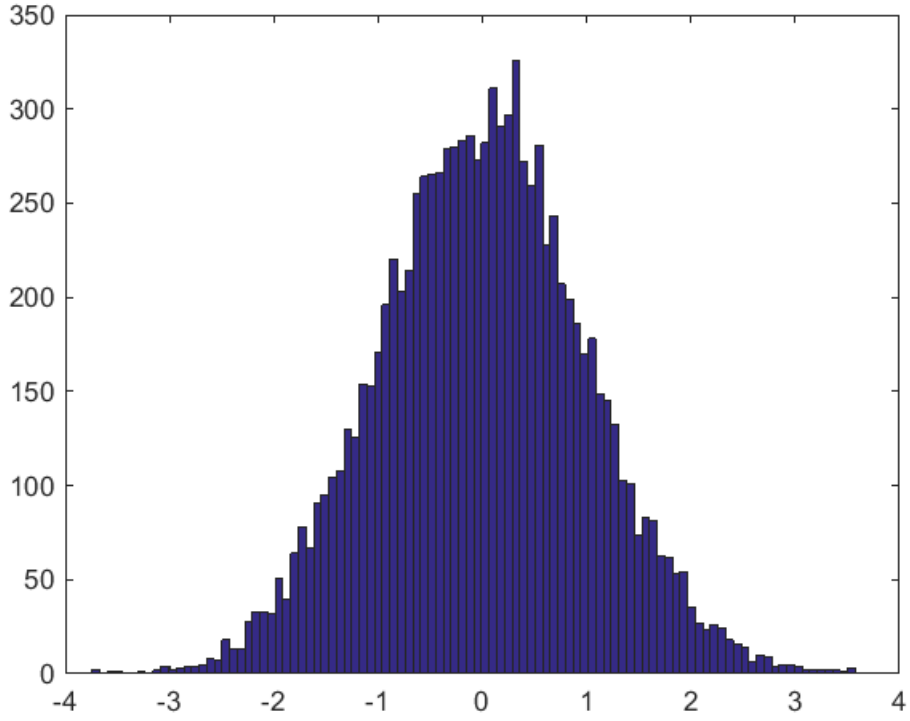
TP

Génération de variables aléatoires



CHASSAGNOL Rémi
ZZ2 - F2 - promo24

Professeur: HILL Davide

11 Octobre 2022

Contents

1 Test de Mercenne Twister

Tout d'abord, il faut tester la répétabilité du générateur. L'archive téléchargée sur le site de **Makoto Matsumoto** contient un fichier `mt19937ar.out` qui correspond à la sortie attendue du programme. Ici, il suffit de tester si le programme affiche bien le résultat attendu. Pour ce faire, on redirige la sortie standard du programme vers un nouveau fichier, puis on utilise la commande `diff` pour tester si la sortie est correcte.

```
$ gcc mt19937ar.c
$ ./a.out > tmp.txt
$ diff tmp.txt mt19937ar.out
```

La commande `diff` n'affiche rien, la sortie du programme correspond bien à celle attendue.

2 Génération uniforme entre A et B

Pour générer un nombre aléatoire dans un intervalle $[a, b]$, nous allons utiliser la fonction `genrand_real1()` qui génère un nombre aléatoire entre 0 et 1 (1 compris). On obtient la fonction suivante:

```
/* G n re un nombre al atoire compris entre `a` et `b`. */
double uniformAB(double a, double b) {
  return a + (b - a)*genrand_real1();
}
```

Si on essais de générer **1000** nombres entre **-89.2** et **56.7** on obtient la répartition suivante:

