BEAUMONT Hélène F2
JEAN Nicolas

Compte rendu du TP 2
Simulation

Tp sur la génération de nombres pseudo-aléatoires avec des distributions non uniformes

Nombres aléatoires uniformément repartis entre A et B

```
#define A 10
#define B 100
#define NBITER 10

int main(int argc, char ** argv)
{
   /* Declarations */
   int      i;
   double   nb;

   srand48( time( NULL ) );

   /* Calcul et affichage des NBITER nombres aleatoires créés */
   for( i = 0 ; i < NBITER ; i++ )
   {
      nb = A + drand48() * (B - A);
      printf("Nombre %d : %f\n", i, nb);
   }

   return 0;
}
```

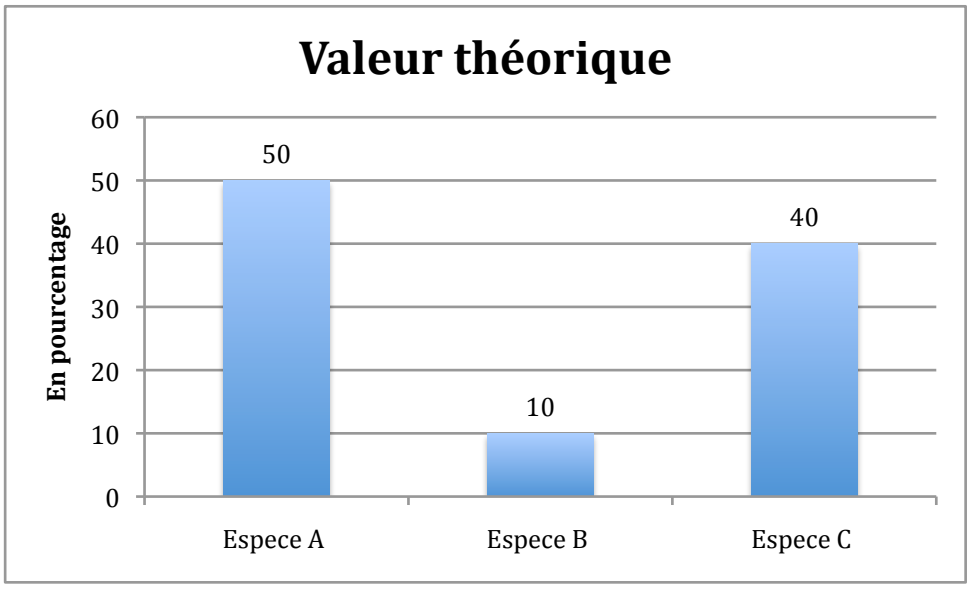
Reproduction de distributions discrètes

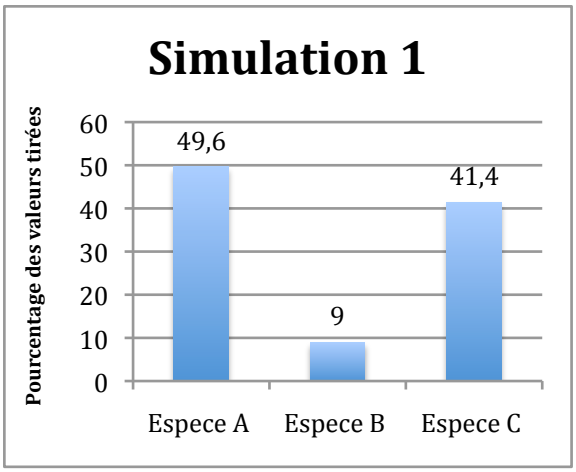
Dans un premier temps, nous allons, à partir d'une table cumulant les répartitions, simuler avec un générateur uniforme la répartition des valeurs selon cette table.

