TP1 : Génération de nombres aléatoires, de variable aléatoire et d'événements

L'objectif de ce TP est de vous familiariser avec la génération de nombres et variable aléatoire en utilisant les fonctions `srandom()` et `random()` de la librairie C. Les nombres aléatoires sont souvent nécessaires pour la génération d'événements et donc pour la simulation d'un système quelconque. Les variables aléatoires permettent de modéliser et donc de simuler la variabilité des valeurs de certains paramètres du système. Par exemple, le nombre de programmes actifs en mémoire, la taille de pages web, le temps de transmission de paquets, le délai séparant l'arrivée de deux paquets successifs (délai d'inter-arrivée), etc. Certaines parties des codes sources des programmes que vous allez implanter vous sont données afin de vous aider.

N.B. La rédaction d'un compte rendu final suivra la réalisation des TPs. Ce compte rendu consignera les codes sources des programmes, les réponses aux questions, les valeurs numériques et les courbes.

Le rapport devra être remis sous format électronique en pdf seulement.

- ***Le fichier est nommé suivant le format SEV18_TP1_NOM1_NOM2.PDF***
- ***Sur la page de garde doivent figurer vos prénoms, noms, numéros d'étudiant et le numéro ainsi que l'intitulé du TP.***
- ***Reportez dans votre rapport le numéro de la question où du test à effectuer. Toutes les questions de ce TP nécessitent une réponse (généralement une ou quelques phrases suffisent) avec une capture d'écran uniquement si nécessaire.***
- ***Les codes sources et/ou commandes Shell doivent être commentés correctement. Ils doivent être écrits en police à pas fixe (par exemple* `Courier`*)***

Les rapports qui ne vérifient pas ces consignes seront pénalisés. En cas de plagiat, les étudiants concernés auront la note globale de 0/20 .

Il est à rendre au plus tard 15 minutes après le TP par email à l'adresse David.Cordova@lip6.fr

A. Génération de nombres aléatoires entiers

A.1 Lancement de dés

On souhaite simuler le lancement de dés. Cela consiste donc à générer deux nombres aléatoires entiers entre 1 et 6. Complétez le code suivant qui génère ces deux nombres et les affiche à l'écran. Utilisez les fonctions `srandom()` et `random()` pour initialiser la graine du générateur et ensuite pour générer les nombres entre 1 et 6. La fonction `random()` génère directement un nombre aléatoire entre 0 et `RAND_MAX`. Ce dernier est une constante définie dans le fichier `stdlib.h`. Il s'agit du plus grand nombre entier supporté. C'est aussi la valeur maximale que peut retourner la fonction `random()`. Consultez les pages du manuel, `man random()` et `man rand()`, pour plus d'informations.

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char** argv) {

  // Déclaration de variables
  int seed, d1, d2;


  // Récupération des paramètres du programme
  // Premier paramètre : la graine pour la génération des
  // nombres pseudo-aléatoire (seed)
  if (argc < 2) {
    fprintf(stderr, "usage: %s <seed>\n", argv[0]);
    exit(-1);
  };
  seed = atoi(argv[1]);

  // Initialisation (à compléter)


  // Lancement de dés : une fois (à compléter)

  d1 = ; // Premier dé
  d2 = ; // Deuxième dé

  printf("%d %d\n", d1, d2);

}; //end main
```

Remarque : La méthode de génération qui se base sur l'expression `random() % 6` est déconseillée.

1. Pour compiler le programme : `gcc -o lancer lancer.c`. S'il y a des warnings, il faut aussi les supprimer en corrigeant votre programme.
2. Exécutez le programme plusieurs fois sans changer la graine (le seed). Que remarquez-vous ? Est-ce que le résultat obtenu est correcte.
3. Exécutez le programme plusieurs fois en changeant cette fois-ci la graine à chaque exécution. Commentez le résultat.

A.2 Lancement de dés plusieurs fois

Maintenant, nous allons lancer les dés `n` fois au lieu d'une seule fois. Pour ce faire, on rajoute un deuxième paramètre d'entrée `n` correspondant au nombre de lancements visé. On encadre la génération des deux nombres aléatoires et leur affichage avec une boucle de 1 à n :

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char** argv) {

  // Déclaration de variables
  int seed, d1, d2;
  int i, n;

  // Récupération des paramètres du programme
  // Premier paramètre : la graine pour la génération des
  // nombres pseudo-aléatoire (seed)
  // Deuxième paramètre : le nombre de lancements de dés
  if (argc < 3) {
    fprintf(stderr, "usage: %s <seed> <n>\n", argv[0]);
    exit(-1);
  };
  seed = atoi(argv[1]);
  n = atoi(argv[2]); // On récupère aussi n

  // Initialisation (à compléter)


  // Lancement de dés : n fois (à compléter)
  for (i=1; i<=n; i++) {
    d1 = ; // Premier dé
    d2 = ; // Deuxième dé
    printf("%d %d\n", d1, d2);
  };

}; //end main
```

Exemple d'affichage pour n = 10 :

```
bash$ ./lancer 12345 10
2 3
1 2
1 1
2 1
2 2
3 2
1 3
4 1
1 6
bash$
```

1. Utilisez la valeur 12345 pour la graine. Lancez les dés 1 fois (n=1), calculez le nombre de fois où le double six est obtenu (normalement 0, mais vous pouvez avoir 1 si le double six apparait du premier coup !) Ensuite, lancez les dés 10 fois (n=10), et calculez aussi le nombre de double six. Continuez avec n=100, 1000, 10000, 100000, 1000000, 10000000 et 100000000. Vous remarquerez que le nombre de double six commence toujours par 27 ou 277 pour les grandes valeurs de n. Pourquoi ?

