Ao XIE

SIMULATION TP2

Génération de variables Aléatoires

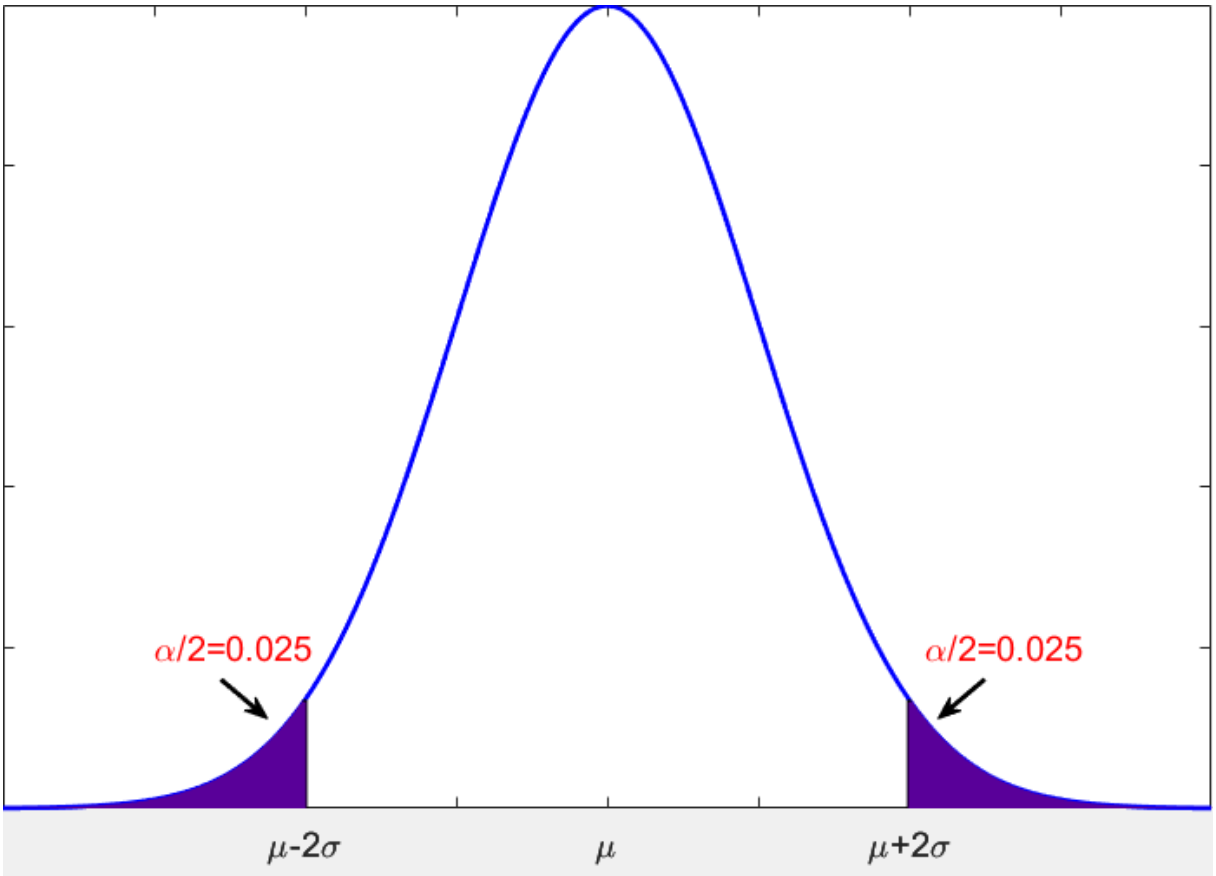

TABLE DES MATERES

TABLE DES FIGURES

TABLE DES EQUATIONS

Introduction

L'objectif de ce TP est de nous permettre de comprendre et d'apprendre à utiliser différents générateurs de nombres aléatoires. Dans ce TP, j'ai appris à écrire différents types de générateurs de nombres aléatoires en C et à générer des nombres aléatoires selon une distribution graduelle et normale.

