Mamadou Bane

SIMULATION LAB # 3

MONTE CARLO SIMULATION & CONFIDENCE INTERVALS

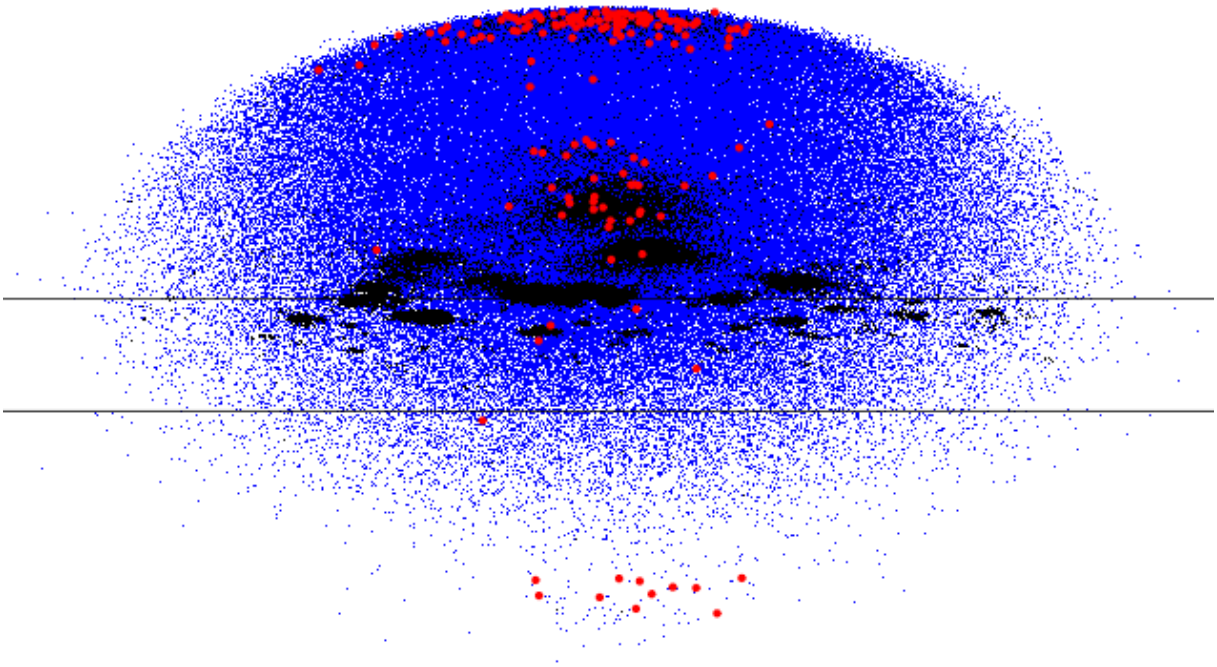


Table des matières

Introduction

Dans ce TP, nous allons utiliser le générateur Mersenne Twister proposé par Matsumoto pour générer des nombres aléatoires vu qu'il est plus adapté aux calculs scientifiques comparé au générateur disponible dans la librairie standard de C (avec la fonction rand() de stdlib).

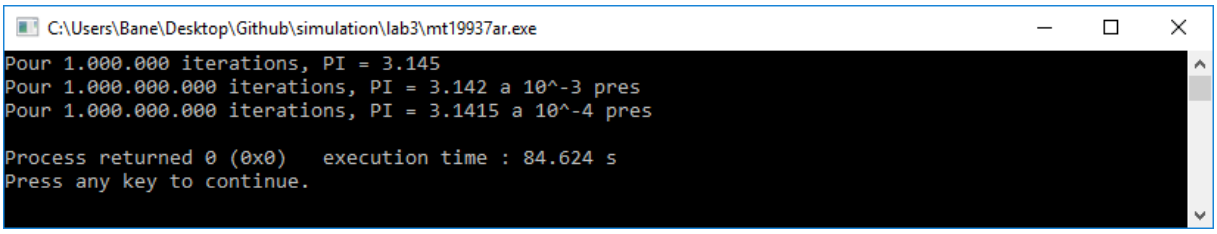
La simulation de Monte-Carlo est une méthode d'estimation d'une quantité numérique qui utilise des nombres aléatoires. Stanisław Ulam et John von Neumann l'appelèrent ainsi, en référence aux jeux de hasard dans les casinos, au cours du projet Manhattan qui produisit la première bombe atomique pendant la Seconde Guerre mondiale.

