Modélisation et Outils Mathématiques
TP génération de nombres aléatoires et probabilités

3IF, INSA de Lyon, 2018/2019
Irène Gannaz

INSA-Lyon, Département Informatique
version 2.00, 2018 – rédigé avec LaTeX

Ce document a été rédigé en collaboration avec Marine Minier.

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 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 avec le logiciel `R`. Attention toutefois, `R` est un logiciel de statistique et il n'est pas approprié pour mettre en œuvre les tests de qualité de générateurs aléatoires sur les séquences de bits (Section 1). Le choix de `R` a été fait ici a but pédagogique. Si par la suite vous êtes amenés à réaliser des tests tels que décrits en Section 1, utilisez `C`.

A la fin des deux séances, vous devrez rendre via moodle à l'adresse `http://moodle2.insa-lyon.fr/` :

- Un compte-rendu rédigé, 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 deuxième séance de TP les réponses aux questions sous forme de tableaux et de graphiques ainsi que des **commentaires** sur ces résultats.
- Les codes que vous avez développés pendant les séances.

- Votre compte-rendu devra s'appeler `NumeroBinome.pdf`.

Il est fortement conseillé de rédiger votre compte-rendu avec `R Markdown`. Des fiches d'introduction au logiciel `R` ainsi qu'un exemple de fichier `R Markdown` sont disponibles sous `moodle`.

Remarques

Vous découvrez un nouveau logiciel, ce qui ralentit nécessairement le déroulement du TP. Ceci est pris en compte dans le TP. Des documents d'aide à la prise en main vous sont fournis sur `moodle`. Les objectifs de ce nouveau logiciels sont double : cela vous permet de découvrir le logiciel `R` (très utilisé en statistique, même si vous en avez ici une autre utilisation) et cela vous permet de montrer que la prise en main d'un nouveau logiciel ne demande pas un temps excessif.

Pensez à commenter et structurer votre code. Et à donner des noms de variables "logiques". Dites vous que quelqu'un d'autre doit pouvoir reprendre votre code ou que vous même devez vous y retrouver dans 3 mois. Vous pouvez regrouper dans un même fichier plusieurs fonctions concernant un même thème. Par exemple `generateurs.R` fourni sur `moodle` regroupe des fonctions concernant des générateurs pseudo-aléatoires. Ensuite `source('generateurs.R')` permet de faire appel à toutes les fonctions contenues dans `generateurs.R`. Faites ensuite un *main*, qui exécute les fonctions des autres fichiers.

Il vous est suggéré d'utiliser un *notebook* pour faire votre compte-rendu. Plus précisément `Rmarkdown` vous permet d'insérer du code `R` et les résultats associés dans un texte. Ne mettez pas tout votre code sur votre fichier Rmarkdown mais préférez utilisez un `source` qui permet d'appeler les fonctions que vous souhaitez.

De manière générale, vous pensez peu à utiliser les capacités du logiciel. Pensez que beaucoup de fonctions ont déjà été programmées et que par exemple il n'est pas nécessaire de recoder un tri de tableau. Un autre exemple : pour calculer la somme du vecteur `vect` de `i0` à `i1`, nous voyons très souvent

```
S <- 0
for(i in i0:i1){
    S <- S + vect[i]
}
```

Cette opération peut aussi se faire directement par

```
S <- sum(vect[i0:i1])
```

1 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. le générateur de Von Neumann,introduit en 1946, qui consiste à élever un nombre au carré puis à retirer le premier et le dernier chiffre, et à itérer cette opération ;
2. un générateur à congruence linéaire usuel, dit Standard Minimal,
3. un générateur à congruence linéaire avec un choix différent des paramètres, dit RANDU,

4. le générateur Mersenne-Twister, qui est le générateur par défaut de `R`.

Le principe de ces générateurs est décrit ci-après. Le générateurs Mersenne-Twister ne sera pas décrit ici. Nous renvoyons à [7] pour une description détaillée.

Afin de comparer ces générateurs, vous aurez ensuite à programmer des tests classiques de probabilité (voir Section 1.2) 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.

