



Rapport TP Simulation

Lab #2 - Generation of Random Variates

BALLEJOS Lilian
ISIMA INP Deuxième année
Spécialité F2
Année Universitaire 2021/2022

Enseignant référant : Mr David HILL

Date du rendu : 17 octobre 2022

Date de la dernière modification : 16 octobre 2022

ISIMA INP

1 rue de la Chebarde - TSA 60125 - CS 60026 - 63178 Aubière CEDEX

Tel : 04 73 40 50 00

Site web : isima.fr

Table des Matières

1 Exercice 1 : Test des sorties des fonctions MT	2
2 Exercice 2 : Tirage aléatoire uniforme sur un intervalle	2
3 Exercice 3 : Reproduction de tirage aléatoire discret	3
3.1 Simulation discrète avec 3 classes A, B et C	3
3.2 Simulation discrète avec n classes	3
4 Exercice 4 : Reproduction de tirage aléatoire continu	4
4.1 Vérification de la moyenne	4
4.2 Vérification de la répartition des sorties	5
5 Exercice 5 : Simulation d'une loi non inversible	6
5.1 Reproduction de la loi normale avec des lancés de dés	6
5.2 Reproduction de la loi normale avec Box and Muller	6
6 Exercice 6 : Les librairies	7
7 Fonctions "Outil"	8

Introduction

Nous allons dans ce TP nous intéresser à la génération de nombre aléatoire uniforme entre $[0, 1]$ en s'appuyant sur les fonctions de génération de Matsumoto que vous pouvez retrouver [ici](#).

Nous allons voir qu'à partir de ces fonctions de génération nous pouvons approcher de manière empirique de nombreuses sorties aléatoire de loi connue (loi normale, loi exponentielle ...).

Quelques remarques sur le fichier .c

Les commentaires `/**/` sont voués à être supprimés Ils sont ici pour rendre l'exécution du main plus lisible et ne pas biaiser les tirages aléatoires en enchainant des tirages.

La compilation s'effectue comme ceci : `gcc -Wall -Wextra -g -o tp2_ballejos tp2_ballejos.c -lm`

Il faut bien penser à créer le répertoire **"sorti"** si vous souhaitez voir les sorties de certaines fonctions dans des fichiers.

1 Exercice 1 : Test des sorties des fonctions MT

Nous allons commencer par vérifier si le code que nous utilisons est portable. Pour cela nous possédons les sorties de code effectuer par Matsumoto sur sa machine qui se trouve dans le fichier "**mt19937ar.out**".

J'ai donc appelé le programme cette fois ci sur ma machine afin de tester si les sorties sont les même. Pour cela j'ai récupéré les sorties du programme à l'exécution dans un fichier **test.out** comme ceci :

```
1      ./mt19937ar > test.out
```

J'ai ensuite comparé le contenu des deux fichiers avec la commande bash **diff** qui affiche toutes les différences présentes entre deux fichiers.

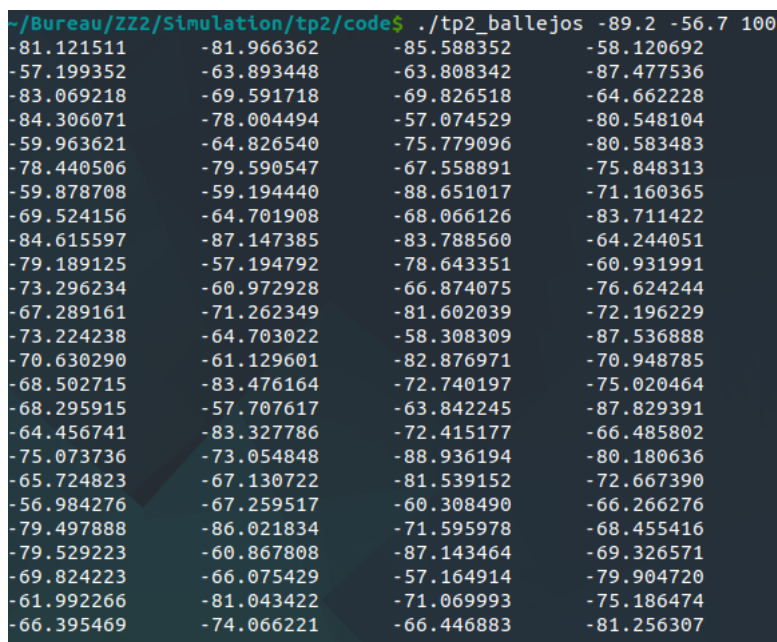
```
1      diff test.out mt19937ar.out
```