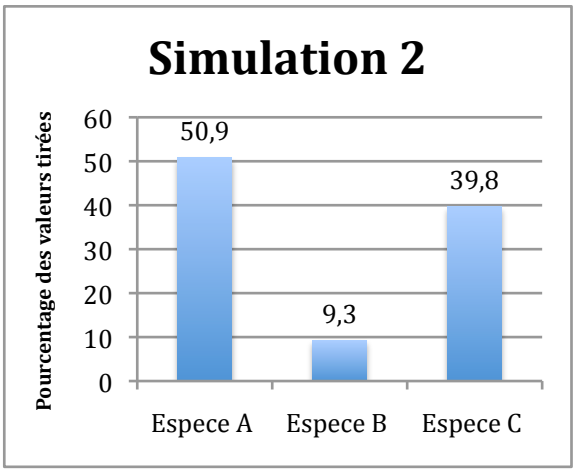
```
/*-------------------------------------------------------------------------*
 * place :  fonction qui retourne l'indice de la classe a laquelle il   *
 *          appartient                                                     *
 *                                                                         *
 * En entree :                                                             *
 *      - tableRep : la table de repartition                               *
 *      - nb : un nombre entre 0 et 1                                      *
 *                                                                         *
 * En sortie : l'indice de la classe correspondante                        *
 *-------------------------------------------------------------------------*/
int place( double * tableRep, double nb )
{
```

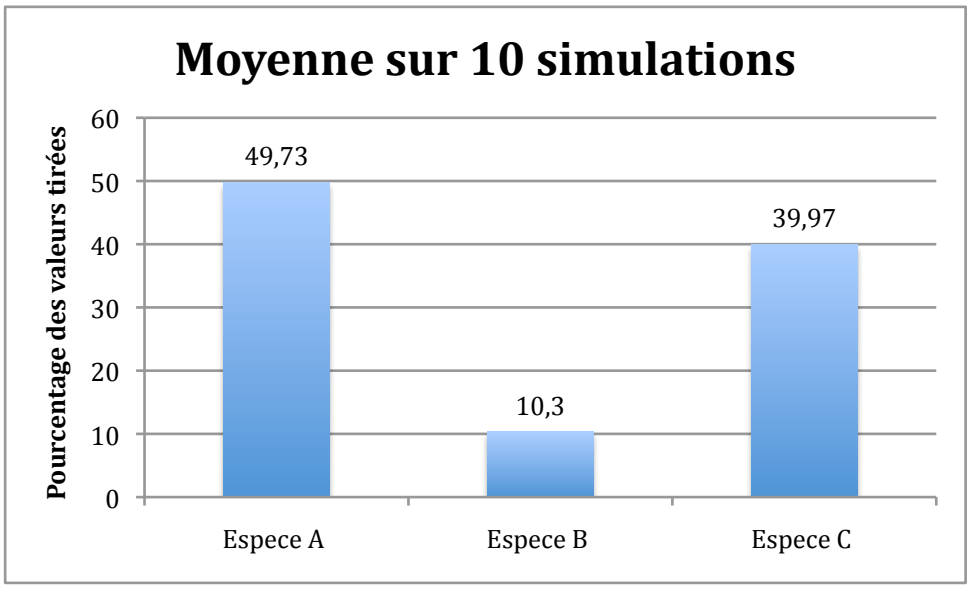
```
   int   i = 0;

   while( i < TAILLE && tableRep[ i++ ] < nb );

   return --i;
}
```

```
#define NBTIRAGE    1000
#define TAILLE      3

int main(int argc, char ** argv)
{
   char         nom[TAILLE];        /* Nom des evenements */
   double       tabRepI[TAILLE],    /* Table de repartition initiale */
                tabEff[TAILLE] ;        /* Table des effectifs */
   double       nb;
   int          i;

   /* Initialisation du germe de drand48 */
   srand48(time(NULL));

   /* Initialisation des noms des evenements */
   nom[ 0 ] = 'A';
   nom[ 1 ] = 'B';
   nom[ 2 ] = 'C';

   /* Initialisation de la table de repartition initiale */
   tabRepI[ 0 ] = 0.5;
   tabRepI[ 1 ] = 0.6;
   tabRepI[ 2 ] = 1.0;

   /* Initialisation du tableau receuillant les effectifs */
   for( i = 0 ; i < TAILLE ; i++ )
   {
       tabEff[ i ] = 0;
   }

   /* Tirage aleatoire */
   for( i = 0 ; i < NBTIRAGE ; i++ )
   {
      nb = drand48();
      tabEff[ place( tabRepI, nb ) ]++;
   }

   /* Affichage de la repartition */
   for( i = 0 ; i < TAILLE ; i++ )
   {
     printf( "Espece %c : %f %% \n", nom[ i ], tabEff[ i ] / NBTIRAGE * 100 );
   }
   printf( "\n" );

   return 0;
}
```









D'après ces simulations, nous obtenons un résultat en accord à la table cumulant les répartitions.

