Langage C et aléa, séance 1

École des Mines de Nancy, séminaire d'option Ingénierie Mathématique

Frédéric Sur
http://www.loria.fr/~sur/enseignement/coursCalea/

1 Rappels Linux

Le compilateur utilisé sera `gcc`, présent sur toutes les distributions Linux. Nous travaillerons donc en environnement Linux.

Quelques rappels :

- ouverture d'un terminal sous KDE : click gauche sur le « menu K » (étoile dans la barre des tâches), puis onglet « système ».
- commandes pour la navigation dans l'arborescence de fichiers :
  `ls` pour la liste des fichiers du répertoire courant
  `cd rep` pour aller dans le répertoire nommé `rep`
  `cd` pour retourner dans votre répertoire `home`
  `pwd` pour afficher le nom du répertoire courant
  `mkdir rep` pour créer le répertoire de nom `rep` (conseil : créez un répertoire réservé à ce cours)
  `man command` pour afficher l'aide de la commande `command` (exemple : `man sort`)
  Bien sûr, tapez sur la touche « entrée » après chaque commande...
- historique : flèche vers le haut pour rappeler les dernières commandes exécutées (intéressant pour recompiler le même code source par exemple).
- copier-coller simplifié : surlignez à la souris un texte à copier, et cliquez sur le bouton du milieu là où vous voulez le coller.
- éditeur de texte : on éditera directement les fichiers sources dans un éditeur de texte. Utilisez `kwrite`, ou `emacs` si vous êtes habitué.

Remarque : les extensions données ici aux noms des exécutables sont totalement arbitraires, et ont pour seul but de faciliter la lecture.

2 Le langage C

Le langage C se veut très rapide à l'exécution, et permet le contrôle du matériel. Une grande partie des systèmes d'exploitation sont écrits en C, et l'histoire du langage C est très liée à celle des systèmes Unix.

Voici un bref historique :

- Début des années 70 : Denis M. Ritchie et Brian W. Kernighan développent un langage de programmation à la suite du langage B pour le développement d'Unix aux "Bell Laboratories" : le langage C.
- 1978 : publication de "The C programming language" par Kernighan et Ritchie, qui donne les bases du langage.
- 1983 : normalisation par l'American National Standards Institute (ANSI) du langage, après l'apparition de multiples compilateurs C différents. On parle du C-ANSI.
- début des années 80 : première version de C++ par Bjarne Stroustrup, extension du langage C avec la notion d'objet.

Attention, selon le couple compilateur / machine utilisé, la normalisation du langage ne sera pas forcément exactement la même... Pour des raisons historiques (de compatibilité), certains logiciels sont écrits en C dit

K&R (Kernighan et Ritchie) et ne respectent donc pas la norme ANSI. De même, les compilateurs Microsoft et Borland ne respectent pas toutes les recommandations de la dernière norme C99 (qui n'est d'ailleurs pas entièrement supportée par `gcc`). Nous ne rentrerons pas dans ces détails[1].

Comme en langage JAVA, le signe de fin d'instruction est le point-virgule, les séparateurs de blocs d'instructions sont les accolades { }, et il y a une différence entre majuscules et minuscules. Le programme principal à exécuter correspond aussi à une fonction `main`.

Pour des raisons de lisibilité, il est conseillé de commenter ses programmes (de la manière suivante : `/* blabla */`) et de les indenter correctement.

2.1 Le compilateur

Un compilateur est un programme qui transforme le fichier source (un fichier texte contenant une suite d'instructions C) en un langage directement compréhensible par le processeur de la machine utilisée. Contrairement à JAVA, il n'y a pas de notion de *machine virtuelle*. Dans ces séances, on utilisera le compilateur gcc[2]. Un fichier source C finit par l'extension .c (attention, une extension .C sera interprétée par gcc comme du langage C++...)

La syntaxe de compilation (dans un terminal) est :

```
gcc nom.c -o nom.exe
```

Ceci a pour effet de compiler le fichier source `nom.c` pour créer l'exécutable `nom.exe`

On lance l'exécutable dans le terminal en tapant :

```
./nom.exe
```

Attention à bien recompiler à chaque modification du fichier source !