1. La « page d’accueil de Matsumoto » et la « dernière implémentation en C » du Mersenne Twister (MT) original.

Cette section concerne le téléchargement d'un générateur de nombres aléatoires de haute qualité et son test. Par conséquent, les résultats générés dans le projet original sont les suivants :

```
1000 outputs of genrand_real2()
0.76275443 0.99000644 0.98670464 0.10143112 0.27933125
0.69867227 0.94218740 0.03427201 0.78842173 0.28180608
0.92179002 0.20785655 0.54534773 0.69644020 0.38107718
0.23978165 0.65286910 0.07514568 0.22765211 0.94872929
0.74557914 0.62664415 0.54708246 0.90959343 0.42043116
0.86334511 0.19189126 0.14718544 0.70259889 0.63426346
0.77408121 0.04531601 0.04605807 0.88595519 0.69398270
0.05377184 0.61711170 0.05565708 0.10133577 0.41500776
0.91810699 0.22320679 0.23353705 0.92871862 0.98897234
0.19786706 0.80558809 0.06961067 0.55840445 0.90479405
0.63288060 0.95009721 0.54948447 0.20645042 0.45000959
0.87050869 0.70806991 0.19406895 0.79286390 0.49332866
0.78483914 0.75145146 0.12341941 0.42030252 0.16728160
0.59906494 0.37575460 0.97815160 0.39815952 0.43595080
```

Figure 1 : Résultats générés par le générateur de nombres aléatoires MT.

```
1000 outputs of genrand_real2()
0.76275443 0.99000644 0.98670464 0.10143112 0.27933125
0.69867227 0.94218740 0.03427201 0.78842173 0.28180608
0.92179002 0.20785655 0.54534773 0.69644020 0.38107718
0.23978165 0.65286910 0.07514568 0.22765211 0.94872929
0.74557914 0.62664415 0.54708246 0.90959343 0.42043116
0.86334511 0.19189126 0.14718544 0.70259889 0.63426346
0.77408121 0.04531601 0.04605807 0.88595519 0.69398270
0.05377184 0.61711170 0.05565708 0.10133577 0.41500776
0.91810699 0.22320679 0.23353705 0.92871862 0.98897234
0.19786706 0.80558809 0.06961067 0.55840445 0.90479405
0.63288060 0.95009721 0.54948447 0.20645042 0.45000959
0.87050869 0.70806991 0.19406895 0.79286390 0.49332866
0.78483914 0.75145146 0.12341941 0.42030252 0.16728160
0.59906494 0.37575460 0.97815160 0.39815952 0.43595080
0.04952478 0.33917805 0.76509902 0.61034321 0.90654701
0.92915732 0.85365931 0.18812377 0.65913428 0.28814566
0.59476081 0.27835931 0.60722542 0.68310435 0.69387186
0.03699800 0.65897714 0.17527003 0.02889304 0.86777366
0.12352068 0.91439461 0.32022990 0.44445731 0.34903686
```

Figure 2 : Des nombres aléatoires MT après la compilation et l'exécution。

2. Génération de nombres aléatoires uniformes entre A et B.

La tâche de cette sous-section est de réécrire une fonction appelée "uniforme", en utilisant la fonction MT, dont le but est de générer un nombre aléatoire entre a et b. Le code est présenté dans la figure 3 :

```
/** ------------------------------------------------------------------------
 * @fn         uniform                                                     *
 * @brief      Randomly generate a pseudo-number between a_rangeMin and b_rangeMax.
 *                                                                         *
 * @param      a_rangeMin  Minimun value in the range                      *
 * @param      b_rangeMax  Maxmun value in the range                       *
 * @return     A pseudo-random number between a_rangeMin and b_rangeMax.
 *                                                                         *
 * @todo       It's for the question TWO                                   *
 * ------------------------------------------------------------------------ */
double uniform(double a_rangeMin, double b_rangeMax)
{
    return a_rangeMin + (b_rangeMax - a_rangeMin) * genrand_real2();
}
```

Figure 3 : Fonction uniforme

Ainsi, sur la base de cette fonction, nous pouvons mettre en œuvre une génération de température aléatoire entre -89,2 degrés Celsius et 56,7 degrés Celsius. Il s'agit de l'expérience dans laquelle un total de 50 températures aléatoires a été générées, comme le montre la figure.

