TP de Probabilités

Clément Giraudon & Swan Maillard

2022-2023

##Compte rendu

TP1

Dans ce 1er TP, nous allons implémenter 2 générateurs de nombres aléatoires et les tester ainsi que deux autres générateurs avec une série de tests que nous allons implémenter.

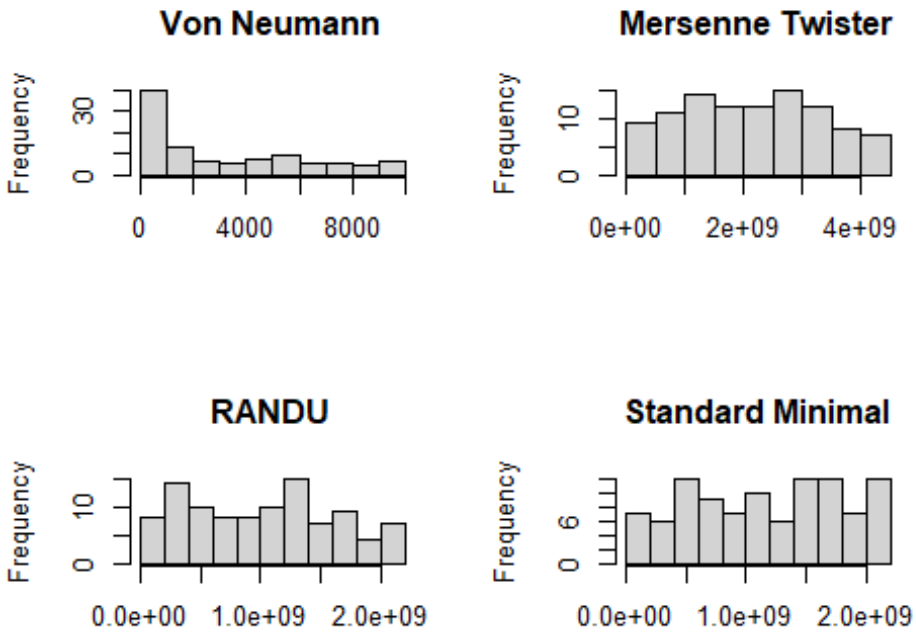
Question 1. et 2.1

Voici le code utiliser pour les générateurs RANDU et Standard Minimal :

```
RANDU <- function(k, p=1, graine)
{
  a <- 65539
  m <- 2^31

  x <-  rep(graine, k*p+1)
  for(i in 2:(k*p+1))
  {
    x[i] <- (a*x[i-1])%%m
  }
  x <- matrix(x[2:(k*p+1)],nrow=k,ncol=p)
  return(x)
}


StandardMinimal <- function(k, p=1, graine)
{
  a <- 16807
  m <- 2^31 - 1

  x <-  rep(graine, k*p+1)
  for(i in 2:(k*p+1))
  {
    x[i] <- (a*x[i-1])%%m
  }
  x <- matrix(x[2:(k*p+1)],nrow=k,ncol=p)
  return(x)
}
```

Et voici les résultats obtenus sous forme d'histogramme :

```
vn <- VonNeumann(Nsimu, Nrepet, graine)
mt <- MersenneTwister(Nsimu,Nrepet,graine)
RANDU <- RANDU(Nsimu, Nrepet, graine)
sm <- StandardMinimal(Nsimu, Nrepet, graine)

Histo(vn, mt, randu, sm)
```



On peut déjà remarquer que la répartition des valeurs obtenues par le générateur de Von Neumann n'est pas du tout homogène étant donné qu'une grande majorité d'entre elles sont entre 0 et 1000. De plus, d'après les histogrammes, le générateur de Mersenne Twister semble être celui qui suit le plus la loi Uniforme.

Question 2.2.

