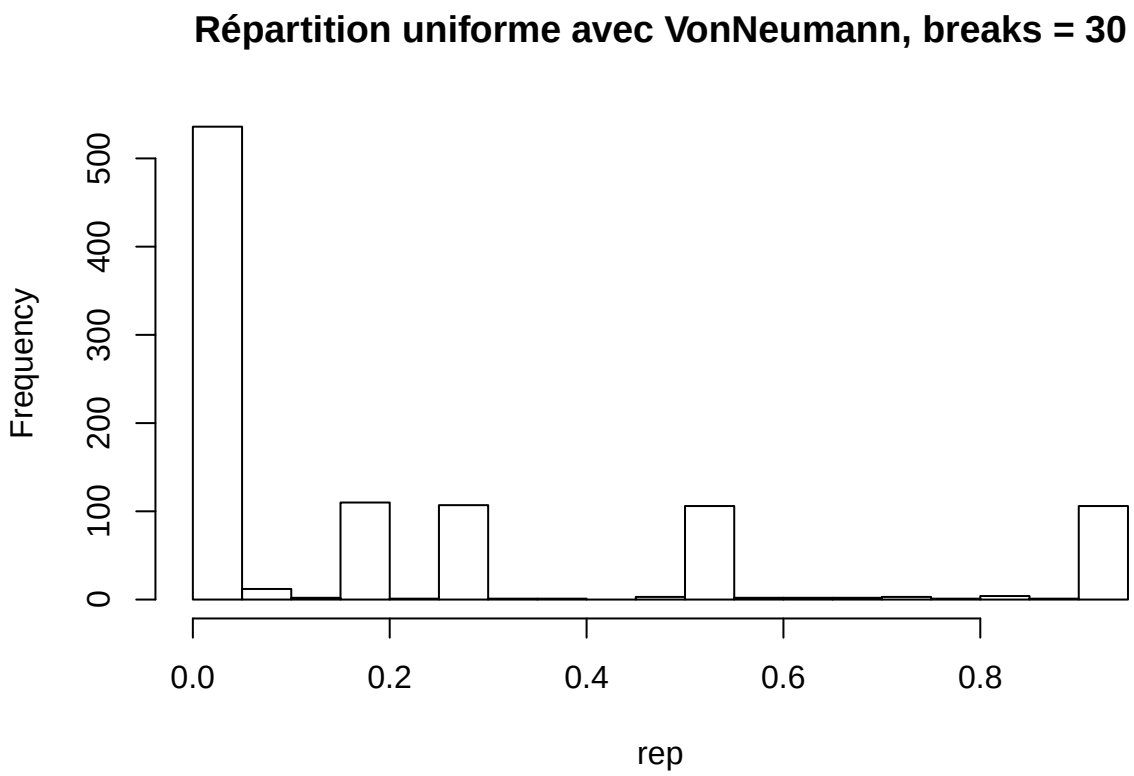
TP génération de nombres aléatoires et probabilités

Aleryc Serrania Marie-Carmen Prévot Oumar Diakhaby

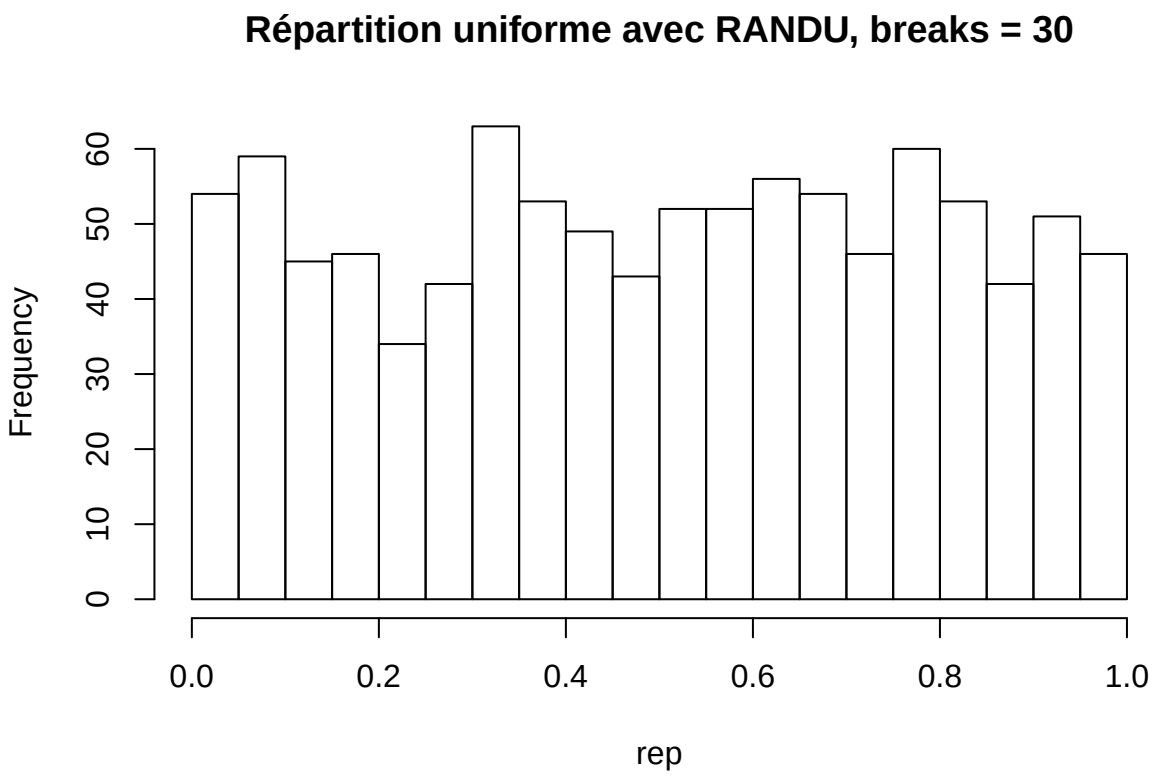
Question 2

Pour VonNeumann, la répartition est très mauvaise et pas visiblement uniforme. Les autres générateurs ont visiblement l'air de répartir uniformément les valeurs

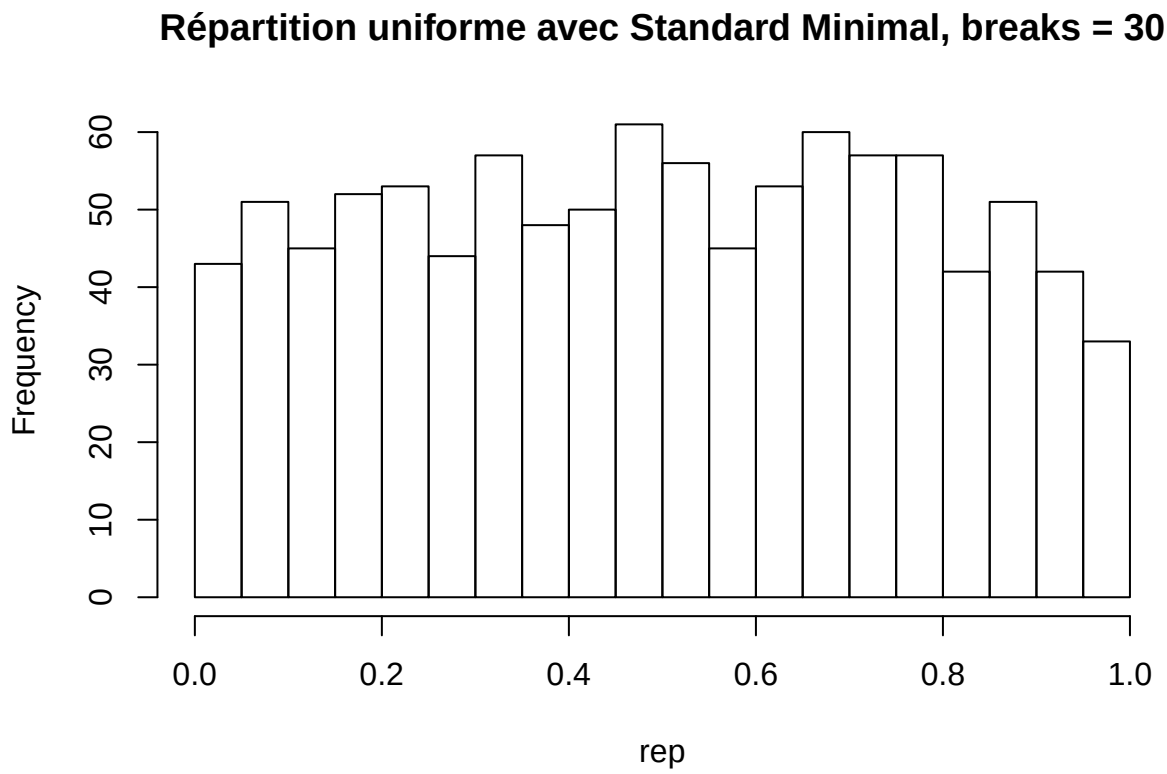
```
breaks = 30
rep <- VonNeumann(k, 1, seed) / (9999)
hist(rep,
     main = paste("Répartition uniforme avec VonNeumann, breaks =", breaks),
     breaks = breaks)
```



