TP Probabilités 3IF

Victor Lezaud

2018

Test de générateur pseudos-aléatoires

Implémentation de deux générateurs à congruences linéaires

On réalise deux générateurs à congruence linéaire (RANDU et Standard Minimal). Ces deux générateurs ne varie que dans les valeurs des paramètres. On crée donc d'abord un générateur à congruence linéaire paramétré puis on crée des fonctions RANDU et StandardMinimal appelant ce générateur avec les valeurs de paramètre correspondantes.

```
GenCongruenceLin <- function(a,b,m,graine,p=1)
{
  x <- graine
  vect <- c()
  for(i in 1:p)
  {
    x <- (a*x+b)%%m
    vect<- c(vect,x)
  }
  return(vect)
}

RANDU <- function(graine,p=1)
{
  return(GenCongruenceLin(65539,0,2^31,graine,p))
}

StandardMinimal <- function(graine,p=1)
{
  return(GenCongruenceLin(16807,0,2^31-1,graine,p))
}
```

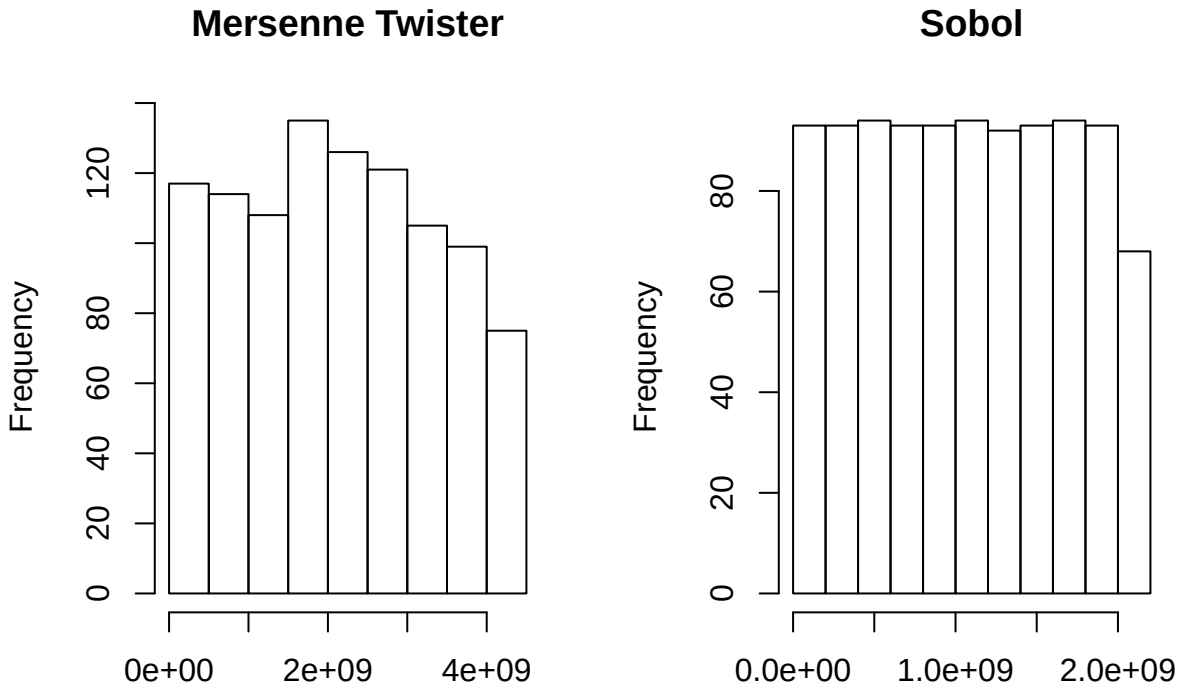
Test visuel

Dans cette partie nous allons chercher à tester à l'aide d'histogramme et de graphe les générateurs ci-dessus ainsi que le Mersenne Twister et la suite de Sobol.

