Modélisation et Outils Mathématiques
TP génération de nombres aléatoires et probabilités

3IF, INSA de Lyon, 2012/2013
Marine Minier, Irène Gannaz

INSA-Lyon, Département Informatique
version 2.00, 12 Février 2013 – rédigé avec LaTeX

1 Introduction

Les objectifs de ce TP sont les suivants :
- Faire passer des tests statistiques à des générateurs pseudo-aléatoires (via trois tests de qualité de séquences) afin de comparer la qualité des séquences produites. Vous aurez à programmer vous-même deux générateurs simples.
- Transformer la distribution uniforme sur $[0, 1]$ en une distribution plus complexe.
- Appliquer l'une de ces méthodes pour modéliser un problème classique de file d'attente.
- Etudier la validité de résultats théoriques sur des simulations, dans le cadre d'une file d'attente.

La première séance devrait porter sur le premier ou les deux premiers points. La deuxième séance consistera en la modélisation et l'étude de files d'attentes, qui constituent les deux derniers points.

L'ensemble des codes durant ces deux séances sera développé en langage C et compilé via l'émulateur de Linux cygwin sous Windows. Tous les fichiers sont disponibles sur `servif-home/fic_eleves/Espace Pedagogique/Modeles et Outils Mathematiques/Probabilites Statistiques/PROBA_Generation_Nb`.

A la fin des deux séances, vous devrez envoyer à l'adresse mail `tpananum@gmail.com` :
- Un compte rendu "papier" au format .pdf qui contiendra pour la première séance de TP les résultats **commentés** des tests obtenus sous forme de graphiques et de tableaux ; et pour la deuxime séance de TP les réponses aux questions sous forme de tableaux et d'histogrammes ainsi que des **commentaires** sur ces résultats.
- Les codes C que vous avez développés pendant les séances. Le `main` de tous vos codes devra suivre le format de celui décrit en Annexe B. Les sorties de vos programmes devront être faites dans un unique fichier appelé ``NumeroBinome.txt''.
- Votre compte-rendu devra s'appeler `NumeroBinome.pdf`, votre main C aussi. Votre numéro de

binôme devra également apparaître dans l'entête du mail envoyé.

2 Tests de générateurs pseudo-aléatoires

Le but de ce TP est de tester la qualité des séquences aléatoires produites par différents générateurs aléatoires. Nous étudierons les générateurs suivants :

1. les 4 bits de poids forts produits par la fonction `rand` du C,
2. les 4 bits de poids faibles produits par la fonction `rand` du C,
3. le générateur de Von Neumann,
4. le générateur Mersenne-Twister,
5. la transformation de l'AES en générateur pseudo-aléatoire.

Les générateurs de Von Neumann, Mersenne-Twister et l'AES ne seront pas décrits ici. Nous renvoyons respectivement à [Wika], [Wikb] et [FIP01] pour une description détaillée.

Les codes pour les générateurs de Von Neumann, Mersenne-Twister et l'AES vous sont fournis mais vous aurez à implémenter les deux générateurs utilisant la fonction `rand` du C. Les fichiers `.c` et `.h` fournissant les générateurs utilisés dans ce TP (Von Neumann, Mersenne-Twister, AES) sont décrits dans l'Annexe A.

Afin de comparer ces générateurs, vous aurez ensuite à programmer des tests classiques de probabilité (voir Section 2.2) permettant de tester la qualité des suites produites. La qualité d'une suite sera mesurée par la probabilité que les valeurs obtenues suivent bien une loi uniforme comme cela est souhaité.

2.1 Définition d'un générateur aléatoire

Dans beaucoup d'applications informatiques (la simulation, les jeux vidéos, la cryptographie, etc.), il est nécessaire de tirer des nombres au hasard pour initialiser différents algorithmes. Pour cela, on utilise ce qu'on appelle des nombres **pseudo-aléatoires** pour souligner leur différence par rapport aux véritables suites de variables aléatoires indépendantes identiquement distribuées. Nous nous intéressons ici à quelques méthodes classiquement utilisées pour générer des nombres pseudo-aléatoires (pour plus de détails, se référer par exemple à [Ber05a]). Plus précisément, il existe deux types de générateurs :

– les générateurs à sorties imprédictibles : ils génèrent des nombres pseudo-aléatoires dont on ne peut prévoir la valeur,
– les générateurs à sorties prédictibles : dans ces générateurs, on initialise (on dit nourrir) l'algorithme à partir d'un nombre connu appelé graine et le générateur produira toujours la même suite s'il est initialisé avec la même valeur.

2.1.1 Définition

Un générateur pseudo-aléatoire est une structure $G = (S, \mu, f, U, g)$ avec :

– S un (grand) ensemble fini d'états,
– μ est une distribution de probabilité sur S (le plus souvent la loi uniforme),
– $f : S \to S$ est la fonction de transition utilisée pour passer d'un état S_i au suivant S_{i+1}.

– U est l'ensemble image de la fonction $g : S \to U$ qui fait correspondre à chaque état S_i un échantillon de U. Très classiquement, on va avoir $U = [0, 1]$ pour une loi uniforme sur [0,1].

Le générateur fonctionne donc en itérant la fonction f à partir d'un état initial S_0 appelé graine, choisi par l'utilisateur. En notant $(X_n)_{n\geq 1}$ la liste des valeurs successives produites par le générateur, on a la relation $X_n = g(f \circ \cdots \circ f(S_0))$ où f est appliquée n fois.