Remarque : Vous pouvez utiliser les commandes unix `grep` et `wc -l` pour compter le nombre d'apparition du double six.

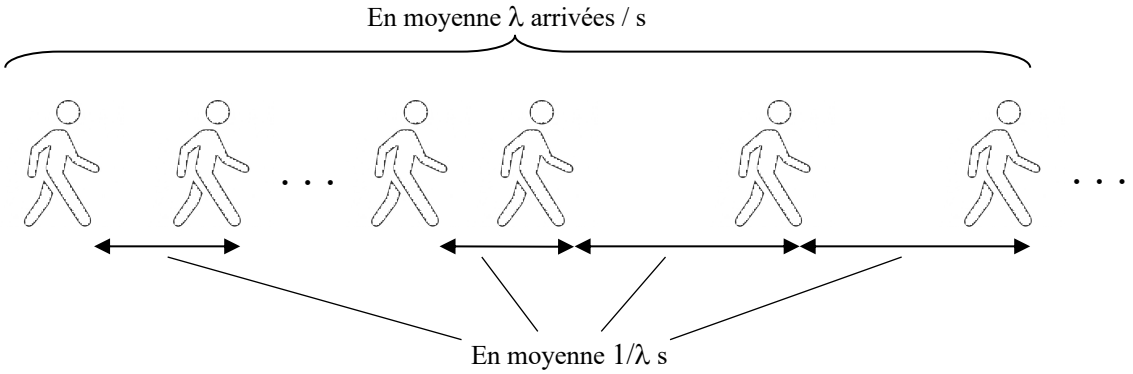
2. Refaites la même chose avec la graine 99658. Y a-t-il une différence avec le test précédent utilisant la graine 12345 ? Pour conclure, vers quelle valeur le nombre de double six converge-t-il exactement ?

3. Le résultat serai-il différent avec le double 1 ? Avec la réalisation '2 6' ?

B. Introduction à la simulation à événements discrets

B.1 Génération de variable aléatoire

Dans cette partie, nous allons simuler des événements d'arrivée. Cela peut correspondre à l'arrivée de paquets à un routeur, l'arrivée d'appels téléphoniques à une station de base, l'arrivée de clients devant un guichet, etc. Nous supposons que le délai entre deux arrivées successives suit la loi exponentielle. Cela revient à dire de façon équivalente que le processus d'arrivée est un processus de Poisson. Le paramètre λ de l'exponentielle correspond au nombre moyen d'arrivées par seconde, c'est-à-dire au taux moyen d'arrivée (λ arrivées/seconde). $1 / \lambda$ correspond à la durée moyenne d'inter-arrivée en seconde (voir figure suivante).

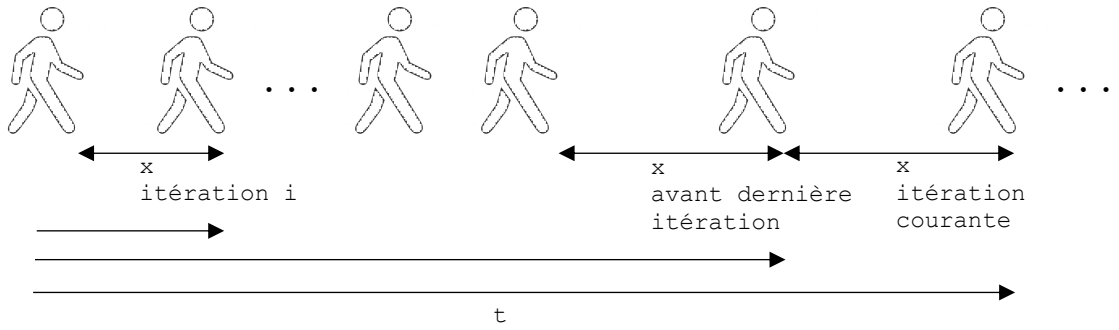


En s'inspirant du programme précédent, écrivez un autre programme afin de générer n valeurs aléatoires suivant la loi exponentielle (le type à utiliser est '`double`'). Récupérez le taux λ comme troisième paramètre d'entrée du programme. Dans la boucle de génération, générez d'abord une valeur aléatoire entre 0 et 1 (`u`), ensuite une valeur `x` suivant la loi exponentielle à partir de `u`. Pour l'affichage à l'écran suivez le modèle suivant :

```
printf("a %f %d\n", t, nb);
```

La première colonne donne une indication de l'événement : la lettre a pour arrivée.
La deuxième colonne représente le temps d'occurrence, c'est-à-dire quand l'événement se produit, autrement dit l'instant d'arrivée. Cet instant est obtenu en additionnant la valeur courante générée avec toutes les valeurs précédentes (`t = t + x` dans la boucle, `t` initialisé à `0`, voir figure ci-dessous).