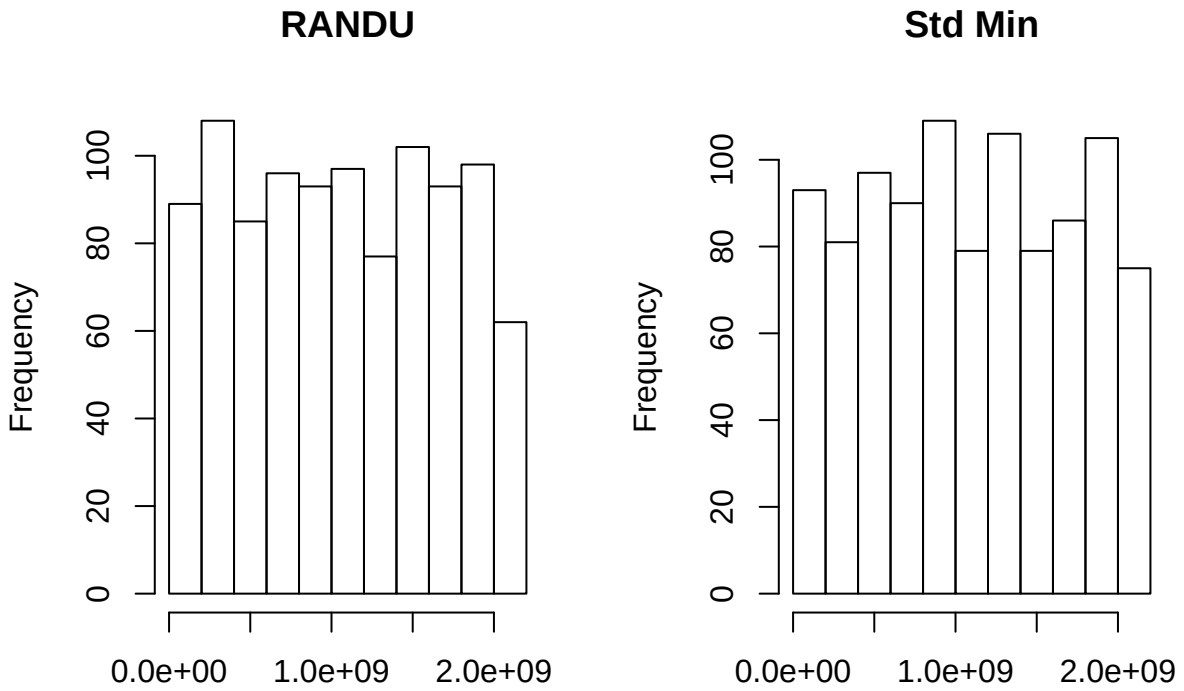
Répartition des valeurs

```
sob <- Sobol(Nsimu,Nrepet)
mt <- MersenneTwister(Nsimu,Nrepet,sMT)$x
randu <- RANDU(sMT,Nsimu)
stdMin <- StandardMinimal(sMT,Nsimu)
par(mfrow=c(1,2))
```

```r
hist(mt[,1],xlab='',main='Mersenne Twister')
hist(sob[,1],xlab='',main='Sobol')
```



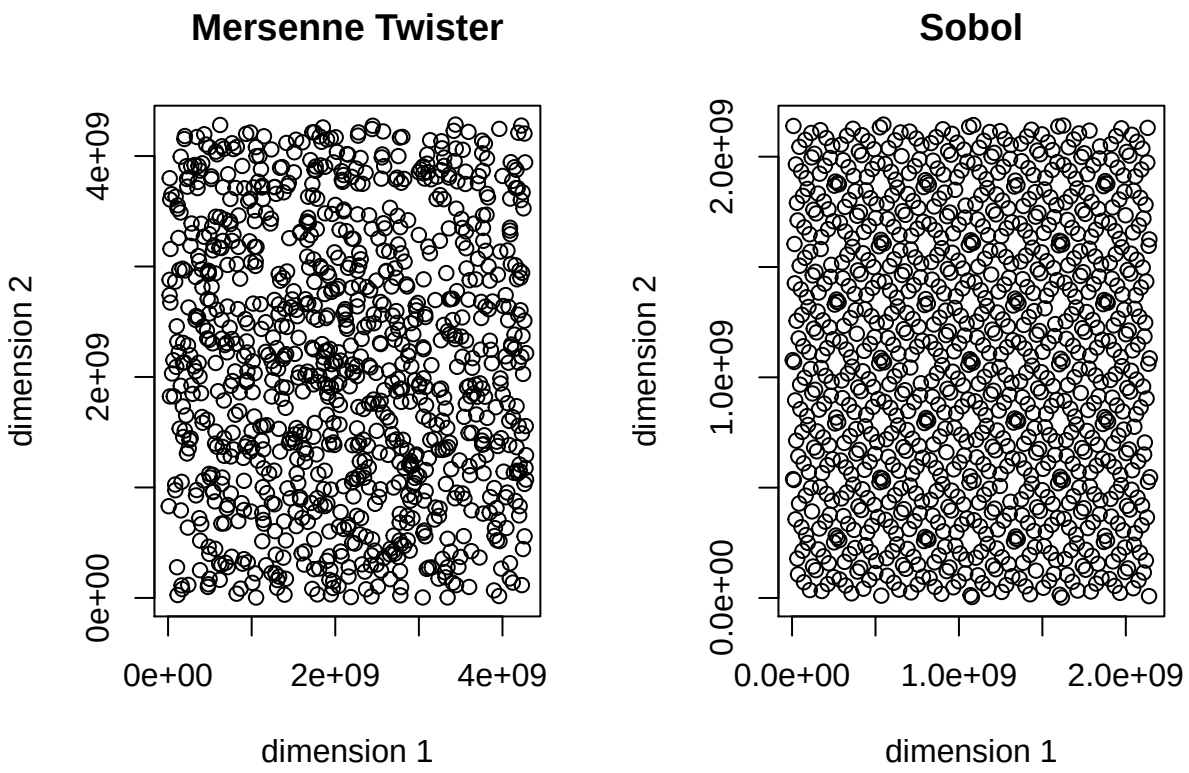

```r
hist(randu,xlab='',main='RANDU')
hist(stdMin,xlab='',main='Std Min')
```



