Projet informatique

Compte rendu - TP 1 et TP 2

Dans ce dossier, je vais vous présenter les divers exercices des TP 1 et 2 portant sur la simulation stochastique, la génération de nombres pseudo – aléatoires.
En effet, pour pouvoir copier un phénomène physique, de manière à le reproduire le plus exactement possible, on est amené à générer des nombres aléatoires qui pourrons ensuite faire fonctionner correctement les logiciels de simulation. Mais en réalité, cet « aléatoire » ne l'est pas vraiment. Il dépend principalement de formules mathématiques, parfois complexes. Nous verrons que recréer le hasard n'est pas chose facile, et qu'il existe une multitude de manière d'y arriver.

Code et rapport rédigé par :

CORCOS Ludovic
L2 Informatique
Université Clermont-Auvergne

Table des matières

I) TP 1 : Génération de nombres pseudo aléatoires

a. Générateur de type élévation au carré

i. Question 1

En 1946, John von Neumann propose un générateur pseudo-aléatoire connu sous le nom de la méthode middle-square (carré médian). Très simple, elle consiste à prendre un nombre, à l'élever au carré et à prendre les chiffres au milieu comme sortie. Celle-ci est utilisée comme graine pour l'itération suivante.
Toutefois, la période du middle-square est faible. La qualité des sorties dépend de la graine, « oooo » produit toujours la même séquence et constitue un « état absorbant » de l'algorithme.

ii. Question 2

Voici les deux fonctions utilisées pour pouvoir implémenter ce type de générateur :

```
1.  //  Renvoi les 4 chiffres au milieu d'un nombre de 8 chiffres
2.  int milieuNb(int nombre)
3.  {
4.      int nbFin = (nombre / 100) % 10000;
5.      return nbFin;
6.  }
7.
8.  //  Renvoi un nombre au carré
9.  int nbCarre(int nb_au_carre)
10. {
11.     return nb_au_carre * nb_au_carre;
12. }
```

En testant avec différents germes, on constate bien que la taille des cycles et des pseudocycles est très variable et dépend fortement de l'état initial.
Par exemple, on peut obtenir le type de résultat suivant avec N = 1301, où on ne constate pas d'état absorbant (en tout cas sur les 1 000 000 premières opérations)

```
N00 = 1301        1301 * 1301 = 01692601
N01 = 6926        6926 * 6926 = 47969476
N02 = 9694        9694 * 9694 = 93973636
N03 = 9736        9736 * 9736 = 94789696

...

N147 = 2100       2100 * 2100 = 04410000
N148 = 4100       4100 * 4100 = 16810000
N149 = 8100       8100 * 8100 = 65610000
```

Cependant, avec le germe N = 1234, on constate que le générateur s'écrase à partir de 54 itérations ; et même pire, avec le germe N = 4100, le générateur s'écrase au bout de 4 opérations seulement, on voit une boucle se former et se répéter.

b. Générateur congruentiel linéaire (LCG)

i. Question 3

Un générateur congruentiel linéaire est un générateur de nombres pseudo-aléatoires dont l'algorithme pour produire des nombres aléatoires, est basé sur des congruences et une fonction affine.
En effet, il repose sur la suite suivante :

$$X_{i+1} = (a \times X_i + c) mod\ \mathrm{m}$$

Avec : a = le multiplicateur
c = l'incrément
m = le modulo

D'ailleurs, on se rend bien compte de son fonctionnement lors de son implémentation :

```
1.  int LCG(int a, int c, int m, int result)
2.  {
3.      // Formule = N(k+1) = (a * N(k) + c ) mod m
4.      int result_fin = (a * result + c) % m;
5.      return result_fin;
6.  }
```

```
Iteration 0 = 003
Iteration 1 = 024
Iteration 2 = 171
Iteration 3 = 176

...

Iteration 350 = 091
Iteration 351 = 128
Iteration 352 = 131

...
```

Dans un contexte sur 8 bits, nous obtenons comme premières itérations :
Pour obtenir ces valeurs, on se place dans un contexte où
A = 7
C = 3
M = 256

Après quelques recherches sur internet, il semble que ces valeurs permettent de fournir un bon LCG sur 8 bits.