Pour obtenir des commentaires et avertissements détaillés lors de la création de l'exécutable, on peut passer les options `-Wall` et `-pedantic` à `gcc` :

```
gcc -Wall -pedantic nom.c -o nom.exe
```

2.2 Fichiers d'en-tête

La commande `#include` permet d'intégrer des parties de code dans le fichier source. En particulier, elle permet l'utilisation de bibliothèques de fonctions en spécifiant la définition de ces fonctions. Ces définitions sont contenues dans un *fichier d'en-tête* (header) que l'on inclut par `#include` en début du fichier source. Le fichier d'en-tête ne contient que les prototypes des fonctions (pas leur code source proprement dit), des valeurs constantes, etc, et permet au compilateur de connaître la syntaxe des fonctions de la bibliothèque utilisée.

Par exemple :
`#include<stdlib.h>` correspond à la *standard library*,
`#include<stdio.h>` correspond à l'*input / output library*.

On verra par la suite quand utiliser ces bibliothèques.

Les fonctions mathématiques se trouvent dans une bibliothèque spécifique. Le fichier d'en tête est `math.h` (inclus par `#include<math.h>`). Attention, cette librairie n'est pas incluse par défaut par gcc (contrairement à `stdlib` et `stdio`), il faut le spécifier en passant l'option `-lm` au compilateur :

```
gcc nom.c -o nom.exe -lm
```

(ou éventuellement : `gcc -Wall -pedantic nom.c -o nom.exe -lm` si on veut plus de commentaires.)

[1] Sur les problèmes de normalisation du langage, et les avantages et inconvénients de C, on pourra lire :
`http ://en.wikipedia.org/wiki/C_programming_language`

[2] `http ://gcc.gnu.org`

2.3 Types et variables

Le langage C ne fournit que des types numériques. En particulier, il n'y a pas de type `boolean` ou `String` comme en JAVA. Pour évaluer des expressions logiques, on utilisera tout type numérique (généralement `int`, ou `unsigned char`) en considérant que 0 correspond à *false* et toute autre valeur à *true*.

Les déclarations et initialisations éventuelles se font comme en JAVA. Par contre en C les variables ne sont pas initialisées par défaut. Comme en JAVA, une variable n'existe qu'à partir de son endroit de déclaration et jusqu'à la fin du bloc dans lequel elle est déclarée.

Un nom de variable peut être constitué de lettres, de chiffres et du caractère « blanc souligné » _ (*underscore* en anglais), mais il ne peut commencer que par une lettre ou un underscore (jamais par un chiffre). Bien sûr, on ne peut pas utiliser les mots réservés du langage C.

La normalisation C90 impose la déclaration des variables en début des blocs d'instruction (contrairement à JAVA ou C++). Ce n'est plus le cas depuis la normalisation C99.

Les types entiers sont : `char`, `short`, `int`, `long` (éventuellement précédés de `unsigned`).

Les types décimaux sont : `float`, `double`.

2.4 Les tableaux

En C, on déclare un tableau (unidimensionnel) par : `type nom_du_tableau[taille];`

Si `taille` est une constante, on peut initialiser tous les éléments d'un tableau à 0 à la déclaration par : `type nom_du_tableau[taille] = {0};`

En C90, la taille du tableau devait être une constante, i.e. à valeur fixée au moment de la compilation (par exemple soit définie par `#define taille 14`, soit par `const int taille=14...`). Ce n'est plus le cas en C99 (comme en JAVA donc).

Exemple de déclaration d'un tableau de `double`s dont la taille est stockée dans la variable `taille_tab` (de type `int`) :

```
double tableau[taille_tab];
```

Attention, les cases d'un tableau de taille T sont numérotées de 0 à $T-1$. Pour stocker 4.3 dans la quatrième case : `tableau[3]=4.3;`

On verra l'utilisation des tableaux plus loin.

2.5 Affichage formaté

On dispose de l'instruction `printf`, qui permet la sortie sur la sortie standard (ici le terminal) d'une chaîne de caractères formatée. Le premier argument est une chaîne de caractères avec des paramètres de contrôle, les arguments suivants sont les variables dont la valeur doit remplacer les paramètres de contrôle.

Exemples de paramètres de contrôle : `%d` pour un entier, `%f` pour un décimal, et `%s` pour une chaîne de caractères.

Le caractère de contrôle `\n` permet un retour à la ligne.