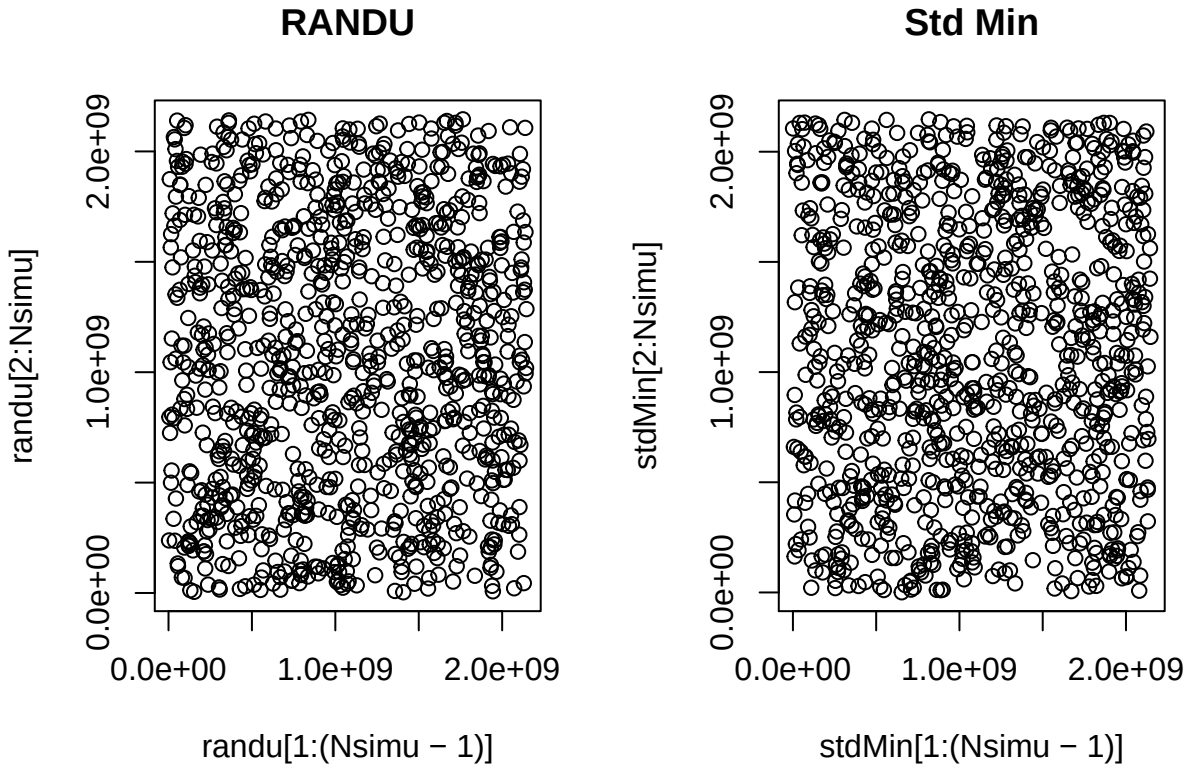
On remarque que les 3 générateurs ont des valeurs de fréquences proches pour l'ensembles des intervalles des histogrammes. La suite de Sobol obtient des valeurs beacoup trop constantes. La chute pour la dernière colonne de l'histogramme s'explique par le fait que la borne de la suite de Sobol est dans l'intervalle de cette colonne ainsi la chute est proportionnelle à $\frac{x-a}{b-a}$ avec x la borne de la suite, et $[a, b]$ l'intervalle