Passons dans un contexte sur 16 bits maintenant, avec les valeurs a = 7, c = 5 et m = 65 536.
Grâce à cela, on peut créer des valeurs aléatoires en 0 et 1 sur 16 bits.
Voici les résultats obtenus :

```
Nombre de valeurs sur [0.0 - 0.1[ = 10010 sur 100000 tirages
Nombre de valeurs sur [0.1 - 0.2[ = 9996 sur 100000 tirages
Nombre de valeurs sur [0.2 - 0.3[ = 9995 sur 100000 tirages
Nombre de valeurs sur [0.3 - 0.4[ = 9984 sur 100000 tirages
Nombre de valeurs sur [0.4 - 0.5[ = 9990 sur 100000 tirages
Nombre de valeurs sur [0.5 - 0.6[ = 10014 sur 100000 tirages
Nombre de valeurs sur [0.6 - 0.7[ = 10004 sur 100000 tirages
Nombre de valeurs sur [0.7 - 0.8[ = 9985 sur 100000 tirages
Nombre de valeurs sur [0.8 - 0.9[ = 10024 sur 100000 tirages
Nombre de valeurs sur [0.9 - 1.0[ = 9999 sur 100000 tirages
```

On peut qualifier ce générateur de bon générateur puisqu'il nous permet de générer des nombres aléatoires de manière homogène sur un intervalle.

Quelques valeurs pertinentes pour les LCG :

a	c	m
69069	0 ou 1	2^{32}
1664525	0	2^{32}
16807	0	$2^{31} - 1$
1103515245	12345	2^{31}
2875A2E7B175	0	2^{48}

D'autres exemples sont dans les commentaires du fichier.

Nous n'avons pas encore vu s'il y a, oui ou non un cycle pour ce type de générateur.
C'est pourquoi j'ai codé la fonction pour afficher la longueur d'un cycle :

```
1.  int LongueurCycle(int Iterations, int result_LCG_init)
2.  {
3.      int i = 0;
4.      int compteur = 0;
5.      int NewVal = 0;
6.
7.      int valeur = LCG(15, 9, BITS_16, result_LCG_init); //  Valeur de début de cycle

8.      printf("Valeur = %d\n\n", valeur);
9.
10.     for (i; i < Iterations; i++)
11.     {
12.         NewVal = LCG(15, 9, BITS_16, NewVal);
13.         printf("Itération %d = %d\n", i, NewVal);
14.         compteur++;
15.         if (i > 1 && valeur == NewVal)
16.         {
17.             return compteur - 2;
18.         }
19.     }
20. }
```

Ici, nous testons la longueur d'un cycle pour le LCG initialisé avec ses valeurs ligne 7.
La condition la plus importante est celle ligne 15 qui nous oblige à être supérieur à 1 (si ce n'étais pas le cas, le générateur renverrai toujours un et sa valeur.
On obtient donc en sorti de ceci :

```
Itération 8189 = 15728
Itération 8190 = 39321
Itération 8191 = 0
Itération 8192 = 9
La longueur d'un cycle avec ce LCG est de : 8191  (On a testé 1000000 itérations)
```

On essaie de le faire pas à pas en testant le plus d'itération possible et en espérant tomber sur un cycle.

c. Génération à base de registre bouclé

Un générateur à base de registre bouclé est initialement utilisé en électronique avec des portes logiques (XOR). Mais vu que l'électronique s'occupe de circuit binaire, on peut représenter le phénomène avec un programme.

La suite récurrente produite par un **LFSR** (de l'anglais linear feedback shift register) est nécessairement périodique à partir d'un certain rang. Les **LFSR** sont utilisés en cryptographie pour engendrer des suites de nombres pseudo-aléatoires. La fonction de rétroaction est alors choisie de façon à obtenir une période la plus grande possible.

```
1.  if (xor == 1)
2.  {
3.      new_nombre = (1 << 3) | new_nombre;
4.  }
5.  else
6.  {
7.      new_nombre = new_nombre & (~(1 << 3));
8.  }
```

```
Itération 0 =   1111
Itération 1 =   0111
Itération 2 =   0011
Itération 3 =   0001
Itération 4 =   1000
Itération 5 =   0100
Itération 6 =   0010
Itération 7 =   1001
Itération 8 =   1100
Itération 9 =   0110
Itération 10 =  1011
Itération 11 =  0101
Itération 12 =  1010
Itération 13 =  1101
Itération 14 =  1110
Itération 15 =  1111
```

Ici, lors du résultat du XOR avec les deux derniers bits, on change le premier bit, ce qu'il fait qu'on simule le « déplacement » du chiffre.