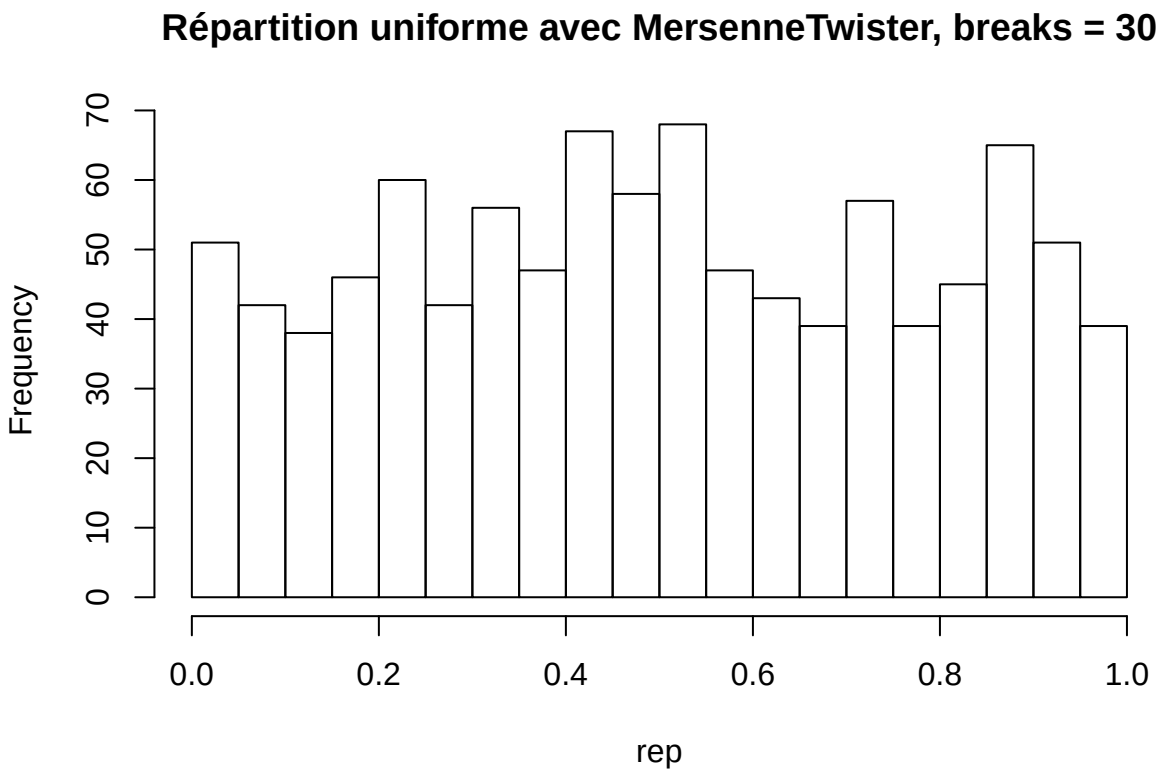

```
rep <- randu(k, seed) / (2^31 - 1)
hist(rep,
     main = paste("Répartition uniforme avec RANDU, breaks =", breaks),
     breaks = breaks)
```




```
rep <- std_minimal(k, seed) / (2^31 - 2)
hist(rep,
     main = paste("Répartition uniforme avec Standard Minimal, breaks =", breaks),
     breaks = breaks)
```

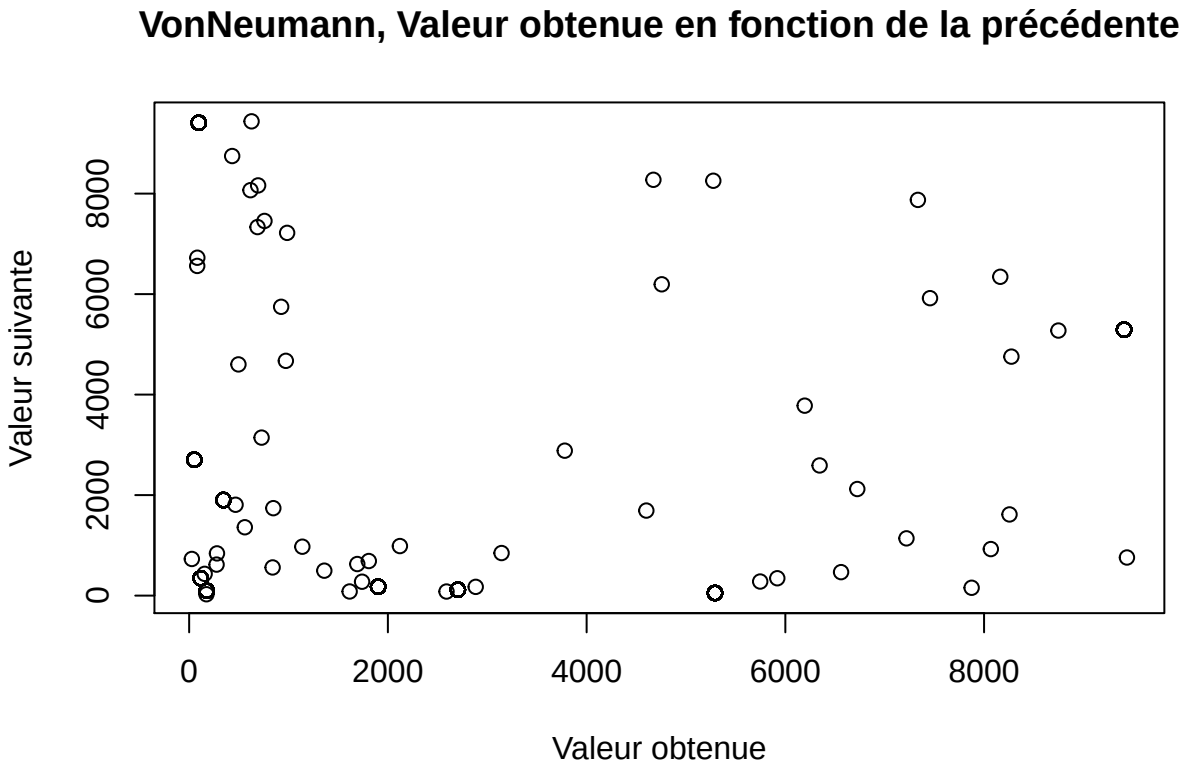



```
rep <- MersenneTwister(k, 1, seed) / (2^32 - 1)
hist(rep,
     main = paste("Répartition uniforme avec MersenneTwister, breaks =", breaks),
     breaks = breaks)
```

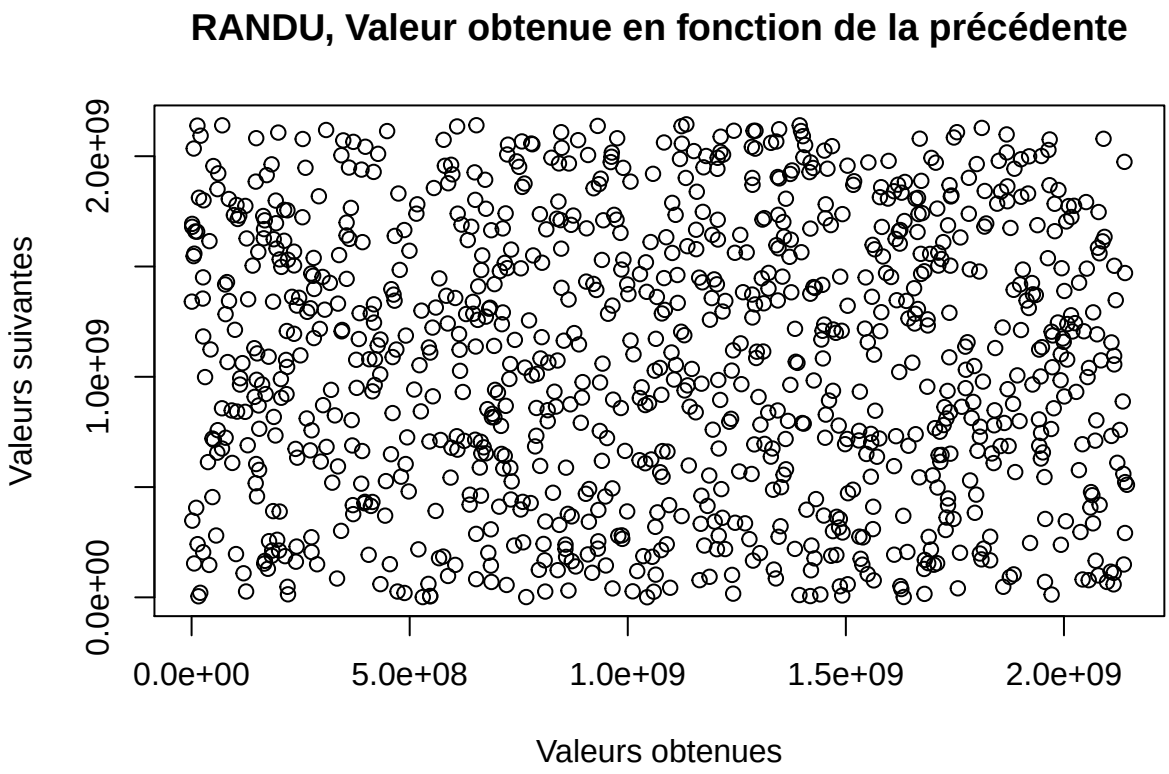


Pour VonNeumann, on remarque les points sont peu étalés. Par exemple, lorsque la valeur actuelle est proche de zéro, les valeurs suivantes restent proche de 0. Cela est due à l'élévation au carré de la valeur. $0^2 = 0$ et $1^2 = 1$, donc une fois la valeur de 0 ou 1 atteinte, le système n'évolue plus. Pour les autres, la génération semble être chaotique : la moindre variation de la valeur actuelle peut engendrer une valeur suivante totalement différente.