Répartition en fonction de la valeur précédente

```
par(mfrow=c(1,2))
plot(mt[,1:2],xlab='dimension 1', ylab='dimension 2', main='Mersenne Twister')
plot(sob[,1:2],xlab='dimension 1', ylab='dimension 2', main='Sobol')
```




```
plot(randu[1:(Nsimu-1)], randu[2:Nsimu],main='RANDU')
plot(stdMin[1:(Nsimu-1)], stdMin[2:Nsimu],main='Std Min')
```



Contrairement aux 3 générateurs pour lesquels rien n'apparaît clairement on observe que la représentation de la suite de Sobol possède un pattern bien particulier qui se répète. Cela met bien en évidence l'aspect non-aléatoire de la suite de Sobol

Principe général des tests

Principe des tests

Tester des générateurs aléatoires

On ne connaît aucun test permétant d'affirmer qu'un générateur est bon. On peut seulement montrer qu'un générateur est mauvais. Même certains générateurs qui valident tous les tests connus à ca jour ne sont pas satisfaisant pour certaines technique de crytptographie.

Raisonnement par l'absurde

Tous les tests que nous allons mettre en oeuvre fonctionne sur la base d'un raisonnement par l'absurde : on suppose qu le générateur est bon. Dans ce cas on peut calculer une grandeur liés à la suite de nombre générés qui est censé tendre vers une certaine valeur lorsqu'on prend un assez grand nombre de tirages.

Règle de décision à 1%

A partir de la grandeur et de sa valeur théorique on peut revenir à une loi normale de moyenne 0 et d'écart-type 1, décrivant la probabilité d'obtenir les différentes valeurs avec un bon générateur. On génère donc un grand nombre de valeurs avec le générateur que l'on souhaite tester. On calcule la valeur de la grandeur en question pour cette suite de nombre. Ensuite on calcule la probabilité d'obtenir une valeur pire que celle-ci (notée P_{valeur}). Enfin si cette valeur tombe en-dessous de 1% on peut considérer que le générateur est mauvais

Outil d'analyse

Problème

Il est évident que l'on ne peut analyser un générateur en ne faisant qu'un seul tirage avec une unique graine. Il faudra donc faire un nombre suffisemment grands d'essais avec des graines différentes pour rendre notre test crédible. Il n'est pas non plus envisageable d'analyser 100 résultats ou plus par générateur et par test.

La moyenne

Nous allons plutôt étudier la moyenne. La moyenne est intéressante car elle donnne une idée du comportement global du générateur pour l'ensemble des graines testés. En faisant un nombre suffisament grand de tirages on s'attend à obtenir des valeurs autour de 50%. La moyenne donne une première appréciation mais à moins d'avoir un générateur vraiment très mauvais nous ne pouvons pas user du critère à 1%. Il faut donc chercher d'autres outils

Le premier décile

Le premier décile est une valeur très pertinente car nous cherchons à mettre en avant les faiblesses du générateur, nous voulons donc chercher les pires valeurs. Par contre nous ne pouvons pas prendre la valeur minimale car par principe même de l'aléatoire il est toujours possible d'obtenir une valeur totalement improbable. En effet si on pouvait faire une infinité de tirage, un générateur parfaitement aléatoire finirait forcément par avoir une P_{valeur} minimale inférieure à 1%. Il est donc important de faire au moins 100 tirages pour que le premier décile contiennent un minimum de valeurs et ait ainsi du sens. Pour un générateur parfait on attendrait des valeurs de premier décile autour de 10%, mais on ne pourra conclure que si la valeur tombe en dessous de 1% auquel cas le générateur échouera au test.