Exemple :

```
printf("la valeur de la variable entière toto est: %d \n",toto);
printf("le double titi1 vaut: %f et l'entier titi2 vaut: %d \n",titi1,titi2);
```

affiche (par exemple) :

```
la valeur de la variable entière toto est: 5
le double titi1 vaut: 4.56 et l'entier titi2 vaut: 3
```

Pour la saisie de valeurs de variables par l'utilisateur, on dispose de `scanf`, au fonctionnement analogue.

Les fonctions `printf` et `scanf` sont définis dans la bibliothèque `stdio` (il faut donc inclure le fichier `.h` correspondant).

Exercice 1 (un premier petit programme)

Écrivez dans un fichier essai1.c le texte suivant. Vous pouvez copier/coller ce texte directement depuis le fichier pdf.

```
#include<stdio.h>
int main(void) {
  int i,j;
  float x;
  printf("entrez un entier (i): ");
  scanf("%d",&i); /* on verra plus tard pourquoi le & */
  printf("entrez un décimal (x) et un entier (j) séparés par un espace\n");
  scanf("%f %d", &x, &j);
  printf("x vaut: %f, i vaut: %d, et j vaut: %d.\n",x,i,j);
}
```

Puis compilez : `gcc essai1.c -o essai1.exe`
Exécutez le programme par : `./essai1.exe`

Exercice 2 (un deuxième petit programme)

Écrivez dans un fichier essai2.c le texte suivant :

```
#include<stdio.h>

int main(void)
{
  int i, j; /* i et j sont déclarés comme entiers */
  double x=5.0, y; /* x est déclaré et initialisé, y seulement déclaré */
  i=2;
  printf("x/i vaut %f\n",x/i);
  j=5;
  printf("mais j/i vaut %d\n",j/i);
  y=(double)j/i;
  printf("par contre y vaut %f\n",y);
}
```

Puis compilez : `gcc essai2.c -o essai2.exe`
Commentez l'affichage.

2.6 Opérateurs

Ils sont les mêmes qu'en JAVA :
Opérateurs de calcul : `+ - * /`
Opérateurs d'assignation : `= += -= *= /=`
Opérateurs d'incrémentation : `++ --`
Opérateurs de comparaison : `== < <= > >= !=`
Opérateurs logiques : `|| && !`

2.7 Structures conditionnelles

La syntaxe de la commande `if` est identique à celle de JAVA :

```
if (condition) {
   bloc à exécuter si condition!=0
}
else {
   bloc à exécuter sinon
}
```

Comme en JAVA, le `else` n'est pas obligatoire s'il n'y a rien à faire lorsque `condition` vaut 0, et les accolades sont facultatives si le bloc est composé d'une unique instruction.

Exemple :

```
#include<stdio.h>
int main(void) {
  int i;
  printf("que vaut i? ");
  scanf("%d",&i);
  if (i%2==0) printf("%d est pair\n",i);
  else printf("%d est impair\n",i);
}
```

On dispose aussi du couple `switch / case` comme en JAVA.

2.8 Structures itératives

La syntaxe des boucles `for` est identique à JAVA à un détail près : en C l'indice utilisé dans une boucle `for` doit être déclaré au préalable.

Donc pas de :

```
for (int i=0; i<10; i++) {
    bloc d'instructions à itérer
}
```

Mais :

```
int i;
instructions diverses
for (i=0; i<10; i++) {
    bloc d'instructions à itérer
}
```

On dispose également de manière identique à JAVA de `while`, du couple `do / while`, de `break` et `continue`.

Syntaxe de `while` :

```
while (condition) {
  bloc à exécuter tant que la condition est vraie
}
```

Exercice 3 (boucle `while`)

Écrivez un programme qui demande à l'utilisateur un entier et qui calcule et affiche la somme des chiffres de son écriture décimale. Vous utiliserez une boucle `while` et remarquerez que si n est de type entier, n/10 est l'entier correspondant à n auquel on a retiré son dernier chiffre et n%10 est ce dernier chiffre.