En pratique, le générateur ne garde en mémoire que l'état courant $S_n \in S$, initialisé à S_0 et mis à jour lors de chaque appel de f : $S_n = f(S_{n-1})$, la valeur renvoyée étant égale à $X_n = g(S_n)$.

Le choix de la graine S_0 détermine donc entièrement la suite de nombres pseudo-aléatoires produite par le générateur car une fois cette graine choisie, le comportement de la suite est complétement déterministe.

2.1.2 La fonction `rand` du C

Vous avez déjà sûrement utilisé la fonction `rand` du C dans vos programmes. Cette fonction est disponible dans la librairie standard ; elle est supposée renvoyer à chaque appel un nombre aléatoire compris entre 0 et RAND_MAX (RAND_MAX vaut en général $2^{15} - 1$). La fonction `rand` est en fait écrite ainsi :

```
int rand () // RAND_MAX assumed to be 32767
{
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % 32768;
}
```

Il s'agit donc ici d'un appel direct à la structure d'un générateur pseudo-aléatoire telle que définie dans la Section 2.1 où la fonction de transition d'un état au suivant est une relation affine. Plus précisément, il s'agit ici d'un générateur linéaire congruentiel qui se définit de la manière générique suivante :

$$S = U = \{0, \cdots, m-1\}, \quad f(S_n) = a * S_{n-1} + b \mod m, \quad X_n = g(S_n) = s/m$$

où / est la division entière. Lorsque la fonction f utilisée par le générateur s'écrit sous cette forme, on dit qu'il s'agit d'un générateur à congruence linéaire (GCL).

Si vous utilisez la fonction `rand` directement dans vos programmes sans changer la graine vous allez retomber systématiquement sur la même suite de nombres et ainsi produire une suite prédictible. Pour produire une suite imprédictible et initialiser la graine "aléatoirement", on emploie généralement la fonction `srand` :

```
void srand(unsigned int seed)
{
  next = seed;
}
```

que l'on "nourrit" l'aide d'un nombre difficilement prévisible et qui varie rapidement. Par exemple, vous pouvez prendre le nombre de cycles utilisés par votre processeur depuis son démarrage. Il peut être obtenu, sur les processeurs x86, avec la commande assembleur `rdtsc`. On peut par exemple utiliser la fonction suivante :

```
int rdtsc()
{
  __asm__ __volatile__("rdtsc");
}
```

puis appeler `srand(rdtsc())` pour initialiser la graine du générateur de façon “aléatoire”.

Par ailleurs, la “qualité aléatoire” des bits de poids faible produits par la fonction `rand` est mauvaise. Ceci signifie que le caractére aléatoire uniforme des valeurs des bits de poids faibles retournées par cette fonction n'est pas satisfaisant. Dans la mesure du possible, mieux vaut garder les bits de poids fort qui ont un meilleur comportement. Par exemple, si vous souhaitez simuler une loi uniforme sur un intervalle du type $[1, N]$ avec N plus petit que RAND_MAX., vous aurez intérêt à programmer plutôt ceci

```
alea = 1 + (int) ($N$*rand()/(RAND\_MAX+1.0));
```

que ceci

```
alea = 1+(rand() \% $N$);
```

Nous reviendrons dans la suite de ce document sur ce qu'est un “bon aléa” et comment on détermine sa qualité.

Il existe beaucoup de méthodes pour générer des nombres pseudo-aléatoires autres que la congruence linéaire sur laquelle repose la fonction `rand`. On peut citer les générateurs à récursivité multiple (GRM), l'algorithme Blum-Blum-Shub, etc. Pour plus de détails, le lecteur peut se référer à [Ber05a] ou à [Wika].

2.1.3 Les autres générateurs pseudo-aléatoires de ce TP

Les autres générateurs pseudo-aléatoires utilisés dans ce TP sont respectivement :

- **Le générateur de Von Neumann.** C'était l'un des premiers algorithmes proposés pour générer des nombres aléatoires. Malheureusement, il n'est pas de bonne qualité, c'est-à-dire que les nombres produits ne passent pas un certain nombre de tests permettant d'être sûr de la qualité des séquences produites.
- **Le générateur Mersenne-Twister.** Ce générateur engendre des séquences pseudo-aléatoires de qualité satisfaisante. Il est très souvent utilisé bien qu'il ne soit pas cryptographiquement sûr.
- **Le générateur AES.**

2.2 Qualité de la séquence produite par un générateur pseudo-aléatoire

Il existe beaucoup de tests permettant de s'assurer de la qualité des nombres aléatoires produits (voir les suites de tests complètes du NIST [oST] ou de DIEHARD [Mar95]). Nous ne les programmerons pas tous, nous allons nous focaliser sur trois tests particuliers que nous appliquerons ensuite sur les cinq générateurs décrits précédemment. Le cours de statistique de quatrième année vous donnera plus de détails sur le principe d'un test statistique. On pourra aussi se référer à [Ber05b] ou [Can06] pour voir les nombreux tests statistiques qui existent.

On vous demande donc d'implémenter en C les tests présentés dans la suite de ce document. Plus précisément, il s'agira de tester tous les générateurs :

– pour le premier test, sur une séquence de $n = 1024$ valeurs.
– pour les deux derniers tests sur une séquence de $n = 1024$ et pour 20 initialisations différentes.