Nous utiliserons de plus le paquet `randtoolbox`. Pour l'installer, utilisez

```
install.packages('randtoolbox')
```

et ensuite

```
library(randtoolbox)
```

permet de préciser à `R` que nous allons faire appel à des fonctions de ce paquet.

1.1 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 [1]). 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.

1.1.1 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.

1.1.2 Les générateurs pseudo-aléatoires étudiés

Les différents générateurs étudiés sont basés sur ce principe. Ainsi dans le générateur de Von Neumann proposé en 1946, la fonction f consiste à élever le nombre S_n au carré et à ôter les premiers et les derniers chiffres, de manière symétrique, de sorte à ce que le nombre obtenu soit compris entre 0 et 9999. Par exemple si $S_n = 1315$ alors $1315^2 = 1729225$ donc $S_{n+1} = 292$.

Introduits en 1948, les générateurs à congruence linéaire ont ensuite eu beaucoup de succès. L'idée est d'appliquer une transformation linéaire suivie d'une opération de congruence. Dans le cas discret :

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

Si on souhaite se ramener à l'intervalle $U = [0, 1]$, on appliquera $X_n = g(S_n) = S_n/m$. D'autres types de congruence peuvent être considérés. Par exemple, la fonction `rand` de `C` utilise par défaut une congruence de type polynomial.

Nous étudierons dans ce TP deux générateurs à congruence linéaire. Le premier est connu sous le nom de Standard Minimal. Il s'agit du générateur défini ci-dessus avec les paramètres

$$a = 16807,\ b = 0,\ m = 2^{31} - 1.$$

Vous générerez également le générateur à congruence linéaire dit RANDU, qui prend pour paramètres

$$a = 65539,\ b = 0,\ m = 2^{31}.$$

Cette méthode ayant toutefois montré ses limites, le générateur pseudo-aléatoire par défaut de `R` est actuellement Mersenne-Twister. C'est aussi le générateur par défaut du logiciel `Matlab`. Il engendre des séquences pseudo-aléatoires de qualité satisfaisante, bien qu'il ne soit pas cryptographiquement sûr. Ce générateur ne sera pas décrit ici. Nous renvoyons à [7] ou [4] pour une description détaillée.

1.1.3 Graine des générateurs

Pour choisir la graine, on emploie la fonction `set.seed` que l'on "nourrit" à l'aide d'un nombre. Si vous souhaitez faire une expérience reproductible, il faut initialiser la graine avec un nombre fixé. Ceci doit être utilisé notamment pour comparer des algorithmes. En effet les comparaisons sont alors réalisées sur deux jeux de données simulées identiques. Si le but est de générer une suite difficilement prévisible et qui varie rapidement, il faut l'initialiser avec un nombre qui varie rapidement. Par exemple, vous pouvez prendre le nombre de cycles utilisés par votre processeur depuis son démarrage.

1.1.4 Génération de séquence avec 4 générateurs

Si vous tapez `RNGkind()` sans arguments, `R` vous retournera le nom actuel du générateur utilsé, et la méthode de génération de la loi normale. Pour utiliser le générateur de Mersenne-Twister avec une graine $S_0 = 215$ on utilisera `set.seed(215,kind='Mersenne-Twister')`. Mersenne-Twister étant le générateur par défaut, on pourra plus simplement appeler `set.seed(215)`. Une fonction

`VonNeumann(n,p,graine)` est fournie sur `moodle`. Elle permet de générer p séquences de n entiers sur $\{0, \ldots, 9999\}$ avec VonNeummann, avec S_0 égal à `graine`.

La majorité des techniques de simulations de lois de probabilité génère des entiers sur un intervalle $\{0, \ldots, m\}$. Cependant en pratique il est plus souvent utile de faire appel à une loi uniforme sur l'intervalle [0, 1]. Cette loi est aussi fort utile dans la génération d'autres lois de probabilités. Par défaut la fonction `runif` de `R` normalise donc les valeurs générées sur [0, 1]. Un appel de la fonction `runif(n,p)` retournera une matrice de taille $n \times p$ dont les valeurs sont obtenues par le générateur choisi. Les valeurs sont normalisées sur [0, 1]. Les générateurs de nombres entiers sont obtenus en utilisant la fonction `sample.int(m,n)` qui génère n entiers sur $\{0, \ldots, m\}$ à l'aide du générateur et de la graine définis dans `set.seed()`. Un exemple est fourni sur `moodle` pour générer des valeurs avec Mersenne-Twister. La fonction `MersenneTwister(n,p,graine)` vous permet ainsi de générer p séquences de n entiers sur $\{0, \ldots, 2^{32} - 1\}$.

