Erreurs fréquentes

Essaie toujours de vérifier d'abord que tu n'as pas fait l'une de ces erreurs avant d'appeler un enseignant en TD !

- Faire une division d'entiers inattendue : $1/2 = 0$.
- Utiliser $=$ au lieu de $==$ dans un `if` (ou autre condition).
- Faute de copier-coller : `for (j = 0; j < 10; i++)`
- Utilisation d'une variable déclarée mais non initialisée.
- Utiliser une virgule au lieu d'un point comme séparateur décimal.
- Oubli d'un $<<$ quelque part dans une commande `cout`.
- Oubli d'accolades autour de deux (ou plus de) commandes dans un bloc, ce qui fait que la deuxième commande est exécutée *après* et non pas *dans* la boucle.
- Essayer d'utiliser le résultat d'une fonction du type `void` : `cout << f(x) << endl;` où `f` est une fonction du type `void`.
- Oublier que le premier élément d'un tableau a l'indice 0 et non pas 1 et que l'élément n d'un tableau de taille n n'existe donc pas $\rightarrow$ erreur de segmentation.
- Écrire les éléments d'une matrice comme `m[i,j]` au lieu de `m[i][j]`.

9. Générateur de nombres aléatoires

En physique on a souvent besoin d'une suite de nombres aléatoires pour simuler des processus statistiques, comme par exemple la radioactivité, le mouvement brownien, ou le fond diffus cosmologique.

- Le C fournit la fonction `drand48()` (nécessite `#include<stdlib.h>`) comme générateur de nombres réels aléatoires distribués de façon uniforme sur l'intervalle $[0.0, 1.0]$ (en anglais *RNG = random number generator*).
- En réalité aucun algorithme ne peut produire des nombres vraiment aléatoires (on parle souvent de nombres pseudo-aléatoires). En particulier il faut toujours spécifier le point de départ de la suite (la graine ou le germe, en anglais *seed*) en fournissant une valeur entière avec la fonction `srand48` avant la première utilisation de `drand48()`. Avec la même graine, on reproduit exactement la même suite de nombres.
- Pour éviter d'avoir la même suite à chaque exécution du programme, on peut utiliser l'heure UNIX (= nombre de secondes depuis le 1er janvier 1970) comme graine avec la commande `srand48(time(NULL));` (nécessite en plus la directive `#include<time.h>`).

`drand48` est un générateur congruentiel linéaire, c'est à dire qu'il utilise la suite $u_{n+1} = (p * u_n + c) \,\%\, q$. La valeur de u_0 est la graine. `drand48` utilise $q = 2^{48} = 281474976710656$, $c = 11$ et $p = 25214903917$.

Pour expliquer la méthode, on regarde le cas plus simple avec $c = 0$, appelé générateur congruentiel multiplicatif : $u_{n+1} = (p * u_n) \,\%\, q$.

On peut montrer (voir poly) que si q est un nombre premier et p une racine primitive modulo q, la suite des u_n va donner toutes les valeurs entières entre 1 et $q-1$ dans un ordre "aléatoire", avant de répéter. La valeur $(u_n - 1.)/(q - 2.)$ sera donc une valeur réelle pseudo-aléatoire et uniforme dans l'intervalle $[0., 1.]$.

(La condition que q soit premier n'est pas nécessaire pour le générateur congruentiel linéaire.)

Exemple : $q = 11$, $p = 2$, $u_0 = 1$

n	0	1	2	3	4	5	6	7	8	9	10	11
u_n	1	2	4	8	5	10	9	7	3	6	1	2

La bibliothèque du magistère (`#include<bibli_fonctions.h>`) contient une fonction `alea()` construite de cette façon (avec $q = 1664713$ et $p = 1288$) et une fonction `germe` pour spécifier la graine :

```
const int q = 1664713, p = 1288; int n = 1;  // variables globales
void germe(int g) {
  n = g % q;
  if(n == 0) ...          // afficher message d'erreur dans ce cas
}
double alea() {
  n = n * p % q;
  return (double)(n-1)/(q-2);
}
```