En l'occurrence, la sortie de mon diff est ici vide ce qui implique que j'ai obtenu les mêmes sorties que Matsumoto et que le code est bien portable sur différentes machines !

2 Exercice 2 : Tirage aléatoire uniforme sur un intervalle

Nous allons maintenant commencer à jouer avec les fonctions MT en tirant des nombres aléatoires uniformément dans un intervalle voulu. Pour cela nous utilisons (et cela jusqu'à la fin du TP) la fonction **genrand_real2()** qui génère des nombres uniformément entre $[0, 1]$ et nous changeons les bornes de l'intervalle en $[a, b]$ en faisant $a + (b - a) * \text{Random}$

J'ai testé la fonction avec des températures entre $-89,2^{\circ}\text{C}$ et $-56,7^{\circ}\text{C}$ sur 100 itérations et les sorties semblent très bien réparties comme nous pouvons le voir ici :



```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos -89.2 -56.7 100
-81.121511    -81.966362    -85.588352    -58.120692
-57.199352    -63.893448    -63.808342    -87.477536
-83.069218    -69.591718    -69.826518    -64.662228
-84.306071    -78.004494    -57.074529    -80.548104
-59.963621    -64.826540    -75.779096    -80.583483
-78.440506    -79.590547    -67.558891    -75.848313
-59.878708    -59.194440    -88.651017    -71.160365
-69.524156    -64.701908    -68.066126    -83.711422
-84.615597    -87.147385    -83.788560    -64.244051
-79.189125    -57.194792    -78.643351    -60.931991
-73.296234    -60.972928    -66.874075    -76.624244
-67.289161    -71.262349    -81.602039    -72.196229
-73.224238    -64.703022    -58.308309    -87.536888
-70.630290    -61.129601    -82.876971    -70.948785
-68.502715    -83.476164    -72.740197    -75.020464
-68.295915    -57.707617    -63.842245    -87.829391
-64.456741    -83.327786    -72.415177    -66.485802
-75.073736    -73.054848    -88.936194    -80.180636
-65.724823    -67.130722    -81.539152    -72.667390
-56.984276    -67.259517    -60.308490    -66.266276
-79.497888    -86.021834    -71.595978    -68.455416
-79.529223    -60.867808    -87.143464    -69.326571
-69.824223    -66.075429    -57.164914    -79.904720
-61.992266    -81.043422    -71.069993    -75.186474
-66.395469    -74.066221    -66.446883    -81.256307
```

FIGURE 1 – Sortie après 100 appels de la fonction $\text{uniform}(-89, 2, -56, 7)$

3 Exercice 3 : Reproduction de tirage aléatoire discret

Nous allons ici reproduire des distributions discrètes d'abord avec 3 classes puis après plus générale.

