Rendu TP probabilités

B3404 et A. Guigal

09/05/2021

Compte rendu Guigal, Penot, Souabi, Collard

09/05/2021

Ce compte-rendu rend compte du TP de probabilités

1 : Tests de générateurs pseudo-aléatoires

Nous allons lors de ce TP tester 4 générateurs : VonNeumann, RANDU, Standard Minimal et Mersenne-Twister.

1.1 : Définition des générateurs

Code de Standards Minimal :

```
StandardMinimal <- function(seed,n)
  a <- 16807
  m <- 2^31-1
  x <- seed
  #b = 0
  x <- (x*a)%%m
  for(i in 1:n)
  {
    x <- c(x,(x[i-1]*a)%%m)
  }
  #x <- x/m #Normalise x
  return(x)
```

Code de RANDU

```
RANDU <- function(seed,n)
{
  a <- 65539
  m <- 2^31
  x <- seed
  #b = 0
  x <- (x*a)%%m
  for(i in 1:n)
  {
    x <- c(x,(x[i-1]*a)%%m)
  }
  #x <- x/m #Normalise x
  return(x)
}
```

Nous noterons que les générateurs de VonNeumann et Mersenne-Twister ont déjà été définis au préalable.

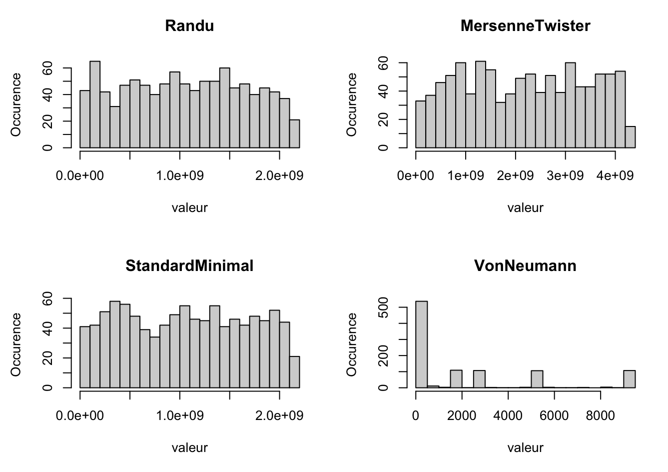
1.2 : Tests de qualité des séquences produites

Afin de tester la qualité des générateurs, nous allons procéder à plusieurs tests.

1.2.1 : Test visuel

Générons une séquence avec chaque générateur et observons la répartition des valeurs :

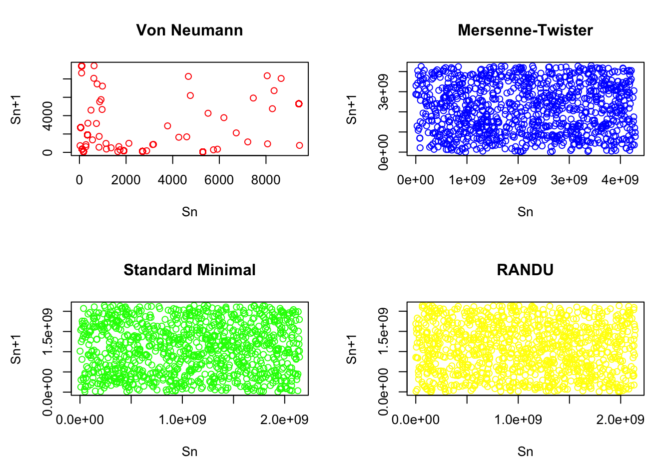
```
par(mfrow=c(2,2)) #(2,2) marche pas -> trop grand?
hist(RANDU(34,1000),xlab='valeur',ylab='Occurence',main='Randu',breaks = 20)
hist(MersenneTwister(1000,1,34),xlab='valeur',ylab='Occurence',main='MersenneTwister',breaks = 20)
hist(StandardMinimal(34,1000),xlab='valeur',ylab='Occurence',main='StandardMinimal',breaks = 20)
hist(VonNeumann(1000,1,34),xlab='valeur',ylab='Occurence',main='VonNeumann',breaks = 20)
```



Nous constatons que chaque générateur semblent avoir une étendue satisfaisante, excepté celui de VonNeumann.