Test de fréquence monobit

Principe

Ce test vérifie la proportion de 1 et de 0 dans les bits d'une séquence d'entiers. On teste la différence entre le nombre de 1 et de 0. Si la séquence est bien aléatoire cette différence tend vers 0. On change donc les 0 en -1 puis on fait la somme de toutes ces valeurs pour obtenir cette différence. On applique ensuite la méthode ci-dessus

Implémentation de la fonction Frequency

La fonction Frequency doit calculer : * La différence entre le nombre de 1 et de 0 S_n * La valeur normalisée $\frac{S_n}{\sqrt{n}}$ * La p-valeur Pour ce faire on utilise la fonction binary donnée sur moodle pour créer une version binary vect qui transforme un vecteur d'entier en une matrice de bits.

```
binaryVect <-function(vect)
{
  m <- c()
  n <- 4
  for(i in 1:n)
  {
    m<-cbind(m,binary(vect[i]))
  }
  return(m)
}

binaryVectToMatrix <-function(vect)
{
```

```
  m <- c()
  n <- length(vect)
  for(i in 1:n)
  {
    m<-cbind(m,binary(vect[i]))
  }
  return(m)
}
```

Ensuite on crée une fonction calculSn qui calcule la différence entre le nombre de 1 et le nombre de 0 parmi les n bits de poids faible de la matrice que l'on passe en paramètre (qui en pratique sera celle que l'on vient de créer avec la fonction binaryVect).

```
calculSn <-function(m,nb)
{
  m <- 2*m-1
  x<-sum(m[(33-nb):32,])
  return(x)
}
```

Enfin la fonction Frequency appelle les deux fonctions précédentes puis normalise le S_n en un S_{obs} et ainsi on obtient la p-valeur à partir de la fonciton `pnorm()` qui est la fonction de répartition de la loi normale $N(0,1)$

```
Frequency <- function(x,nb=32)
{
  Sn <- calculSn(binaryVectToMatrix(x),nb)
  Sobs <- abs(Sn)/sqrt(nb*length(x))
  p <- 2*(1-pnorm(Sobs))
  return(p)
}
```

Test et interprétation

Il faut maintenant créer un test à partir de cette fonction. Pour qu'un test soit crédible il faut faire des calculs pour un grand nombre de graines différentes (. On définit donc la fonction TestFrequency. Afin de faciliter l'analyse des résultats, on fait des calculs statistiques pour n'observer que quelques valeurs. J'ai ici mis en avant la moyenne et le premier décile. La moyenne nous donne une information sur les valeurs obtenues par l'ensemble des graines. Le premier décile est important parce que notre test nous permet seulement de dire qu'un générateur est mauvais s'il obtient une p-valeur inférieur à 1%.

```
TestFrequency <- function(nbSerie)
{
  pSobol <- Frequency(Sobol(Nsimu,1),10)
  vecMt <- c()
  vecRandu <- c()
  vecStdMin <- c()

  for (i in 1:nbSerie) {
    seed <- sample.int(100000,1)


    pMt <- Frequency(MersenneTwister(Nsimu,1,seed)$x,10)
    vecMt <- c(vecMt,pMt)

    pRandu <- Frequency(RANDU(seed,Nsimu),10)
```

```
    vecRandu <- c(vecRandu,pRandu)

    pStdMin <- Frequency(StandardMinimal(seed,Nsimu),10)
    vecStdMin <- c(vecStdMin,pStdMin)
  }

  decileSobol <- pSobol
  decileMt <- quantile(vecMt, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileRandu <- quantile(vecRandu, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileStdMin <- quantile(vecStdMin, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]

  moyenneSobol <- pSobol
  moyenneMt <- mean(vecMt)
  moyenneRandu <- mean(vecRandu)
  moyenneStdMin <- mean(vecStdMin)

  tableau <- data.frame(moyenne = c(moyenneSobol, moyenneMt, moyenneRandu, moyenneStdMin), PremDecile =
  tableau <- round(tableau, 3)*100
  print(tableau, digits = 3)
}
```

On réalise un test sur 100 graines différentes :

```
TestFrequency(100)
```

```
##                   moyenne PremDecile
## Sobol                84.1       84.1
## Mersienne-Twister    52.6       13.8
## RANDU                29.7        0.0
## Standard Min         49.1       10.4
```

On remarque avant tout les valeurs obtenus par le générateur RANDU, sa moyenne est plutôt basse et surtout son premier décile est bloqué à 0%. Ce générateur a donc échoué au test, on peut conclure que c'est un mauvais générateur. Les deux autres générateurs sont proches des valeurs attendus, on ne peut donc rien conclure pour l'instant, il valide le test. Il en va de même pour la suite de Sobol. Cela prouve qu'un mauvais générateur peut réussir au moins une partie des tests.