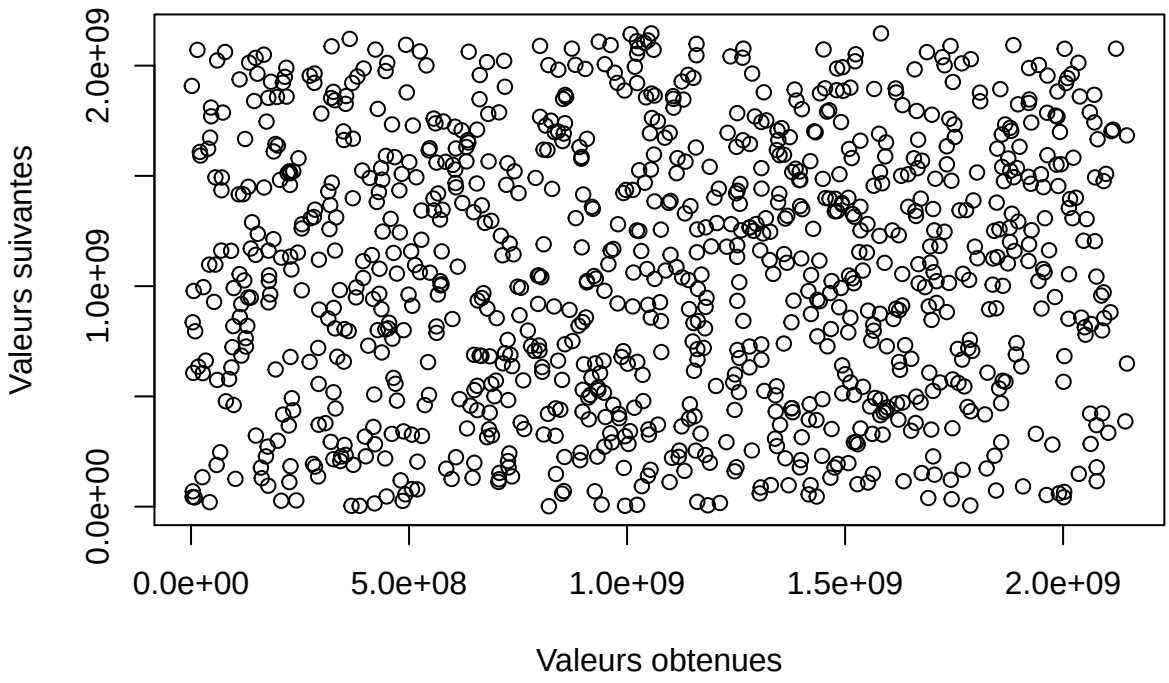
```
rep <- VonNeumann(k, 1, seed)
plot(rep[1:k-1], rep[2:k],
     main = "VonNeumann, Valeur obtenue en fonction de la précédente",
     xlab = "Valeur obtenue",
     ylab = "Valeur suivante")
```



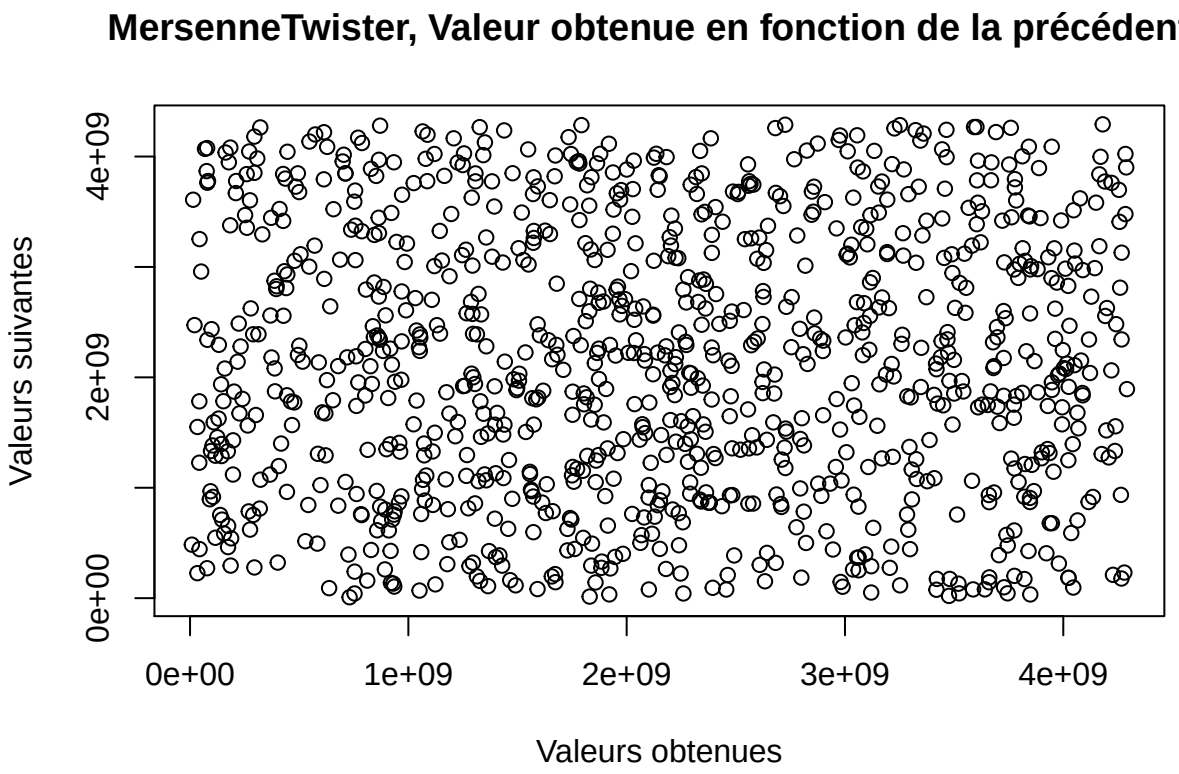

```
rep <- randu(k, seed)
plot(rep[1:k-1], rep[2:k],
     main = "RANDU, Valeur obtenue en fonction de la précédente",
     xlab = "Valeurs obtenues",
     ylab = "Valeurs suivantes")
```




```
rep <- std_minimal(k, seed)
plot(rep[1:k-1], rep[2:k],
     main = "StdMinimal, Valeur obtenue en fonction de la précédente",
     xlab = "Valeurs obtenues",
     ylab = "Valeurs suivantes")
```




```r
rep <- MersenneTwister(k, 1, seed)
plot(rep[1:k-1], rep[2:k],
     main = "MersenneTwister, Valeur obtenue en fonction de la précédente",
     xlab = "Valeurs obtenues",
     ylab = "Valeurs suivantes")
```



Question 3

Pour VonNeumann et RANDU, on observe pour la plupart des tests que la p valeur est en dessous des 1 %.

Pour Standard Minimal et MersenneTwister, on observe très peu de p valeurs inférieures à 1 %.

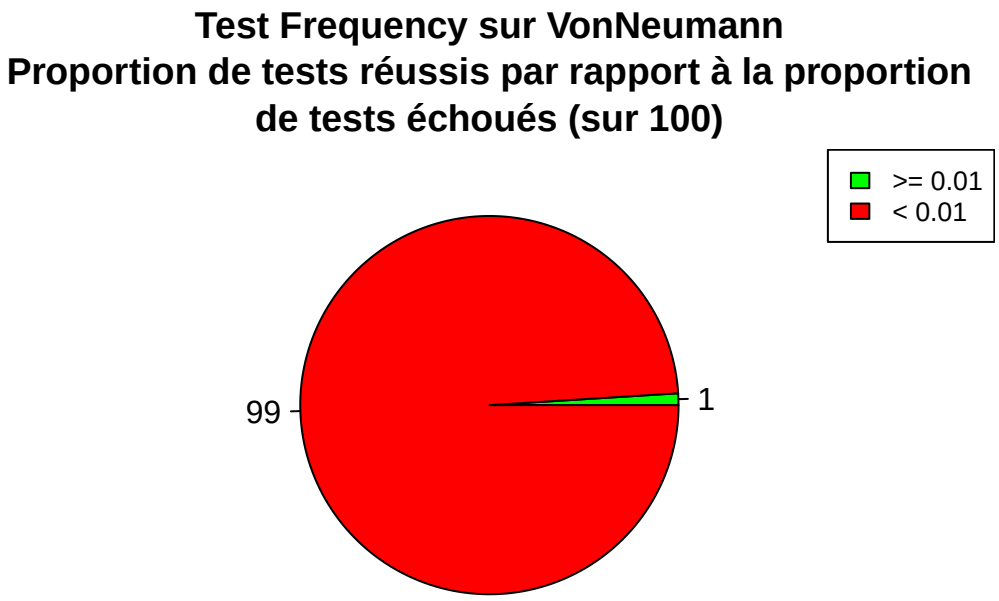
Ainsi, RANDU et VonNeumann ne passe le premier test et n'est donc pas un bon générateur aléatoire.

Pour Standard Minimal et MersenneTwister, on ne peut rien conclure pour l'instant.