Génération de π avec la méthode Monte Carlo

Pour la génération de π nous avons récupérer l'implémentation du générateur Mersenne Twister pour obtenir des nombres aléatoires.

```
/* ---------------------------------------------------------------------------- */
/* computePi    : Generation de Pi avec la méthode Monte Carlo      */
/* En entrée    : ni, nombre entier désignant e nombre d'itération   */
/* En sortie    : la valeur réelle obtenue pour le calcul de Pi           */
/* ---------------------------------------------------------------------------- */
double computePi(int ni) // ni nombre d'itération
{
   double rand1, rand2, nbInside=0. , rate, pi;
   int i;
   for(i=0;i<ni;i++)
   {
      rand1 = genrand_real1();
      rand2 = genrand_real1();
      if(rand1*rand1 + rand2*rand2 < 1)
         nbInside++;
   }
   rate = nbInside/ni;
   pi = 4*rate;
   return pi;
}
```

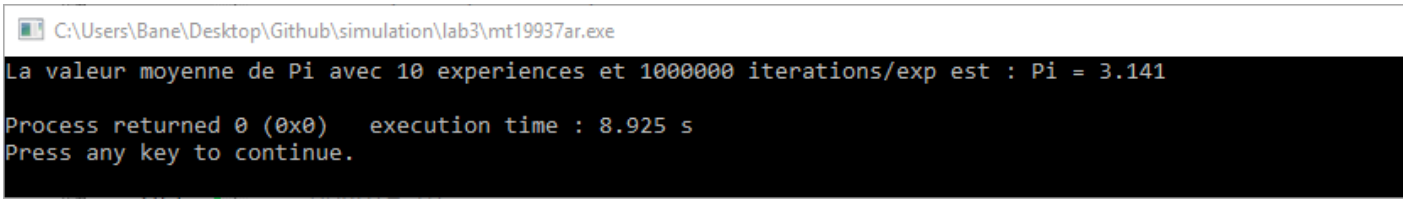
La figure suivante montre les résultats obtenus lors des différents tests.



On peut par ailleurs utiliser cette fonction dans un nombre finis d'expériences indépendantes afin de générer une valeur moyenne de PI.

```
/* ------------------------------------------------------------------------------------------ */
/* compute_n_Pi : Valeur moyenne de Pi avec n expériences indépendantes   */
/* En entrée    : n, nombre d'expérience indépendantes à réaliser          */
/*              ni, nombre entier désignant le nombre d'itération          */
/*              pour chaque expérience                                     */
/* En sortie    : la valeur moyenne de Pi obtenue                          */
/* ------------------------------------------------------------------------------------------ */
double compute_n_Pi(int n, int ni) // n : nombre d'experience, ni:nombre d'iteration pour chaque experience
{
   int i;
   double pi = 0.;
   for(i=0; i<n; i++)
      pi += computePi(ni);
   pi /= n;
   return pi;
}
```

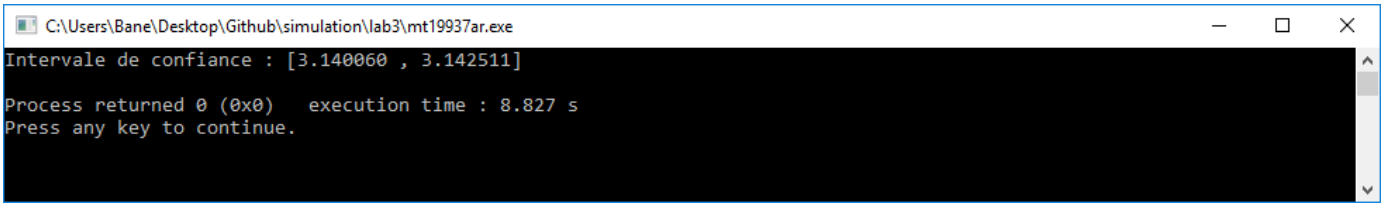
Ce qui nous donne le résultat suivant :