2.9 Les fonctions

Définition d'une fonction :

```
type_de_sortie nom_fonction(type_arg1 arg1, type_arg2 arg2, ...) {
  corps de la fonction
}
```

La fonction doit être définie avant le `main` (à moins d'en donner le prototype...) et après les instructions au préprocesseur (les `#include` ou `#define` par exemple). Elle doit renvoyer une valeur de type `type_de_sortie` à l'aide de l'instruction `return`.

Une fonction ne renvoyant rien (car par exemple elle se limite à un affichage) a un type de sortie égal à `void`.

L'appel d'une fonction se fait par : `nom_fonction(arg1, arg2,...)` Si `sortie` est une variable de type `type_de_sortie`, on peut stocker la valeur de retour de la fonction par l'affectation : `sortie = nom_fonction(arg1,arg2, ...)`

Récursivité et appels croisés se font comme en JAVA.

Cas particulier de la fonction `main` : son type de retour est `int` (la valeur retournée peut être capturée par le système d'exploitation et interprétée comme un code d'erreur), et elle peut éventuellement accepter des arguments (qui correspondent aux arguments passés en ligne de commande, comme en JAVA).

Nous n'utiliserons pas d'arguments en entrée, d'où la syntaxe utilisée `int main(void)`

Exercice 4 (utilisation d'une fonction, suite de Syracuse)

On considère la fonction suivante :

```
int syracuse(int n) {
  int sortie;
  if (n%2==0) sortie=n/2;
  else sortie=3*n+1;
  return sortie;
}
```

Que se passe-t-il si on itère la fonction `syracuse` sur l'entier 1 ?

Écrivez un programme qui demande à l'utilisateur un entier, et affiche les itérés successifs de cet entier par la fonction tant que l'itéré est différent de 1.

Modifiez votre programme pour afficher 1) le nombre d'itérations pour arriver à 1 et 2) le plus grand entier atteint.

---

3 Génération de nombres aléatoires

Dans le chapitre 7 du livre *Numerical recipes in C*[3] figure l'avertissement suivant :

> *Now our first, and perhaps most important, lesson in this chapter is : be very, very suspicious of a system-supplied rand() [...]. If all scientific papers whose results are in doubt because of bad rand()s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist.*

La question des générateurs (pseudo-)aléatoires est critique dans de nombreuses applications (loteries, cryptographie, méthodes de Monte-Carlo que l'on verra plus tard...). Pour améliorer le caractère aléatoire du générateur, on peut perturber (intelligemment) un générateur pseudo-aléatoire par des phénomènes *a priori* non prédictibles, comme par exemple des événements observés sur le réseau. Pour des applications très critiques, on utilise des générateurs purement physiques ou mécaniques : pensez aux machines à sous ou au tirage du Loto.

Pour plus de détails sur les générateurs de nombres aléatoires et une discussion sur leurs qualités et défauts, lisez le chapitre 7 de *Numerical recipes in C*, ainsi que le chapitre 3 (tome 2) de *The art of computer programming* par D.E. Knuth.

3.1 RANDU

RANDU est un générateur congruentiel linéaire[4] qui a été largement utilisé à partir des années 60. Il est défini par la relation de récurrence :

$$V_{j+1} = 65539 \cdot V_j \mod 2^{31},$$

avec V_0 impair. Remarquez que les V_j sont tous impairs.

On va voir qu'il s'agit d'un très mauvais générateur. Pour citer Knuth : « its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists ! »

Exercice 5 (RANDU, un générateur défectueux)

Voici un code pour RANDU.

```
int next=1;
int randu(void) {
  next= (next*65539) % 2147483648U;
  return next;
}
```

La variable `next` est ici globale, l'appel à la fonction `randu` renvoie un entier (pseudo-)aléatoire.

Pensez-vous que le fait que la quantité `next*65539` puisse dépasser la taille du plus grand entier représentable soit gênant ?

Nous allons mettre en évidence une des caractéristiques de RANDU qui font que cet algorithme ne peut pas être considéré comme un générateur d'aléa fiable. Il s'agit d'un test graphique illustrant un *test spectral*.

En incorporant la fonction `randu` à votre code, écrivez un programme affichant une succession de lignes formées de trois réalisations successives de RANDU (par exemples 10 000 ou 100 000 lignes).

Ensuite, lancez votre programme par :

```
./nom_programme.exe > randu.dat
```

[3] On y dit des choses très intéressantes sur les mathématiques et la programmation en général, et le langage C en particulier : `http ://www.nrbook.com/a/bookcpdf.php`

[4] i.e. obtenu par itération de $f(x) = ax + b \mod m$.