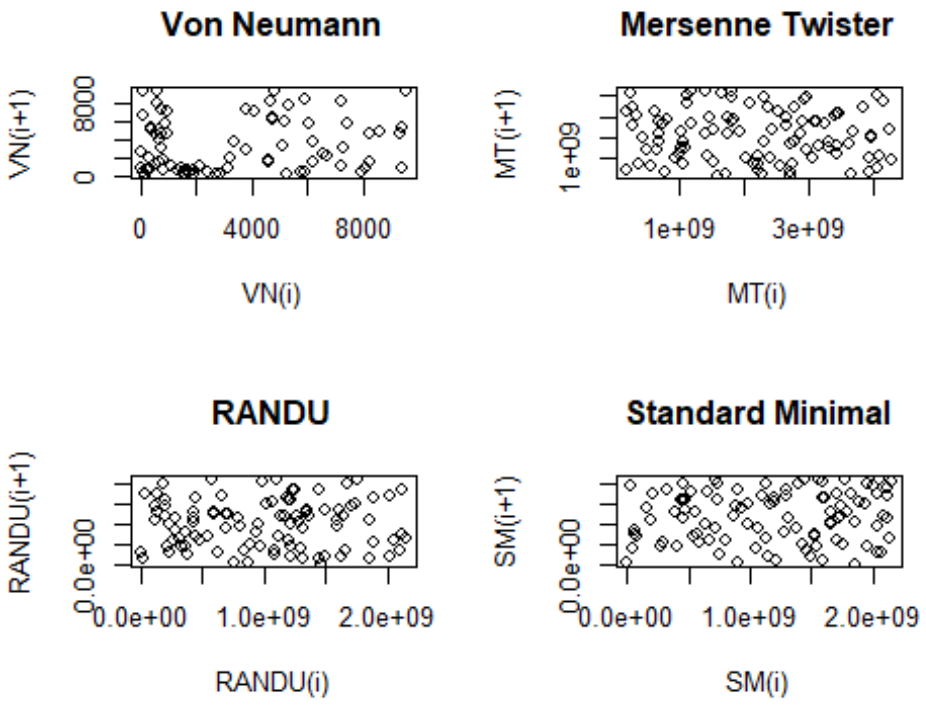
Ici, on trace les graphiques représentants les états obtenus en fonction de l'état précédant grâce à ce code :

```
Plot <- function(vn, mt, randu, sm)
{
  par(mfrow=c(2,2))
  plot(vn[1:(Nsimu-1),1],vn[2:Nsimu,1],xlab='VN(i)', ylab='VN(i+1)',
main='Von Neumann')
  plot(mt[1:(Nsimu-1),1],mt[2:Nsimu,1],xlab='MT(i)', ylab='MT(i+1)',
main='Mersenne Twister')
  plot(RANDU[1:(Nsimu-1),1],RANDU[2:Nsimu,1],xlab='RANDU(i)',
ylab='RANDU(i+1)', main='RANDU')
  plot(sm[1:(Nsimu-1),1],sm[2:Nsimu,1],xlab='SM(i)', ylab='SM(i+1)',
```

```
main='Standard Minimal')
}
```

Ce qui donne cela :

```
Plot(vn, mt, randu, sm)
```



Encore une fois, le générateur de Von Neumann n'apparaît pas homogène de par la non-occupation de tout le graphe qu'il devrait présenter à l'instar des générateurs de Mersenne Twister, RANDU, et Standard Minimal.

Question 3.

Ici, nous allons appliquer le test de Frequency 100 fois à nos algorithmes de générations de nombre aléatoire puis tirer une moyenne des résultats obtenus afin de définir si ces générateurs sont fiables d'après ce test.

Voici les résultats :

```
frequency_vn = rep(0, Nrepet)
frequency_mt = rep(0, Nrepet)
frequency_randu = rep(0, Nrepet)
frequency_sm = rep(0, Nrepet)

for (i in 1:Nrepet)
{
  frequency_vn[i] <- Frequency(vn[,i], 14)
  frequency_mt[i] <- Frequency(mt[,i], 32)
```

```
  frequency_randu[i] <- Frequency(RANDU[,i], 31)
  frequency_sm[i] <- Frequency(sm[,i], 31)
}

# Test Fréquence monobit
# Von Neumann
(mean(frequency_vn))

## [1] 0

# Mersenne Twister
(mean(frequency_mt))

## [1] 0.5055268

# RANDU
(mean(frequency_randu))

## [1] 0.4806587

# Standard Minimal
(mean(frequency_sm))

## [1] 0.5192751
```

Et voici le code utilisé :

```
Frequency <- function(x, nb)
{
  s <- 0
  n <- length(x)*nb
  for (i in 1:length(x))
  {
    bits <- binary(x[i])
    for (j in 1:nb)
    {
      s = s + 2*bits[j] - 1
    }
  }
  sobs <- abs(s)/sqrt(n)
  P <- 2*(1-pnorm(sobs))
  return(P)
}
```

D'après ces résultats, tous les générateurs sont proches de 0.5 qui représente une parfaite distribution des valeurs sauf Von Neumann qui a une valeur inférieure à 0.01, il est donc défini comme de mauvaise qualité d'après le test de Frequency.

Question 4.

Ici, nous allons la même méthodologie que précédemment mais avec le test des runs pour définir si nos générateurs sont fiables selon le test des runs.