2.2.1 Test Visuel

Question 1. Tracez, pour chacun des cinq générateurs, le graphique des sorties observées pour une suite de $n = 1024$ valeurs. Que constatez-vous ? Expliquez.

Les tableurs classiques types Excel ou Open Office ne savent pas tracer de représentations graphiques adaptées à la visualisation de données telles que celles que vous avez générées. Afin de vous évitez un pré-traitement de vos sorties, nous vous conseillons d'utiliser le logiciel Matlab, qui propose une fonction `hist` traçant des histogrammes.

2.2.2 Test de fréquence monobit

Principe du test : Le but de ce test est de s'intéresser à la proportion de zéros et de uns dans une séquence entière. On teste donc si le nombre de uns et de zéros d'une séquence sont approximativement les mêmes comme attendu dans une séquence vraiment aléatoire.

Définition de la fonction à implémenter : la fonction à implémenter devra s'écrire : `double Frequency(int` n`)` où n est la longueur de la séquence observée (vous pouvez également prendre comme paramètre d'entrée le tableau stockant les valeurs à tester). Elle effectue les opérations suivantes sur la séquence de bits $(\epsilon) = \epsilon_1 \cdots \epsilon_n$:

– Conversion en +1 ou -1 : les zéros et uns de la séquence d'entrée (ϵ) sont convertis en -1 pour 0 et 1 pour 1. Cela revient à réaliser l'opération $X_i = 2\epsilon_i - 1$. Ces valeurs sont ensuite additionnées : $S_n = X_1 + \cdots + X_n$. Par exemple, la séquence $\epsilon = 1011010101$ avec $n = 10$ devient $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

– Calcul de s_{obs} : $s_{obs} = \frac{|S_n|}{\sqrt{n}}$. Pour l'exemple précédent, on obtient : $s_{obs} = \frac{|2|}{\sqrt{10}} = 0.632455532$.

  Si nous avons bien indépendance dans la séquence de bits, alors le théorème de la limite centrale assure que pour n grand, $\frac{|S_n|}{\sqrt{n}}$ est une variable aléatoire qui suit approximativement une loi normale $\mathcal{N}(0, 1)$. L'idée est alors de regarder la valeur de la fonction de répartition de la loi $\mathcal{N}(0, 1)$ pour la valeur s_{obs} observée. Cette valeur, appelée P_{valeur} donne la probabilité d'avoir bien observé s_{obs} lorsqu'on a la loi $\mathcal{N}(0, 1)$. Ainsi, si cette P_{valeur} est petite, cela signifie qu'il était improbable d'avoir obtenu s_{obs}, donc qu'a priori le théorème de la limite centrale ne peut pas s'appliquer. Ceci signifie que l'indépendance dans la suite de bits n'est pas vérifiée.

  Le calcul de la P_{valeur} en C peut se faire ainsi : $P_{valeur} = \text{erfc}(\frac{s_{obs}}{\sqrt{2}})$ où `ercf()` est une fonction incluse dans `math.h` (voir également Appendice C). Pour notre exemple, on obtient $P_{valeur} = \text{erfc}(\frac{0.632455532}{\sqrt{2}}) = 0.527089$.

Règle de décision à 1% : Au vu du principe décrit ci-dessus, plus la P_{valeur} est petite plus on peut rejetter de manière sûre le fait que le séquence est aléatoire. En pratique, si la P_{valeur} calculée est inférieure à 0.01 alors la séquence n'est pas aléatoire. Sinon, on peut conclure qu'elle l'est au sens de ce test. Dans l'exemple précédent, comme $P_{value} = 0.527089$, on peut conclure que la séquence est aléatoire au sens de ce test. Il est recommandé que chaque séquence testée fasse au minimum

100 bits (i.e. $n \geq 100$) afin que l'application du théorème de la limite centrale ait un sens.

Question 2. Implémentez la fonction décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

2.3 Test des runs

Principe du test : Le but de ce test est de s'intéresser à la longueur des suites successives de zéros et de uns dans la séquence observée. Il teste donc la longueur moyenne de ce qu'on appelle les "runs", i.e. les suites consécutives de 0 ou de 1. Il s'appuie sur la propriété suivante :

Propriété 1 *Soit $s = (s_i)_{i\in\mathbb{N}}$ une suite de période T sur l'alphabet $\mathcal{A}$. On appelle run de longueur k un mot de longueur k constitué de symboles identiques, qui n'est pas contenu dans un mot de longueur $k+1$ constitué de symboles identiques, i.e. $(s_i, \cdots, s_{i+k-1})$ est un run de longueur k si : $(s_i = \cdots = s_{i+k-1})$ et $(s_{i-1} \neq s_i)$ et $(s_{i+k} \neq s_{i+k-1})$. La séquence s possède la propriété des runs si le nombre $N(k)$ de runs de longueur k sur une période T vérifie : $\lfloor \frac{T\cdot(|A|-1)^2}{|A|^{k+1}} \rfloor \leq N(k) \leq \lceil \frac{T\cdot(|A|-1)^2}{|A|^{k+1}} \rceil$.*

Définition de la fonction à implémenter : La fonction à implémenter devra s'écrire : `double Runs(int n)` où n est la longueur de la séquence de bits observés. Elle effectue les opérations suivantes sur la séquence de bits $(\epsilon) = \epsilon_1 \cdots \epsilon_n$:

- Pre-test : Calculer la proportion de 1 dans la séquence observée : $\pi = \frac{\sum_{j=1}^{n} \epsilon_j}{n}$. Par exemple, si $\epsilon = 1001101011$, alors $n = 10$ et $\pi = 6/10 = 3/5$.