Exemple : jeu de la vie de Conway (automate cellulaire). Pour chaque cellule dans une grille à deux dimensions on regarde les 8 voisines les plus proches. Une cellule vivante possédant 2 ou 3 voisines vivantes le reste, sinon elle meurt. Une cellule morte possédant 3 voisines vivantes naît, sinon elle reste morte. La configuration initiale est déterminée de façon aléatoire.

```
#include<Python.h>
#include<fstream>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>  // pour usleep
using namespace std;
const int dimx = 101, dimy = 101, N = 100; const double seuil = 0.8;
void initialisation(int** m) { // Determination de la configuration initiale
  int i, j;
  for (i = 0; i < dimy; i++)
    for (j = 0; j < dimx; j++)
      if (drand48() >= seuil) m[i][j] = 1;  // 1 = vivante
      else m[i][j] = 0;                     // 0 = morte
}
int nombre_voisins(int** m, int i0, int j0) { // Compte le nombre de voisines
  int i, j, s = 0;                            // vivantes
  for (i = i0-1; i <= i0+1; i++)
    for (j = j0-1; j <= j0+1; j++)
      if (!(i == i0 && j == j0))
        s += m[(i+dimy)%dimy][(j+dimx)%dimx]; // conditions aux limites
  return s;                                   // periodiques
}
```

```
void applique_regles(int** m, int** m1) { // m contient la config actuelle
  int i, j, s;                              // a l'entree, m1 contiendra la
  for (i = 0; i < dimy; i++)                // nouvelle config a la sortie
    for (j = 0; j < dimx; j++) {
      s = nombre_voisins(m,i,j);
      if (s == 3) m1[i][j] = 1;
      else if (s == 2) m1[i][j] = m[i][j];
      else m1[i][j] = 0;
    }
}
int main() {
  int i, j, k;
  int **m, **m1;
  fstream fich;
  srand48(time(NULL));
  m = (int**)malloc(dimy * sizeof(int*));
  for (i = 0; i < dimy; i++)
    m[i] = (int*)malloc(dimx * sizeof(int));
  m1 = (int**)malloc(dimy * sizeof(int*));
  for (i = 0; i < dimy; i++)
    m1[i] = (int*)malloc(dimx * sizeof(int));
/*On ne peut pas utiliser la fonction make_plot_py ici, parce qu'il faut faire
  appel a Python plusieurs fois (on ne peut avoir qu'une seule par programme)*/
  Py_Initialize();
  PyRun_SimpleString("from numpy import *");
  PyRun_SimpleString("from matplotlib.pyplot import *");
  PyRun_SimpleString("ion()");
  PyRun_SimpleString("fig = figure()");
```

```
  for (k = 0; k < N; k++) {
    if (k == 0)
      initialisation(m);
    else {
      applique_regles(m, m1);
      for (i = 0; i < dimy; i++)
        for (j = 0; j < dimx; j++)
          m[i][j] = m1[i][j];
    }
    fich.open("jeu.dat", ios::out);
    for (i = 0; i < dimy; i++) {
      for (j = 0; j < dimx; j++)
        fich << m[i][j] << " ";
      fich << endl;
    }
    fich.close();
    PyRun_SimpleString("A = loadtxt('jeu.dat')");
    PyRun_SimpleString("clf()");
    PyRun_SimpleString("imshow(A)");
    PyRun_SimpleString("fig.canvas.draw()");
    usleep(3e5);     // pause, en microsecondes (donc 0.3 secondes)
  }
  Py_Finalize();
  for (i = 0; i < dimy; i++) {
    free(m[i]); free(m1[i]); }
  free(m); free(m1);
  return 0;
}
```

Distribution non uniforme

Pour avoir des nombres aléatoires uniformes dans $[a, b]$ au lieu de $[0, 1]$ on peut simplement prendre `(b-a)*drand48()+a`. Mais quoi faire si l'on veut une distribution non uniforme ?
Soit x une valeur aléatoire uniforme dans $[0, 1]$. On montre deux méthodes pour en déduire une valeur aléatoire y de densité de probabilité $g(y)$:

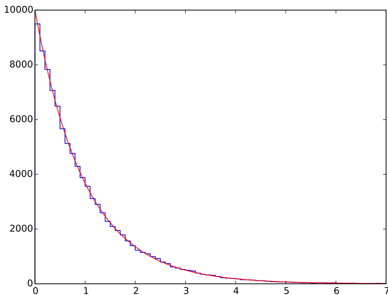
1. Si l'on peut calculer analytiquement la primitive G de g et son inverse G^{-1}, on peut écrire directement $y = G^{-1}(x - x_0 + G(y_0))$, où la valeur y_0 que l'on associe à une valeur x_0 de référence a fixé la constante d'intégration (**méthode de l'inversion de la primitive**).

   Exemple : $g(y) = \frac{1}{\alpha} \exp\left(-\frac{y}{\alpha}\right)$ pour $y \geq 0$ et 0 ailleurs, avec $\alpha > 0$.

   $\Rightarrow\ G(y) = -\exp(-y/\alpha)$ et $G^{-1}(z) = -\alpha \ln(-z)$.
   En fixant $y_0 = 0$ quand $x_0 = 0$ on trouve $y = -\alpha \ln(-x + 1)$.
   Vu que x et $1 - x$ ont la même distribution, $y = -\alpha \ln(x)$ marche aussi.

   Donc `y = -alpha*log(drand48());` donne des valeurs aléatoires avec une densité de probabilité exponentielle $g(y) = \frac{1}{\alpha} \exp\left(-\frac{y}{\alpha}\right)$.

On voit que ça marche :



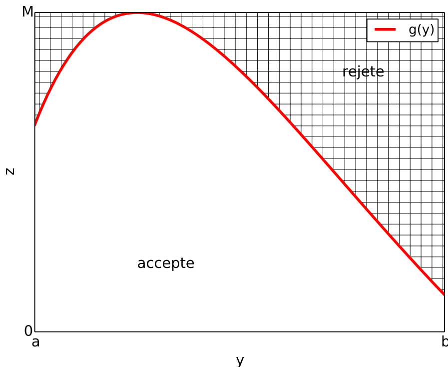

```
#include<bibli_fonctions.h>
int main() {
  fstream fich;
  int i, n = 100000;
  double y, alpha = 1.;
  srand48(time(NULL));
  fich.open("proba_exp.res", ios::out);
  for(i = 0; i < n; i++) {
    y = -alpha * log(drand48());
    fich << y << endl;
  }
  fich.close();
  ostringstream pyth;
  pyth
    << "A = loadtxt('proba_exp.res')\n"
    << "hist(A, 70, (0,7), histtype='step')\n"
    << "x = linspace(0,7,100)\n"
    << "plot(x, " << n/10/alpha << " * exp(-x/" << alpha << "), 'r')\n"
   ;
  make_plot_py(pyth);
  return 0;
}
```

Rappel : On cherche des valeurs aléatoires y de densité de probabilité $g(y)$ sur $[a, b]$. Soit M la valeur supérieure de $g(y)$ sur $[a, b]$.

2. Si l'on ne peut pas calculer analytiquement la primitive G de g ou son inverse G^{-1}, on peut utiliser la **méthode de rejet de Von Neumann** :

   On tire deux valeurs aléatoires : y avec densité uniforme sur $[a, b]$ et z avec densité uniforme sur $[0, M]$. Si le point de coordonnées (y, z) est au-dessus de la courbe $g(y)$, on le rejette, sinon on l'accepte. Les valeurs de y ainsi sélectionnées suivent la loi de densité $g(y)$ (voir poly).

   Remarque : Tirer des valeurs de z uniformes sur $[0, \lambda M]$ ($\lambda > 0$) et rejeter les points au-dessus de la courbe $\lambda g(y)$ donne le même résultat.