3.1 Simulation discrète avec 3 classes A, B et C

Pour commencer, j'ai implémenté **SimulDiscreteABC** qui construit le tableau des probabilités cumulées et observe pour un certain nombre d'itération si nous arrivons à retrouver empiriquement les valeurs attendues. Le résultat est concluant dès 1000 itérations comme nous pouvons le voir ici

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 1000
Sortie sur 1000 itérations:
Class A : Attendu 0.350000      Sortie:0.343000
Class B : Attendu 0.450000      Sortie:0.455000
Class C : Attendu 0.200000      Sortie:0.202000
```

FIGURE 2 – Sortie de ma fonction *SimulDiscreteABC* sur 1000 itérations

Plus nous augmentons le nombre de tirage plus le résultat empirique tend vers le résultat théorique! Ainsi pour 100 000 itérations la sortie est très convaincante.

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 100000
Sortie sur 100000 itérations:
Class A : Attendu 0.350000      Sortie:0.350770
Class B : Attendu 0.450000      Sortie:0.449900
Class C : Attendu 0.200000      Sortie:0.199330
```

FIGURE 3 – Sortie de ma fonction *SimulDiscreteABC* sur 100000 itérations

3.2 Simulation discrète avec n classes

Je me suis inspiré du dernier code que j'ai généralisé. Les différentes étapes de la fonction sont les suivantes : on remplit automatiquement le tableau des probabilités cumulées à l'aide d'un tableau **proba** passé en argument. On effectue ensuite n nombre de tirage dont le résultat est stocké dans un tableau **sorti**. Une fois les tirages terminés, on affiche les résultats.

On peut déjà voir que la fonction semble bien fonctionner car nous obtenons les mêmes résultats que la fonction précédente en recréant les 3 classes A, B et C.

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 100000
Sortie sur 100000 itérations:
Class A : Attendu 0.350000      Sortie:0.350770
Class B : Attendu 0.450000      Sortie:0.449900
Class C : Attendu 0.200000      Sortie:0.199330
```

FIGURE 4 – Sortie de ma fonction *SimulDiscreteGeneric* sur 100000 itérations

Maintenant, je me suis amusé à créer un exemple plus poussé pour tester la fonction avec le tableau de proba : $\{0.5, 0.3, 0.1, 0.05, 0.025, 0.025\}$. Le résultat est très concluant car on arrive à retrouver empiriquement quasiment les mêmes résultats que ceux théorique.

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 100000
Sortie sur 100000 itérations:
Class A : Attendu 0.500000      Sortie:0.499840
Class B : Attendu 0.300000      Sortie:0.300830
Class C : Attendu 0.100000      Sortie:0.100250
Class D : Attendu 0.050000      Sortie:0.048960
Class E : Attendu 0.025000      Sortie:0.025400
Class F : Attendu 0.025000      Sortie:0.024720
```

FIGURE 5 – Sortie de ma fonction *SimulDiscreteGeneric* avec 6 classes

Avec 1 000 000 de tirage le résultat empirique tend encore plus vers les probabilités théoriques

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 1000000
Sortie sur 1000000 itérations:
Class A : Attendu 0.500000      Sortie:0.499811
Class B : Attendu 0.300000      Sortie:0.300098
Class C : Attendu 0.100000      Sortie:0.100177
Class D : Attendu 0.050000      Sortie:0.049625
Class E : Attendu 0.025000      Sortie:0.025229
Class F : Attendu 0.025000      Sortie:0.025060
```

FIGURE 6 – Sortie de *SimulDiscreteGeneric* avec 6 classes et 1000000 de tirages

4 Exercice 4 : Reproduction de tirage aléatoire continu

Dans cette exercice nous allons reproduire une distribution cette fois-ci continue et cela en inversant la fonction exponentielle.

4.1 Vérification de la moyenne

Après avoir implémenté la fonction `negExp()` qui renvoie des valeurs suivant la fonction $-mean * \log(1 - Random)$, j'ai codé en premier temps **SimulNegExp** qui effectue n tirages aléatoires et vérifie que la moyenne théorique que l'on doit obtenir et la même que celle obtenu empiriquement.

Et voici les résultats pour une moyenne de 11.

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 11 1000
Moyenne sur 1000 itérations: 11.099904
```

FIGURE 7 – Sortie de *SimulNegExp* avec $mean = 11$ et 1000 tirages

Dès 1000 tirage le résultat et concluant mais voyons ce qu'il en est avec 1 000 000 de tirage.

```
~/Bureau/ZZ2/Simulation/tp2/code$ ./tp2_ballejos 11 1000000
Moyenne sur 1000000 itérations: 11.000462
```

FIGURE 8 – Sortie de *SimulNegExp* avec $mean = 11$ et 1 000 000 tirages

Le résultat est de plus en plus précis quand on augmente le nombre de tirage!