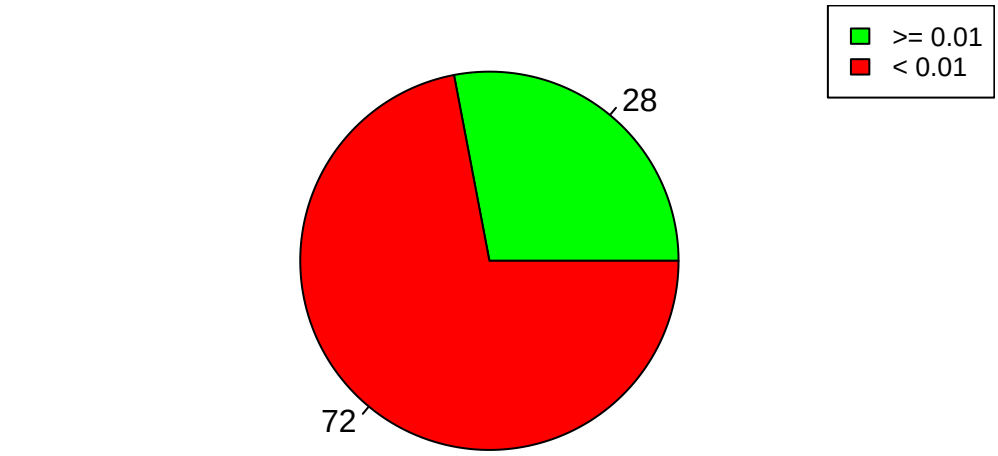
```
nb_seed <- 100
seed <- sample.int(2^31, nb_seed)
ShowTestResults <- function(frequence, name_generator, name_test)
{
  name =  paste(paste("Test", name_test, "sur", name_generator),
                "Proportion de tests réussis par rapport à la proportion",
                  paste("de tests échoués (sur ", nb_seed, ")", sep=""),
                sep="\n")
  nb_reussis <- PercentageTest(frequence, 0.01)
  percentage <- nb_reussis / nb_seed
  # plot(frequence, xlim = c(0, 100), ylim = c(0, 1.0))
  # abline(h = 0.01, col = "red")
  # mtext(paste("Proportion de tests réussis : ", percentage * 100, " %"), side = 3)
  pie(c(percentage, 1 - percentage), labels = c(nb_reussis, nb_seed - nb_reussis),
      col = c("green", "red"), main = name)
```

```
  legend("topright", c(">= 0.01", "< 0.01"), cex = 0.8, fill = c("green", "red"))
}
```

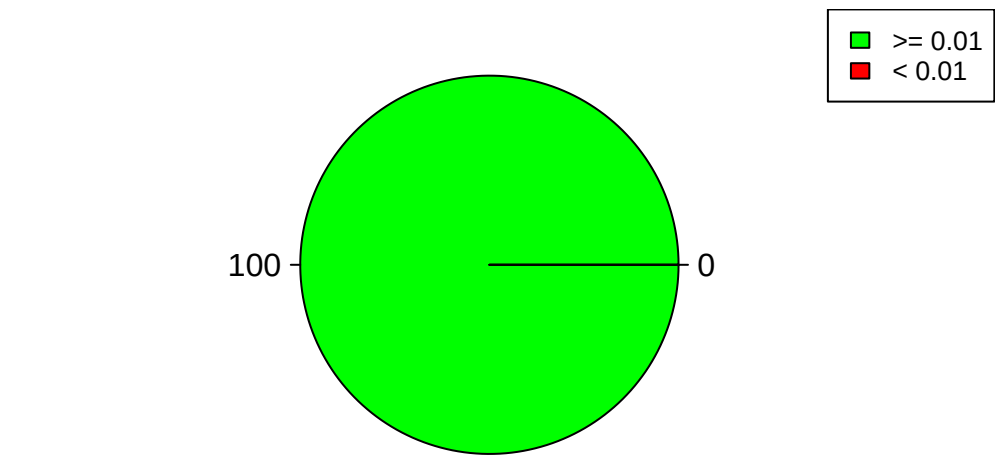
```
frequence <- sapply(seed, function(s) Frequency(VonNeumann(k, 1, s), 14))
ShowTestResults(frequence, "VonNeumann", "Frequency")
```



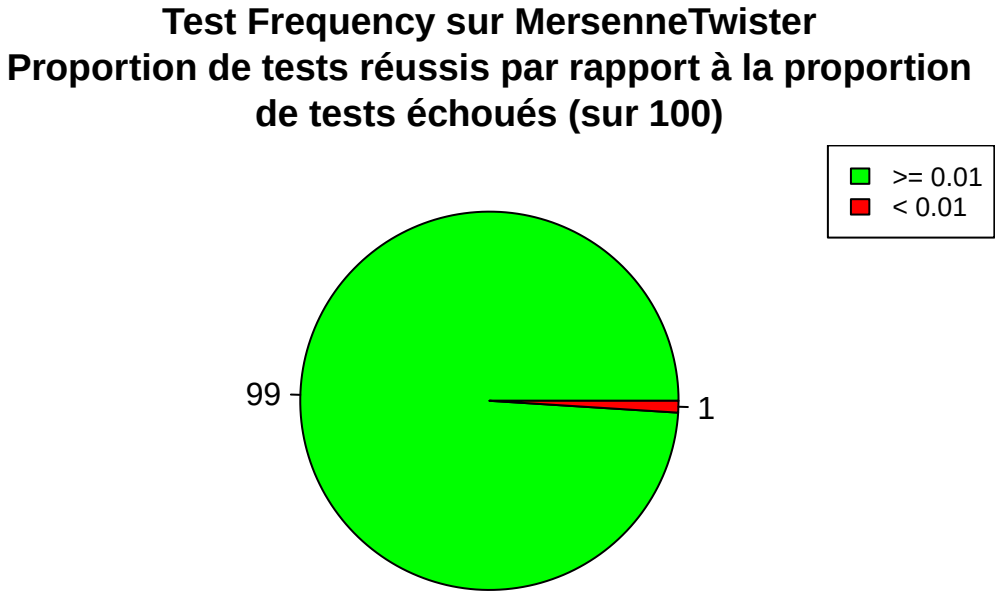

```
frequence <- sapply(seed, function(s) Frequency(randu(k, s), 31))
ShowTestResults(frequence, "RANDU", "Frequency")
```




```
frequence <- sapply(seed, function(s) Frequency(std_minimal(k, s), 31))
ShowTestResults(frequence, "StdMinimal", "Frequency")
```




```
frequence <- sapply(seed, function(s) Frequency(MersenneTwister(k, 1, s), 32))
ShowTestResults(frequence, "MersenneTwister", "Frequency")
```



Question 4

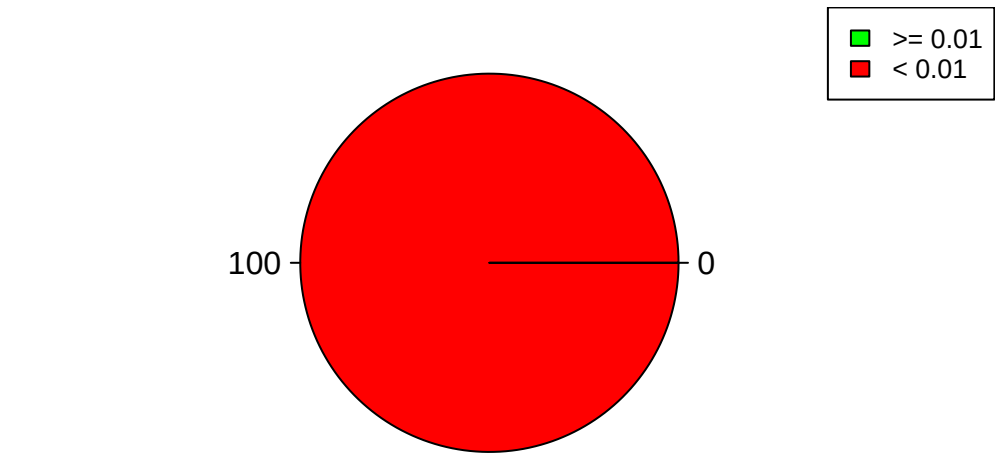
Pour VonNeumann et RANDU, on observe pour la plupart des tests que la p valeur est en dessous des 1 %.

Pour Standard Minimal et MersenneTwister, on observe très peu de p valeurs inférieures à 1 %.

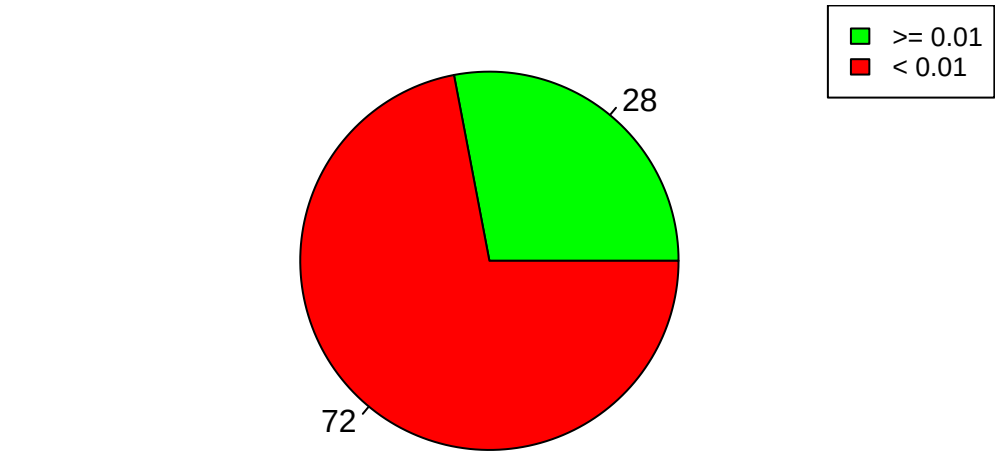
RANDU et VonNeumann ne passe donc pas le second test non plus.

Pour Standard Minimal et MersenneTwister, on ne peut rien conclure pour l'instant.