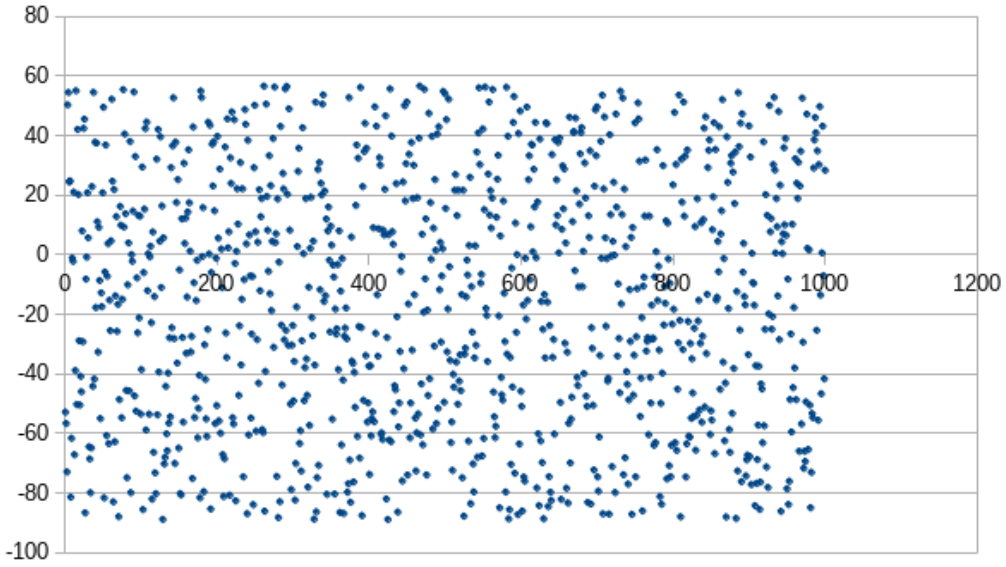


Figure 1: Répartition génération uniforme

On observe que les points sont répartit de manière assez uniforme sur l'espace, il n'y a pas de cluster apparent, on peut donc en conclure que le générateur est assez performant.

3 Reproduction d'une distribution discrète empirique

3.1 Cas spécifique à 3 classes

Tout d'abord, on souhaite générer reproduire une distribution à partir de 3 classes A, B, et C. On dispose des informations suivantes:

- 350 observations de la classe A **(35%)**
- 450 dans la classe B **(45%)**
- 200 dans la classe c **(20%)**

Pour ce faire, on utilise la fonction `genrand_real1()` et le principe suivant:

- si elle retourne un nombre inférieur à `0.35`, on est dans la classe **A** (on retourne **0**)
- pour un nombre compris entre `0.35` et `0.80` on est dans la classe **B** (on retourne **1**)
- sinon on est dans la classe **C** (on retourne **2**)

Avec cette technique et sachant que `genrand_real1()` retourne des nombre pseudo-aléatoires uniformément répartit s dans $[0, 1]$, on a bien 35% de chance de tirer un individus de la classe A, 45% de la classe B et 20% de la classe C.

Le code source de la fonction est le suivant:

```
/* Reproduction d'une distribution discr te empirique avec les 3 classes A, B, C
 */
int genrand_discEmpDist() {
  double r = genrand_real1();
  int output = 2; // classe C par d faut

  if (r <= 0.35) { // classe A
    output = 0;
  } else if (r <= 0.80) { // classe B
    output = 1;
  }
  return output;
}
```

A noter que pour éviter une condition inutile, on se place dans la classe C par défaut et on change en si besoin.

On teste ensuite ce générateur en faisant plusieurs tirages que l'on comptabilise dans un tableau, puis on calcule le pourcentage d'individus trouvés dans chaque classes.

```
discEmpDist - 1000
A: 0.3430000000
B: 0.4550000000
C: 0.2020000000

discEmpDist - 10000
A: 0.3545000000
B: 0.4447000000
C: 0.2008000000
```

```
discEmpDist - 100000
A: 0.3493800000
B: 0.4503200000
C: 0.2003000000
```

On observe déjà de bons résultats pour 1000 tirages, et on peut effectivement voir que la précision augmente avec le nombre de tirages. On passe de près de **2%** d'erreur sur la classe A pour 1000 à **0.2%** pour 100000 tirages.