-52.93°C	-2.23°C	-40.90°C	5.67°C	-17.80°C
-56.73°C	20.96°C	-46.06°C	-64.56°C	37.52°C
-72.99°C	-67.23°C	7.95°C	-68.62°C	11.03°C
50.32°C	-38.94°C	-29.26°C	-79.99°C	-32.74°C
54.46°C	55.02°C	42.43°C	-64.91°C	9.16°C
24.41°C	-50.36°C	45.50°C	22.83°C	-8.67°C
24.79°C	42.05°C	-86.74°C	-44.26°C	-55.09°C
-81.47°C	20.22°C	-8.22°C	54.48°C	-12.87°C
-61.68°C	-28.95°C	-0.87°C	-41.81°C	-17.48°C
-1.17°C	-50.52°C	20.78°C	37.70°C	20.77°C

Figure 4 : 50 températures générées aléatoirement.

3. Reproduction de distributions empiriques discrètes.

La tâche de cette sous-section est de simuler une distribution discrète d'une série aléatoire en utilisant la fonction MT.

a. Implémentez et testez

L'objectif de cette section est de générer des nombres aléatoires proportionnellement à leur taille. Pour garantir la réutilisabilité de la fonction, la proportion est définie comme l'entrée de la fonction. La fonction est illustrée dans la figure 5.

```
1.  /** ------------------------------------------------------------------------
    ------------------------------------------------------------ *
2.    * @fn         Ques3_A_numTest
                                                                  *
3.    * @brief      Install the three ratios entered to generate a pseudo-
    random number.                                                *
4.    * @param      testTimes   Number of random numbers generated.
                                                                  *
5.    * @param      ratio1ST    The first ratio
                                                                  *
6.    * @param      ratio2NE    The secong ratio
                                                                  *
7.    * @param      ratio3ND    THe third ratio
                                                                  *
8.    * @return     An address that points to the result
                                                                  *
9.    * @todo       It's for the question THREE PART A
                                                                  *
10.   * ------------------------------------------------------------------------
    ------------------------------------------------------------ */
11. int * Ques3_A_proportionPseudoNumber(int testTimes, int ratio1ST, int ratio2N
    D, int ratio3ND)
12. {
13.     /* Deckaring local variables */
14.     static int resultQues3[3] = {0};
15.     int sum = ratio1ST + ratio2ND + ratio3ND;
16.
17.
18.     for(int i=0; i<testTimes; i++)
19.     {
20.         /* Generate random numbers */
21.         int val = genrand_real2() * sum;    // For a random number between 0
    and sum
22.
23.         /* Categorized random numbers */
24.         if(val < ratio1ST){
25.             resultQues3[0] += 1;
26.         }
27.         else if(val >= (ratio1ST + ratio2ND)){  // Only retio3ND remians in t
    he range sum
28.             resultQues3[2] += 1;
29.         }
```

```
30.         else{
31.             resultQues3[1] += 1;
32.         }
33.     }
34.
35.     return resultQues3;
36. }
```

Figure 5 : Fonction de nombre aléatoire proportionnel

Le rapport utilisé dans le test était de 35 : 45 : 20 et certains des résultats sont présentés dans la figure 6.