Ceci a pour effet de rediriger la sortie du programme dans le fichier texte `randu.dat` (que vous pouvez visualiser avec votre éditeur de texte favori, par exemple `kwrite`). Si vous obtenez un message d'erreur (fichier existant), tapez dans le terminal l'instruction suivante : `unset noclobber` et recommencez.

On va visualiser la répartition de ces triplets (vus comme des coordonnées 3D) dans l'espace avec Gnuplot. Dans un *autre* terminal, placez-vous dans le répertoire où est stocké le fichier `randu.dat`, et lancez Gnuplot en tapant `gnuplot` (logique) en ligne de commande.

L'instruction Gnuplot : `splot 'randu.dat'` permet d'afficher le nuage de points dans l'interface de visualisation 3D. A l'aide de la souris vous pouvez faire tourner le nuage de points.

Que constatez-vous ? Comment devraient être répartis les points si le générateur fournissait des réalisations effectivement indépendantes ?

Pour quitter Gnuplot : `quit`.

3.2 Le générateur de `stdlib` : rand()

Nous allons à présent étudier la fonction `rand` de la bibliothèque `stdlib`. Cette fonction simule une loi uniforme et renvoie un entier pseudo aléatoire entre 0 et `RAND_MAX`. Son prototype est : `int rand(void)` (c'est-à-dire : la fonction renvoie un entier mais n'a pas d'argument). Il s'agit d'un générateur par méthode congruentielle linéaire dont la période correspond en fait à `RAND_MAX`.

Exercice 6 (Premières expériences)

Écrivez un programme affichant la valeur de `RAND_MAX`, et tirant au hasard 10 entiers. (Ne pas oublier d'inclure les fichiers d'en-tête des bibliothèques `stdio` et `stdlib`)

Qu'obtenez-vous en relançant l'exécution de votre programme ? Qu'obtient votre voisin ?

On dispose de la fonction `srand` pour fixer la graine (*seed*) du générateur. Un appel à `time(NULL)` (inclure `time.h`) renvoie un entier correspondant à l'heure système. Initialisez le générateur en ajoutant la ligne `srand(time(NULL));` dans votre programme, et expérimentez.

Comparez la répartition des points 3D obtenus comme dans l'exercice 4 à ce que vous obteniez avec RANDU.

Exercice 7 (Simulation d'un dé et test statistique)

Écrivez une fonction qui simule un dé (équilibré, à 6 faces). Elle ne prend pas d'argument et renvoie un entier (entre 1 et 6). Proposez (au moins) deux solutions différentes. Laquelle vous semble la plus adaptée ?

On va chercher à évaluer si notre dé n'est pas pipé. Pour ce faire, on calcule, pour un nombre fixé N de lancers de dé :

$$C_N = \sum_{j=1}^{6} \frac{(E_{obs}(j) - E_{th}(j))^2}{E_{th}(j)},$$

où $E_{obs}(j)$ est le nombre de tirages j observé et $E_{th}(j)$ est le nombre de tirages de j attendu (qui est donc $N/6$).

Si le générateur suit la loi théorique (uniforme sur $\{1, 2, 3, 4, 5, 6\}$, il se trouve que la loi de C_N (considéré comme une variable aléatoire) tend vers une loi du χ^2 à 5 degrés de liberté. On peut donc calculer (par approximation avec la loi du χ^2 pour N assez grand) $\Pr(C_N < \alpha)$, et réciproquement, le plus petit α tel qu'on ait 95 chances sur 100 d'observer $C_N < \alpha$. En regardant la table du χ^2 à 5 degrés de liberté, on trouve $\alpha = 11.07$ pour le degré de confiance 0.95. Un test statistique consiste donc à dire : si $C_N > 11.07$, alors on rejette l'hypothèse « le générateur est uniforme sur {1, 2, 3, 4, 5, 6} » avec un risque d'erreur de 5%.

Votre dé est-il pipé selon ce test ?

Et si vous remplacez rand() par randu() ?