Voyons maintenant la valeur Sn+1 en fonction de Sn

```
par(mfrow=c(2,2))
n<-1000
u <- VonNeumann(1000,1,34)
plot(u[1:(n - 1)], u[2:n], main = "Von Neumann", xlab = 'Sn', ylab = 'Sn+1', col = 'red')
u <- MersenneTwister(1000, 1, 34)
plot(u[1:(n - 1)], u[2:n], main = "Mersenne-Twister", xlab = 'Sn', ylab = 'Sn+1', col = 'blue')
u <- StandardMinimal(34, 1000)
plot(u[1:(n - 1)], u[2:n], main = "Standard Minimal", xlab = 'Sn', ylab = 'Sn+1', col = 'green')
u <- RANDU(34, 1000)
plot(u[1:(n - 1)], u[2:n], main = "RANDU", xlab = 'Sn', ylab = 'Sn+1', col = 'yellow')
```

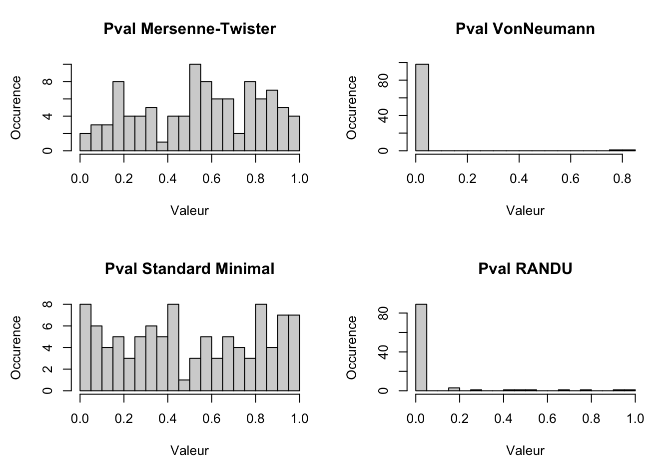


Mersenne-Twister, RANDU et Standard Minimal semble ne pas avoir de dépendance entre les différentes valeurs. Cependant, ce n'est pas le cas de VonNeumann, où la valeur précédente impacte grandement la suivante.

1.2.2 : Test de fréquence monobit

Nous allons tester donc si les nombres de uns et de zéros d'une séquence sont approximativement les mêmes, comme attendu dans une séquence vraiment aléatoire.

On calcule une somme Sn, qui ajoute +1 pour chaque bit à 1 et -1 pour chaque bit à 0. Plus cette somme est éloignée de 0, moins la séquence de bits générée est homogène. On en déduit ensuite Sobs en divisant par l'écart-type. On place la valeur de Sobs sur la courbe de densité de la loi normale centrée réduite, de manière symétrique, en on en déduit la Pvaleur. En effet, la Pvaleur correspond à l'intégrale de -∞ à - Sobs de f(x)dx + l'intégrale de +sobs à +∞ de f(x)dx, f(x) étant la densité de la loi normale centrée réduite.



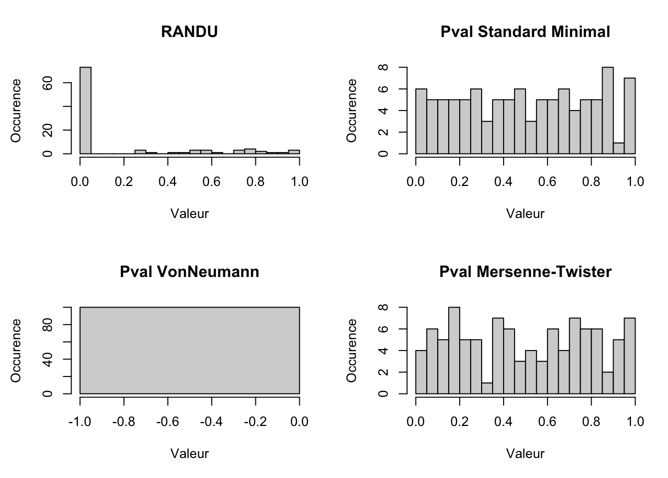
Règle de décision à 1% : Plus la Pvaleur est petite plus on peut rejeter de manière sûre le fait que le séquence est aléatoire. En pratique, si la Pvaleur calculée est inférieure à 0.01 alors la séquence n'est pas aléatoire. Sinon, on ne peut pas conclure pour autant qu'elle l'est, mais rien n'infirme cette hypothèse, au sens de ce test. On répète l'expérience 100 fois afin de générer 100 Pvaleur, pour être sûr qu'une Pvaleur trop faible ne serait pas liée à de la malchance.