Voici les résultats :

```
run_vn = rep(0, Nrepet)
run_mt = rep(0, Nrepet)
run_randu = rep(0, Nrepet)
run_sm = rep(0, Nrepet)

for (i in 1:Nrepet)
{
  run_vn[i] <- Runs(vn[,i], 14)
  run_mt[i] <- Runs(mt[,i], 32)
  run_randu[i] <- Runs(RANDU[,i], 31)
  run_sm[i] <- Runs(sm[,i], 31)
}

# Test Runs
# Von Neumann
(mean(run_vn))

## [1] 0

# Mersenne Twister
(mean(run_mt))

## [1] 0.5162844

# RANDU
(mean(run_randu))

## [1] 0.5306652

# Standard Minimal
(mean(run_sm))

## [1] 0.5108292
```

Et voici le code utilisé :

```
Runs <- function(x, nb)
{
  pi <- 0
  V <- 0
  n <- length(x)*nb
  epsilon <- rep(0, n)
  for (i in 1:length(x))
  {
    bits <- binary(x[i])
    for (j in 1:nb)
    {
      epsilon[(i-1)*nb + j] = bits[j]
    }
  }
```

```
  for (i in 1:n)
  {
    pi = pi + epsilon[i]
    if (i == n || epsilon[i] != epsilon[i+1]) {
      V = V + 1
    }
  }

  pi = pi/n
  to = 2/sqrt(n)

  if (abs(pi - 0.5) >= to) {
    return(0.)
  }

  P <- 2 * (1 - pnorm(abs(V-2*n*pi*(1-pi))/(2*sqrt(n)*pi*(1-pi))))
  return(P)

}
```

D'après le test des runs, le générateur de Von Neumann est encore une fois considéré comme de mauvaise qualité dû à un score moyen retourné nul. Les autres générateurs passent ce test haut la main.

Question 5.

Encore une fois, nous allons itérer 100 fois ce test à nos générateurs pour définir s'ils réussissent à le passer ou non de manière fiable.

Voici le code utilisé :

```
Ordre <- function(x)
{
  P = order.test(x, d=4, echo=FALSE)
  return(P[["p.value"]])
}
```

Et voici les résultats obtenus :

```
ordre_vn = rep(0, Nrepet)
ordre_mt = rep(0, Nrepet)
ordre_randu = rep(0, Nrepet)
ordre_sm = rep(0, Nrepet)

for (i in 1:Nrepet)
{
  ordre_vn[i] <- Ordre(vn[,i])
  ordre_mt[i] <- Ordre(mt[,i])
  ordre_randu[i] <- Ordre(RANDU[,i])
  ordre_sm[i] <- Ordre(sm[,i])
```

```
}

# Test Ordre
# Von Neumann
(mean(ordre_vn))

## [1] 0.0066

# Mersenne Twister
(mean(ordre_mt))

## [1] 0.5

# RANDU
(mean(ordre_randu))

## [1] 0.5

# Standard Minimal
(mean(ordre_sm))

## [1] 0.53
```

D'après ce test, le générateur de Von Neumann est encore une fois rejeté car sa valeur de retour est inférieure à 1%. Quant aux autres générateurs, ils passent tous le test d'ordre.

Conclusion des tests

Pour résumé, le générateur de Von Neumann ne passe aucun des tests, à savoir, le test de Frequency, le test des runs et le test d'Ordre. Les générateurs de RANDU, de Standard Minimal et de Mersenne Twister passent tous les tests haut la main et ne peuvent être différencié grâce à ses tests uniquement.

TP2

Dans ce 2e TP, nous allons simuler une file d'attente et étudier son comportement en fonction de la vitesse d'arrivée des clients et de la vitesse de leur départ.

Question 6.

Dans un 1er temps, nous avons implémenter une fonction nous renvoyant la liste des horaires d'arrivées, de départs et le temps d'attente des clients apparus durant une durée déterminée.