- Déterminer si ce pre-test est passé ou non en vérifiant si $|\pi - 1/2| \geq \tau$ avec $\tau = \frac{2}{\sqrt{n}}$. Si oui, arrêter le test ici (dans ce cas, on renvoie $P_{valeur} = 0.0$). Sinon, continuer à l'étape suivante. Avec l'exemple précédent, on obtient : $\tau = 2/\sqrt{2} \approx 0.6346$ et $|\pi - 1/2| = 0.6 - 0.5 = 0.1 < \tau$, donc on continue à appliquer le test.

- Calculer la statistique $V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$ avec $r(k) = 0$ si $\epsilon_k = \epsilon_{k+1}$ et $r(k) = 1$ sinon. En reprenant l'exemple précédent, on obtient : $V_{10}(obs) = (1+0+1+0+1+1+1+1+0)+1 = 7$.

- On calcule alors la P_{valeur} comme étant :

$$P_{valeur} = \text{erfc}(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}).$$

  Si on reprend l'exemple précédent, on obtient : $P_{value} = \text{erfc}(\frac{|7 - \frac{3}{5}(1-\frac{3}{5})|}{2\sqrt{2\cdot 10}\cdot(3/5)\cdot(1-(3/5))}) \approx 0.8359$.

Règle de décision à 1% : Si la P_{valeur} calcule est inférieure à 0,01 alors on peut conclure que la séquence n'est pas aléatoire. Dans l'exemple précédent avec une P_{valeur} égale à 1,02, on peut donc conclure que la séquence l'est. Il est recommandé que chaque séquence testée fasse au minimum 100 bits (i.e., $n \geq 100$).

Question 3. Implémentez la fonction décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

3 Simulation d'une loi de probabilité exponentielle

Nous souhaitons ici simuler une loi exponentielle afin de l'appliquer à un cas usuel de file d'attente. Nous procédons ici en deux étapes : dans un premier temps nous vous demandons d'implémenter une fonction simulant des observations issues d'une loi uniforme sur [0, 1] puis dans un second temps de l'exploiter pour simuler une loi exponentielle.

3.1 Loi uniforme sur [0 ; 1]

Comme spécifié en annexe, la majorité des techniques de simulations de lois de probabilité reposent sur la simulation préalable d'une loi uniforme sur l'intervalle [0, 1]. C'est pourquoi nous vous demandons à l'aide d'un des générateurs pseudo-aléatoires étudiés précédemment d'implémenter une fonction simulant une telle loi.

On pourra remarquer que si U suit une loi uniforme sur l'intervalle [0, N] alors U/N suit une loi uniforme sur l'intervalle [0, 1].

Définition de la fonction à implémenter : La fonction à implémenter devra s'écrire : `double Alea()`. Elle retourne une réalisation issue d'une loi uniforme sur [0, 1].

Question 4. Implémenter la fonction simulant une réalisation de loi uniforme sur l'intervalle unité décrite ci-dessus.

3.2 Loi exponentielle

Supposons que l'on veuille simuler une loi de probabilité de fonction de répartition F. Les propriétés des fonctions de répartition assurent que F est inversible. Notons F^{-1} son inverse. Soit U suivant une loi uniforme sur $[0, 1]$. Alors nous pouvons montrer que $X = F^{-1}(U)$ a pour fonction de répartition F.

Dans le cas d'une loi exponentielle de paramètre λ, la fonction de répartition vaut

$$F(x) = 1 - \exp(-\lambda x).$$

Par conséquent nous avons

$$F^{-1}(u) = -\ln(1-u)/\lambda.$$

Définition de la fonction à implémenter : la fonction à implémenter devra s'écrire : `double Exponentielle(double lambda)` où `lambda` est le paramètre de la loi exponentielle considérée. Cette fonction doit retourner une valeur issue de la réalisation d'une loi exponentielle de paramètre `lambda`.

Question 5. Implémenter la fonction simulant une réalisation de loi exponentielle décrite ci-dessus.

3.3 Simulations de lois quelconques

Nous ne vous demandons pas pour des raisons de temps de simuler ici d'autres lois de probabilité mais nous présentons en annexe quelques algorithmes classiques le permettant.

Question supplémentaire. Implémentez les algorithmes de simulation par rejet et par inversion comme spécifié dans l'annexe D.

4 Application aux files d'attentes

Nous ne considérerons ici que des files d'attentes dites PAPS (Premier Arrivé Premier Servi) ou FCFS (First Come first Served), ce qui signifie que les clients sont servis dans l'ordre d'arrivée. Il existe bien entendu des modèles sans cette hypothèse, mais nous ne les aborderons pas ici.

Le principe est que nous disposons de n serveurs répondant aux attentes des clients. Le but est d'étudier le nombre de clients en attente, le temps moyen d'attente, etc dans le système.

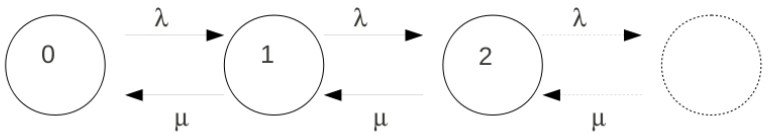
Notation de Kendall.
Nous désignerons une file d'attente par la notation A/B/m. La lettre A désigne la loi du temps écoulé entre deux arrivées et la lettre B la loi du temps nécessaire pour répondre au client pour un serveur. m représente le nombre de serveurs.