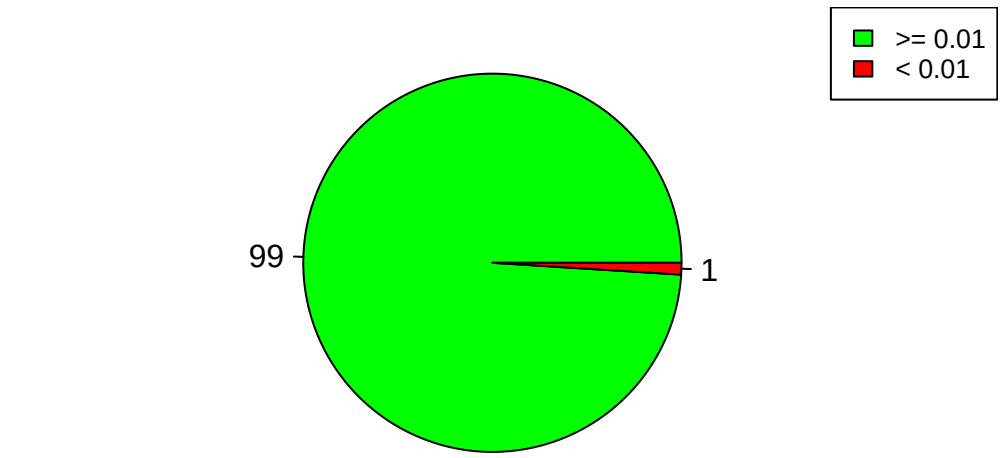
```
frequence <- sapply(seed, function(s) Runs(VonNeumann(k, 1, s), 14))
ShowTestResults(frequence, "VonNeumann", "Runs")
```



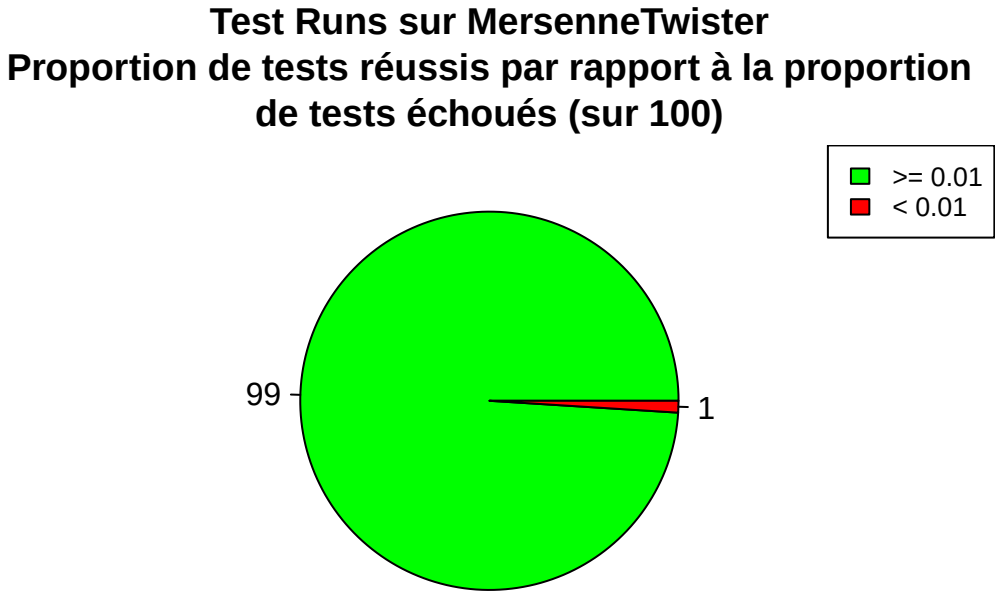

```
frequence <- sapply(seed, function(s) Runs(randu(k, s), 31))
ShowTestResults(frequence, "RANDU", "Runs")
```




```
frequence <- sapply(seed, function(s) Runs(std_minimal(k, s), 31))
ShowTestResults(frequence, "StdMinimal", "Runs")
```




```
frequence <- sapply(seed, function(s) Runs(MersenneTwister(k, 1, s), 32))
ShowTestResults(frequence, "MersenneTwister", "Runs")
```



Question 5

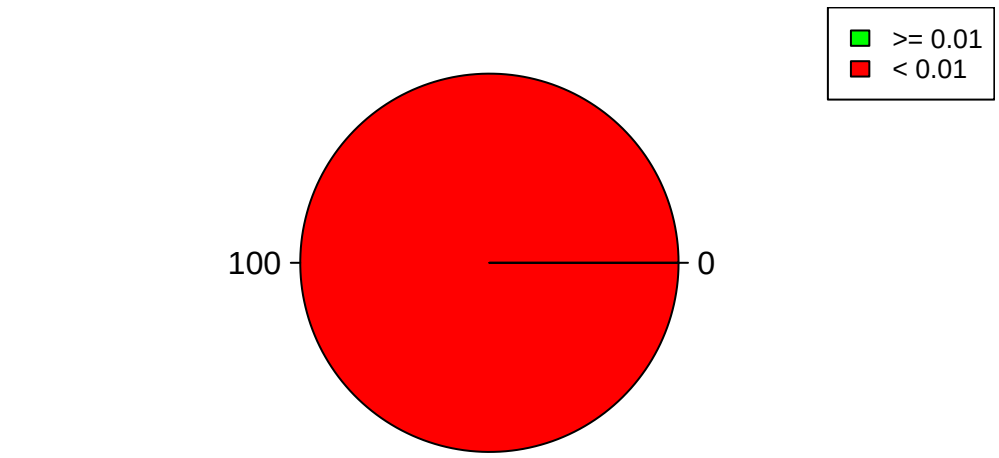
Cette fois-ci c'est seulement pour VonNeumann que l'on observe un grand nombre de tests dont la p valeur est inférieure à 1%.

Pour Standard Minimal, MersenneTwister et RANDU, on observe très peu de p valeurs inférieures à 1 %.

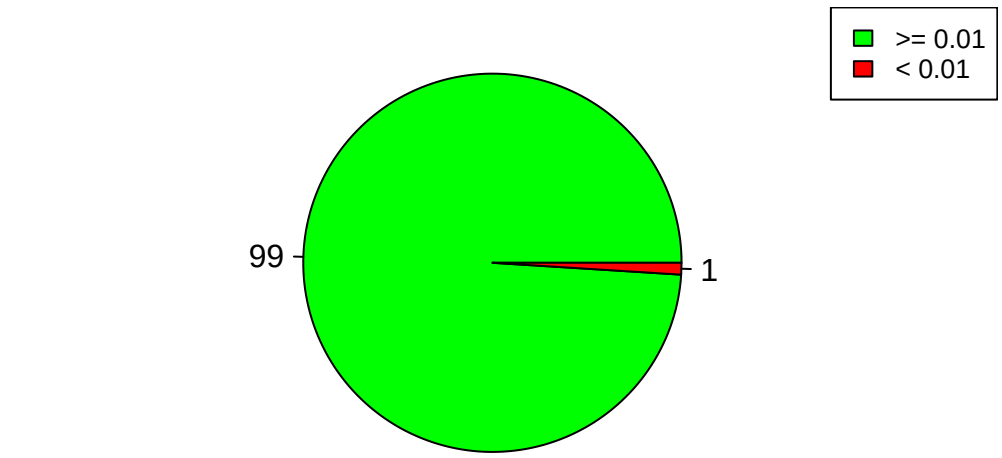
VonNeumann ne passe donc pas le troisième test non plus.

Pour Standard Minimal et MersenneTwister, on ne peut toujours rien conclure pour l'instant mais ils semblent être de très bon générateurs puisqu'ils ont passés les trois tests.

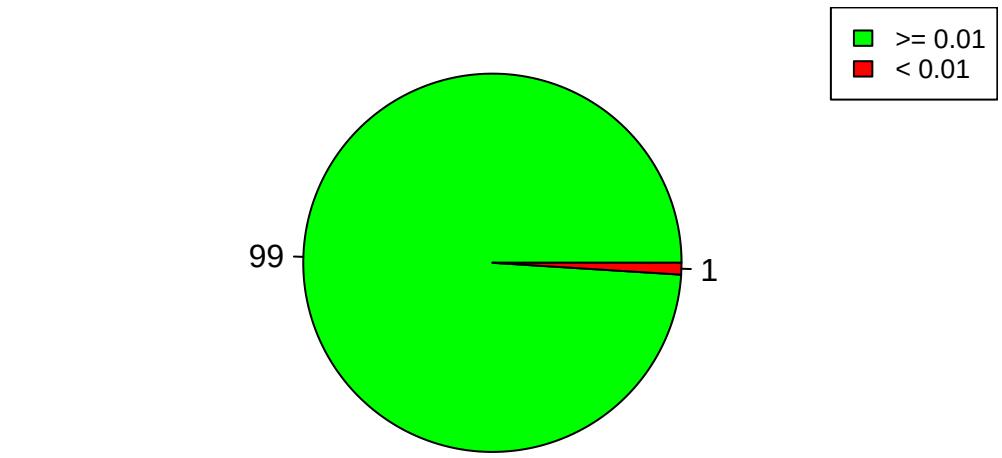
```
frequence <- sapply(seed, function(s) {
  order.test(as.vector(VonNeumann(k, 1, s)), d = 4, echo=FALSE)$p.value
})
ShowTestResults(frequence, "VonNeumann", "Order")
```



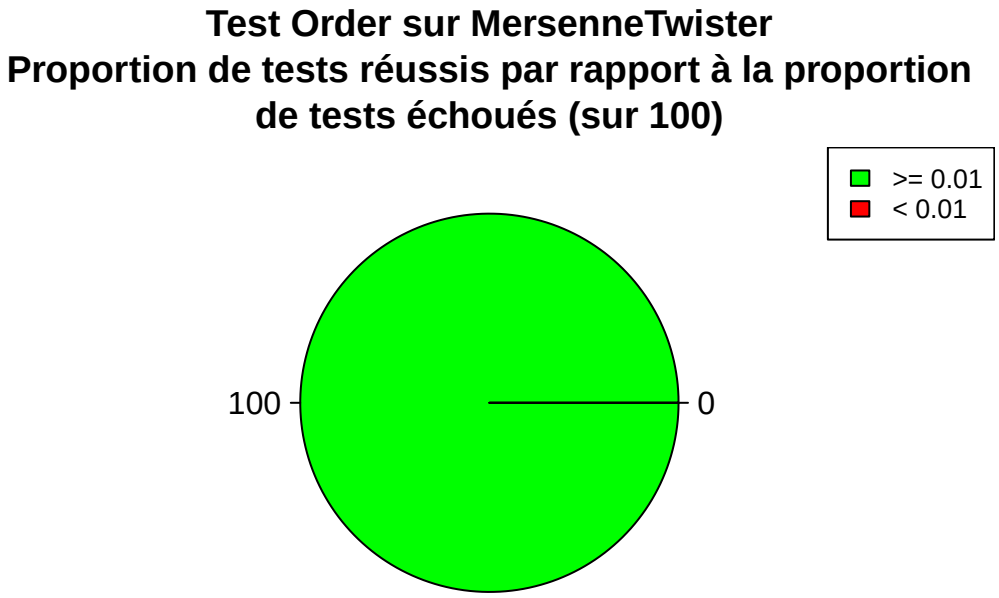

```
frequence <- sapply(seed, function(s) {
  order.test(randu(k, s), d = 4, echo=FALSE)$p.value
})
ShowTestResults(frequence, "RANDU", "Order")
```




```
frequence <- sapply(seed, function(s) {
  order.test(std_minimal(k, s), d = 4, echo=FALSE)$p.value
})
ShowTestResults(frequence, "StdMinimal", "Order")
```




```
frequence <- sapply(seed, function(s) {
  order.test(as.vector(MersenneTwister(k, 1, s)), d = 4, echo=FALSE)$p.value
})
ShowTestResults(frequence, "MersenneTwister", "Order")
```