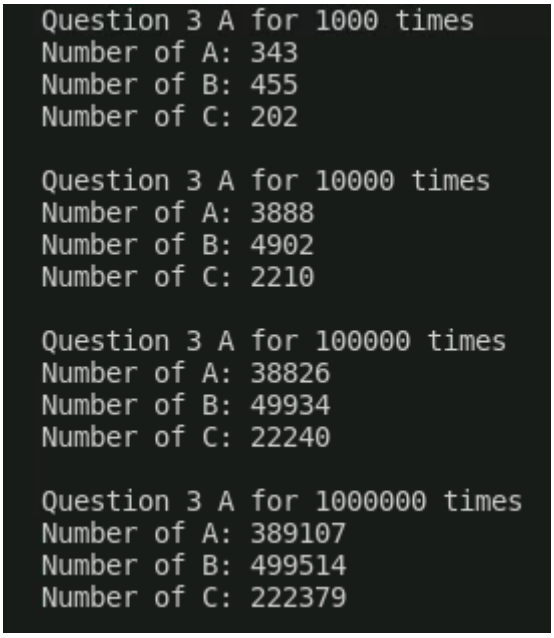

Figure 6 : Résultats partiels de la génération de nombres aléatoires à l'échelle

b. Implémenter une fonction plus générique.

L'objectif de cette section est de prendre les valeurs du tableau d'entrée et de les utiliser comme échelle, la fonction est donc présentée dans la figure 7.

```
1.  /** ------------------------------------------------------------------------
    ------------------------------------------------------- *
2.    * @fn         Ques3_B_HDL
                                                            *
3.    * @brief      Install the three ratios entered to generate a pseudo-
    random number.                                              *
4.    * @param      size   Number of random numbers generated.
                                                            *
5.    * @param      array    The first ratio
                                                            *
6.    * @return     An address that points to the result
                                                            *
7.    * @todo       It's for the question THREE PART B
                                                            *
8.    * ------------------------------------------------------------------------
    ------------------------------------------------------- */
9.  double * Ques3_B_HDL(int size, int *array)
10. {
11.     /* Deckaring local variables */
12.     static double resultQues3B[1000000] = {0};  // Use the first address of t
    he output to store the number of arrays
```

```
13.     double sum = 0;    // The sum of the numbers in the array
14.     int i = 0;
15.
16.     /* Calcul the sum */
17.     for(i=0; i<size; i++)
18.     {
19.         sum += array[i];
20.     }
21.
22.     /* Store the numbers of the array */
23.     resultQues3B[0] = i + 1;
24.
25.     /* Another method to calcul the sum */
26.     /*
27.     int i = 0;
28.     while(array[i] != '\0')
29.     {
30.          sum += array[i];
31.          i++;   // Number of the classes
32.     }
33.     */
34.
35.     /* Calculate the probability of each class*/
36.     for(int j=1; j<=size; j++)
37.     {
38.         resultQues3B[j] = array[j-1] / sum;
39.     }
40.
41.     return resultQues3B;
42.
43. }
```

Figure 7 : Générer des nombres aléatoires en proportion du tableau

Des valeurs issues du cours, des valeurs aléatoires artificielles et deux tableaux aléatoires générés par MT ont été utilisés dans cette partie du test et certains des résultats sont présentés dans la figure 8.

```
Result of question THREE PART B SUBPART A
There are 10 values in the array
The probability of category 1 is 0.12
The probability of category 2 is 0.08
The probability of category 3 is 0.08
The probability of category 4 is 0.04
The probability of category 5 is 0.02
The probability of category 6 is 0.02
The probability of category 7 is 0.10
The probability of category 8 is 0.31
The probability of category 9 is 0.10
The probability of category 10 is 0.12
Result of question THREE PART B SUBPART B
There are 6 values in the array
The probability of category 1 is 0.21
The probability of category 2 is 0.36
The probability of category 3 is 0.02
The probability of category 4 is 0.02
The probability of category 5 is 0.37
The probability of category 6 is 0.03
The probability of category 3785 is 0.000002
The probability of category 3786 is 0.000001
The probability of category 3787 is 0.000000
The probability of category 3788 is 0.000000
The probability of category 3789 is 0.000001
The probability of category 3790 is 0.000000
The probability of category 3791 is 0.000002
The probability of category 3792 is 0.000001
The probability of category 3793 is 0.000002
The probability of category 3794 is 0.000000
The probability of category 3795 is 0.000002
The probability of category 3796 is 0.000000
The probability of category 3797 is 0.000000
The probability of category 3798 is 0.000001
The probability of category 3799 is 0.000001
The probability of category 3800 is 0.000002
The probability of category 3801 is 0.000000
The probability of category 3802 is 0.000001
```

Figure 8 : Résultats partiels pour les valeurs aléatoires mises à l'échelle par des tableaux

4. Reproduction de distributions continues

La tâche de cette sous-section est d'obtenir une distribution continue de nombres aléatoires en inversant la fonction de distribution. Le résultat final généré est donc un graphique d'un nombre décroissant de nombres aléatoires.