Question 1. On vous demande dans un premier temps d'implémenter deux fonctions, qui prennent toutes pour paramètre la taille `k` de la séquence que vous souhaitez générer. Ces fonctions retournent les `k` valeurs obtenues par les générateurs pseudo-aléatoires étudiés. Vous devez ainsi implémenter :

- une fonction `RANDU` qui retourne les valeurs données par le générateur de congruence linéaire RANDU décrit plus haut sur $\{0, \ldots, 2^{31} - 1\}$;
- une fonction `StandardMinimal` retournant les valeurs données par le générateur de congruence linéaire homonyme sur $\{0, \ldots, 2^{31} - 2\}$.

Vous aurez besoin du modulo, noté `%%` en `R` : $x \bmod m$ est donné par `x%%m`.

Nous testerons dans la suite les générateurs pseudo-aléatoires `VonNeumann`, `RANDU`, `Standard Minimal` et `Mersenne- Twister`, donnés sur `moodle` ou construits dans les fonctions ci-dessus. Nous considérerons les valeurs générées par `Mersenne- Twister` sur $\{0, \ldots, 2^{32} - 1\}$.

1.2 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 [6] ou de DIEHARD [5]). Nous ne les programmerons pas tous, nous allons nous focaliser sur des tests particuliers que nous appliquerons ensuite sur les 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 [2] ou [3] pour voir les nombreux tests statistiques qui existent.

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

- pour le premier test, visuel, sur une séquence de $k = 100$ valeurs.
- pour les trois derniers tests sur une séquence de $k = 100$ et pour 100 initialisations différentes.

1.2.1 Test Visuel

Question 2.1. Tracez, pour chacun des générateurs, l'histogramme des sorties observées pour une suite de $k = 100$ valeurs. Que constatez-vous ? Expliquez. Nous vous conseillons d'utiliser la fonction `hist`, dont vous pouvez ajuster la finesse si besoin. Rappel : `help(hist)` ou `?hist` permettent d'accéder à l'aide sur la fonction `hist`.

Question 2.2. Tracez la valeur obtenue en fonction de la valeur précédente de l'algorithme. Plus précisément, à partir d'un vecteur `u` de taille `n`, exécutez

```
plot(u[1:(n-1)], u[2:n])
```

Commentez.

1.2.2 Test de fréquence monobit

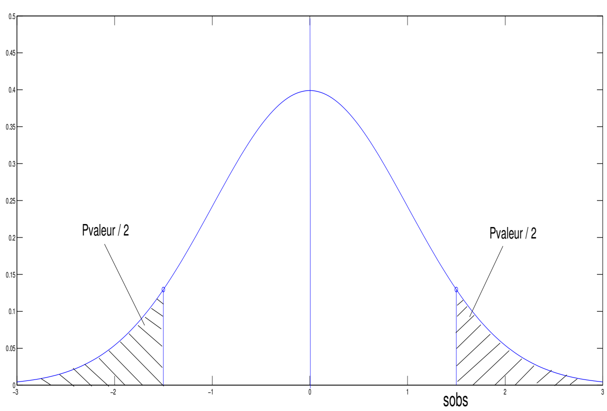
Principe du test : Le but de ce test est de s'intéresser à la proportion de zéros et de uns dans les bits d'une séquence entière : on regarde ici tous les bits des 100 réalisations de la séquence. On teste donc si le nombre de uns et de zéros d'une séquence sont approximativement les mêmes comme attendu dans une séquence vraiment aléatoire.

Définition de la fonction à implémenter : la fonction à implémenter devra s'écrire : `Frequency <- function(x, nb)` où `x` est le vecteur des nombres observés et `nb` le nombre de bits à considérer pour chacun de ces nombres. 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 : $s_{obs} = \frac{|2|}{\sqrt{10}} = 0.6325$.