En calculant les Pvaleurs et en effectuant les tests de validation, on obtient ce tableau

Algorithme	Pvaleur	Taux de réussite
Mersenne-Twister	0.5446661	100
RANDU	0.0557227	11
Standard Minimal	0.501776	99
VonNeumann	0.0161289	2

1.2.3 : Test des runs

Nous allons observer les suites ininterrompues de 0 et de 1

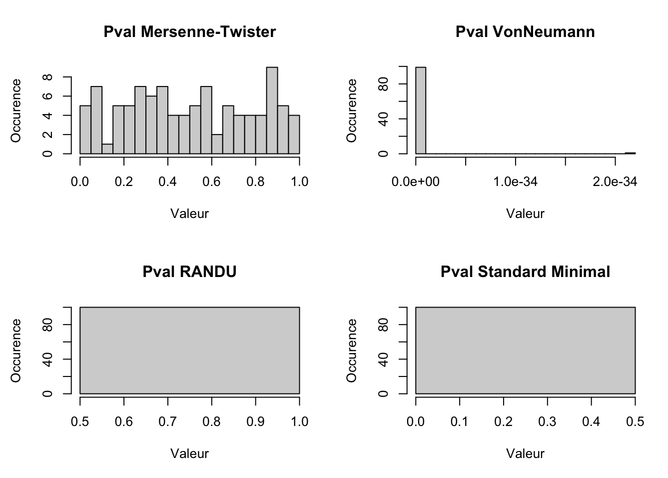


De la même manière qu'auparavant, on obtient

Algorithme	Pvaleur	Taux de réussite
Mersenne-Twister	0.5007325	100
RANDU	0.1756119	28
Standard Minimal	0.4981495	98
VonNeumann	0	0

1.2.4 : Test d'ordre

De la même manière qu'auparavant, on obtient



Algorithme	Pvaleur	Taux de réussite
Mersenne-Twister	0.5	98
RANDU	0.78	100
Standard Minimal	0.49	100
VonNeumann	2.19^{-36}	0

2 : Simulations de lois de probabilités quelconques

Nous allons maintenant étudier la simulation de loi de probabilités quelconques. Nous allons tout au long nous baser sur la loi uniforme, et voir comment celle-ci peut nous aider à simuler différents phénomènes aléatoires.

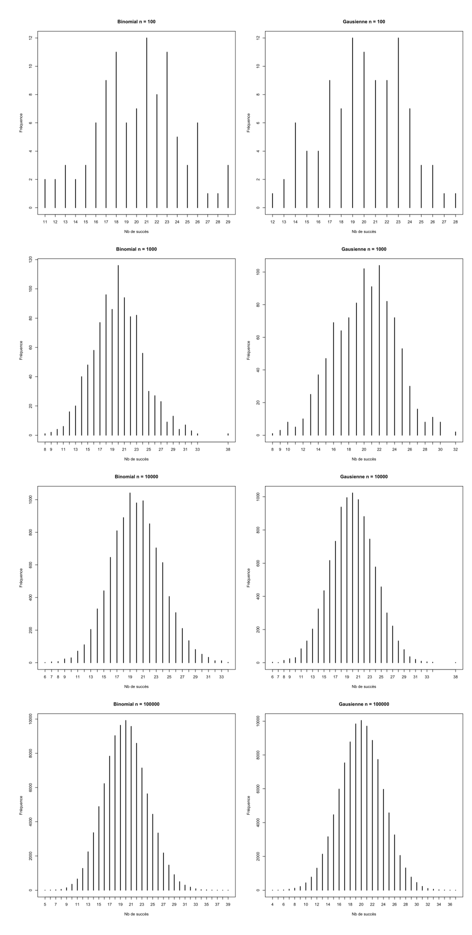
2.1 : Lois discrètes

Simulons une loi binomiale à partir de la loi uniforme U. Faisons de même avec la loi Gausienne, puis comparons les deux entre elles. Les algos sont les suivants :

```
LoiBinomiale <- function(n, p)
{
  u = runif(1)
  k=0
  somme = 0
  while(u > somme){
    k = k + 1
    pk = choose(n, k) * (p^k)*((1-p)^(n-k))
    somme = somme + pk
  }
  return(k)
}

LoiGausienne <- function(n, p)
{
  u = runif(1)
  k=0
  somme = 0
  while(u > somme){
    k = k + 1
    pk = dnorm(k,n*p,sqrt(n*p*(1-p)))
    somme = somme + pk
  }
  return(k)
}
```