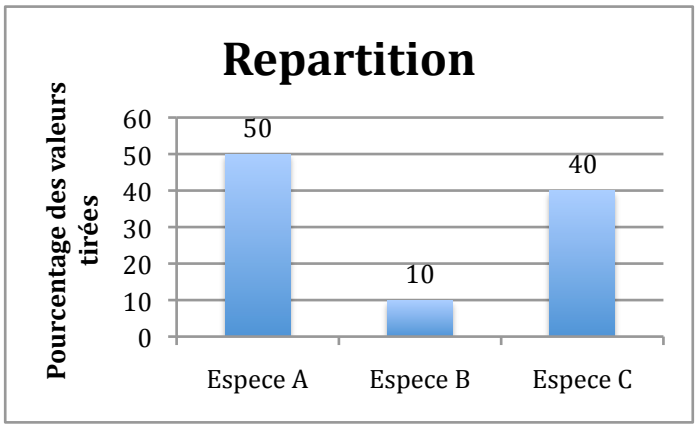
Maintenant, à partir d'un échantillon, nous allons déduire la table des répartitions cumulées, et à partir de celle-ci réaliser une simulation pour s'assurer que notre résultat est correct.

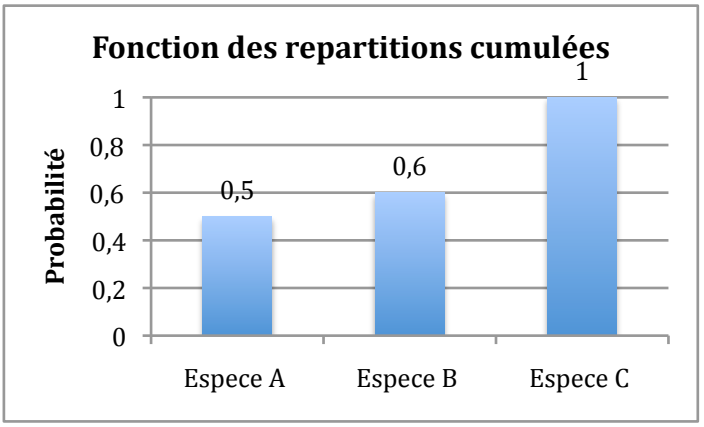
```
/*-------------------------------------------------------------------------*
 * proba :  Fonction qui retourne la table de repartitions sous forme     *
 *          de probabilite                                                 *
 *                                                                         *
 * En entree :                                                             *
 *      - tabEffectif : la table ou l'on trouve les effectifs de chaque    *
 *          classe ;                                                       *
 *      - taille : entier qui indique le nombre de classes                 *
 *                                                                         *
 * En sortie : adresse de la table                                         *
 *-------------------------------------------------------------------------*/
double * proba ( double * tabEffectif, int taille )
{
   double       tailleEch;
   int          i;

   /* Calcul du nombre d'elements */
   for( i = 0 ; i < taille ; tailleEch += tabEffectif[ i++ ]);

   /* Calcul des probalites */
   for( i = 0 ; i < taille ; tabEffectif[ i++ ] /= tailleEch );

   return tabEffectif;
}

/*-------------------------------------------------------------------------*
* repartition : Fonction qui retourne la table cumulant les               *
*               repartitions                                              *
*                                                                         *
* En entree :                                                             *
*       - tabEffectif : la table ou l'on trouve les effectifs de chaque   *
*           classe ;                                                      *
*       - taille : entier qui indique le nombre de classes                *
*                                                                         *
* En sortie : adresse de la table                                         *
*-------------------------------------------------------------------------*/
double * repartition ( double * tabEffectif, int taille )
{
   int          i;

   /* Realisation de la table de probabilite */
   tabEffectif = proba(tabEffectif, taille);

   /* Realisation de la table de probabilite cumullee */
   for( i = 1 ; i < taille ; i++ )
   {
      tabEffectif[i] += tabEffectif[ i - 1 ];
   }
```