Calcul de l'intervalle de confiance

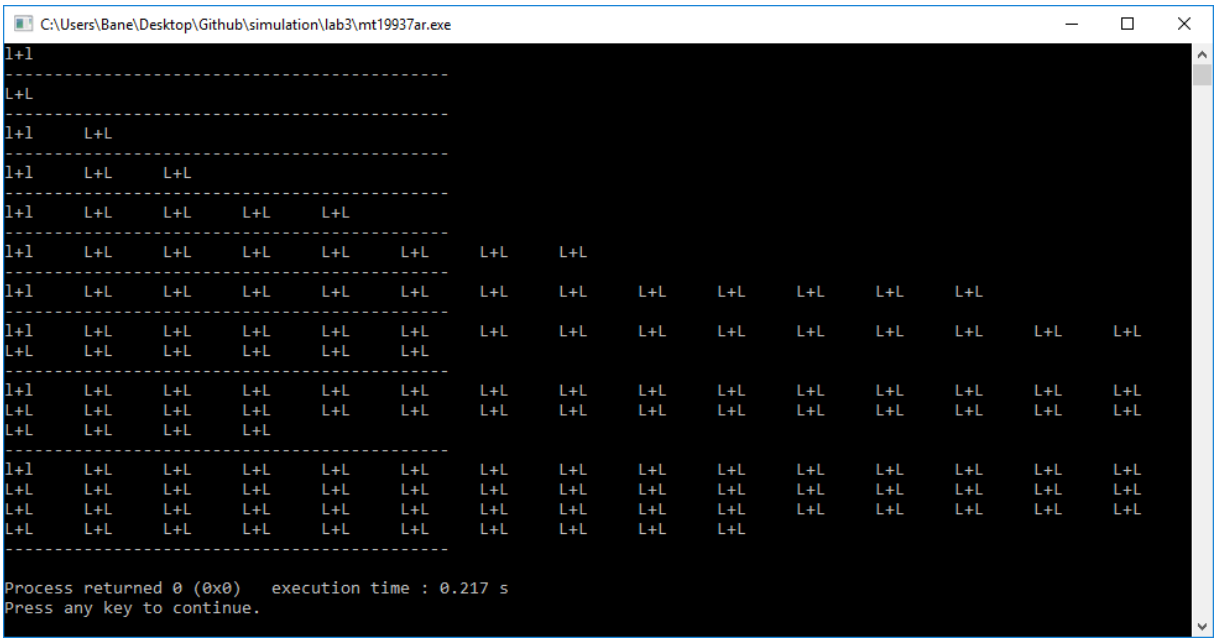
L'intervalle de confiance permet de définir la marge d'erreur entre les résultats obtenus de notre estimation de la valeur de PI.

```
void confidence_interval(int n, int ni)
{
  double t[30] = {12.706,4.303,3.182, 2.776, 2.571, 2.447, 2.365, 2.308, 2.262,
  2.228, 2.201, 2.179, 2.160, 2.145, 2.131, 2.120, 2.110, 2.101, 2.093, 2.086, 2.080,
  2.074, 2.069, 2.064, 2.060, 2.056, 2.052, 2.048, 2.045, 2.042};
   double s2=.0, r, pi_ = .0;
   double* pi = NULL;
   pi =  malloc(n*sizeof(double));
   int i;
   for(i=0; i<n; i++)
   {
      pi[i] = computePi(ni);
      pi_ += pi[i];
   }
   pi_ /= n;
   for(i=0; i<n; i++)
   {
      s2 += pow ( (pi[i]-pi_) , 2);
   }
   s2 /= n-1;
   r = t[n-1]*sqrt(s2/n);
   printf("Intervale de confiance : [%f , %f]\n", pi_-r, pi_+r);
   free(pi);
}
```



Augmentation de la population des lapins

```
int fib(int n)
{
   if(n==0 || n==1)
      return n;
   else
      return fib(n-1)+fib(n-2);
}
void rabbit_population_growth(int n)
{
   int i,j, paire;
   for(i=1; i<=n; i++)
   {
      paire = fib(i);
      if(i==2)
      {
         printf("L+L\t");
      }
      else
         printf("l+l\t");
      for(j=2;j<=paire;j++)
      {
         printf("L+L\t");
      }
      printf("\n----------------------------------------------\n");
   }
}
```


C:\Users\Bane\Desktop\Github\simulation\lab3\mt19937ar.exe
l+l
L+L
l+l L+L
l+l L+L L+L
l+l L+L L+L L+L L+L
l+l L+L L+L L+L L+L L+L L+L L+L
l+l L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
l+l L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L L+L L+L
l+l L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L
l+l L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
L+L L+L L+L L+L L+L L+L L+L L+L L+L L+L
Process returned 0 (0x0) execution time : 0.217 s
Press any key to continue.