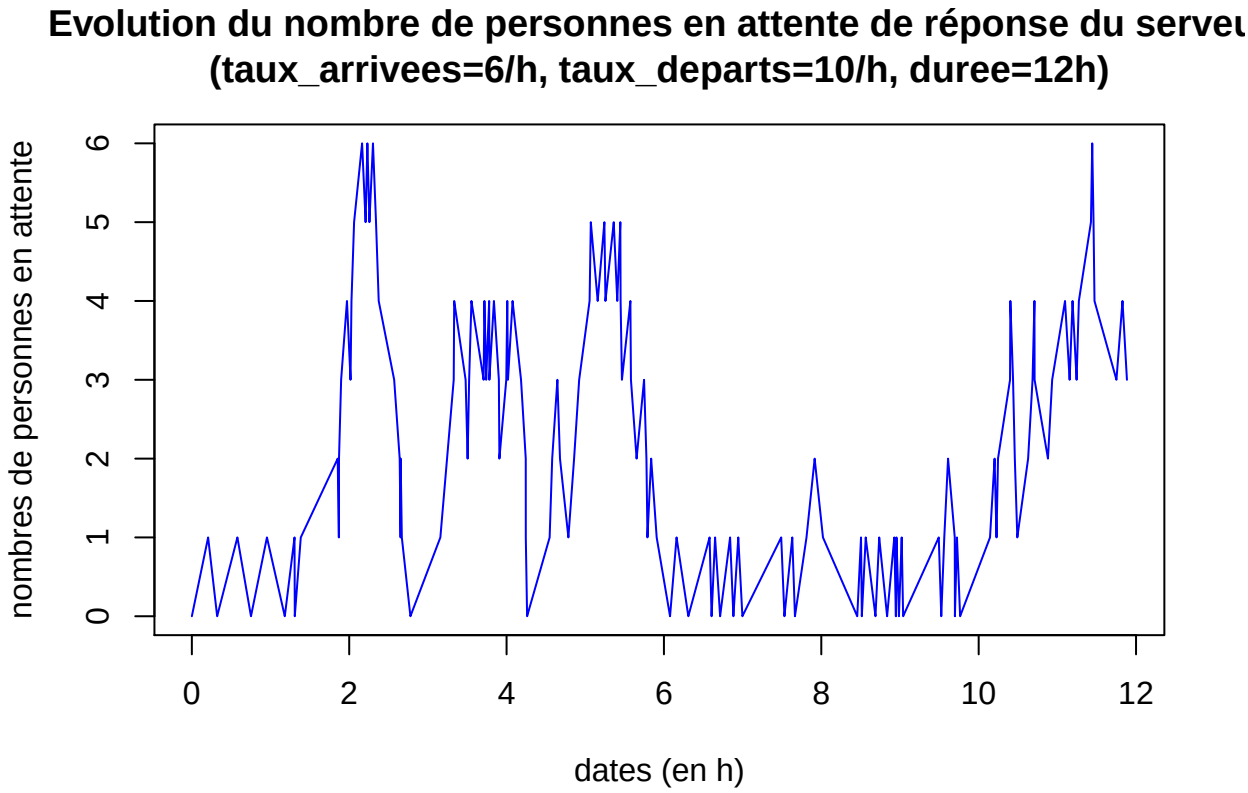


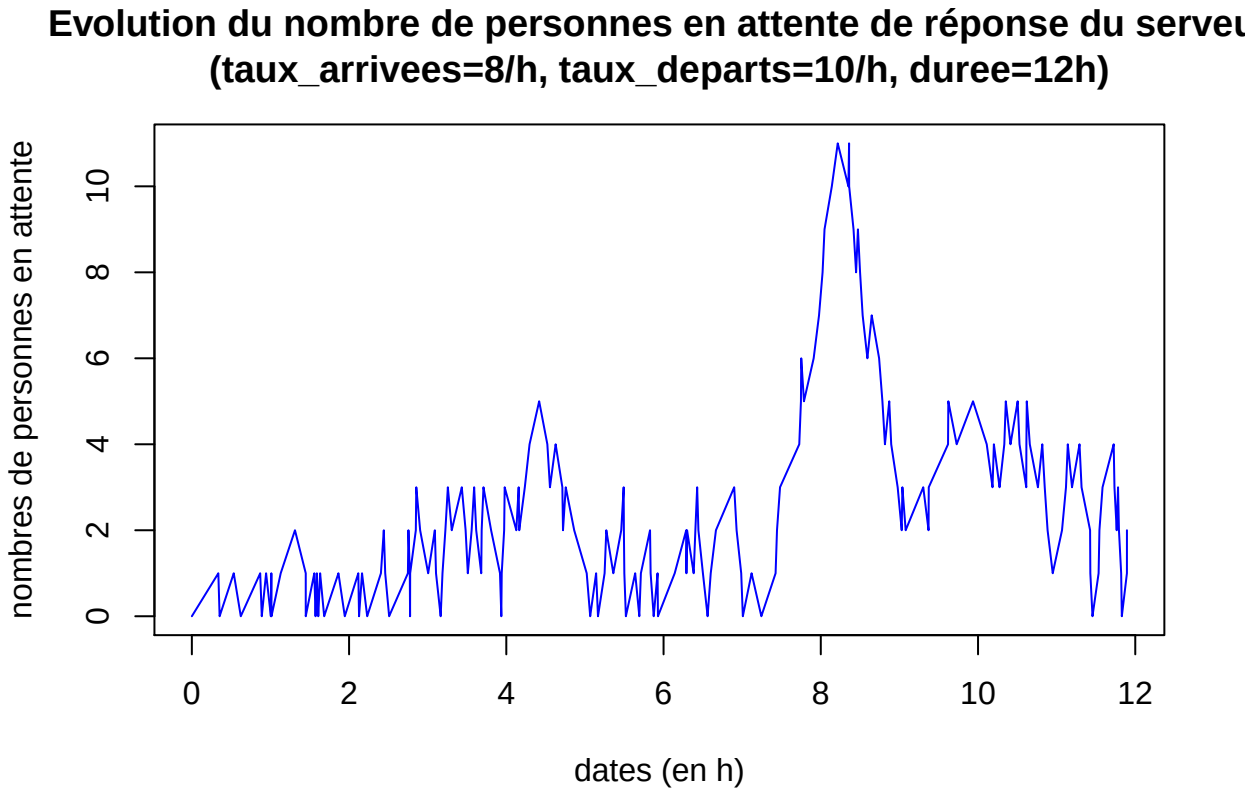
Question 7

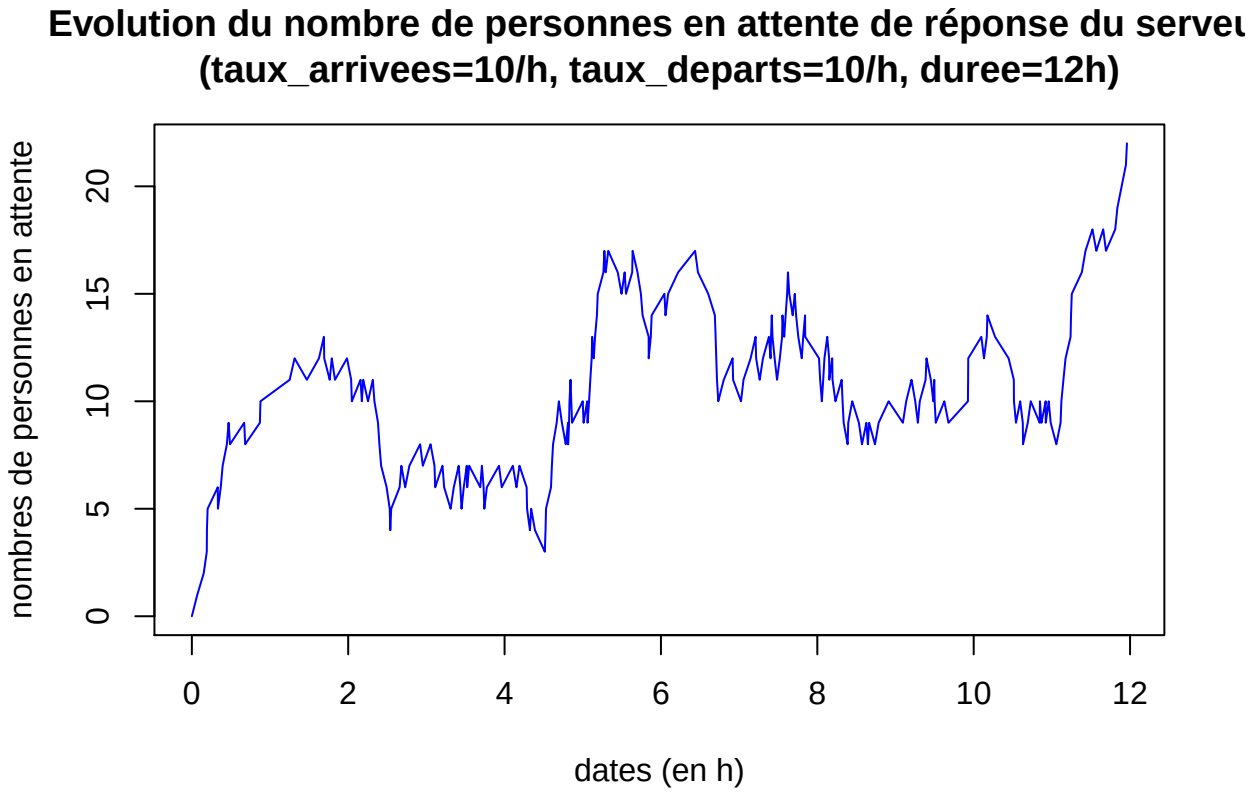
Les résultats montrent que plus on augmente la fréquence d'arrivées, moins la file d'attente a de chances de se vider et plus la file est vite surchargée.