En lançant les algorithmes n fois, nous obtenons les diagrammes suivants :



On constate que, plus *n* est grand, plus les simulations de loi se rapprochent d'une forme gaussienne. Nous pouvons donc considérer que, lorsque n est grand, nous pouvons simuler la loi Binomiale par la loi Gausienne, et cela avec une qualité satisfaisante. En effet, le calcul de la fonction de masse de la loi binomiale devient rapidement fastidieux lorsque n est grand, il est alors possible d'utiliser des approximations par d'autres lois de probabilité telles que la loi de Poisson ou la loi normale et d'utiliser des tables de valeurs.

2.2 : Lois continues

Nous allons maintenant simuler une loi continue, grâce à la loi uniforme. Nous allons étudier les performances de deux algorithmes : la simulation par inversion, et celle par rejet. Pour cela, nous allons simuler la loi suivante : $F(x) = 2/ln(2)^2 * ln(1+x)/(1+x) * U(x)$

Pour la simulation par inversion, on simule une loi de probabilité de fonction de répartition F. Dans notre cas, on sait que F est inversible. Soit U la loi uniforme sur [0,1]. Cet algorithme renvoie la variable $X = F^-1(U)$ qui a une fonction de répartition F.

Pour simuler par rejet, on pose que $F \leq c * G$ avec c une constante positive et G une densité de probabilité que l'on peut aisément simuler. G est une loi de densité uniforme sur [0,1] dans notre cas.

Simulons la fonction F à travers les deux méthodes :

```
SimInversion <- function(){ #Difficile à appliquer généralement
  u = runif(1)
  return(res = exp(sqrt(u) * log(2))-1)
} #F-1, donc F est la fct de répartition

SimRejet <- function(){
  u<-runif(1)
  y<-runif(1)
  while(u>(log(1+y)/(1+y))){ #Tant qu'on est pas sup à U...
    u<-runif(1) #... on re-simule u et y
    y<-runif(1)
  }
  return(y)
}
```

Voici la mesure des performances :

Unité : nanosecondes

expr	min	lq	mean	mediam	uq	max	neval
SimInversion()	2000	2250	3251	3100	3900	7600	100
SimRejet	3700	7750	22359	16000	28700	129600	100

Le package microbenchmark() nous permet de comparer les propriétés des algorithmes de simulation par rejet et simulation par inversion tel que la moyenne, la médiane, les quartiles , etc… Nous remarquons que la Simulation par Inversion est bien plus performante. Cela est normal, puisque nous l'inverse de la fonction est codée en dûr, et ne boucle pas. Cependant, celle-ci est rarement applicable, puisqu'il est souvent difficile, voir impossible de trouver la fonction inverse. Au contraire, la simulation par rejet est employable dans la plupart des cas : il suffit juste de bien déterminer la fonction G.

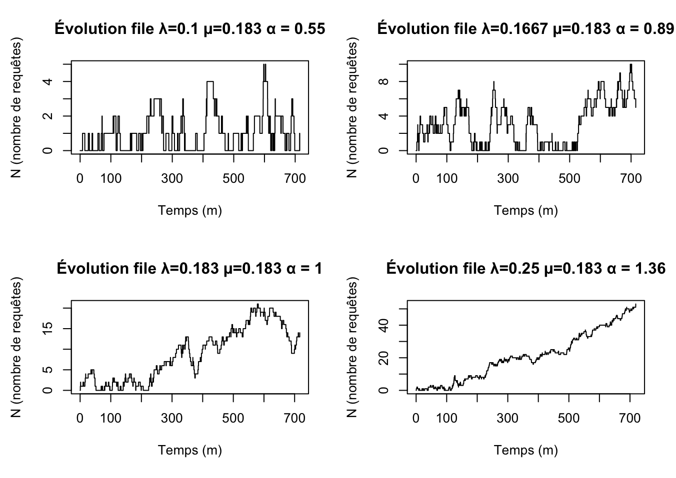
3 : Application aux files d'attentes

3.1 : Files M/M/1

Nous allons générer deux listes qui renvoient les dates d'arrivée et de départ des clients. Ces dates sont générées grâce à une loi exponentielle, ainsi chaque date d'arrivée est bien indépendante de la précédente. Il s'agit de l'absence de mémoire qui caractérise le comportement d'une file d'attente.