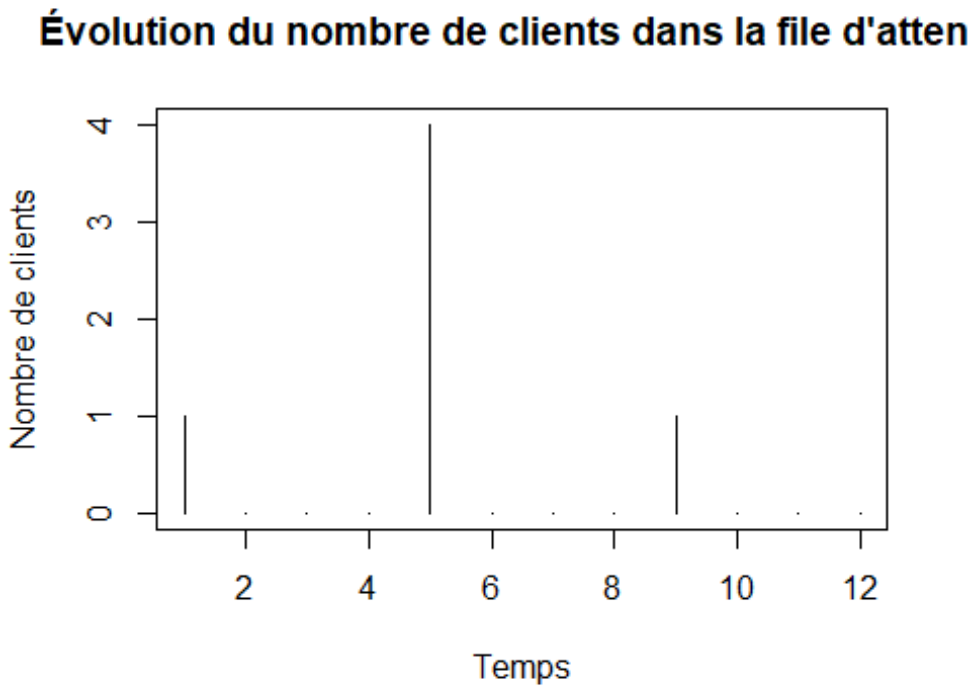
```
result <- FileMM1(lambda, mu, D)

FileMM1 <- function(lambda, mu, D) {
  arrivee <- c()
  depart <- c()
  attente_service <- c()

  temps_arrivee <- 0
```

```
  temps_depart <- 0

  while (temps_arrivee < D)
  {
    intervalle_arrivee <- rexp(1, lambda)
    temps_arrivee <- temps_arrivee + intervalle_arrivee

    if (temps_arrivee < D) {
      arrivee <- c(arrivee, temps_arrivee)
      duree_attente <- rexp(1, mu)
      if (temps_arrivee > temps_depart)
      {
        temps_attente_service <- 0;
        temps_depart <- temps_arrivee + duree_attente
      }
      else
      {
        temps_attente_service <- temps_depart - temps_arrivee
        temps_depart <- temps_depart + duree_attente
      }
      depart <- c(depart, temps_depart)
      attente_service <- c(attente_service, temps_attente_service)
    }
  }

  return(list(arrivee = arrivee, depart = depart, attente_service =
attente_service))
}
```

Question 7.

Afin de mieux visualiser le nombre de clients qui attendent à chaque instant, nous avons rajouté ce bout de code :

```
temps <- seq(1, D, 1)
nb_clients <- sapply(temps, function(t) sum(result$arrivee <= t &
result$depart > t))
nb_clients_attente <- sapply(temps, function(t) max(0, sum(result$arrivee <=
t & result$depart > t)-1))

plot(temps, nb_clients, type = "h", xlab = "Temps", ylab = "Nombre de
clients", main = "Évolution du nombre de clients dans la file d'attente")
```



Nous pouvons observer qu'avec une moyenne de 10 clients par heure à qui arrivent et une moyenne de 20 clients par heure qui partent, la file d'attente est fluide et aucun client n'attend plus d'une heure.

Question 8. Nous calculons les valeurs théoriques suivantes : nombre moyen de clients dans le système (E_N), temps de présence moyen d'un client dans le système (E_W), le nombre moyen de clients en attente d'être servis (E_Na) et le temps moyen d'attente pour un client d'être servi (E_Wa). On le compare ensuite aux valeurs trouvées expérimentalement.

```
alpha <- lambda / mu
E_N_th <- alpha / (1 - alpha)
E_W_th <- E_N_th / lambda
E_Wa_th <- E_W_th - 1/mu
E_Na_th <- lambda * E_Wa_th

temps_attente <- result$depart - result$arrivee

E_N = mean(nb_clients)
E_W = mean(temps_attente)
E_Wa = mean(result$attente_service)
E_Na = mean(nb_clients_attente)
```

Ici, result$attente_service correspond à une nouvelle variable que nous avons rajouté dans FileMM1.R et qui rajoute le temps d'attente de service.

Pour les valeurs initiales de lambda=10, mu=20 et D=12, nous obtenons les valeurs suivantes, montrant des valeurs expérimentales proches des valeurs théoriques :

```
> E_N_th
[1] 1
> E_N
[1] 1.2
>
> E_W_th
[1] 0.1
> E_W
[1] 0.096
>
> E_Wa_th
[1] 0.05
> E_Wa
[1] 0.046
>
> E_Na_th
[1] 0.5
> E_Na
[1] 0.67
```