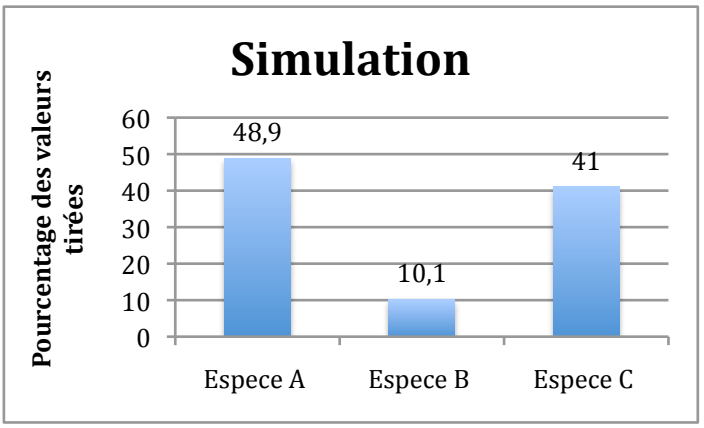
```
   return tabEffectif;
}
```

```
#define TAILLE      3
#define NBTIRAGE    1000

int main(int argc, char ** argv)
{
   char         nom[TAILLE];        /* Nom des evenements */
   double       tabEffI[TAILLE],    /* Table de repartition initiale */
              * tabRep,             /* Table de repartition a partir de la
                                      table de repartition initiale */
                resultat[ TAILLE ]; /* Table de repartition */
   double       nb;
   int          i;

   /* Initialisation du germe de drand48 */
   srand48(time(NULL));

   /* Initialisation des noms des evenements */
   nom[ 0 ] = 'A';
   nom[ 1 ] = 'B';
   nom[ 2 ] = 'C';

   /* Initialisation de la table de repartition initiale */
   tabEffI[ 0 ] = 50;
   tabEffI[ 1 ] = 10;
   tabEffI[ 2 ] = 40;

   /* Affectation de la table de repartition */
   tabRep = repartition ( tabEffI, TAILLE );

   /* Affichage de la table des effectifs */
   printf( "table de repartition : \n" );
   for ( i = 0 ; i < TAILLE ; i++ )
   {
      printf( "Espece %c : %f\n", nom[ i ], tabRep[ i ] );
   }
   printf( "\n" );

   /* Initialisation du tableau recueillant les effectifs */
   for( i = 0 ; i < TAILLE ; i++ )
   {
       resultat[ i ] = 0;
   }

   /* Tirage aleatoire */
   for( i = 0 ; i < NBTIRAGE ; i++ )
   {
      nb = drand48();
      resultat[ place( tabRep, nb ) ] ++ ;
   }
```

```
    for ( i = 0 ; i < TAILLE ; i++ )
    {
        printf( "Espece %c : %f %% \n", nom[ i ], resultat[ i ] / NBTIRAGE * 100 );
    }
    printf( "\n" );

    return 0;
}
```







D'après ces graphiques, on obtient la table des répartitions cumulées escomptée, et la simulation concorde également.