L'équation 4-1 permet d'inverser un nombre aléatoire avec une distribution discrète pour obtenir un nombre aléatoire avec une distribution continue.

$$x = F^{-1} \qquad (4-1)$$

Par conséquent, des nombres aléatoires continus peuvent être générés à l'aide de la fonction de la figure 9

```
1.  /** ------------------------------------------------------------------------
    ------------------------------------------------------- *
2.    * @fn         negExp                                                    *
3.    * @brief      A random number generator with continuous distribution    *
4.    * @param      Mean    Generate the average of the random numbers        *
5.    * @return     Result of the generated random number.                    *
6.    * @todo       It's for the question FOUR                                *
7.    * ------------------------------------------------------------------------
    ------------------------------------------------------- */
8.   double negExp(double Mean)
9.   {
10.     return - Mean * log(1.0 - uniform(0, 1));
11.  }
```

Figure 9 : Fonction de nombre aléatoire continu

Un grand nombre de valeurs aléatoires sont générées pendant le test et leur valeur moyenne est calculée pour obtenir les résultats du test. La fréquence des nombres aléatoires dans chaque intervalle est également calculée et le résultat final obtenu est présenté dans la figure 10.

```
Finally, the average of these 1000 values is 11.099904
Finally, the average of these 1,000,000 values is 11.000834
Int the array Test22bin, values are:
81      70      87      65      53      62      49      42
     39      28      45      38      33      30      27
  2223       6      22      14      16      13
Int the array Test22bin, values are:
87090   79665   72242   66156   60068   54860   50612   459
41   42415   38449   34833   31869   29055   26853   24324
  21897      20294   18703   16841   15383   14154   12798
```

Figure 10 : Statistiques sur le nombre de nombres aléatoires dans chaque intervalle

5. Simulation de lois de distribution non réversibles

La tâche de cette sous-section est de générer des fonctions qui écrivent des fonctions capables de générer des nombres aléatoires normalement distribués.

a. Générer des nombres aléatoires normalement distribués en utilisant un algorithme de rejet.

L'idée centrale de l'algorithme de rejet est de générer des nombres aléatoires dans une plage cible, puis de comparer l'amplitude de la valeur du nombre aléatoire généré avec son résultat correspondant sur une distribution normale. Si la valeur est inférieure au résultat de la distribution normale, elle est retenue. Sinon, un nouveau numéro aléatoire est régénéré.

L'équation 5-1 mathématique de la fonction de distribution orthogonale est la suivante :

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma}} \qquad (5-1)$$

Il est donc écrit dans le programme comme deux fonctions comme la figure 11 :

```
1.  /** ------------------------------------------------------------------------
    ------------------------------------------------------------ *
2.    * @fn        normalDis                                                    *
3.    * @brief     Gaussian function calculator                                 *
4.    * @param     x        x in Gaussian function                              *
5.    * @param     mu                                                           *
6.    * @param     sita                                                         *
7.    * @return    Result of x in Gaussian function.                            *
8.    * @todo      Prepare for question 5                                       *
9.    * ------------------------------------------------------------------------
    ------------------------------------------------------------ */
10. double normalDis(double x, double mu, double sita)
11. {
12.    double Pi = acos(-1.0);
13.    double e = 2.71828182;
14.    double fx;
15.    fx  = (1 / (sqrt(2 * Pi) * sita)) * pow(e, -1 * (pow(x-
    mu, 2) / (2 * (pow(sita, 2)))));
```

```
16.
17.     return fx;
18.  }
```