4.2 Vérification de la répartition des sorties

Nous allons maintenant regarder la répartition des sorties aléatoires.

Pour cela j'ai créé une fonction **DiscretDitribNegExp** qui effectue n tirage aléatoire avec la fonction **negExp()** et regarde la répartition des données sur 22 intervalles entre 0 et 22.

Regardons les sorties pour 1000 itérations et la courbe associée :

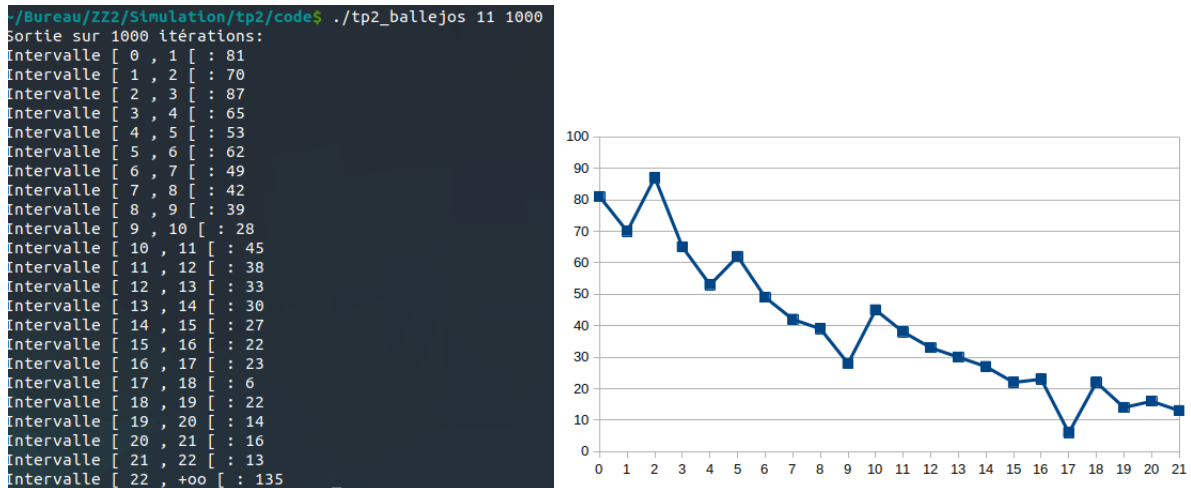


FIGURE 9 – Sortie de *DiscretDitribNegExp* avec $\text{mean} = 11$ et 1000 tirages

La courbe est encore bancale et nous n'arrivons pas à discerner la moyenne, le résultat n'est pas exploitable

Voyons maintenant avec 1 000 000 de tirage :