Au final, on retrouve bien les valeurs indiquées sur le diapo, avec un cycle de $2^4 - 1 = 15$

Il semblerait que les *hardware random number generator* (**HRNG**) or *true random number generator* (**TRNG**) soient des bons générateurs à base de LFSR

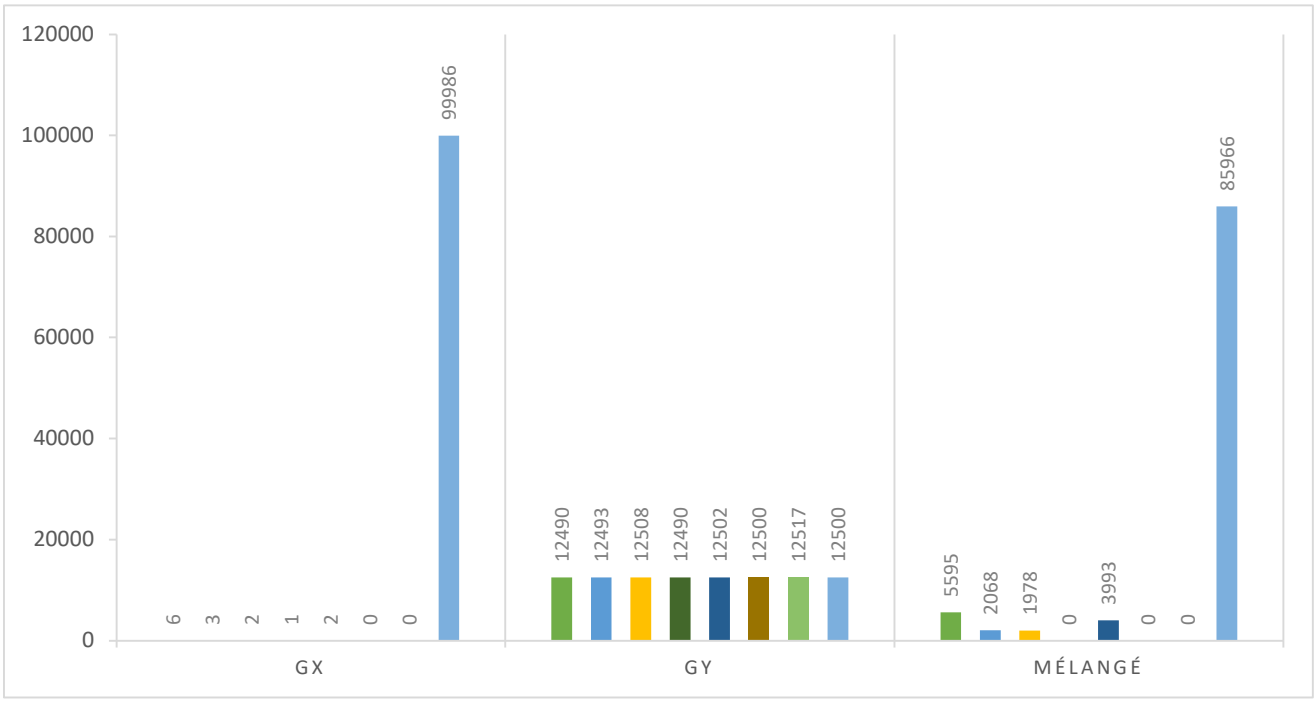
https://tel.archives-ouvertes.fr/tel-00757007v1/document

d. Brassage de générateurs

On a vu que, même s'il existe un cycle de répétition au sein des LCGs, ils peuvent avoir une bonne uniformité. On peut donc imaginer un programme qui prend deux LCGs et qui se sert de l'un pour mélanger l'autre. En effet, imaginons que nous avons un bon LCG que nous allons appeler GY et un autre, très mauvais que nous appelons GX, avec aucune uniformité. On peut alors se servir de GY pour mélanger GX et ainsi obtenir une meilleure uniformité que celle qui est donnée initialement avec GX.