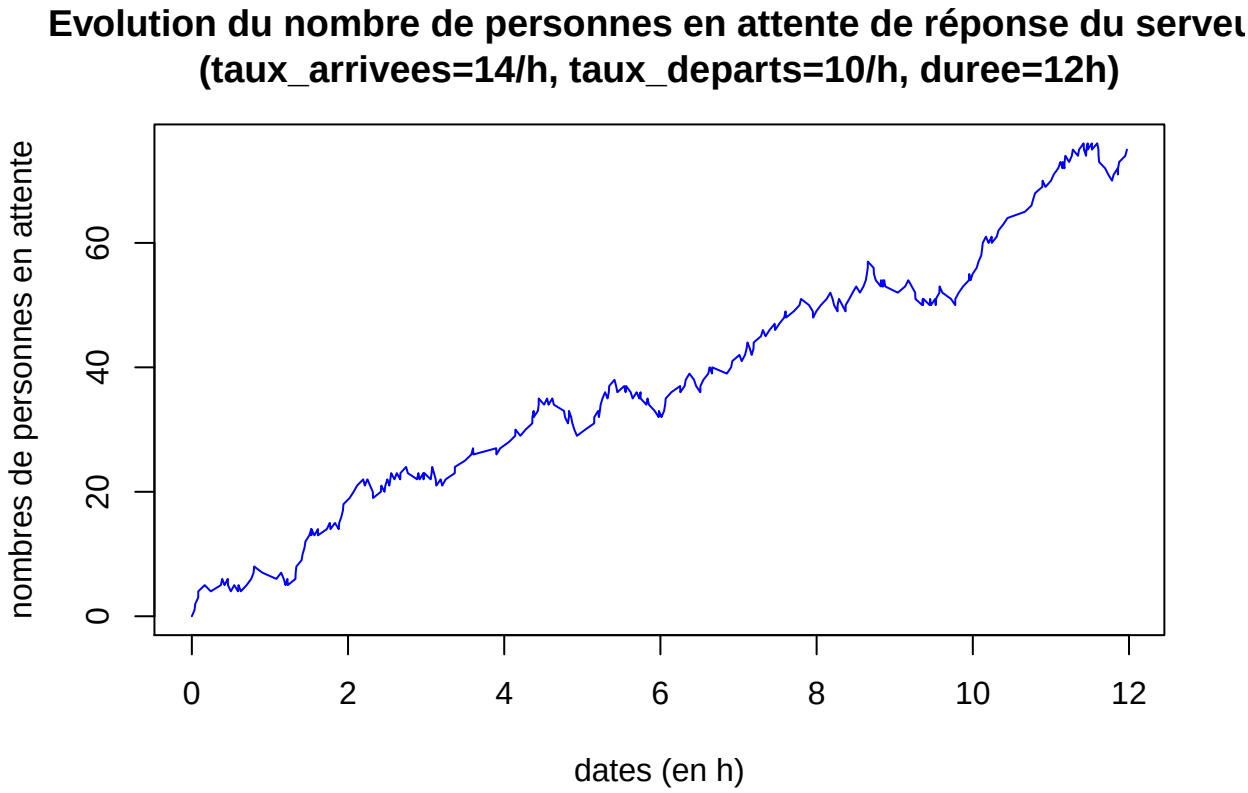
```
ShowEvolutionFile <- function(taux_arrivees, taux_departs, duree)
{
  title = paste("Evolution du nombre de personnes en attente de réponse du serveur",
                paste("(taux_arrivees=", taux_arrivees, "/h, taux_departs=", taux_departs,
                      "/h, duree=", duree, "h)",  sep=""),
                sep="\n")
  file = FileMM1(taux_arrivees, taux_departs, duree)
  evolution = EvolutionFileMM1(file$arrivees, file$departs, duree)
  plot(x=evolution$dates, y=evolution$nombres, type="l", col = "blue", main = title,
       xlab = "dates (en h)", ylab = "nombres de personnes en attente")
}
```

```
taux_departs = 10
duree = 12

for (taux_arrivees in c(6, 8, 10, 14))
{
  ShowEvolutionFile(taux_arrivees, taux_departs, duree)
}
```


Evolution du nombre de personnes en attente de réponse du serveu
(taux_arrivees=6/h, taux_departs=10/h, duree=12h)
nombres de personnes en attente
0
1
2
3
4
5
6
0
2
4
6
8
10
12
dates (en h)


Evolution du nombre de personnes en attente de réponse du serveu
(taux_arrivees=8/h, taux_departs=10/h, duree=12h)
nombres de personnes en attente
0
2
4
6
8
10
0
2
4
6
8
10
12
dates (en h)


Evolution du nombre de personnes en attente de réponse du serveu
(taux_arrivees=10/h, taux_departs=10/h, duree=12h)
nombres de personnes en attente
0
5
10
15
20
0
2
4
6
8
10
12
dates (en h)



Question 8

On remarque que plus le taux d'arrivées est proche du taux de départs, moins la moyenne empirique est proche de la moyenne théorique : il faut une durée plus longue pour s'en rapprocher.

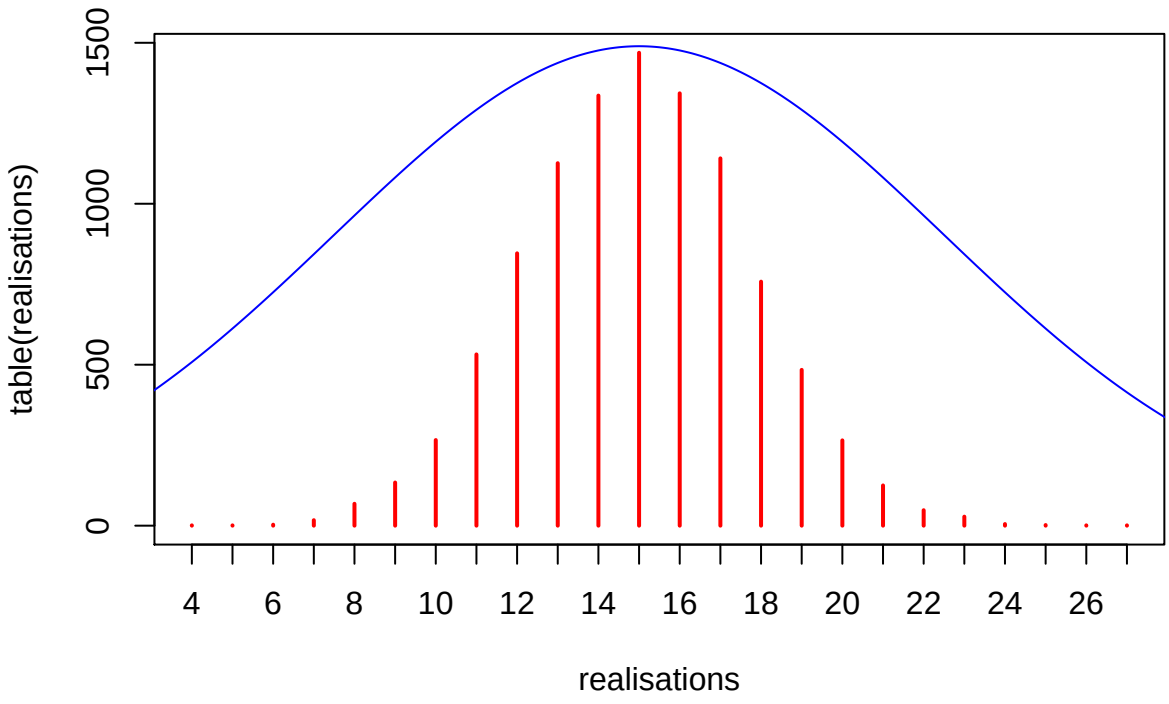
```
VerificationFormuleLittle <- function(taux_arrivees, taux_departs, duree)
{
  iterations <- 100
  means <- vector(length=iterations)
  temps_moyens <- vector(length=iterations)
  lambda <- taux_arrivees
  mu <- taux_departs
  for (i in 1:iterations)
  {
    file = FileMM1(lambda, mu, duree)
    evolution = EvolutionFileMM1(file$arrivees, file$departs, duree)
    means[i] <- weighted.mean(evolution$nombres, evolution$durees)
    temps_moyens[i] <- mean(file$temps_attente_moyen)
  }

  cat(paste("Taux d'arrivees = ", taux_arrivees, "/h, Taux de départs = ", taux_departs, "/h, Durée = "
  # plot(means, type="p")
  cat(paste("Nombre moyen de personnes sur le serveur (moyenne sur", iterations ,"itérations) = ", mean
  cat(paste("Nombre moyen de personnes sur le serveur (théorique) = ", MoyenneTheoriqueN(taux_arrivees,
```

```
  # plot(temps_moyens, type="p")
  cat(paste("Temps d'attente moyen sur le serveur (moyenne sur", iterations ,"itérations) = ", mean(temp
  cat(paste("Temps d'attente moyen sur le serveur (théorique) = ", MoyenneTheoriqueW(taux_arrivees, taux
  cat("\n")
}


for (taux_arrivees in c(2, 6, 8, 9.9))
{
  VerificationFormuleLittle(taux_arrivees, 10, 100)
}
```

```
## Taux d'arrivees =  2 /h, Taux de départs =  10 /h, Durée =  100 h
## Nombre moyen de personnes sur le serveur (moyenne sur 100 itérations) =  0.253329961350039
## Nombre moyen de personnes sur le serveur (théorique) =  0.25
## Temps d'attente moyen sur le serveur (moyenne sur 100 itérations) =  0.125182203471357 h
## Temps d'attente moyen sur le serveur (théorique) =  0.125 h
##
## Taux d'arrivees =  6 /h, Taux de départs =  10 /h, Durée =  100 h
## Nombre moyen de personnes sur le serveur (moyenne sur 100 itérations) =  1.51921354839162
## Nombre moyen de personnes sur le serveur (théorique) =  1.5
## Temps d'attente moyen sur le serveur (moyenne sur 100 itérations) =  0.251047320052432 h
## Temps d'attente moyen sur le serveur (théorique) =  0.25 h
##
## Taux d'arrivees =  8 /h, Taux de départs =  10 /h, Durée =  100 h
## Nombre moyen de personnes sur le serveur (moyenne sur 100 itérations) =  3.95792201169473
## Nombre moyen de personnes sur le serveur (théorique) =  4
## Temps d'attente moyen sur le serveur (moyenne sur 100 itérations) =  0.488901477516282 h
## Temps d'attente moyen sur le serveur (théorique) =  0.5 h
##
## Taux d'arrivees =  9.9 /h, Taux de départs =  10 /h, Durée =  100 h
## Nombre moyen de personnes sur le serveur (moyenne sur 100 itérations) =  22.1347765723668
## Nombre moyen de personnes sur le serveur (théorique) =  98.9999999999999
## Temps d'attente moyen sur le serveur (moyenne sur 100 itérations) =  2.20747432813267 h
## Temps d'attente moyen sur le serveur (théorique) =  9.99999999999999 h
```