4.1 Files M/M/1

Notons T le temps écoulé entre deux arrivées de clients et D la durée de réponse du serveur. Une modélisation classique de ces durées consiste à considérer qu'elles suivent des loi exponentielles. Nous supposerons ainsi que T suit une loi $\mathcal{E}(\lambda)$ et que D suit une loi $\mathcal{E}(\mu)$. Nous supposerons de plus ces deux durées indépendantes. La loi exponentielle étant désignée par la lettre M (M pour Markovien), un tel modèle est appelé un modèle $M/M/1$.

Rappelons que les temps moyens respectivement d'attente et de service valent $1/\lambda$ et $1/\mu$. Le nombre d'arrivées de clients est alors un processus de Poisson et le nombre moyen d'arrivées par unité de temps vaut λ. Les paramètres λ et μ seront par la suite exprimés en minutes^{-1}.

Nous sommes alors amenés à étudier une Chaîne de Markov continue. Les états de cette chaîne correspondent au nombre de clients dans le système, dans la file d'attente ou en train d'être servi. On représente en général une telle chaîne par le schéma suivant :



Nous souhaiterions implémenter une fonction `FileMM1` qui retourne l'évolution du système au cours du temps dans un modèle $M/M/1$ pendant un intervalle de temps de durée D.

Cette fonction aura pour paramètres d'entrée `(double lambda, double mu, double D)` où
– `lambda` est le paramètre des loi exponentielle des arrivées,
– `mu` est le paramètre des loi exponentielle des départs,
– `D` est le temps d'observation de la chaîne,

Afin de construire cette fonction, nous vous suggérons de construire des tableaux stockant les temps auxquels partent et arrivent respectivement les clients. Comme vous ne savez pas combien de clients

vont transiter via le système, vous pouvez allouer de la mémoire de manière arbitraire, en prenant une grandeur suffisament élevée.

Ainsi, vous pouvez définir une variable globale dans l'entête de la fonction :

```
#define ARRAY_MAX_SIZE 1000
```

puis allouer la mémoire de vos tableaux à cette taille à l'aide de l'instruction `calloc`. Pensez ensuite à conserver la taille réellement utilisée dans vos tableaux.

Par ailleurs, si vous souhaitez récupérer ces tableaux (et leur taille), nous vous rappelons qu'il est possible en C de définir des structures, afin de manipuler plusieurs grandeurs simultanément.

Vous pouvez ainsi construire une structure

```
struct file_attente
{
    double *arrivee;
    int taille_arrivee;
    double *depart;
    int taille_depart;
};
typedef struct file_attente file_attente;
```

qui contient les heures d'arrivées et de départ des clients, puis construire une fonction retournant une variable de type `file_attente`, ayant pour argument les valeurs des paramètres λ et μ et le temps d'observation du système.

Définition de la fonction à implémenter : la fonction à implémenter sera ainsi de la forme `file_attente FileMM1(double lambda, double mu, double D)` avec :
– `lambda` est le paramètre des loi exponentielle des arrivées,
– `mu` est le paramètre des loi exponentielle des départs,
– `D` est le temps d'observation de la chaîne.
Si la fonction `FileMM1` retourne la variable `resultat`, celle-ci vérifiera :
– `resultat.arrivee` est le vecteur des dates d'arrivées de clients,
– `resultat.taille_arrivee` est le nombre de clients qui sont arrivés durant l'intervalle de temps `D`,
– `resultat.depart` est le vecteur des dates de départs de clients,
– `resultat.taille_depart` est le nombre de clients qui sont sortis du système durant l'intervalle de temps `D`.
Bien entendu, d'autres approches sont les bienvenues !

Question 6. Construire une fonction qui à partir des paramètres des lois exponentielles d'une file d'attente M/M/1 et du temps d'observation de la file retourne les dates d'arrivées et de sorties des clients du système, comme décrite ci-dessus.

Question 7. Construire une fonction qui à partir des dates d'arrivées et de sorties du système retourne l'évolution du nombre de clients dans le système.

Application : Prendre λ et μ tels qu'arrivent en moyenne 12 clients par heure et repartent en moyenne 20 clients par heure. Représentez l'évolution du nombre de clients dans le système pendant 3 heures de fonctionnement.

Même question avec 18 arrivées par heure en moyenne et le même taux de départ. Comparez les résultats obtenus.

Indication : On pourra éventuellement introduire une structure

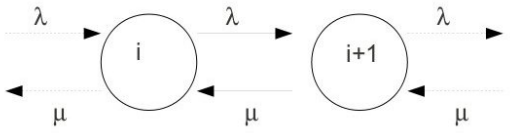
```
struct evolution
{
    double *temps;
    unsigned int *nombre;
};
typedef struct evolution evolution;
```

qui à chaque valeur t du tableau `temps` associe le nombre de personnes dans le système à partir de t dans le tableau `nombre`.

On pourra ensuite faire la représentation graphique sous Open Office Calc ou sous Matlab selon les préférences.

Remarquons que le nombre moyen d'arrivées dans un intervalle de temps t vaut alors λt et que le nombre de client qui part durant cet intervalle vaut en moyenne μt. L'intensité du trafic sur un serveur peut alors être mesuré par le rapport $\alpha = \frac{\lambda}{\mu}$. L'unité de mesure associée à cette grandeur est en général le *Erlang*.