3.2 Fonction plus générique

Ici, on suit le même principe que précédemment sauf que le but est d'implémenter une fonction qui prend en compte un nombre indéterminé de classes. Cette fonction prend en entrée le nombre de classe et un tableau d'effectifs, qui correspond aux effectifs que l'on souhaite observer pour chaque classes. On commence par calculer l'effectif total puis, à l'aide d'une boucle `for`, on test pour toutes les classes, si l'effectif cumulé divisé par l'effectif total est supérieur ou égal à un nombre aléatoire `r` tiré à l'aide de `genrand_real1()`. Quand $r > \frac{effectifcumulé}{effectiftotal}$ alors `r` appartient à la classe `i`.

```
int genrand_discEmpDist2(int nbClasses, int effectifs[]) {
  double r = genrand_real1();
  int output = 2;
  int effectifTotal = 0;
  int effectifCum = 0;
  int i;

  // calcul effectif total
  for (i = 0; i < nbClasses; ++i){
    effectifTotal += effectifs[i];
  }

  // calcul du resultat
  for (i = 0; i < nbClasses; ++i){
    effectifCum += effectifs[i];
    if (r <= (double) effectifCum / effectifTotal) {
      output = i;
      i = nbClasses;
    }
  }
  return output;
}
```

On test la fonction avec 5 classes et des effectifs choisis arbitrairement et on obtient des résultats similaires à la précédente question.

Pour `effectifs[5] = {20, 5, 50, 10, 15}`:

```
discEmpDist2 - 100000
0: 0.2026500000
1: 0.0494200000
2: 0.4980800000
3: 0.0993600000
4: 0.1504900000
```

4 Reproduction de distribution continues

4.1 Fonction `negEx`

Dans cette partie, on va générer des nombre aléatoires suivant une loi exponentielle négative de moyenne donnée. Pour ce faire on utilise la technique d'anamorphose pour inverser la loi. On obtient donc la fonction suivante:

```
double negExp(double mean) {
  return -mean*log(1 - genrand_real2());
}
```

On test avec le code ci-dessous que la suite de nombre retournée par `negExp()` après un certain nombre de tirage a bien une moyenne de 11.

```
void testExp() {
  double r;
  int i;

  for (i = 0; i < 1000; ++i) {
    r += negExp(11);
  }
  printf("mean for 1000: %.10lf\n", (double) r/1000);

  for (i = 0; i < 1000000; ++i) {
    r += negExp(11);
  }
  printf("mean for 1000000: %.10lf\n", (double) r/1000000);
}
```

On voit bien que la moyenne trouvé est proche de 11, on a donc bien le résultat souhaité. On constate aussi que plus le nombre de tirages est grand plus la précision augmente. C'est un résultat encourageant puisqu'on souhaite que la suite de nombres retournée par `negExp` converge vers une loi exponentielle négative de moyenne 11.

```
mean for 1000: 11.0999044573
mean for 1000000: 11.0119343037
```

4.2 Distribution discrète

On souhaite maintenant utiliser la fonction précédente pour générer une distribution discrète de 1 à 22. On va donc utiliser un tableau pour récupérer les fréquences d'apparition des nombre entre 0 et 1, 1 et 2 et ainsi de suite. La dernière case contiendra la fréquence d'apparition des nombre plus grand que 22. Voici la fonction utilisé:

```
void testDiscretisationNegExp(int n) {
  int test23bins[23] = {0};
  int i, j;
  double r;

  for (i = 0; i < n; ++i) {
    r = negExp(11);
    // recherche de la position dans le tableau
    for (j = 0; j < 21; ++j) {
      if (r >= j && r < j + 1) {
  test23bins[j]++;
      }
    }
    if (r >= 22) {
      test23bins[22]++;
    }
  }

  for (i = 0; i < 22; ++i) {
    printf("%d: %d\n", i, test23bins[i]);
  }
}
```

4.2.1 Test pour 1000 tirages

Dans un premier temps, on test avec **1000** tirages et on obtient le résultat suivant:

