, 45%, et 20%.

Implémentation

Nous avons implémenté la simulation comme suit :

```
void simulate_discrete(int n) {
    int countA = 0, countB = 0, countC = 0;

    for (int i = 0; i < n; i++) {
        double r = genrand_real1();
        if (r < 0.35) countA++;
        else if (r < 0.80) countB++;
        else countC++;
    }

    printf("Classe A: %.2f%%\n", (double)countA / n * 100);
    printf("Classe B: %.2f%%\n", (double)countB / n * 100);
    printf("Classe C: %.2f%%\n", (double)countC / n * 100);
}
```

Résultats

Voici les résultats pour un échantillon de 1 000 tirages :
— Classe A : 34.20%
— Classe B : 45.50%
— Classe C : 20.30%

Visualisation

Voici la répartition des classes sous forme d'un graphique à barres :

Commentaires

Les résultats de la simulation sont conformes aux proportions attendues, ce qui valide l'efficacité de l'implémentation.

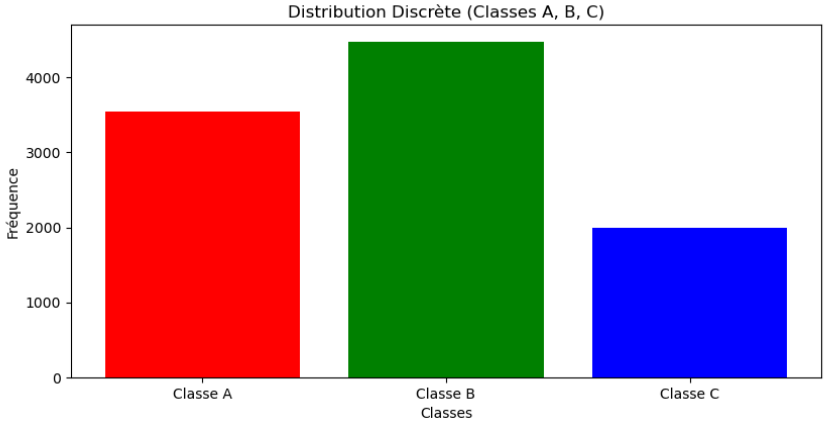

FIGURE 2 – Simulation de la distribution discrète (Classes A, B, C)

Partie 4 : Reproduction de Distributions Continues

But de la Partie

Nous allons reproduire une distribution continue en utilisant une loi exponentielle négative et la méthode d'inversion.

Implémentation

La fonction 'negExp' est définie comme suit :

```
double negExp(double mean) {
    double random_number = genrand_real1();
    return -mean * log(1 - random_number);
}
```

Visualisation

L'histogramme ci-dessous montre la répartition des 10 000 nombres générés suivant la loi exponentielle négative :

Commentaires

La distribution générée montre que la plupart des nombres sont concentrés autour de petites valeurs, ce qui est typique d'une loi exponentielle négative.

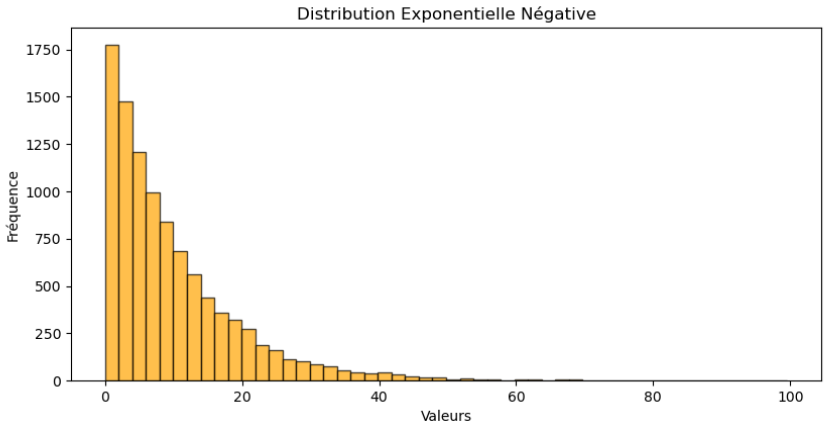

FIGURE 3 – Distribution exponentielle négative des nombres générés

Partie 5 : Simulation de Lois Non-Réversibles

But de la Partie

Nous allons utiliser la méthode de rejet pour générer des nombres selon une loi normale.

Implémentation

La méthode de Box-Muller est utilisée pour générer des nombres suivant une loi normale.

Visualisation

L'histogramme ci-dessous montre la répartition des nombres générés par la méthode de Box-Muller :

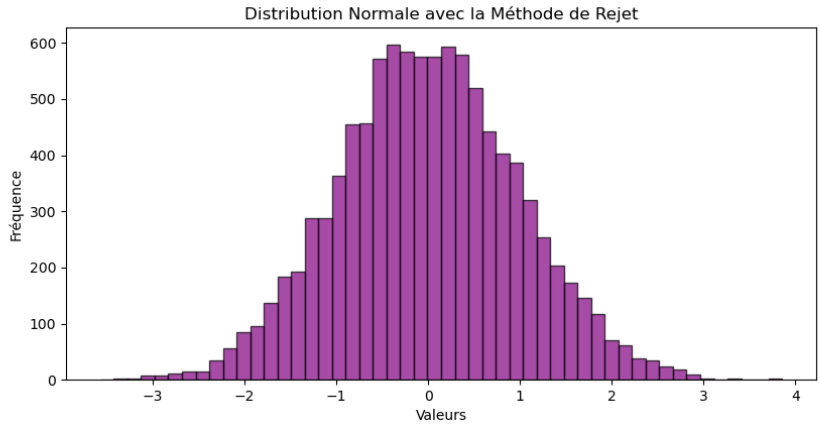

FIGURE 4 – Distribution normale générée par la méthode de Box-Muller

Commentaires

La courbe obtenue est symétrique autour de zéro, ce qui confirme l'efficacité de la méthode de Box-Muller pour générer des nombres suivant une distribution normale.

Ouverture sur la Théorie de la Percolation

Dans le cadre de ce laboratoire, il est intéressant de faire un lien entre les techniques de génération de nombres aléatoires et la théorie de la percolation. En effet, la percolation est une branche des mathématiques qui étudie le comportement des réseaux et des milieux désordonnés. La génération de réseaux aléatoires, par exemple, repose sur des algorithmes de génération de nombres aléatoires, permettant de simuler des phénomènes tels que la propagation d'infections ou la diffusion de fluides.

L'utilisation de générateurs de haute qualité, comme le Mersenne Twister, est cruciale pour obtenir des simulations précises et fiables. En appliquant ces techniques à des modèles de percolation, nous pouvons explorer des comportements critiques dans des systèmes complexes, mettant en lumière l'importance de la génération de nombres aléatoires dans des contextes variés.

Conclusion

Dans ce rapport, nous avons exploré plusieurs méthodes pour générer des nombres pseudo-aléatoires suivant différentes lois de probabilité (uniforme, exponentielle, discrète et normale). À l'aide du générateur Mersenne Twister, nous avons pu vérifier la qualité des nombres générés et visualiser les résultats sous forme de graphiques. Chaque méthode a montré sa capacité à générer des échantillons représentatifs des distributions théoriques, soulignant ainsi l'importance de choisir la bonne méthode de génération pour chaque application.