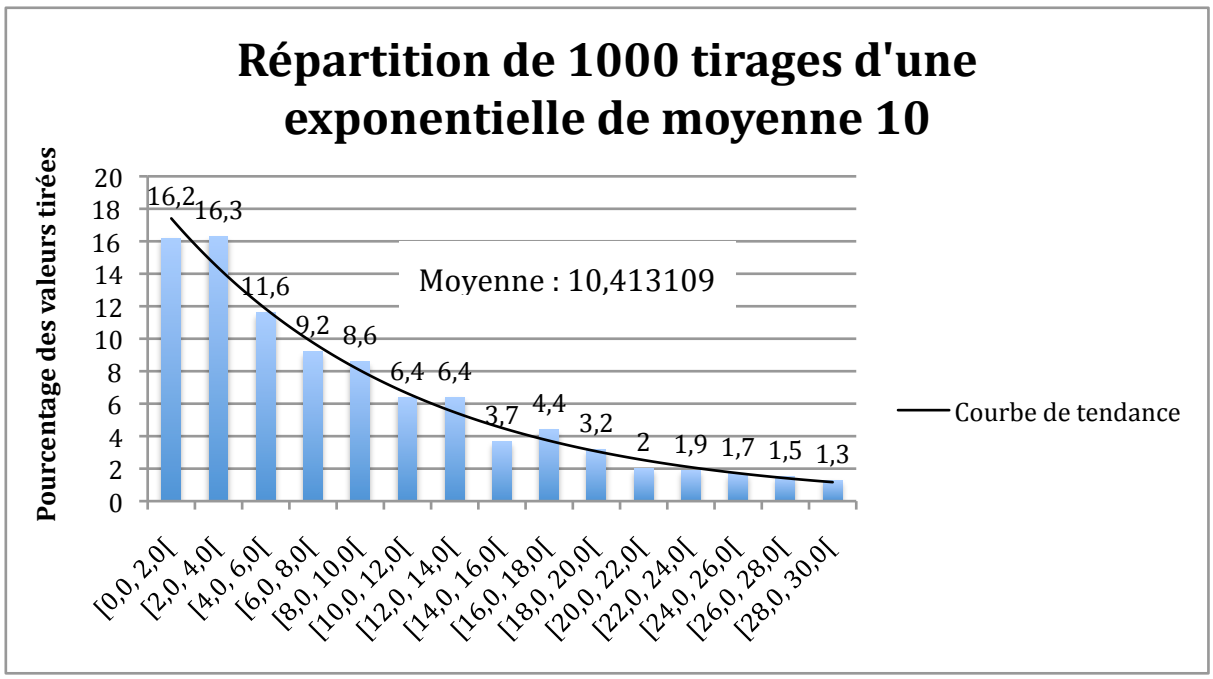
Distribution exponentielle négative de moyenne 10

Nous intéressons maintenant à une loi exponentielle négative. Cette fonction est bijective, donc inversible. Nous allons donc appliquer une *génération par loi inverse,* également connue sous le nom d'*anamorphose*.

```
/*-------------------------------------------------------------------------*
 * expo :   Fonction qui, a une image retourne le nombre qui l'a genere *
 *                                                                         *
 * En entree :                                                             *
 *      - nb : le nombre dont on veut l'antecedant                         *
 *      - moyenne : la valeur moyenne de la fonction exponentielle         *
 *                                                                         *
 * En sortie : l'antecedant de nombre                                      *
 *-------------------------------------------------------------------------*/
double expo(double nb, double moyenne)
{
   double      res;

   /* Calcul de l'antecedant */
   res = - moyenne * log(1 - nb);

   return res;
}
```

```
#define NBTIRAGE 1000
#define MOYENNE 10
#define NBINTERVALLE 15


int main(int argc, char ** argv)
{
  double        nbTire,
                res,
                moyenne = 0;
  double        histo[NBINTERVALLE];
  int           i;

  /* Initialisation du germe de drand48 */
  srand48( time( NULL ) );

  /* Initialisation de l'histogramme */
  for( i = 0 ; i < NBINTERVALLE ; histo[i++] = 0 );

  /* Remplissage de l'histogramme */
  for( i = 0 ; i < NBTIRAGE ; i++ )
  {
    /* Tirage du nombre aleatoire */
    nbTire = drand48();

    /* Calculd e l'antecedent */
    res = expo( nbTire, MOYENNE );
```

```
    /* Verification de la moyenne */
    moyenne += ( res / NBTIRAGE );

    if( res < NBINTERVALLE * 2 )
    {
      histo[ (int) res / 2 ]++;
    }
  }

  /* Affichage */
  for( i = 0 ; i < NBINTERVALLE ; i++ )
  {
    printf( "[%0.1f, %0.1f[ : %f %%\n", (double) i * 2, (double) (i + 1) *
2, histo[i] / NBTIRAGE * 100 );
  }
  printf( "Moyenne obtenue : %f\n", moyenne );

  return 0;
}
```



D'après le graphique obtenu, on note que les valeurs sont reparties selon une exponentielle de moyenne 10.