Ensuite, ces deux derniers tableaux nous permettent de calculer l'évolution du nombre de clients dans la file d'attente, au cours du temps.

On choisit λ et µ tels qu'arrivent en moyenne 6 clients par heure et repartent en moyenne 11 clients par heure. Si 6 clients arrivent par heure, il y a donc 6/60 = 0.1 client par minute, donc λ=0.1. De même, si 11 clients repartent par heure, on a donc µ=11/60=0.183.



Dans les deux premiers cas, on a α<1, ce qui produit au bout de 12h une file d'attente avec un régime stable. Dans le cas où λ=0.183 et µ=0.183, on a alors α=1, et on obtient ainsi un régime instable : la file d'attente semble croître mais de manière incertaine et non régulière. Pour λ=0.25 et µ=0.183, on a alors α>1, ce qui veut dire que le temps moyen de traitement d'un client est supérieur à l'intervalle moyen d'arrivée entre deux clients. AInsi, les clients s'accumulent dans la file d'attente, c'est un régime divergent.

La durée entre deux arrivées suit une loi exponentielle de paramètre λ, dont l'espérance est 1/λ. C'est-à-dire que le temps moyen d'attente d'une nouvelle arrivée est de 1/λ. Autrement dit, on a une moyenne de λ arrivées par unité de temps. Ainsi, sur un intervalle de temps t, on a bien une moyenne de λt arrivées.

Dans tous les cas, 15 personnes partent en moyenne par heure. Si seulement 8 clients arrivent par heure (graph 1), tout le monde est vite servi, et la file est principalement vide. Pour 14 et 15, malgré des pics d'attente, la file reste souvent vide. Enfin, pour 20 clients / heure, le serveur n'arrive pas à servir assez de monde : le flux entrant est trop important, la file s'allonge et le serveur sature. On en conclue qu'il ne faut pas excéder les possibilités du serveur en terme de capacité, sous peine de voir une saturation rapide.

3.2 : Formule de Little

On va étudier un régime stable, c'est-à-dire dans le cas où on a α = λ/µ < 1. Cela veut dire que le nombre moyen d'arrivées en un certain temps est inférieur au nombre de départs, et que le nombre de personnes dans la file d'attente va se stabiliser. Ainsi, la file ne s'encombre pas et ne saturera pas.

Il s'agit ici de calculer le nombre moyen de clients dans le système E(N) ainsi que le temps de présence d'un client dans le système E(W). Nous devons vérifier la formule de Little : E(N) = λ E(W), λ représentant le nombre moyen d'arrivée de clients par minute. Pour calculer E(N), on reprend les résultats de la question 7 et on fait la moyenne du nombre de clients en attente. Attention, il ne suffit pas de sommer les N et de diviser par le nombre de valeurs, car les mesures de N ne sont pas faites à intervalles réguliers, mais seulement lorsqu'un client arrive ou repart. Ainsi, il faut multiplier la valeur de N par le temps pendant lequel ce N vaut. Enfin, on divise le résultat par le temps total en minutes.

```
filemm1 <- FileMM1(0.1, 0.183, 12 * 60)
evolution <- fileEvolution(filemm1[[1]], filemm1[[2]])
e1 <- esperanceFile(filemm1[[1]], filemm1[[2]], evolution[[1]], evolution[[2]])

filemm1 <- FileMM1(2, 5, 10 * 12 * 60)
evolution <- fileEvolution(filemm1[[1]], filemm1[[2]])
e2 <- esperanceFile(filemm1[[1]], filemm1[[2]], evolution[[1]], evolution[[2]])
```

λ	$\mathbb{E}(W)$ calculée	$\mathbb{E}(N)$ calculée	$\lambda\mathbb{E}(W)$ calculée
0.1	9.92	1.01	0.99
2	0.34	0.67	0.67

On trouve bien deux valeurs très proches pour E(N) et λE(W). Les valeurs sont faibles, ce qui signifie qu'il y a très peu de personnes dans la file d'attente. (ex : si E(N)=1.3, alors il y a en moyenne 1.3 personnes dans le système, donc 1 personne dont la requête est en train d'être traitée et en moyenne 0.3 personne en train d'attendre son tour).