Comme on peut le constater sur le graphique ci-dessous, on voit que l'on part d'un GX très mauvais et un bon GY (uniforme), et qu'au final, on se retrouve avec un nouveau LCG (certes, toujours pas uniforme) qui a évolué par rapport à l'état initial.
On peut alors imaginer faire ce genre d'algorithme avec plus de deux LCG, comme **l'algorithme de Wichmann – Hill** [1] où il est question de mettre en relation 3 LCG avec différents modulos.

[1] Est-ce vous ?



Résultats originaux en sorti d'algorithme :

```
Nombre de valeurs sur [0 - 8 192[ = 6 sur 100000 tirages
Nombre de valeurs sur [8 192 - 16 384[ = 3 sur 100000 tirages
Nombre de valeurs sur [16 384 - 24 576[ = 2 sur 100000 tirages
Nombre de valeurs sur [24 576 - 32 768[ = 1 sur 100000 tirages
Nombre de valeurs sur [32 768 - 40 960[ = 2 sur 100000 tirages
Nombre de valeurs sur [40 960 - 49 152[ = 0 sur 100000 tirages
Nombre de valeurs sur [49 152 - 57 344[ = 0 sur 100000 tirages
Nombre de valeurs sur [57 344 - 65 536[ = 99986 sur 100000 tirages


Resultat pour tab_GY :

Nombre de valeurs sur [0 - 8 192[ = 12490 sur 100000 tirages
Nombre de valeurs sur [8 192 - 16 384[ = 12493 sur 100000 tirages
Nombre de valeurs sur [16 384 - 24 576[ = 12508 sur 100000 tirages
Nombre de valeurs sur [24 576 - 32 768[ = 12490 sur 100000 tirages
Nombre de valeurs sur [32 768 - 40 960[ = 12502 sur 100000 tirages
Nombre de valeurs sur [40 960 - 49 152[ = 12500 sur 100000 tirages
Nombre de valeurs sur [49 152 - 57 344[ = 12517 sur 100000 tirages
Nombre de valeurs sur [57 344 - 65 536[ = 12500 sur 100000 tirages


Resultat pour uniforme :

Nombre de valeurs sur [0 - 8 192[ = 5995 sur 100000 tirages
Nombre de valeurs sur [8 192 - 16 384[ = 2068 sur 100000 tirages
Nombre de valeurs sur [16 384 - 24 576[ = 1978 sur 100000 tirages
Nombre de valeurs sur [24 576 - 32 768[ = 0 sur 100000 tirages
Nombre de valeurs sur [32 768 - 40 960[ = 3993 sur 100000 tirages
Nombre de valeurs sur [40 960 - 49 152[ = 0 sur 100000 tirages
Nombre de valeurs sur [49 152 - 57 344[ = 0 sur 100000 tirages
Nombre de valeurs sur [57 344 - 65 536[ = 85966 sur 100000 tirages
```

```
1.  for (i = 0; i < 100000; i++)
2.          {
3.
4.              alea = rand() % 100000;
5.              indice = tab_GY[alea] % 100;
6.
7.              result = tab_GX[indice];
8.              new_indice = tab_GY[alea] % 100;
9.
10.             new_result = tab_GX[new_indice];
11.
12.             tab_GX[indice] = new_result;
13.             tab_result[i] = result;
14.         }
```

Remarquez ici, on se sert en plus d'un générateur absolument pas fiable pour le calcul scientifique, le générateur rand de linux.
C'est lui qui nous permet de générer aléatoirement l'indice qui sera choisi pour effectuer les opérations.

e. Bibliothèques supplémentaires

Quelques possibles librairies où sont implémentés des générateurs aléatoires :

- Tina's Random Number Generator Library (TRNG) → C++
- Rng: Random numbers generation (libraire GNU)
- GNU scientific library (GSL)
- Et d'autres encore…