Figure 11 : Fonction de distribution normale

L'algorithme dérivé de cette fonction de distribution normale est illustré à la figure 12.

```
1.   /** ------------------------------------------------------------------------
    ------------------------------------------------------------ *
2.     * @fn         rejectionAlgo                                              *
3.     * @brief      A Gaussian distributed random number generator.            *
4.     * @param      MinX    Minimun value of X                                 *
5.     * @param      MaxX    Maximum value of X                                 *


6.     * @param      MaxY    Maximum value of Y                                 *
7.     * @return     Result of the generated random number.                     *
8.     * @todo       It's for the question FIVE POINT ONE                       *
9.     * ------------------------------------------------------------------------
    ------------------------------------------------------------ */
10. double rejectionAlgo(double MinX, double MaxX, double MaxY)
11. {
12.     double x;
13.     double y;
14.     double naOne;
15.     double naTwo;
16.     double res;
17.     double realY;
18.
19.     do
20.     {
21.         naOne = uniform(0.0, 1.0);
22.         naTwo = uniform(0.0, 1.0);
23.         x = MinX + naOne * (MaxX - MinX);
24.         y = MaxY * naTwo;
25.         realY = normalDis(x, (MaxX - MinX) / 2, 20);
26.         if(y <= realY)
27.         ///((1 / (sqrt(2 * pi))) * pow(2.71828182, -1 * (pow(x, 2) / 2)))
28.         {
29.             res = x;
30.             break;
31.         }
32.     }while(1);
33.
34.     return res;
```

Figure 12 : algorithme de rejet

L'image finale de la distribution des dés et des points de dés générée par ces deux fonctions dans Matlab est illustrée à la figure 13.