Loi normale de moyenne 10 et d'écart-type 3

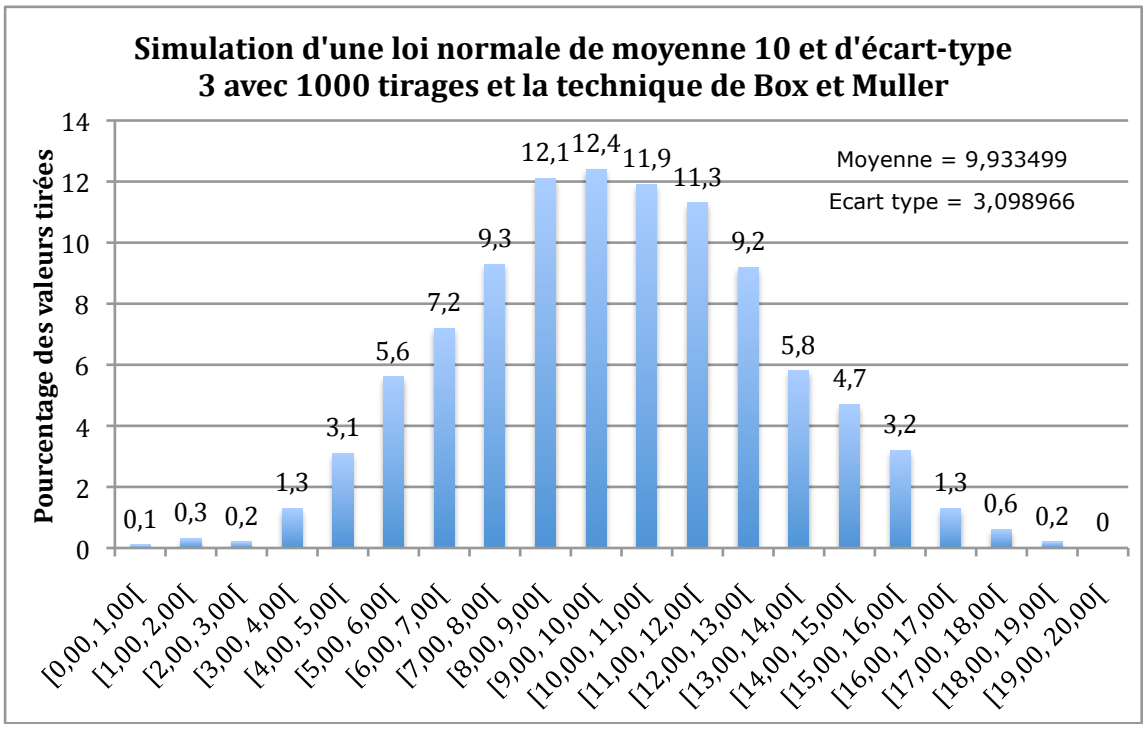
Cette loi n'est pas inversible. On ne peut donc pas utiliser la méthode précédente pour la représenter. Nous allons donc utiliser la méthode de Box et Muller.

```
#define PI 3.141592

/*-------------------------------------------------------------------------*
 * paireNbAlea :    Retourne deux nombres aleatoires suivant la methode *
 *                  de Box et Muller                                       *
 *                                                                         *
 * En entree :                                                             *
 *      - x : Un tableau de deux nombres aleatoires                        *
 *                                                                         *
 * En sortie : Un tableau contenant les deux nombres crees                 *
 *-------------------------------------------------------------------------*/
double * paireNbAlea( double * x )
{
   double       na1,
                na2;

   /* Tirages aleatoires */
   na1 = drand48();
   na2 = drand48();

   /* Calcul pour loi N(0,1) */
   x[ 0 ] = cos( 2 * PI * na2 ) * sqrt( - 2 * log( na1 ) );
   x[ 1 ] = sin( 2 * PI * na2 ) * sqrt( - 2 * log( na1 ) );

   /* Calcul pour une loi N(10,3) */
   x[ 0 ] *= 3;
   x[ 0 ] += 10;

   x[ 1 ] *= 3;
   x[ 1 ] += 10;

   return x;
}

/*-------------------------------------------------------------------------*
 * distribution :   Place dans le tableau de distribution les nombres   *
 *                  obtenus grace a Box et Muller                          *
 *                                                                         *
 * En entree :                                                             *
 *      - distrib : Tableau contenant la distribution                      *
 *      - nbIntervalle : Nombre d'intervalle, ie de cases de distrib       *
 *      - intervalle : Longueur des intervalles                            *
 *      - stat : Contient la moyenne et l'ecart type                       *
 *      - nbTirage : Nombre de tirages effectues (necessaire pour le       *
 *          calcul de la moyenne et de l'ecart-type                        *
 *-------------------------------------------------------------------------*/
void distribution( int * distrib, int nbIntervalle, double intervalle,
double * stat, int nbTirage )
```

```
{
   double       x[ 2 ];
   int          i;

   /* Creation de deux nombres avec la methode de Box et Muller */
   paireNbAlea( x );

   /* Calcul de la moyenne */
   stat[ 0 ] += ( ( x[ 0 ] + x[ 1 ] ) / nbTirage );