Quelques librairies fournissant des options de test de ces générateurs :

- RapidMiner (plateforme)
- GSL - The GNU Scientific Library
- TurboPower SysTools
- StatsLib
- Lib/statistics.py

II) TP 2 : Génération de variantes aléatoires

a. Génération uniforme entre A et B

Après avoir vu les générateur pseudo aléatoire avec une période assez courte, ici, on travaillera avec des générateurs scientifiques beaucoup plus précis, notamment celui de Makoto Matsumoto.
On importe la fonction suivante :

```
genrand_real2()
```

Elle permet de générer un nombre aléatoire entre 0 et 1.
Selon la formule vue en cours, on peut donc générer un ensemble de valeurs aléatoires réparti de manière uniforme entre A et B.

Voici un résultat obtenu (entre 0 et 1) :

```
[0,0 - 0,1[    99814
[0,1 - 0,2[    99284
[0,2 - 0,3[    100404
[0,3 - 0,4[    100779
[0,4 - 0,5[    99830
[0,5 - 0,6[    100257
[0,6 - 0,7[    99996
[0,7 - 0,8[    99871
[0,8 - 0,9[    99835
[0,9 - 1,0[    99930
```

Et l'implémentation de la fonction :

```
//  Permet de générer un nombre aléatoire entre borne_inf et borne_sup de manière uniforme
double uniform (double borne_inf, double borne_sup) {
    return (borne_inf + (borne_sup - borne_inf) * genrand_real2());
}
```

b. Reproduction d'une distribution empirique

i. Implémentation sur les 3 groupes proposés

Ici, le problème est assez facile à modéliser.

Comme vous pouvez le voir ci-contre, plus on va faire des essais, plus on va gagner en précision dans la répartition selon les groupes.

```
Valeurs pour 1000 essais :

Class 1 = 51 %
Class 2 = 9 %
Class 3 = 39 %


Valeurs pour 1000000 essais :

Class 1 = 50 %
Class 2 = 10 %
Class 3 = 39 %
```

ii. Implémentation d’une fonction plus générale

```c
1.  cumulative_tab[0] = tab_val_gr[0];
2.
3.      for (i = 0; i < taille_tab; i++) {
4.
5.          //  Calcul de la probabilité
6.          float proba = tab_val_gr[i] / nb_tests;
7.
8.          printf("Probabilité d'etre dans le groupe %d = %g, soit %d %%\n",
9.          i + 1, proba, (tab_val_gr[i] * 100) / (int)nb_tests);
10.
11.         cumulative_tab[i] = cumulative_tab[i - 1] + tab_val_gr[i];
12.     }
```

Pour la fonction cumulative, on crée un nouveau tableau qui sera renvoyé à la fin, et où seront intégrées les valeurs cumulées.
La ligne 11 nous indique comment se font les opérations d’addition avec la case précédente.

On obtient donc en sorti les résultats suivants :

```
Vérification des valeurs pour la fonction discrete_distributions :

Pour test_tab_simu_3classes (1 000 essais) :
Probabilité d'etre dans le groupe 1 = 0.517, soit 51 %
Probabilité d'etre dans le groupe 2 = 0.092, soit 9 %
Probabilité d'etre dans le groupe 3 = 0.391, soit 39 %

Pour test_tab_simu_3classes (1 000 000 essais) :
Probabilité d'etre dans le groupe 1 = 0.500079, soit 50 %
Probabilité d'etre dans le groupe 2 = 0.100293, soit 10 %
Probabilité d'etre dans le groupe 3 = 0.399628, soit 39 %

Affichage des histogrammes cumulatifs:

Pour test_tab_simu_3classes (1 000 essais)
[0]     █████████████████ 517
[1]     ████████████████████ 609
[2]     █████████████████████████████████ 1000

Pour test_tab_simu_3classes (1 000 000 essais)
[0]     █████████████████████████████████ 500079
[1]     ████████████████████████████████████████ 600372
[2]     ██████████████████████████████████████████████████████████████████ 1000000
```

Les résultats parlent d’eux-mêmes.

c. Reproduction d’une distribution continue

i. Fonction negExp

La fonction negExp se base sur la formule suivante :

$$x = -M \ln(1 - RND)$$

RND = nombre généré aléatoirement.