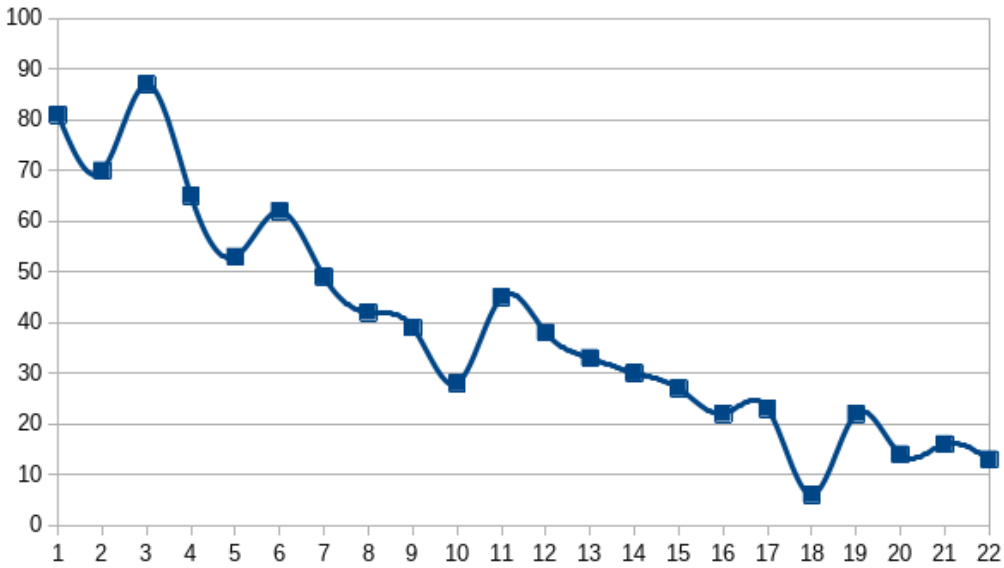


Figure 2: Graphe discrétisation entre 0 et 22 pour 1000 tirages.

On constate que la courbe décroît bien comme une loi exponentielle inverse, cependant on voit clairement des irrégularités.

4.2.2 Test pour 1000000 tirages

Ensuite on reproduit la même expérience mais cette fois ci avec **1000000** tirages:

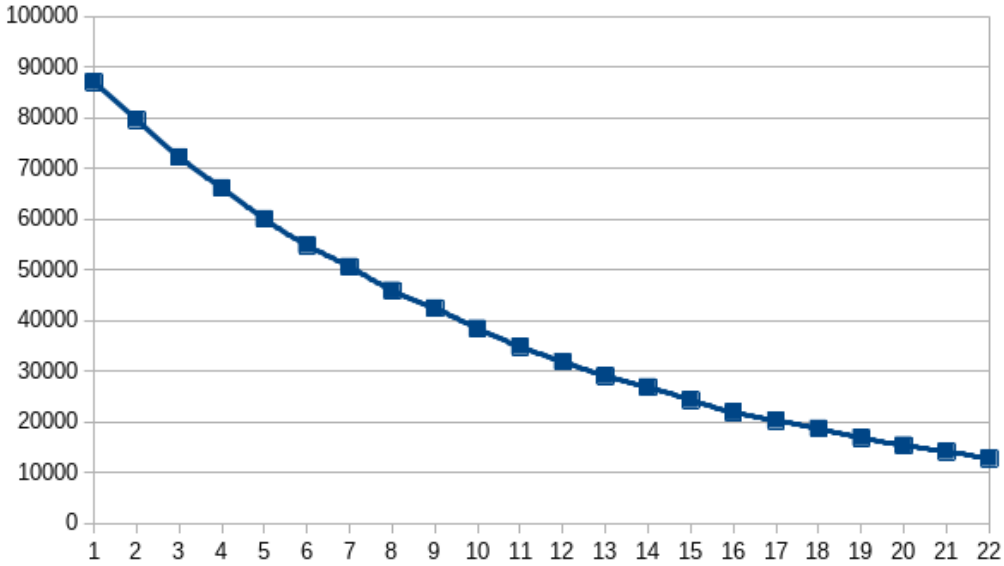


Figure 3: Graphe discrétisation entre 0 et 22 pour 1000000 tirages.

Ici on voit bien apparaître une loi exponentielle négative et la courbe est assez régulière. La suite de nombres retournée par `negExp` converge bien vers une loi exponentielle négative pour un grand nombre de valeurs. De plus, comme **Mercenne twister** est un générateur fiable (il ne boucle jamais sur les mêmes valeurs), on peut conclure que notre suite de nombre convergera toujours vers la loi souhaité même pour un très grand nombre de tirages.