Test des runs

Principe du test

Principe du pré-test

On peut mettre en place un pré-test en comptant n le nombre de 1 dans la séquence. Soit N le nombre de bits dans la séquence et $\pi = \frac{n}{N}$, le pré-test est échoué si $|\frac{n}{N} - \frac{1}{2}| \geqslant \frac{2}{\sqrt{N}}$. Dans ce cas on a $P_{valeur} = 0$.

Principe du test principal

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. On calcule le nombre $V_N = \sum_{k=1}^{n-1} r(k) + 1$ avec $r(k) = 0$ si $\epsilon_k = \epsilon_{k+1}$ où ϵ_k est la valeur du k^{eme} bit et $r(k) = 1$. Ainsi V_N est le nombre de fois où l'on change de valeur de bit au cours de la séquence de n bits

plus 1. Enfin on peut calculer la P_{valeur} avec la formule suivante :

$$P_{valeur} = 2\left(1 - \left(\frac{|V_N - 2\pi(1-\pi)N|}{2\pi(1-\pi)\sqrt{N}}\right)\right)$$

Implémentation du test

Implémentation du pré-test

La première étape de l'implémentation consiste à transformer la liste de nombre donnés aléatoirement en un grand vecteur de bits. Pour cela on réutilise la fonction binaryVectToMatrix et on le tansforme en vecteur avec la fonction as.vector() de R. On obtient donc n en faisant la somme de toutes les valeurs du vecteur et N est la taille du vecteur. On implémente donc le pré-test de la façon suivante :

```
PreTestRun <- function(vect)
{
  n <- sum(vect)
  N <- length(vect)
  if(abs(n/N-0.5)>=2/sqrt(N))
  {
    return(0)
  }
  else
  {
    return(1)
  }
}
```

Implémentation du test principal

On implémente le calcul de V_n à partir du vecteur de bits obtenu comme ci-dessus :

```
CalculVn <- function(vect)
{
  Vn <- 1
  N <- length(vect)
  for (i in 1:(N-1))
  {

    if(vect[i]==vect[i+1])
    {
      r <- 1
    }
    else
    {
      r <- 0
    }
    Vn <- Vn + r
  }
  return(Vn)
}
```

On implémente ensuite le calcul de la P_{valeur} :

```
CalculPvaleurRun <- function(vect)
{
  Vn <- CalculVn(vect)
  N <- length(vect)
  pi <- sum(vect)/N
  num <- abs(Vn-2*N*pi*(1-pi))
  denom <- 2*pi*(1-pi)*sqrt(N)
  Pvaleur <- 2*(1-pnorm(num/denom))
  return(Pvaleur)
}
```

Test complet

On met ensemble toutes les fonctions ci-dessous pour créer une fonction TestCompletRun

```
TestCompletRun <- function(alea)
{
  vectBin <- as.vector(binaryVectToMatrix(alea))
  if(PreTestRun(vectBin)==0)
  {
    return(0)
  }
  else
  {
    return(CalculPvaleurRun(vectBin))
  }
}
```

Test et interprétation

Nous allons ici fonctionner de la même manière que pour le test précédent. On définit donc une fonction TestDesRun ci-dessous:

```
TestDesRuns <- function(nbSerie)
{
  aleaSobol <- Sobol(Nsimu,1)
  pSobol <- TestCompletRun(aleaSobol)

  vecMt <- c()
  vecRandu <- c()
  vecStdMin <- c()

  for (i in 1:nbSerie) {
    seed <- sample.int(100000,1)

    aleaMt <- MersenneTwister(Nsimu,1,seed)$x
    pMt <- TestCompletRun(aleaMt)
    vecMt <- c(vecMt,pMt)

    aleaRandu <- RANDU(seed,Nsimu)
    pRandu <- TestCompletRun(aleaRandu)
    vecRandu <- c(vecRandu,pRandu)
```

```
    aleaStdMin <- StandardMinimal(seed,Nsimu)
    pStdMin <- TestCompletRun(aleaStdMin)
    vecStdMin <- c(vecStdMin,pStdMin)
  }

  decileSobol <- pSobol
  decileMt <- quantile(vecMt, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileRandu <- quantile(vecRandu, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileStdMin <- quantile(vecStdMin, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]

  moyenneSobol <- pSobol
  moyenneMt <- mean(vecMt)
  moyenneRandu <- mean(vecRandu)
  moyenneStdMin <- mean(vecStdMin)

  tableau <- data.frame(moyenne = c(moyenneSobol, moyenneMt, moyenneRandu, moyenneStdMin), PremDecile =
  tableau <- round(tableau, 3)*100
  print(tableau, digits = 3)
}
```