Remarque : Il ne s'agit que d'un seul test, sur la bonne répartition des tirages. Sauriez-vous trouver un simulateur de dé, pas aléatoire du tout, vérifiant le test du χ^2 ? Pour évaluer les performances d'un générateur (pseudo-)aléatoire, il faut lui faire passer plusieurs tests différents (le test spectral évoqué plus haut est l'un d'eux). Plusieurs procédures de tests sont décrites dans : *The art of computer programming* (tome 2) par D.E. Knuth.

3.3 Simulation d'une loi normale

Pour simuler une variable aléatoire suivant la loi normale de moyenne μ et de variance σ^2 :

$$\mathcal{N}_{\mu,\sigma^2} : x \mapsto \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right),$$

on peut utiliser la méthode suivante (Box-Muller). Si U_1 et U_2 sont deux variables aléatoires uniformes sur $[0, 1]$ alors $X = \cos(2\pi U_1)\sqrt{-2\log(U_2)}$ suit une loi normale centrée réduite. Donc $\mu + \sigma X$ suit une loi normale de moyenne μ et variance σ^2.

Exercice 8 (loi normale)

Écrivez une fonction `loinormale` qui génère des `double` suivant une loi normale dont la moyenne et la variance sont passées en arguments.

La bibliothèque mathématique (rappel : `#include<math.h>` et compilation avec l'option `-lm`) comprend les fonctions : `cos sin exp log sqrt`...ainsi que la constante `M_PI` (valeur de π).

Vérifiez expérimentalement pour ce générateur la convergence de la moyenne empirique (resp. la variance empirique) vers la moyenne (resp. la variance). Pour ce faire, vous pouvez stocker les valeurs successives générées dans un tableau.

On se propose aussi de vérifier graphiquement la convergence de la fonction de répartition empirique vers la fonction de répartition (lemme de Glivenko-Cantelli). Modifiez votre programme pour afficher seulement les valeurs successives générées par une boucle sur : `printf("%f\n", loinormale(moy,var));` puis exécutez votre programme en redirigeant la sortie dans un fichier par :

```
./nom_programme.exe > loinormale.dat
```

Triez le fichier et stockez le résultat dans `loinormalesort.dat` en exécutant, toujours dans le terminal :

```
sort -n loinormale.dat -o loinormalesort.dat
```

Si le tri n'est pas correct, tapez :

```
export LC_ALL=POSIX
```

si votre shell est Bash

ou `setenv LC_ALL POSIX` si votre shell est Tcsh
puis recommencez le tri.
On va visualiser la fonction de répartition empirique avec Gnuplot. Dans un *autre* terminal, placez-vous dans le répertoire où est stocké le fichier `loinormalesort.dat`, et lancez Gnuplot.
Exécutez la commande :
`plot 'loinormalesort.dat' using 1:0 with line`
Comme `using 1:0` précise d'inverser abscisse et ordonnée, vous obtenez la fonction de répartition empirique, multipliée par le nombre de tirages… Êtes-vous d'accord ?
Si votre simulation est basée sur 100 tirages, vous pouvez normaliser en exécutant :
`plot 'loinormalesort.dat' using 1:($0)/100 with line`

On va superposer à ce graphe celui de la fonction de répartition de $\mathcal{N}_{(\mu,\sigma)}$. La *fonction d'erreur* (erf) est définie par :

$$\forall\, x \in \mathbb{R},\ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt.$$

En utilisant $\lim_{x\to+\infty} \text{erf}(x) = 1$, montrez que la fonction de répartition associée à la loi normale de paramètres (μ, σ^2) vaut en $x \in \mathbb{R}$:

$$\frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sqrt{2}\,\sigma}\right)\right).$$

Sous Gnuplot, définissez la fonction f par :
`f(x)=(1+erf((x-mu)/(sqrt(2)*sigma)))/2`
et spécifiez les valeurs : `mu=5` et `sigma=2.3` (par exemple).

Affichez les deux graphes superposés par :
`plot 'loinormalesort.dat' using 1:($0)/100 with line, f(x)`

Vous pouvez afficher une grille en faisant précéder l'appel à `plot` par : `set grid`.

Expérimentez avec différents nombres de tirages générés par votre programme C.