Voici son implémentation en C :

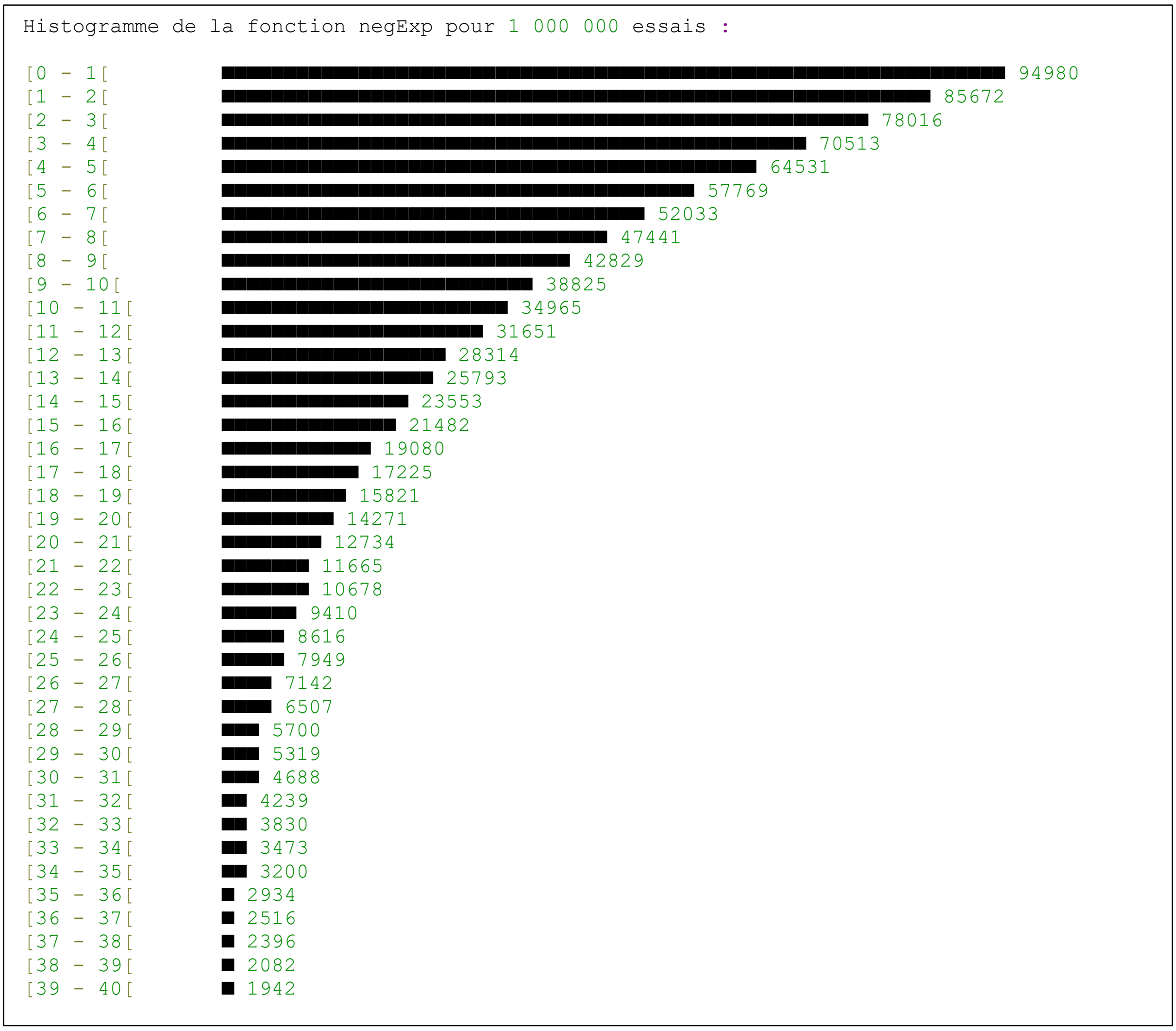
```c
//  Générateur selon une loi exponentielle
double negExp(int moyenne) {
    return - moyenne * log(1 - genrand_real2());

}
```