   /* Calcul de l'ecart-type */
   stat[ 1 ] += ( ( x[ 0 ] * x[ 0 ] + x[ 1 ] * x[ 1 ] ) / nbTirage );

   /* Placement dans le tableau de distribution */
   for( i = 0 ; i < 2 ; i++ )
   {
      if( x[ i ] >= 0 && x[ i ] < nbIntervalle )
      {
         distrib[ (int) ( x[ i ] / intervalle ) ]++;
      }
   }
}
```

```
#define NBPOINT 500
/*  en realite, NBPOINT est le nombre d'iteration, pour avoir le nombre de
point, il faut multplier par 2 */

#define NBINTERVALLE 20
#define INTERVALLE 1

int main(int argc, char ** argv)
{
   int          distrib[NBINTERVALLE];
   double       stat[2];
   int          i;

   /* Initialisation de la moyenne et de l'ecart-type */
   stat[ 0 ] = 0;
   stat[ 1 ] = 0;

   /* Initialisation du germe de drand48 */
   srand48( time ( NULL ) );

   /* Initialisation du tableau de distribution */
   for( i = 0 ; i < NBINTERVALLE ; i++ )
   {
      distrib[ i ] = 0;
   }

   /* Realisation des tirages et de l'histogramme */
   for( i = 0 ; i < NBPOINT ; i++ )
   {
      distribution(distrib, NBINTERVALLE, INTERVALLE, stat, NBPOINT * 2);
```

```
   }

   /* Dernier calcul pour l'ecart-type */
   stat[ 1 ] = sqrt( stat[ 1 ] - stat[ 0 ] * stat[ 0] );

   /* Affichage */
   for( i = 0 ; i < NBINTERVALLE ; i++ )
   {
      printf("[%0.2f, %0.2f[ : %f %%\n", (double) i * INTERVALLE,
             (double) (i + 1) * INTERVALLE,
             (double) distrib[i] / ( NBPOINT * 2 ) * 100 );
   }
   printf("Moyenne = %f\nEcart type = %f\n", stat[ 0 ], stat[ 1 ]);

   return 0;
}
```



On remarque que la distribution obtenue ressemble à une loi normale.

Quelques bibliothèques qui implémentent la génération de nombre pseudos-aléatoires suivant des loi de distribution non uniforme

- **GSL - GNU Scientific Library** : http://www.gnu.org/software/gsl/
  Bibliothèque **C/C++** ;
  Les différentes distributions réalisables sont indiquées à cet endroit :
  http://www.gnu.org/software/gsl/manual/html_node/Random-Number-Distributions.html
- **Boost** : http://www.boost.org/doc/libs/1_36_0/libs/libraries.htm
  Bibliothèque **C++** ;
  Les différentes distributions réalisables sont indiquées à cet endroit :
  http://www.boost.org/doc/libs/1_36_0/libs/random/random-distributions.html
- **UNU.RAN - Universal Non-Uniform RANdom number generators** :
  http://statmath.wu-wien.ac.at/unuran/
  Bibliothèque **C** ;
- **Newran02C - a random number generator Library** :
  http://www.robertnz.net/nr02doc.htm
  Bibliothèque **C++** ;
- **SSJ : A Java library for a stochastic simulation** :
  http://www.iro.umontreal.ca/~simardr/ssj/doc/html/overview-summary.html
  Bibilothèque **Java**.