3.4 Étude expérimentale du mouvement d'une particule

La propriété suivante permet de générer les réalisations d'une variable aléatoire suivant une loi exponentielle : si U suit une loi uniforme sur [0,1] alors $X = -\log(U)/\lambda$ suit une loi exponentielle de paramètre λ (rappel : $\mathbb{E}(X) = 1/\lambda$ et $\text{Var}(X) = 1/\lambda^2$).

Exercice 9 (loi exponentielle)

Écrivez une fonction qui génère des `double` suivant une loi exponentielle dont le paramètre est passé en argument.

On peut modéliser le parcours d'une particule dans un milieu (plan) par un mouvement brownien comme suit. Entre deux chocs avec les particules du milieu, la particule parcourt une distance dont la loi est exponentielle ($1/\lambda$ correspondant au *libre parcours moyen*), avec une orientation uniformément distribuée sur $[0, 2\pi]$.

On va vérifier expérimentalement la propriété suivante : la distance quadratique moyenne des particules à leur point de départ est asymptotiquement proportionnelle à la racine carrée du temps de parcours. Ce point, relatif aux phénomènes de diffusion, intervient dans l'un des trois articles célèbres d'Albert Einstein publiés en 1905.

Exercice 10 (mouvement d'une particule)

En utilisant une fonction pour simuler la loi exponentielle et une autre pour simuler la loi uniforme sur $[0, 2\pi]$, écrivez un programme qui calcule les T positions successives d'une particule. À chaque étape, vous afficherez les résultats sous la forme : `printf("%f  %f \n",posx,posy);`

Pour visualiser un parcours de la particule, on peut utiliser l'utilitaire gnuplot. En exécutant le programme par : `./nom_programme.exe > particule.dat` on redirige l'affichage vers le fichier `particule.dat`. Dans un *autre* terminal, placez-vous dans le répertoire où est stocké ce fichier, et lancez gnuplot. Exécutez la commande : `plot 'particule.dat' with line`

Recommencez avec différentes valeurs du temps de parcours T.

Si X_t est la variable aléatoire correspondant à la distance à l'origine d'une particule au temps t, la distance quadratique moyenne à l'origine est : $d_t = \sqrt{E(X_t^2)}$.

Modifiez votre programme pour calculer une approximation de cette distance à chaque pas de temps (pensez à la loi des grands nombres...). On pourra stocker $(d_t)_{t\in\{0,\dots,T\}}$ dans un tableau de taille $T + 1$.

Finissez votre programme en affichant les éléments de (d_t) par une boucle sur `printf("%f \n",d[t]);` et visualisez le résultat à l'aide de gnuplot. Vous pouvez aussi visualiser le carré de d_t pour mettre en évidence une relation linéaire entre d_t^2 et t...

3.5 Générateur Linux `urandom`

On va expérimenter un générateur de nombres aléatoires différent de la fonction `rand()` fournie par le langage C.

Le noyau linux comporte un générateur aléatoire basé sur l'observation de propriétés physiques difficilement prédictibles sur l'ordinateur, comme le temps entre l'appui sur les touches du clavier, la variation de la vitesse de rotation du disque dur, des événements sur le réseau...Le résultat est stocké dans le fichier `/dev/random`. Néanmoins, le nombre de bits aléatoires obtenus par cette méthode est limité par l'activité sur la machine, donc on ne peut pas se contenter de `/dev/random` si on a besoin d'une grande quantité de nombres aléatoires. Dans ce cas on utilise le fichier `/dev/urandom` qui complète par des bits pseudo-aléatoires.

Pour plus d'informations, tapez : `man 4 random` dans un terminal.

Exercice 11 (générateur aléatoire matériel)

La fonction suivante renvoie un entier aléatoire entre 1 et 6 en lisant le contenu du fichier `/dev/urandom`.

```
int de_modifie(void){
  FILE * f;
  int random;
  if ((f = fopen("/dev/urandom","r")) == NULL) {
    perror("probleme fopen");
```

```
    exit(1);
  }
  fread(&random, sizeof(random), 1, f);
  fclose(f);
  return 1+abs(random)%6;
}
```

Modifiez votre programme de simulation d'un dé pour utiliser ce générateur aléatoire (limitez le nombre d'itérations à moins de 100 000 pour conserver un temps d'exécution raisonnable). Est-ce satisfaisant ? Ce nouveau dé passe-t-il le test du χ^2 ?