Lorsque $\alpha > 1$, cela signifie qu'il y a en moyenne plus d'arrivées que de départs aux serveurs, donc que la file d'attente s'allonge et finira par saturer. Le cas qui nous intéresse est donc le cas $\alpha < 1$. Alors le système va se stabiliser ; on dit qu'il admet un régime stationnaire (voir votre cours de probabilités). Décrivons ce régime stationnaire :



Soit π_i la probabilité d'être dans l'état i. Alors nous avons $\lambda\pi_i = \mu\pi_{i+1}$, ou encore $\pi_{i+1} = \alpha\pi_i$. Nous reconnaissons une suite géométrique de raison $\alpha < 1$. La solution est $\pi_i = \alpha^i(1-\alpha)$.

Soit N le nombre de clients dans le système. Lorsque le système est stabilisé, N a pour loi $(\pi_1, \ldots, \pi_k, \ldots)$. Nous avons ainsi $\mathbb{E}(N) = \sum_i i\,\pi_i = \frac{\alpha}{1-\alpha}$.

De plus, dans une file M/M/1, il existe une relation fondamentale reliant les différentes grandeurs du système. Cette relation est la formule de Little. Soit W le temps durant lequel un client reste dans le système. Alors nous avons

$$\mathbb{E}(N) = \lambda\,\mathbb{E}(W).$$

Ceci signifie qu'en moyenne un client restant un laps de temps $\mathbb{E}(W)$ voit arriver derrière lui $\mathbb{E}(W)\lambda$ autres clients, avec λ fréquence d'entrée des clients.

Question 8. Estimez le nombre moyen de clients dans le système ainsi que le temps de présence d'un client dans le système (c'est-à-dire les temps moyens passés en attente et à être servi) après 3 heures de fonctionnement. Retrouvez-vous la formule de Little ? Commentez.

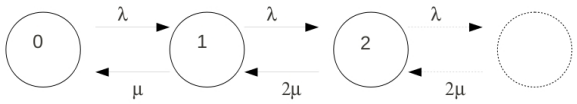
On répondra à cette question en prenant $\lambda = 0.20\,\mathrm{min}^{-1}$ et $\mu = 0.33\,\mathrm{min}^{-1}$.

Nous pouvons affiner un peu l'étude en étudiant aussi le temps passé à attendre dans la file avant d'être servi que nous noterons W_a ou le nombre moyen d'individus dans la file d'attente, non servis, que nous noterons N_a. Nous pouvons montrer que

$$\mathbb{E}(W_a) = \mathbb{E}(W) - 1/\mu \quad \text{et} \quad \mathbb{E}(N_a) = \lambda\,\mathbb{E}(W_a).$$

4.2 Files M/M/n

Considérons le cas où il y a n serveurs dans le système. Les lois d'arrivées et de départ seront de nouveau prise selon des lois exponentielles. Le modèle est alors noté $M/M/n$. Nous prendrons ici $n = 2$. Le schéma de la chaîne de Markov associée est le suivant :



Question 9. Qu'avez-vous à modifier dans vos codes précédents pour simuler un tel système ?

Simulez une file d'attente $M/M/2$ de paramètres $\lambda = 0.2\,\mathrm{min}^{-1}$ et $\mu = 0.33\,\mathrm{min}^{-1}$. Estimez le nombre moyen de clients dans le système et le temps moyen passé dans le système par un client sur une durée de fonctionnement de 3 heures. Comparez aux résultats obtenus pour la file $M/M/1$.

4.3 Files M/G/n

La durée D de réponse du serveur suit maintenant une loi générale, qui n'est plus une loi exponentielle. On note alors le système $M/G/n$. La chaîne des états du système associée ne vérifie plus alors nécessairement la propriété de Markov. Le calcul d'un régime permanent est alors possible, mais nécessite des calculs plus complexes. En général, la méthode de résolution consiste à discrétiser la chaîne.

Notons T_n l'instant de départ du $i^{\text{ème}}$ client et X_i le nombre de clients dans le système à l'instant T_i. Désignons par ailleurs par Y_i le nombre d'arrivées de clients entre les instants T_i et T_{i+1}. Ces trois grandeurs sont bien entendu aléatoires. Elles sont reliées par $X_{i+1} = Y_i + (X_i - 1)_+$, où $u_+ = u$ si $u > 0$ et 0 sinon. Les variables Y_i sont supposées indépendantes et de même loi.

La condition d'existence d'un régime permanent est alors donnée par $\alpha = \mathbb{E}[Y_0] < 1$: il faut qu'en moyenne il arrive moins de clients qu'il n'en parte. On peut alors trouver des majorations ou des résultats partiels, mais on ne peut établir de relations comme cela a été fait dans des modèles M/M/n.

Parmi les relations que l'on peut montrer, nous avons

$$\mathbb{E}[X_i] = \frac{2\alpha - \alpha^2 + \lambda^2 Var(D)}{2(1-\alpha)}.$$

Ceci se montre à l'aide des fonctions génératrices. Lorsque D suit une loi exponentielle, on retrouve bien les formules ci-dessus.

Nous ne demandons pas dans ce TP de simuler un tel régime, en raison du temps limité dont vous disposez.

Références

[Ber05a] J. Berard. *Fiche 1 - Nombres pseudo-aléatoires.* ISTIL, 2005. disponible à `http://math.univ-lyon1.fr/~jberard/genunif-www.pdf`.