5 Simulation de lois irréversibles

5.1 Méthode avec des lancés de dés

Dans cette partie, on va essayer de simuler une loi normale en utilisant un générateur de nombre pseudo-aléatoires. Le but est de sommer les résultats de **30 lancés de dés 6** puis de comptabiliser cette somme dans un tableau d'effectifs d'intervalles discrets entre 30 et 180.

5.1.1 Le générateur aléatoire

Tout d'abord, implémentons le générateur qui retourne la somme des résultats de 30 lancés de dés 6. Pour cela, on va utiliser la fonction `genrand_int32()%6 + 1` car on veut un nombre entier entre 1 et 6. La somme des résultats est stockée dans la variable `acc`.

```
int genrand_gaussienne() {
  int i;
  int acc = 0;

  for (i = 0; i < 30; ++i) {
    acc += genrand_int32()%6 + 1;
  }

  return acc;
}
```

La fonction retourne un nombre entre 30 et 180.

5.1.2 Calcule de la moyenne

Le but et de vérifier que notre méthode permet bien de simuler une loi normale. Tout d'abord il faut vérifier que la moyenne est bien environ égale à $\frac{30+180}{2} = 105$. Pour cela, on affiche le résultat de la fonction suivante:

```
double moyenneGaussienne(int n) {
  int i;
  double acc = 0;

  for (i = 0; i < n; ++i) {
    acc += genrand_gaussienne();
  }
  return acc / n;
}
```

```
104.953200
```

Pour 1000000 tirages, on a bien une moyenne proche de 105, ce qui est logique par rapport au résultat souhaité.

5.1.3 Courbe de Gauss

Maintenant, on va vérifier que la simulation s'approche bien d'une loi normale, pour ce faire, nous allons utiliser un tableau qui va contenir les effectifs des résultats trouvés. On procède de manière similaire que pour `negExp` sauf qu'ici, on peut directement incrémenter la bonne case de tableau car on ne traite plus avec des intervalles mais directement les valeurs.

Les nombre tirés sont compris entre 30 et 180, il faut donc un tableau de taille 151 pour stocker tous les résultats possibles. Le principe est de tirer un nombre pseudo-aléatoires à l'aide de `genrand_gaussienne()` puis d'incrémenter la cases du tableau des effectifs. La fonction est la suivante:

```
void simulationCourbeGauss(int n) {
  int i;
  int r;
  int testBins[151] = {0};

  // remplissage du tableau
  for (i = 0; i < n; ++i) {
    r = genrand_gaussienne();
    testBins[r - 30]++;
  }

  // affichage des effectifs
  for (i = 0; i < 150; ++i) {
    printf("%d\n", testBins[i]);
  }
}
```

On obtient le résultat suivant:

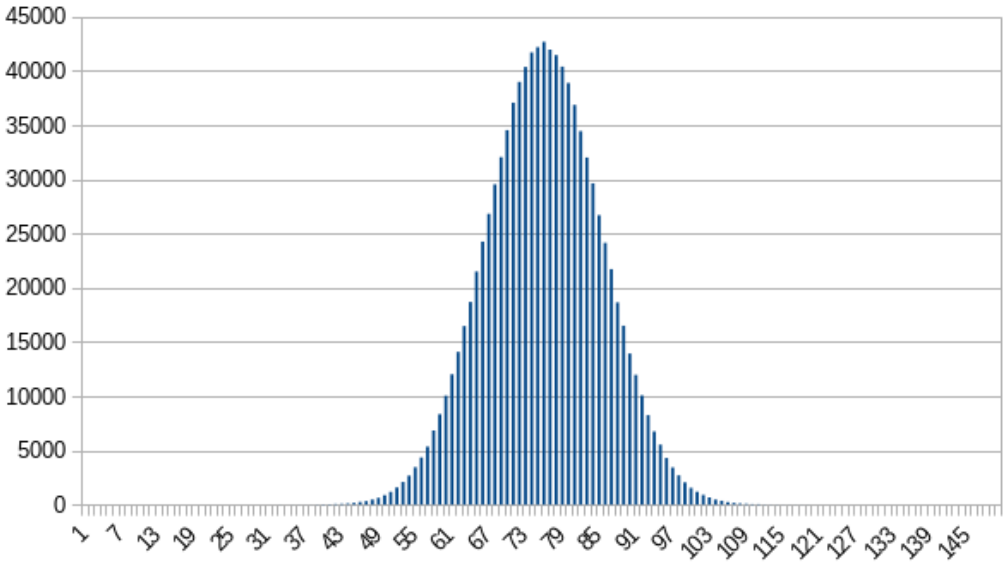


Figure 4: résultats simulation courbe de Gauss

On voit bien sur ce graphique que les résultats sont cohérents, on a bien simuler une soi normale.