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} sous `R` peut se faire ainsi : $P_{valeur} = 2 * (1 - \text{pnorm}(s_{obs}))$. Pour notre exemple, on obtient $P_{valeur} = 0.5271$.

L'appel à la fonction `Frequency(x, nb)` devra retourner la P_{valeur} du test de fréquence monobit sur x.

Règle de décision à 1% : Au vu du principe décrit ci-dessus, plus la P_{valeur} est petite plus on peut rejeter 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 ne peut pas conclure pour autant qu'elle l'est, mais rien n'infirme cette hypothèse, au sens de ce test. Dans l'exemple précédent, comme $P_{valeur} = 0.527089$, on peut valider 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 3. Implémentez la fonction décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

Vous aurez besoin de transformer les nombres générés en séquences de bits. `R` propose la fonction `intToBits`, mais qui malheureusement ne peut s'appliquer que sur les entiers compris entre $-(2^{31}-1)$ et $+(2^{31}-1)$ car il prend en compte le signe. Je vous fournit sur `moodle` une fonction `binary(x)` qui convertit l'entier positif `x` en séquence de 32 bits.

1.2.3 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 $\mathcal{A}$. Notons A le nombre d'éléments de $\mathcal{A}$. 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 : $\left\lfloor \frac{T\cdot(A-1)^2}{A^{k+1}} \right\rfloor \leq N(k) \leq \left\lceil \frac{T\cdot(A-1)^2}{A^{k+1}} \right\rceil$.

Pour comprendre d'où vient cette propriété, notons p la probabilité d'avoir un symbole donné. Alors $p = 1/|A|$ et la probabilité d'avoir une séquence de longueur k exactement vaut $(1-p)^2p^k = \frac{(|A|-1)^2}{|A|^{k+1}}$.

Définition de la fonction à implémenter : La fonction à implémenter devra s'écrire : `Runs <- function(x,nb)` où `x` est le vecteur des nombres observés et `nb` le nombre de bits à considérer pour chacun de ces nombres. Elle retournera la P_{valeur} du test, obtenue en effectuant 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 : $\tau = 2/\sqrt{10} \approx 0.6346$ 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 :

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

Si on reprend l'exemple précédent, on obtient : $P_{valeur} = 2 \cdot \left(1 - \text{pnorm}\left(\frac{|7-2\cdot 10\cdot \frac{3}{5}(1-\frac{3}{5})|}{2\sqrt{10}\cdot(3/5)\cdot(1-3/5)}\right)\right) \approx 0.1472$.

Règle de décision à 1% : Si la P_{valeur} calculée 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 0.1472, on ne peut donc conclure que la séquence est de mauvaise qualité. Il est recommandé que chaque séquence testée fasse au minimum 100 bits (i.e., $n \geq 100$).

Question 4. Implémentez la fonction décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

1.2.4 Test d'ordre

Le dernier test n'étudie pas la suite de bits générés mais directement la suite de nombres obtenus.

Supposons que nous ayons $U_1^{(k)}, \ldots U_n^{(k)}$ indépendants de loi uniforme $\mathcal{U}[0, 1]$, avec $k = 1, \ldots, d$. Alors si nous comparons $U_i^{(1)}$, $U_i^{(2)}$ et $U_i^{(3)}$, nous devons avoir la même probabilité que $U_i^{(1)} \leq U_i^{(2)} \leq U_i^{(3)}$ et que $U_i^{(1)} \leq U_i^{(2)} \leq U_i^{(3)}$. Nous avons en fait $d!$ ordres possibles de $(U_i^{(1)}, \ldots U_i^{(d)})$ à i fixé, chacun arrivant avec une probabilité $1/d!$. L'idée de ce test est donc de compter pour chaque ordre le nombre n_j d'apparation de celui-ci et de comparer ce nombre à $n/d!$. Ceci peut être fait à l'aide d'un test du χ^2 (qui sera vu en 4IF).

Ce test est implémenté dans le paquet `randtoolbox`, [4]. Si `u` est un vecteur de valeurs sur $\{0, \ldots, m\}$, ou sur $[0, 1]$. Alors

```
order.test(u, d=3, echo=FALSE)$p.value
```

retourne la P_{valeur} du test d'ordre décrit ci-dessus avec $d = 3$. L'hypothèse que les observations de `u` sont issues d'une loi uniforme (discrète ou uniforme) est rejetée si la P_{valeur} est inférieure à 1%.