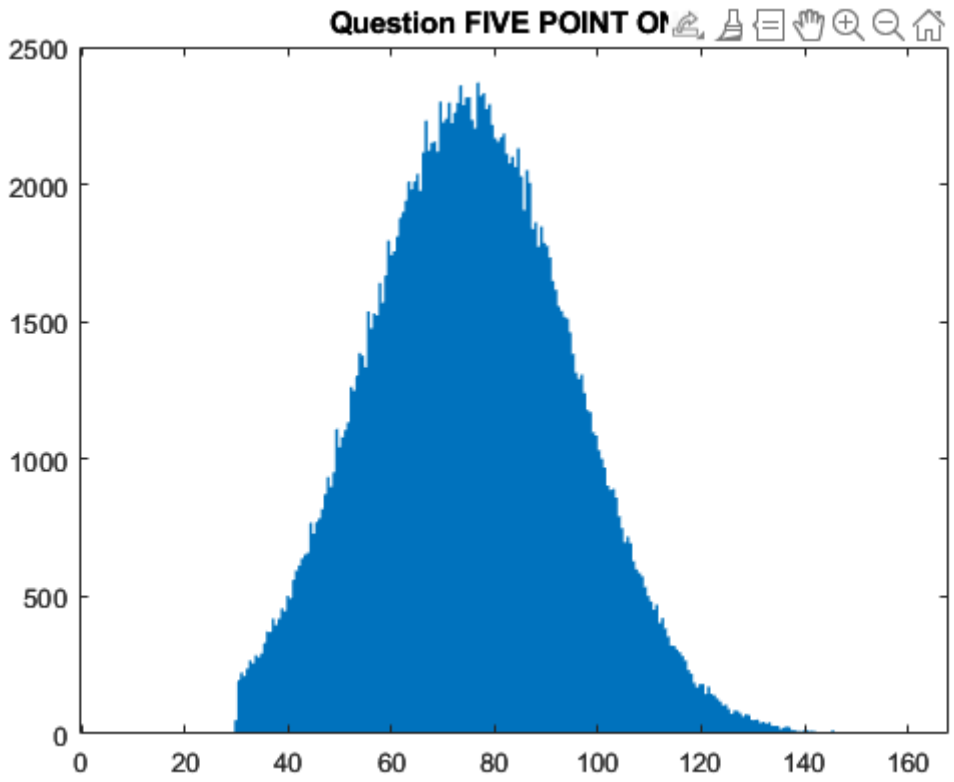

Figure 13 : Somme des lancers de dés

b. L'algorithme proposé par Box et Muller

Dans cette sous-section, l'utilisation du théorème central limite proposé par Box et Muller est utilisée pour générer des nombres aléatoires normalement distribués, et l'algorithme implémenté est illustré à la figure 14.

```
1.  /** ------------------------------------------------------------------------
    ------------------------------------------------------------ *
2.    * @fn         BoxMuller                                                  *
3.    * @brief      A high quality random number generator                     *
4.    * @param      mean    Generate the middle of the range                   *
5.    * @param      stdc    Generate range                                     *

6.    * @return     A random number                                            *
7.    * @todo       It's for the question FIVE PART TWO                        *
8.    * ------------------------------------------------------------------------
    ------------------------------------------------------------ */
9.  double BoxMuller(double mean, double stdc)
10. {
11.     double v1 = 0;
12.     double v2 = 0;
13.     double s = 0;
14.     int phase = 0;
15.     double x;
```

```
16.     if(phase == 0)
17.     {
18.         do
19.         {
20.             double u1 = genrand_real2() / RAND_MAX;
21.             double u2 = genrand_real2() / RAND_MAX;
22.
23.             v1 = 2 * u1 - 1;
24.             v2 = 2 * u2 - 1;
25.             s = v1 * v1 + v2 * v2;
26.         }while(s >= 1 || s ==0);
27.
28.         x = v1 * sqrt(-2 * log(s) / s);
29.     }
30.     else
31.     {
32.         x = v2 * sqrt(-2 * log(s) / s);
33.     }
34.
35.     phase = 1 - phase;
36.
37.     return mean + x * stdc;
38. }
```

Figure 14 : Le théorème central limite proposé par Box et Muller

Sous l'algorithme de ce théorème central limite, une série de nombres aléatoires est générée et tracée dans matlab, comme illustré à la figure 15.

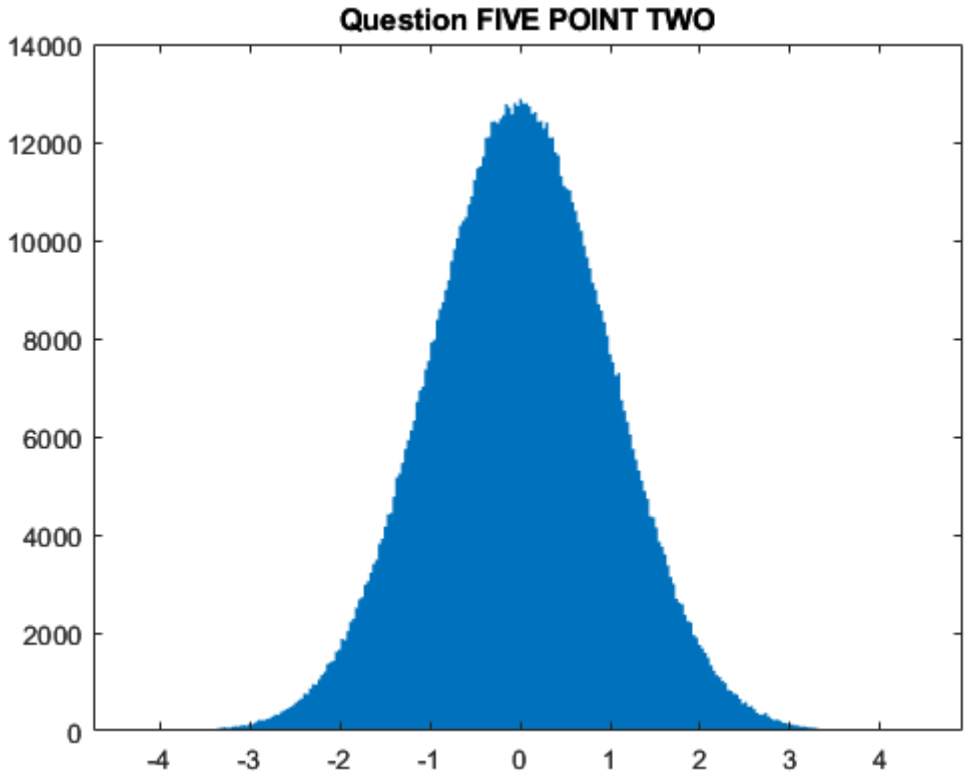

Figure 15 : Nombres aléatoires normalement distribués générés par l'algorithme de Box et Muller.