ii. Test de la fonction

Voici les résultats obtenus :

```
Moyenne réelle pour negExp à 1 000 valeurs = 9.34505

Moyenne réelle pour negExp à 1 000 000 valeurs = 9.64509

Histogramme de la fonction negExp pour 1 000 essais :

[0 - 1[      85
[1 - 2[      88
[2 - 3[      89
[3 - 4[      73
[4 - 5[      59
[5 - 6[      65
[6 - 7[      63
[7 - 8[      52
[8 - 9[      28
[9 - 10[     34
[10 - 11[    43
[11 - 12[    31
[12 - 13[    32
[13 - 14[    19
[14 - 15[    24
[15 - 16[    22
[16 - 17[    22
[17 - 18[    21
[18 - 19[    9
[19 - 20[    9

Histogramme de la fonction negExp pour 1 000 000 essais :

[0 - 1[      94980
[1 - 2[      85672
[2 - 3[      78016
[3 - 4[      70513
[4 - 5[      64531
[5 - 6[      57769
[6 - 7[      52033
[7 - 8[      47441
[8 - 9[      42829
[9 - 10[     38825
[10 - 11[    34965
[11 - 12[    31651
[12 - 13[    28314
[13 - 14[    25793
[14 - 15[    23553
[15 - 16[    21482
[16 - 17[    19080
[17 - 18[    17225
[18 - 19[    15821
[19 - 20[    14271
```

Le résultat est encore plus flagrant si on agrandit la taille du tableau de l'histogramme :



TADAM !! Tout ceci est généré dans le Shell grâce eux fonction présente dans histrogramme.c.

d. Simulation d'une loi de distribution non réversible (loi normale)

Les fonctions de Box et Muller nous permettent d'obtenir les résultats suivants :

```
Affichage des résultats :

Pour la fonction cosinu :

Avec 1 000 nombres générés :
[0]     ███████████████████████████████ 31
[1]     ████████████████████████████████████████████ 44
[2]     █████████████████████████████ 29
[3]     ███████████████████████████████████████ 39
[4]     ███████████████████████████████ 31
[5]     ████████████████████████████████ 32
[6]     ██████████████████████ 22
[7]     █████████████████████████████████████████████ 45
[8]     ████████████████████████████████████████ 40
[9]     ███████████████████████████████ 31
[10]    █████████████████████████████████████ 37
[11]    ██████████████████████████████████ 34
[12]    ██████████████████████████████████████████ 42
[13]    █████████████████████████████████████████ 41
[14]    █████████████████████████████████ 33
[15]    ████████████████████████████ 28
[16]    ██████████████████████████████████ 34
[17]    ████████████████████████████████████ 36
[18]    █████████████████████████████ 29
[19]    ██████████████████████ 22

Avec 1 000 000 nombres générés :
[0]     █████████████████████████ 25556
[1]     ███████████████████████████ 27960
[2]     ██████████████████████████████ 30242
[3]     ████████████████████████████████ 32282
[4]     ██████████████████████████████████ 34248
[5]     ███████████████████████████████████ 35959
[6]     █████████████████████████████████████ 37948
[7]     ██████████████████████████████████████ 38196
[8]     ███████████████████████████████████████ 39593
[9]     ███████████████████████████████████████ 39987
[10]    ███████████████████████████████████████ 39874
[11]    ███████████████████████████████████████ 39550
[12]    ██████████████████████████████████████ 38736
[13]    █████████████████████████████████████ 37403
[14]    ████████████████████████████████████ 36033
[15]    █████████████████████████████████ 33757
[16]    ████████████████████████████████ 32101
[17]    ██████████████████████████████ 30259
[18]    ████████████████████████████ 28028
[19]    █████████████████████████ 25689
```

On voit très clairement que plus on augmente le nombre de nombres générés, plus on obtient une distribution gaussienne. Un essai a été effectué pour n = 100 000 000 000, mais ce fut un échec, mon ordinateur ne supportant pas ce genre de valeurs.

ANNEXES