[Ber05b] J. Berard. *Fiche 2 - Génération de variables pseduo-aléatoires.* ISTIL, 2005. disponible à `http://math.univ-lyon1.fr/~jberard/genloi-www.pdf`.

[Can06] Anne Canteaut. Présentation des principaux tests statistiques de générateurs aléatoires, 2006. disponible sur `http://www.picsi.org/parcours_63.html`.

[FIP01] FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001. U.S. Department of Commerce/N.I.S.T.

[Mar95] George Marsaglia. Diehard battery of tests of randomness, 1995. disponible à `http://i.cs.hku.hk/~diehard/cdrom/`.

[oST] National Institute of Standards and N.I.S.T Technology. Random number generation. disponible à `http://csrc.nist.gov/groups/ST/toolkit/rng/index.html`.

[Wika] Wikipédia. Générateur de nombres pseudo-aléatoires. disponible à `http://fr.wikipedia.org/wiki/Generateur_de_nombres_pseudo-aleatoires`.

[Wikb] Wikipédia. Mersenne Twister. disponible à `http://fr.wikipedia.org/wiki/Mersenne_Twister`.

A Détails des codes fournis pour les générateurs pseudo-aléatoires

Les fichiers suivants vous sont fournis :

- `Generateurs.h` contient la fonction : `word16 Von_Neumann(word16 *next)` qui calcule une itération du générateur de Von Neumann ;
- `Rijndael.h` et `Rijndael.c` fournissent le code du générateur de l'AES. La fonction `u32 AES(u32* Px, const u32* Kex)` calcule une itération de l'AES ; Ce dernier générateur doit être initialisé à l'aide de la fonction `void init_rand(u32* key, u32* plain, const size_t nk, const size_t nb, int params)` où `params` est un paramètre qui doit être différent à chaque initialisation. On doit ensuite avant d'appliquer la fonction `AES` construire les sous-clés à partir de la clé maître `key` à l'aide de la fonction `void KeyExpansion(u32* Key_ex, const u32* Key);`.
- `mt19937p.h` et `mt19937p.c` fournissent le code du générateur de Mersenne-Twister. Ce générateur est initialisé à l'aide de la fonction `void sgenrand(unsigned long seed, struct mt19937p* config)` où `seed` est la graine. Pour calculer une itération du générateur, la fonction est : `unsigned long genrand(struct mt19937p* config)`.

- `essai_main.c` vous donne un exemple de comment utiliser ces générateurs (initialisation et génération des nombres).

B Description du main attendu

Le main que vous nous envoyez devra avoir cette forme :

```
int main ()
{
  Pour chacun des 5 générateurs
   Pour i variant de 1 à 20 faire
     Générer une séquence de 1024 blocs
     Test de fréquence monobits // sortie dans un fichier
     Test des Runs // sortie dans le même fichier

  Récupération des temps d'arrivées et de sorties du système d'une file M/M/1
  // sortie dans le même fichier
  Etude de l'évolution du nombre de clients dans le système
  // sortie dans le même fichier et graphique
  Etude du temps d'attente // sortie dans le même fichier
}
```

C Définition de la fonction `erfc`

La fonction erf est la fonction d'erreur. Plus précisément, quand les résultats d'une série de mesures sont décrits par une distribution normale d'écart type σ et de moyenne 0, alors erf $\left(\frac{a}{\sigma\sqrt{2}}\right)$ est la probabilité que l'erreur d'une unique mesure soit comprise entre $-a$ et $+a$, pour un a positif. Cette fonction a beaucoup d'applications et permet notamment de calculer le "bit error rate" d'un communication digitale. Son expression mathématique (liée à la fonction de répartition de la loi normale) est :

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

La fonction erfc est, quant à elle, la fonction d'erreur complémentaire. C'est-à-dire :

$$\begin{aligned} \operatorname{erfc}(x) &= 1 - \operatorname{erf}(x) && (1) \\ &= \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt. && (2) \end{aligned}$$

D Simulations de lois de probabilités quelconques

Nous devons distinguer deux types de lois de probabilité fort différentes à manipuler : les lois discrètes et les lois continues.

D.1 Lois discrètes

Considérons X une variable aléatoire discrète à valeurs dans $\{e_1, \dots e_k, \dots\}$ vérifiant :

$$\text{pour tout } k \in \mathbb{N}, \quad \mathbb{P}(X = e_k) = p_k.$$

Remarquons alors que si U suit une loi uniforme sur $[0,\,1]$ nous avons :

$$\text{pour tout } a \in \mathbb{R}, \quad \mathbb{P}(U \in [a,\, a + p_k]) = p_k.$$

Ainsi, le principe de simulation d'une loi discrète est le suivant :

```
Soit u realisation d'une loi uniforme sur [0,1],
k = 0;
somme = 0;
Tant que (u < somme)
     k = k + 1;
     somme = somme + p_k;
fin;
x = e_k;
```

Nous obtenons ainsi `x` réalisation de la variable X définie ci-dessus.

D.2 Lois continues

Le principe de simulations de lois continues est de se ramener à des simulations d'autres lois plus aisées à simuler. Ainsi, l'idée est de savoir simuler certaines familles de lois grâce aux simulations de lois uniformes, puis d'autres lois plus complexes à l'aide des premières lois que l'on a su simuler. Nous ne détaillerons pas les méthodologies pour toutes les lois usuelles, mais indiquons simplement la méthode usuelle de simulation d'une loi gaussienne étant donnée son importance. Nous présentons aussi deux algorithmes usuels dits d'inversion et de rejet s'appliquant à de nombreuses lois de probabilité.