5.2 Box and Muller

Ici l'objectif est le même que précédemment, sauf qu'on va utiliser la méthode de **Box et Muller** qui consiste à tirer deux nombres aléatoires suivant cette formule:

$$x_1 = \cos 2\pi Rn_2 \times \sqrt{(} - 2\log(Rn_1))$$
$$x_2 = \sin 2\pi Rn_2 \times \sqrt{(} - 2\log(Rn_1))$$

La fonction est la suivante:

```
void boxAndMuller(int n) {
  double x[2];
  int testBins[20] = {0};
  int i, j;
  double borne;

  for (i = 0;  i <= n; ++i) {
    genrand_boxAndMuller(x);

    j = 0;
    for (borne = -5; borne < 5; borne += 0.5) { // place x1 et x2 dans `testBins`
      if ((x[0] >= borne && x[0] < borne + 0.5)
    || (x[1] >= borne && x[1] < borne + 0.5)) {
  testBins[j]++;
      }
      ++j;
    }
  }
  // affichage
  for (i = 0; i < 20; ++i) {
    printf("%d\n", testBins[i]);
  }
}
```

On commence par remplir la tableau `testBins` qui contient les effectifs des tirages appartenant à des intervalles discrets de taille 0.5 sur $[-5; 5]$, ensuite on affiche `testBins`. Voici les résultats dans un tableur:

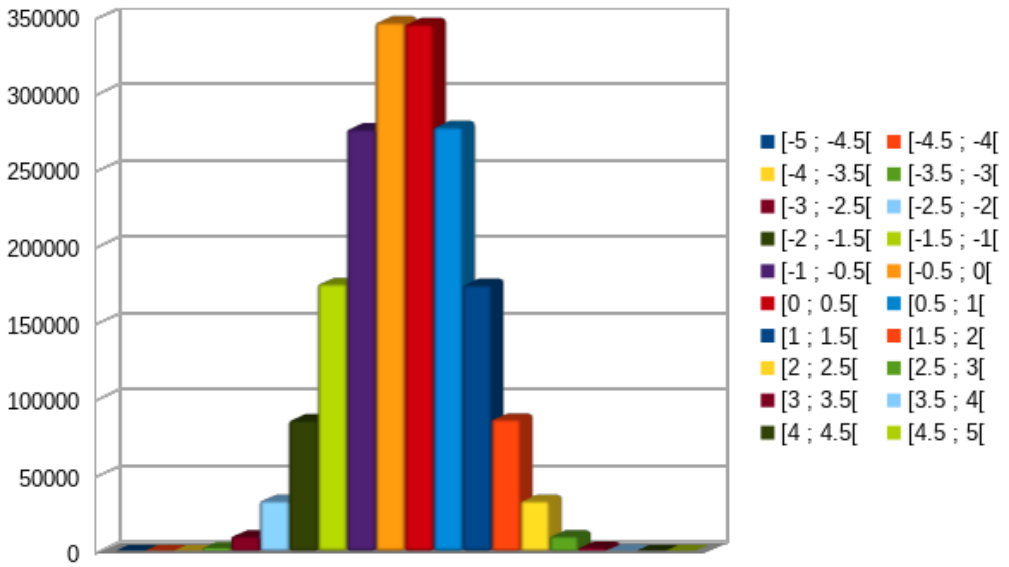


Figure 5: Box and Muller sur [-5; 5]

On constate à nouveau que les effectifs sur les intervalles sont répartit suivant une loi normale.

6 Bibliothèque `Box and Muller`

- **C/C++:** `https://github.com/lfarizav/pseudorandomnumbergenerators`
- **Java:** `https://github.com/mmiklavc/box-muller`