Exemple de la méthode de l'inversion de la primitive : le **mouvement brownien** d'une particule dans le plan. Entre deux chocs contre les molécules du milieu elle parcourt en ligne droite une distance r dans une direction aléatoire. Chaque choc est considéré comme le tirage au hasard d'un angle θ entre 0 et 2π avec densité uniforme et d'une longueur r entre 0 et ∞ avec densité exponentielle (avec paramètre $a =$ libre parcours moyen).

```
#include<bibli_fonctions.h>
int main() {
  int i, N = 500;
  double r, th, x = 0., y = 0., a = 1.;
  fstream fich;
  srand48(time(NULL));
  fich.open("mvtbrownien.dat", ios::out);
  for (i = 0; i < N; i++) {
    fich << x << " " << y << endl;
    th = 2.*M_PI * drand48();   // Tirer au hasard l'angle theta
    r = -a * log(drand48());    // Tirer au hasard la longueur r
    x += r * cos(th);
    y += r * sin(th);
  }
  fich.close();
  ostringstream pyth;
  pyth << "A = loadtxt('mvtbrownien.dat')\n"
       << "plot(A[:,0], A[:,1])\n";
  make_plot_py(pyth);
  return 0;
}
```

Exemple : on considère un **gaz parfait** de N_p particules (pas de collisions entre elles, collisions élastiques avec les parois) dans une boîte à une dimension (entre 0 et L). On tire au hasard les positions initiales (uniformes dans $[0, L]$) et les vitesses initiales (selon la distribution de Maxwell $f(v) = \sqrt{\frac{m}{2\pi kT}} \exp(-\frac{mv^2}{2kT})$).
Puis on fait évoluer le système et on calcule à chaque instant l'impulsion transférée aux deux parois ($2mv$ par collision), ce qui représente la pression. On calcule la moyenne sur le temps et on compare à la valeur analytique $N_p kT/L$.
On utilise la **méthode de rejet** pour trouver des valeurs aléatoires pour les vitesses initiales de densité de probabilité gaussienne.

```
#include<iostream>
#include<math.h>
#include<stdlib.h>
#include<time.h>
using namespace std;
double gaussienne(double var) { // Fournit valeur alea. de densite gaussienne
  double x, y, ecart = sqrt(var); // avec variance var et moyenne 0.
  do {  // On prend x dans l'intervalle [-5sigma,5sigma], on suppose que la
    x = 10.*ecart * drand48() - 5.*ecart; // gaussienne est nulle ailleurs.
    y = drand48();
  } while(y > exp(-x*x/(2.*var)));
  return x;
}
void evolution(double* x, double* v, int Np, double L, double m,
               double* p1, double* p2) {
  int i;
  *p1 = 0.; *p2 = 0.;    // p1 pression sur paroi x = 0 et p2 sur x = L
```

```
  for (i = 0; i < Np; i++) {
    x[i] += v[i];          // Evolution de x, on prend dt = 1
    if (x[i] <= 0) {
      x[i] = -x[i]; v[i] = -v[i];
      *p1 += 2. * m * v[i]; }
    if (x[i] >= L) {
      x[i] = 2.*L - x[i]; v[i] = -v[i];
      *p2 += -2. * m * v[i]; }
  } }
int main() {
  int i, Np = 100000, Nt = 1000;  // Np nombre particules, Nt nombre temps
  double p1, p2, L = 10., kT = 10., m = 3.;
  double sp1 = 0., sp2 = 0., sp1c = 0., sp2c = 0., moy, var;
  double* x = (double*)malloc(Np * sizeof(double));
  double* v = (double*)malloc(Np * sizeof(double));
  srand48(time(NULL));
  for (i = 0; i < Np; i++) {      // Initialisation
    x[i] = L * drand48();
    v[i] = gaussienne(kT/m); }
  for (i = 0; i < Nt; i++) {      // Evolution et calcul
    evolution(x, v, Np, L, m, &p1, &p2);
    sp1 += p1; sp2 += p2; sp1c += p1*p1; sp2c += p2*p2; }
  moy = sp1 / Nt; var = sp1c / Nt - moy*moy;
  cout << "p(x=0) = " << moy << " +/- " << sqrt(var) << endl;
  moy = sp2 / Nt; var = sp2c / Nt - moy*moy;
  cout << "p(x=L) = " << moy << " +/- " << sqrt(var) << endl;
  cout << "p = N k T / V = " << Np * kT / L << endl;
  free(x); free(v); return 0; }
```