Question Bonus 1

```
nb_iterations <- 10000
realisations <- vector(length = nb_iterations)
n <- 30
p <- 0.5
for (i in 1:nb_iterations)
{
  realisations[i] <- LoiBinomiale(n, p)
}
plot(table(realisations), col = "red")

# par(new = TRUE)
xseq<-seq(0, n, 0.1)
lines(xseq, 28000*dnorm(xseq, n*p, n*p*(1-p)), col='blue', type='l')
```



Question Bonus 2

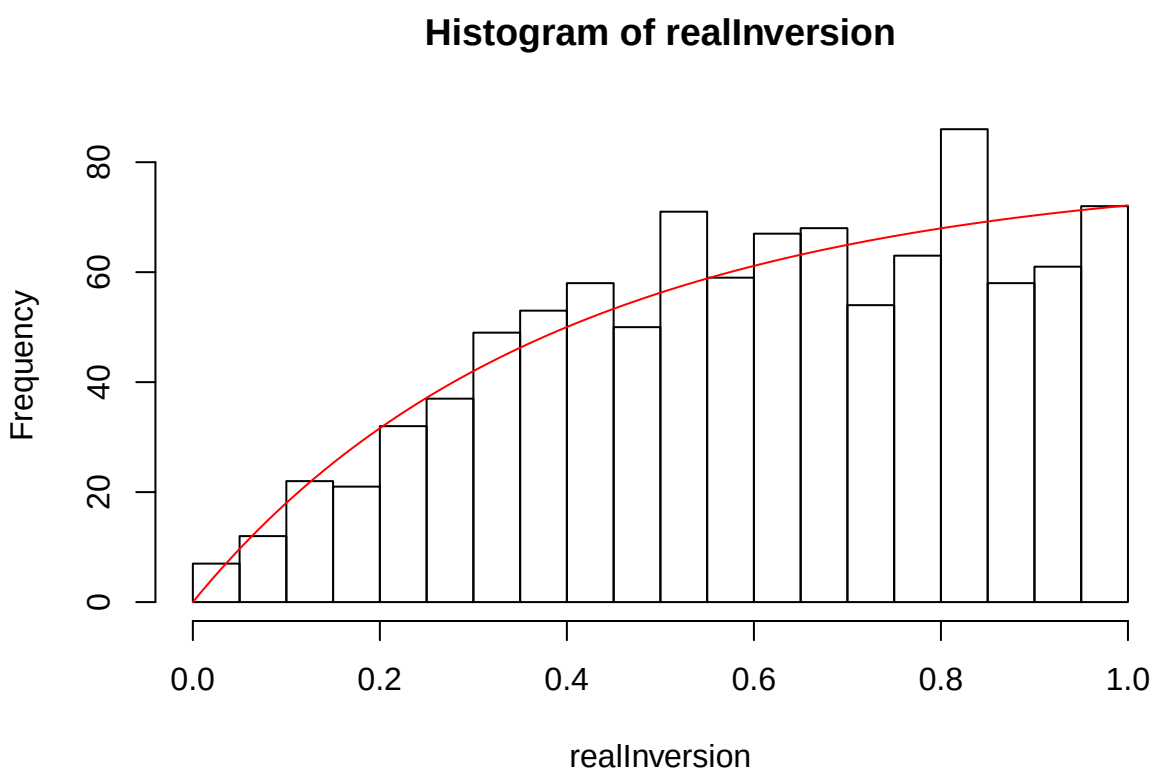
Pour le même résultat, le temps d'exécution moyen de la simulation par rejet est plus grand (d'un facteur 10) que celui de la simulation par inversion. Il faut donc favoriser l'utilisation de la simulation par inversion dès que possible.

```
nb_iterations <- 1000
microbenchmark::microbenchmark(times=nb_iterations, LoiBonusInversion(), LoiBonusRejet())
```

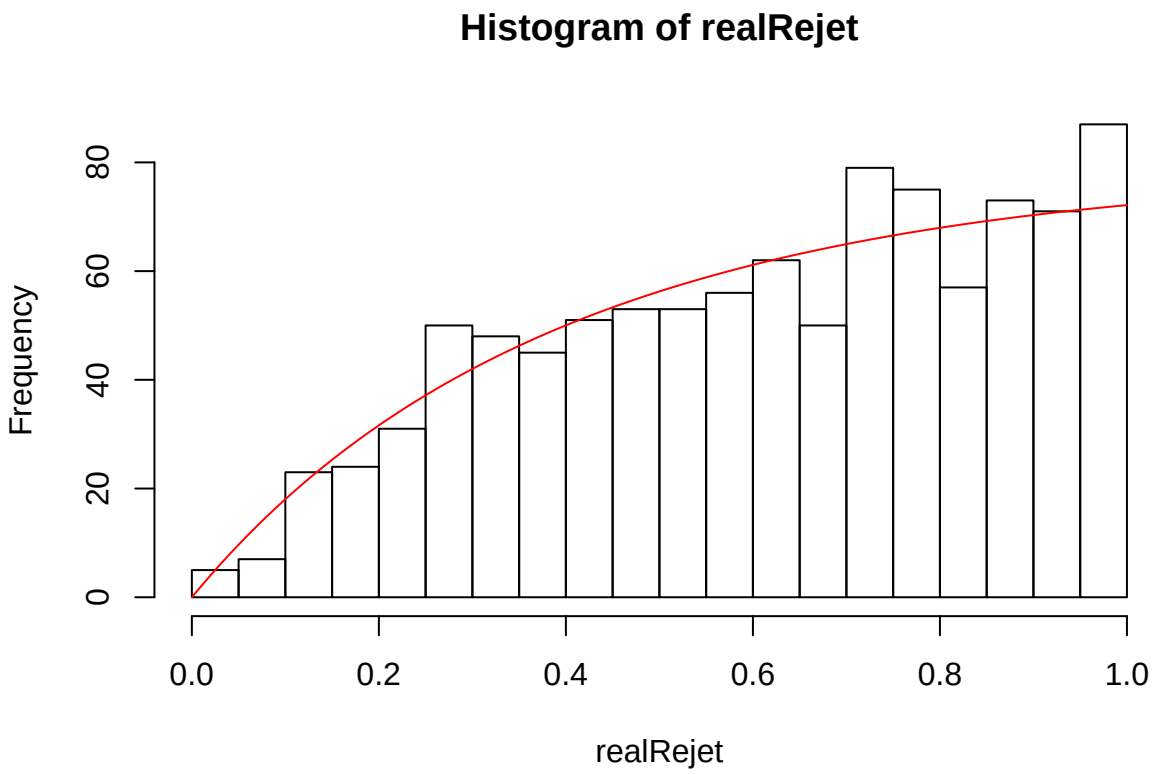
```
## Unit: microseconds
##                 expr min  lq mean median   uq  max neval
##  LoiBonusInversion() 1.1 1.2  3.4    1.3  1.4 2052  1000
##      LoiBonusRejet() 2.4 4.8 21.3    7.5 14.5 7306  1000
```

```
realInversion <- vector(length = nb_iterations)
realRejet <- vector(length = nb_iterations)
for (i in 1:nb_iterations)
{
  realInversion[i] <- LoiBonusInversion()
  realRejet[i] <- LoiBonusRejet()
}
```

```r
xseq<-seq(0, 1, 0.01)

hist(realInversion, breaks = 20)
lines(xseq, nb_iterations/20*FonctionRepartitionBonus(xseq), col='red')
```




```r
hist(realRejet, breaks = 20)
lines(xseq, nb_iterations/20*FonctionRepartitionBonus(xseq), col='red')
```


Histogram of realRejet
Frequency
0
20
40
60
80
0.0
0.2
0.4
0.6
0.8
1.0
realRejet