La troisième colonne donne le nombre total d'arrivées à l'instant t. Il est obtenu en utilisant un compteur à incrémenter à chaque génération et à initialiser à 0 avant la boucle de génération.



1. Fixez la graine à 12345 et n à une grande valeur. Testez votre programme avec $\lambda = 5$ et $\lambda = 6.42$. Pour les deux tests, calculez le taux moyen d'arrivée et vérifiez qu'il est presque égale à λ. Si le taux moyen calculé n'est pas égal à λ alors la génération est incorrecte, et il faut donc revoir votre code.

2. Redirigez l'affichage du programme avec **$\lambda = 5$** dans un fichier nommé par exemple arrivees. Ensuite, obtenez et observez l'évolution des arrivées en fonction du temps pendant les 5 premières secondes, c'est-à-dire `nb` en fonction de `t` pour `t` entre `0` et `5`, en tapant :
   `gnuplot` ensuite
   ```
   plot [0:5] [0:50] "arrivees" using 2:3
   ```

B.2 Génération de départs et création d'un fichier d'événements

Supposons que les arrivées générées dans le test précédent correspondent à des clients arrivant et faisant la queue devant un guichet. Quand une personne arrive devant le guichet, il lui faut exactement S secondes pour être servi avant de partir et quitter la file d'attente. Ainsi, le guichetier sert une personne de la file d'attente pendant exactement S secondes, ensuite il passe à la personne suivante. Donc, les instants de départ *potentiel* des clients peuvent être obtenus avec une simple variable `td` incrémentée de `S` à chaque itération dans une boucle, et initialisée à la valeur de l'instant de la première arrivée (première ligne du fichier des arrivées). Il n'y a donc pas de génération de variable aléatoire dans ce cas puisque nous considérons des départs déterministes.

Générez un fichier de départ avec un autre programme C dans lequel vous suivez le format d'affichage suivant :

```
printf("d %f %d\n", t, nb);
```

La première colonne donne une indication de l'événement : la lettre d pour départ.
La deuxième colonne représente le temps d'occurrence, c'est-à-dire quand l'événement se produit, autrement dit l'instant de départ.
La troisième colonne donne le nombre total de départs à l'instant `t`. Il est obtenu en utilisant un compteur à incrémenter à chaque départ et à initialiser à 0 avant la boucle de génération des départs.

1. Redirigez l'affichage du programme précédent avec n = 100000 et **S = 0.3** dans un fichier nommé par exemple departs. Ensuite, obtenez et observez l'évolution des départs en même temps que les arrivées en fonction du temps pendant les 50 premières secondes en tapant :
   `gnuplot` ensuite
   ```
   plot [0:50] [0:200] "arrivees" using 2:3, "departs" using 2:3
   ```

2. Pour avoir un fichier regroupant ensemble les événements d'arrivée et de départ utilisez les deux commandes suivantes :

   ```
   cat arrivees departs  > events # pour concaténer les deux fichiers
   sort -k2,2g events > sevents   # pour trier les événements dans l'ordre de
                                  # leur occurrence.
   ```

   Le fichier `sevents` est un peu similaire au résultat obtenu par certains logiciels de simulations. Le fichier des événements permet de mesurer certains paramètres du système. Par exemple en parcourant le fichier ligne par ligne et en incrémentant un compteur à chaque événement 'a', et en décrémentant ce compteur à chaque événement 'd', on obtient l'évolution du nombre de clients dans la file d'attente en fonction du temps (Ne le faites pas).

3. Calculez en examinant le fichier des événements `sevents`, sans écrire un programme, le nombre de clients en attente dans la file à l'instant 10 secondes. Expliquez.

4. A votre avis, est-ce que cette méthode de simulation est correcte dans tous les cas ? Si non, donnez un cas où la simulation est fausse, autrement dit un cas où les événements obtenus et leur instant d'occurrence sont faux, et donc ne reflètent pas le système réel.

Remarques à lire :

Cette méthode de simulation peut être utile dans certains cas où on veut effectuer une simulation particulière sans passer par un logiciel de simulation. Cette méthode est impraticable dans le cas général et notamment pour simuler des systèmes complexes.

Normalement, pour que la simulation soit correcte, il faut ***ordonnancer*** les événements ***au fur et à mesure*** en ***mémoire*** et non dans un fichier. Autrement dit, il faut ordonnancer et trier les événements dans le bon ordre correctement au fur et à mesure et non « offline ». La gestion des événements doivent être faite « online » pendant la génération de ces mêmes événements.

Les simulateurs et en particulier le simulateur OMNET++ offre des outils et une interface graphique ergonomique et puissante permettant de générer les événements et de les gérer correctement, de façon très maîtrisée. Néanmoins, le *principe* de génération d'événements est le même vu dans ce TP.