Question 5. Testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés en prenant $d = 4$.

2 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.

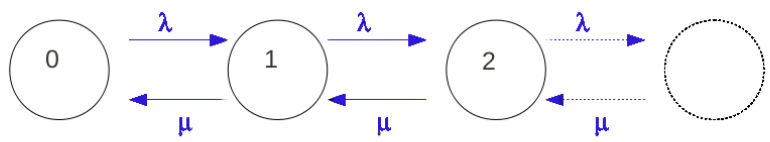
Notation de Kendall.
Nous désignerons une file d'attente par la notation A/B/m. La lettre A désigne la loi du temps écoulé entre deux arrivées et la lettre B la loi du temps nécessaire pour répondre au client pour un serveur. m représente le nombre de serveurs.

2.1 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 lois 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 `(lambda, mu, D)` où

— `lambda` est le paramètre de la loi exponentielle des arrivées,

— `mu` est le paramètre de la loi exponentielle des départs,

— `D` est le temps d'observation de la chaîne,

Les réalisations de lois exponentielles peuvent être obtenues à l'aide de la fonction `rexp` de `R`.

Définition de la fonction à implémenter : la fonction à implémenter sera ainsi de la forme `FileMM1 <- function(lambda, mu, D)` avec :

— `lambda` est le paramètre de la loi exponentielle des arrivées,

— `mu` est le paramètre de la loi exponentielle des départs,

— `D` est le temps d'observation de la chaîne.

La fonction `FileMM1` retournera une liste contenant les éléments `arrivee` et `depart`, tels que :

— `arrivee` est le vecteur des dates d'arrivées de clients durant l'intervalle de temps `D`,

— `depart` est le vecteur des dates de départs de clients, qui sont sortis du système durant l'intervalle de temps `D`.

Rappel : Une liste en `R` se construit comme suit : `ma_liste <- list(x=2, y=c(1,2))`. Ce type permet de mélanger des objets de nature différentes. Alors pour accéder au premier élément de la liste, on utilisera `ma_liste$x` ou `ma_liste[[1]]`.

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.

Une fois cette fonction construite, nous souhaiterions visualiser l'évolution de la file d'attente. Plus précisément nous souhaiterions savoir le nombre de clients dans la file d'attente au cours du temps.

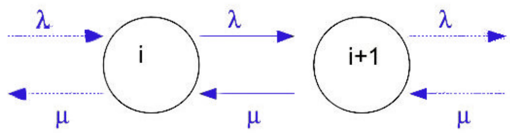
Question 7. Construire une fonction qui à partir des dates d'arrivées et de sorties du système retourne l'évolution du nombre de clients dans le système.

Application : Prendre λ et μ tels qu'arrivent en moyenne 10 clients par heure et repartent en moyenne 20 clients par heure. Représentez l'évolution du nombre de clients dans le système pendant 12 heures de fonctionnement.

Même question avec 14, 20 puis 30 arrivées par heure en moyenne et le même taux de départ. Comparez les résultats obtenus.

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 12 heures de fonctionnement. Retrouvez-vous la formule de Little ? Commentez.

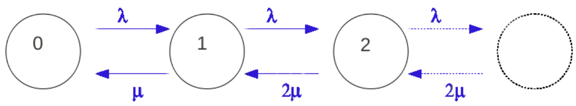
On répondra à cette question en prenant les valeurs de λ et μ des questions précédentes.

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

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

2.2 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 prises 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 Bonus. Simuler un tel système, en reprenant la fonction `FileMM1` précédente, que l'on notera `FileMM2`.

Simulez une file d'attente $M/M/2$ de paramètres λ et μ tels qu'arrivent en moyenne 10 clients par heure et repartent en moyenne 20 clients par heure. 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 12 heures. Comparez aux résultats obtenus pour la file $M/M/1$.

2.3 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

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

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

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

[4] Dutang Christophe and Savicky Petr. *randtoolbox : Generating and Testing Random Numbers*, 2015. R package version 1.17.

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

[6] 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`.

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