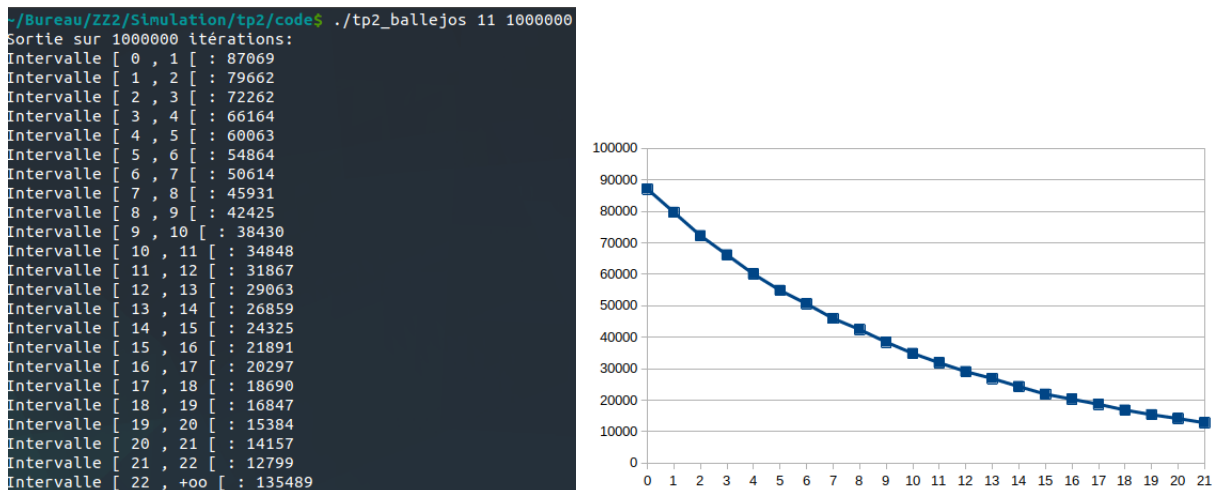


FIGURE 10 – Sortie de *DiscretDitribNegExp* avec $\text{mean} = 11$ et 1 000 000 tirages

Le résultat est déjà beaucoup plus intéressant, on distingue bien la courbe exponentielle négative ayant 11 pour moyenne.

5 Exercice 5 : Simulation d'une loi non inversible

Certaines lois n'ont aujourd'hui toujours pas été inversées et nous allons essayer ici de les reproduire empiriquement.

5.1 Reproduction de la loi normale avec des lancers de dés

Après avoir créé une fonction qui simule un lancer de dés (nombre aléatoire entre 1 et 6) j'ai créé la fonction **SimulDice** qui effectue 30 tirages de dés, additionne les sorties et cela n fois. Grâce à cette expérience nous allons réussir à approcher une courbe gaussienne entre 30 et 180 (qui correspond aux bornes min et max de notre expérience)

Regardons les sorties pour différents nombres de tirage.

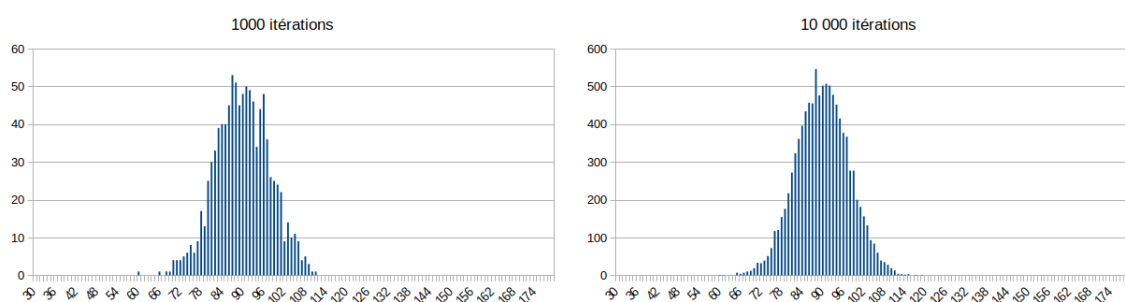


FIGURE 11 – Sortie de *SimulDice* avec 1000 et 10 000 tirages

Ici, on remarque qu'une gaussienne est en train de se créer mais celle-ci n'est pas parfaite. Voyons voir ce que cela donne avec plus d'itérations.