D.2.1 Simulation d'une loi gaussienne

Si U et V suivent des lois uniformes indépendantes sur $[0, 1]$, alors nous pouvons établir que $X = \sqrt{-\ln(U)}\cos(2\pi V)$ et $Y = \sqrt{-\ln(U)}\sin(2\pi V)$ suivent une loi de Gauss $\mathcal{N}(0, 1)$ et sont indépendants. Ceci résulte d'un changement de variables. Ce résultat permet de simuler facilement une loi normale.

Notons que si X a pour loi $\mathcal{N}(0, 1)$, alors $m + \sigma X$ a pour loi $\mathcal{N}(m, \sigma^2)$. Ainsi une fois que vous savez simuler la loi $\mathcal{N}(0, 1)$, vous pouvez aisément simuler une loi $\mathcal{N}(m, \sigma^2)$.

Sur le même principe, beaucoup de lois résultent de changement de variables sur des lois usuelles. Ainsi avec des transformations simples de la loi gaussienne, il est facile de simuler des lois lognormale, de Student, de Fisher-Snedecor,du χ^2, etc. La méthode de simulation par inversion exposée ci-après est similaire.

D.2.2 Simulation par inversion

Il s'agit de la méthode utilisée dans ce TP pour simuler la loi exponentielle.

Rappelons le principe. Nous souhaitons simuler une loi de probabilité de fonction de répartition F. Nous sommes assurés de l'existance de la fonction inverse F^{-1}. Soit U suivant une loi uniforme sur $[0, 1]$. Alors nous pouvons montrer que $X = F^{-1}(U)$ a pour fonction de répartition F.

D.2.3 Simulation par rejet

Supposons que vous souhaitiez simuler une loi de densité f. Soit g une densité de probabilité telle que $f \leq cg$ avec c constante positive. Si vous savez simuler la loi de densité g, l'idée est alors de simuler selon cette loi, puis d'accepter ou de rejeter la valeur obtenue avec une certaine probabilité afin d'obtenir la loi de densité f.

Détaillons un peu. Soit U de loi uniforme sur $[0,1]$. Alors pour tout y, nous avons

$$\mathbb{P}\left(U \leq \frac{f(y)}{cg(y)}\right) = \frac{f(y)}{cg(y)}.$$

Par conséquent

$$\mathbb{P}\left(U \leq \frac{f(Y)}{cg(Y)}\right) = \int_{\mathbb{R}} \mathbb{P}\left(U \leq \frac{f(y)}{cg(y)}\right) g(y)dy = \int_{\mathbb{R}} \frac{f(y)}{cg(y)} g(y)dy = \frac{1}{c}.$$

De plus,

$$\mathbb{P}\left(Y \leq t, U \leq \frac{f(Y)}{cg(Y)}\right) = \int_{\mathbb{R}} \mathbb{1}_{\{y \leq t\}} \mathbb{P}\left(U \leq \frac{f(y)}{cg(y)}\right) g(y)dy = \int_{]-\infty,t]} \frac{f(y)}{c} dy.$$

Nous pouvons déduire de ces résultats que la densité de Y sachant $\{U \leq \frac{f(Y)}{cg(Y)}\}$ est f.

On remarquera que la simulation par inversion n'est pas envisageable pour certaines lois. L'algorithme de simulation par rejet permet ainsi de simuler des lois dont la fonction de répartition est plus complexe. En pratique, il permet de simuler plus de lois que la méthode d'inversion.

D.2.4 Application

Considérons une variable aléatoire X de densité

$$f(x) = \frac{2}{\ln(2)^2} \frac{\ln(1+x)}{1+x} \mathbb{1}_{[0,1]}(x).$$

Alors nous pouvons remarquer que $f(x) \leq cg(x)$ avec $c = \frac{2}{\ln(2)^2}$ et g densité de la loi uniforme sur [0, 1]. Ceci permet d'appliquer l'algorithme de simulation par rejet.

De plus, la fonction de répartition vaut

$$F(x) = \int_0^x f(u)du = \frac{\ln(1+x)^2}{\ln(2)^2}.$$

(On pourra retrouver ce résultat en réalisant le changement de variable $v = \ln(1+u)$ dans l'intégrale.) Ainsi

$$F^{-1}(u) = \exp(\sqrt{u}\ln(2)) - 1.$$

L'algorithme de simulation par inversion est donc utilisable.

Question supplémentaire. Ecrire des fonctions de simulation par inversion et par rejet de la loi de densité f. Comparer les temps de calcul pour simuler 1000 valeurs.

Vérifiez que la distribution des valeurs simulées est proche de la distribution théorique cherchée à l'aide d'un histogramme. On pourra utiliser la fonction `hist(.)` de Matlab.

La fonction `time` en C n'étant a priori pas assez précise (ne permet pas de considérer des temps inférieurs à la seconde), nous vous conseillons d'utilser la fonction `gettimeofday` dont la syntaxe est la suivante :

```
struct timeval tps1, tps2;
gettimeofday(&tps1, NULL);
gettimeofday(&tps2, NULL);
printf("Temps ecoule entre tps2 et tps1 : %ld\n",tps2.tv_usec-tps1.tv_usec);
```