6. Les bibliothèques en C/C++ et Java qui génèrent des variables

La tâche de cette sous-section est d'introduire les fonctions de la bibliothèque de nombres aléatoires en C/C++ et Java

a. Les bibliothèques en C/C++

Dans les deux langages, les fonctions permettant de générer des nombres aléatoires sont incluses dans la bibliothèque « stdlib.h ». Ils sont définis comme suit :

```
1.  #include <stdlib.h>
2.
3.  int rand(void);
4.  void srand (unsigned int seed);
```

Figure 16 : Les bibliothèques en C/C++

int rand(void);

La valeur de retour de la fonction rand() est un nombre entier aléatoire compris entre 0 et RAND_MAX.

RAND_MAX est une variable macro dans le fichier d'en-tête « stdlib.h », mais sa valeur exacte n'est pas spécifiée dans le langage C/C++, seulement que sa valeur minimale est 32767.

Cette fonction, lorsqu'elle est appelée, génère des nombres aléatoires basés sur une "graine" et leur relation est une fonction de distribution normale. Notez toutefois que la graine n'est aléatoire qu'au moment du démarrage de l'ordinateur, mais qu'elle ne change pas une fois l'ordinateur démarré. Par conséquent, si l'on fait référence à cette fonction à plusieurs reprises dans un programme, on obtient la même valeur aléatoire.

void srand(unsigned int seed) ;

Cette fonction peut être utilisée pour changer la "graine" dans l'ordinateur, mais elle prend un argument. La manière habituelle de l'utiliser est la suivante :

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
```

```
3.  #include <time.h>
4.
5.  int main()
6.  {
7.          int a;
8.          srand((unsigned) time(NULL));
9.          a = rand();
10. }
```

Figure 17 : L'utilisation de srand() ;

Dans ce cas, la valeur aléatoire générée par rand va changer.

b. Les bibliothèques en Java

En Java, il existe deux façons de générer des nombres aléatoires. La première façon est d'utiliser la méthode random() de la classe Math, et la deuxième façon est d'utiliser la classe Random.

La méthode random() de la classe Math

La méthode random() de la classe mathématique n'a pas d'arguments et cette méthode génère un nombre à virgule flottante en double précision entre zéro et un.

La classe Random

La classe random est située dans le paquet java.util et cette classe a les deux constructeurs communs suivants :

- Random() : Ce constructeur utilise un nombre correspondant à l'heure actuelle du système comme nombre d'amorçage et utilise ensuite ce nombre d'amorçage pour construire l'objet aléatoire.
- Random(long seed) : Créer un nouveau générateur de nombres aléatoires avec un seul argument long.

Les utilisations courantes de la classe aléatoire sont énumérées dans Figure 7.

Méthode	Analyse
boolean nextBoolean()	Générer une valeur booléenne aléatoire, avec une probabilité égale de générer des valeurs vraies et fausses
double nextDouble()	Génère une valeur double aléatoire entre 0 et 1, avec 0 mais sans 1.0
int nextlnt()	Génère une valeur aléatoire int dans l'intervalle de int
int nextlnt(int n)	Génère une valeur aléatoire int entre 0 et n, avec 0 mais sans n
void setSeed(long seed)	Réinitialise le nombre de graines dans l'objet aléatoire
long nextLong()	Renvoie un nombre entier long aléatoire
boolean nextBoolean()	Renvoie une valeur booléenne aléatoire

float nextFloat()	Renvoie un nombre aléatoire à virgule flottante
double nextDouble()	Renvoie un nombre aléatoire en double précision à virgule flottante

Figure 18 : Les utilisations courantes de la classe Random