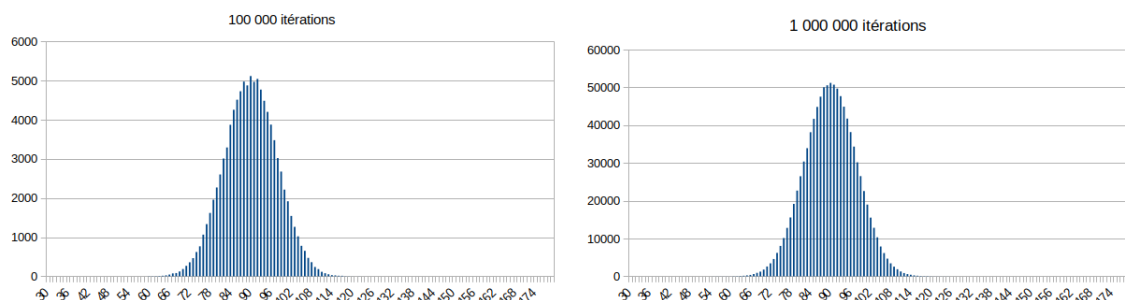


FIGURE 12 – Sortie de *SimulDice* avec 100 000 et 1 000 000 tirages

Avec 100 000 itérations, la courbe n'est pas encore totalement parfaite. Avec 1 000 000, on distingue une gaussienne parfaite. On a donc réussi avec cette expérience de lancer de dés à reproduire une gaussienne. Ceci dit le coût d'exécution est très cher et il faut beaucoup d'itération pour obtenir une très belle courbe.

Regardons maintenant si il n'est pas possible d'obtenir cette même courbe avec un moyen plus efficace.

5.2 Reproduction de la loi normale avec Box and Muller

Nous allons maintenant reproduire une gaussienne en utilisant le couple d'équation de Box and Muller qui permet à l'aide de deux nombres tirés aléatoirement d'obtenir 2 points de la gaussienne centrée en 0.

J'ai donc implémenté deux fonctions **BoxAndMullerX1** et **BoxAndMullerX2** qui renvoie le résultat des équations de Box and Muller en prenant en argument deux nombres supposés tirés aléatoirement.

$$x_1 = \cos(2\pi Rn_2)(-2 \ln(Rn_1))^{\frac{1}{2}} \quad (1)$$

$$x_2 = \sin(2\pi Rn_2)(-2 \ln(Rn_1))^{\frac{1}{2}} \quad (2)$$

FIGURE 13 – Les deux équations de Box and Muller

Ensuite j'ai créé la fonction **SimulBoxAndMuller** qui prend en argument un nombre d'itération. Elle va faire tirer deux nombres aléatoires et faire appel aux deux fonctions de Box and Muller afin de stocker ensuite les sorties.

Afin d'éviter un *if* à rallonge, j'ai écrit une petite ligne de code permettant de stocker les résultats sur des intervalles de 0.5 (par exemple $[-5, -4.5]$) dans un tableau de taille 20.

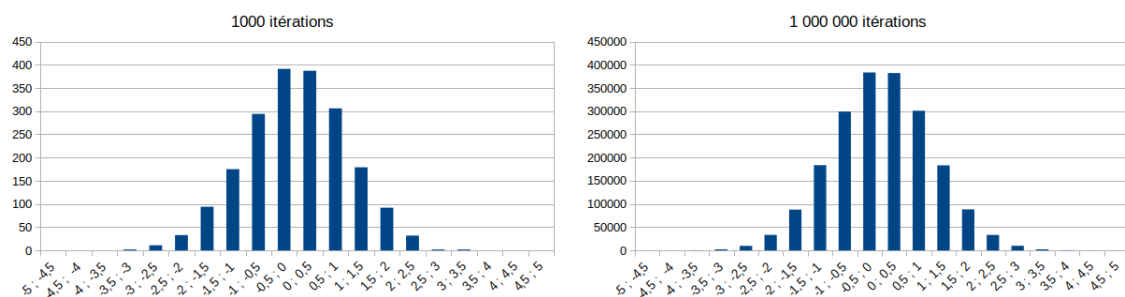
La voici :

```

1 // Réajustement sur un tableau de taille 20 entre 0 et 19 (* 10 + 50 / 5)
2 repartition[(int)((BoxAndMullerX1(u1, u2) * 10 + 50) / 5)]++;
3 repartition[(int)((BoxAndMullerX2(u1, u2) * 10 + 50) / 5)]++;

```

Au final voici la sortie obtenue tout d'abord avec seulement 1000 itérations puis avec 1 000 000.

FIGURE 14 – Sortie de *SimulBoxAndMuller* avec 1000 et 1 000 000 tirages

On remarque qu'avec 1000 itérations la gaussienne est quasi parfaite! On a donc trouvé une méthode bien plus efficace et rapide pour simuler empiriquement une courbe gaussienne.

J'ai tout de même aussi représenté la courbe pour 1 000 000 d'itérations et on remarque qu'à part sur le milieu de l'échantillon le résultat est quasiment le même.

6 Exercice 6 : Les librairies

En C/C++, la bibliothèque qui semble pouvoir effectuer tout ce que l'on a fait précédemment est **Boost Random Number Library Distributions**. Toutes les informations sont disponibles sur le site de la librairie que vous pouvez trouver en cliquant [ici](#).

En Java, **The Apache Commons Mathematics Library** semble posséder toutes les méthodes nécessaires permettant de faire ce que l'on a fait durant le TP. Cette bibliothèque possède par exemple la méthode *NormalDistribution*. Le site de la librairie se trouve [ici](#).

7 Fonctions "Outil"

Pour m'aider lors de mon travail j'ai codé deux fonctions annexes dont je vais prendre quelques lignes pour expliquer leur but.

La première est la fonction **exportData** qui reçoit un tableau ainsi que la taille de celui-ci et une chaîne de caractère. Elle permet d'exporter les données de sortie du tableau dans un fichier formaté qui est facilement manipulable sur Excel ou LibreOffice Calc.

La seconde est **afficheHistogramme** qui permet d'afficher dans mon terminal une pré visualisation très basique de l'histogramme des données stockés dans un tableau.

La fonction à l'aide de **valeurMax** va automatiquement trouver la donnée la plus grande afin de représenter proprement l'histogramme.

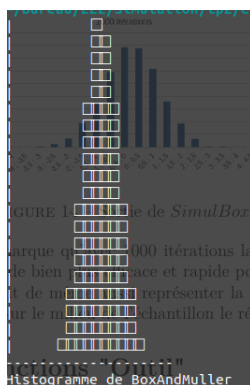


FIGURE 15 – Exemple d'histogramme généré par ma fonction

Table des figures

1	Sortie après 100 appels de la fonction <i>uniform</i> (−89, 2, −56, 7)	2
2	Sortie de ma fonction <i>SimulDiscreteABC</i> sur 1000 itérations	3
3	Sortie de ma fonction <i>SimulDiscreteABC</i> sur 100000 itérations	3
4	Sortie de ma fonction <i>SimulDiscreteGeneric</i> sur 100000 itérations	3
5	Sortie de ma fonction <i>SimulDiscreteGeneric</i> avec 6 classes	4
6	Sortie de <i>SimulDiscreteGeneric</i> avec 6 classes et 1000000 de tirages	4
7	Sortie de <i>SimulNegExp</i> avec mean = 11 et 1000 tirages	4
8	Sortie de <i>SimulNegExp</i> avec mean = 11 et 1 000 000 tirages	4
9	Sortie de <i>DiscretDitribNegExp</i> avec mean = 11 et 1000 tirages	5
10	Sortie de <i>DiscretDitribNegExp</i> avec mean = 11 et 1 000 000 tirages	5
11	Sortie de <i>SimulDice</i> avec 1000 et 10 000 tirages	6
12	Sortie de <i>SimulDice</i> avec 100 000 et 1 000 000 tirages	6
13	Les deux équations de Box and Muller	7
14	Sortie de <i>SimulBoxAndMuller</i> avec 1000 et 1 000 000 tirages	7
15	Exemple d'histogramme généré par ma fonction	8