On réalise un test sur 100 graines :

```
TestDesRuns(100)
```

```
##                   moyenne PremDecile
## Sobol                 0.0        0.0
## Mersienne-Twister    45.3        7.2
## RANDU                 4.7        0.0
## Standard Min          0.0        0.0
```

La suite de Sobol ne valide même pas le pré-test de ce test des Runs. On peut donc affirmer le résultat attendu, la suite de Sobol n'est pas un bon générateur aléatoire. Les deux générateurs RANDU et Standard Minimum ont échoué au test. Ce sont donc des mauvais générateurs. Le générateur Mersienne-Twister est le seul à réussir le test. C'est le seul pour lequel on ne peut pas conclure

Test d'ordre

Principe du test

Le dernier test n'étudie pas la suite de bits générés mais directement la suite de nombres obtenus. ### Implémentation du test Le test est déjà implémenté dans le package randtoolbox, on utilisera donc cette implémentation. On crée juste une fonction pour simplifier l'utitlisation :

```
Ordre <- function(vect)
{
  return(order.test(vect, d=4, echo=FALSE)$p.value)
}
```

Test et interprétation

Nous allons ici fonctionner de la même manière que pour les tests précédents. On définit donc une fonction TestOrdre ci-dessous:

```
TestOrdre <- function(nbSerie)
{
  aleaSobol <- Sobol(Nsimu,1)
  pSobol <- Ordre(aleaSobol)

  vecMt <- c()
  vecRandu <- c()
  vecStdMin <- c()

  for (i in 1:nbSerie) {
    seed <- sample.int(100000,1)

    aleaMt <- MersenneTwister(Nsimu,1,seed)$x[,1]
    pMt <- Ordre(aleaMt)
    vecMt <- c(vecMt,pMt)

    aleaRandu <- RANDU(seed,Nsimu)
    pRandu <- Ordre(aleaRandu)
    vecRandu <- c(vecRandu,pRandu)

    aleaStdMin <- StandardMinimal(seed,Nsimu)
    pStdMin <- Ordre(aleaStdMin)
    vecStdMin <- c(vecStdMin,pStdMin)
  }

  decileSobol <- pSobol
  decileMt <- quantile(vecMt, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileRandu <- quantile(vecRandu, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]
  decileStdMin <- quantile(vecStdMin, probs=seq(0, 1, 0.1),TRUE,FALSE)[2]

  moyenneSobol <- pSobol
  moyenneMt <- mean(vecMt)
  moyenneRandu <- mean(vecRandu)
  moyenneStdMin <- mean(vecStdMin)

  tableau <- data.frame(moyenne = c(moyenneSobol, moyenneMt, moyenneRandu, moyenneStdMin), PremDecile =
  tableau <- round(tableau, 3)*100
  print(tableau, digits = 3)
}
```

On réalise des test sur 100 graines.

```
TestOrdre(100)
```

```
##                   moyenne PremDecile
## Sobol                 0.0        0.0
## Mersienne-Twister    51.6       10.7
## RANDU                48.6        6.5
## Standard Min         48.9        9.5
```

La suite de Sobol échoue au test, on confirme ici encore que cette suite n'est pas aléatoire. Les 3 générateurs obtiennent des valeurs de moyenne autour de 50% et de premier décile autour de 10%, ce qui correspond à nos attentes, ils valident bien le test. On ne peut donc rien conclure sur la qualité de ces générateurs grâce à ce test.

Conclusion sur les tests des générateurs

La suite de Sobol

La suite de Sobol a validé le test de fréquence monobit mais elle a échoué aux tests des run et d'ordre. On a donc réussi à montrer que cette suite n'est pas aléatoire comme on le présentait avec les graphiques.

Les générateurs RANDU et Standard Minimum

Le générateur RANDU a validé le test d'ordre mais il a échoué aux tests des runs et de fréquence monobit. Le générateur Standard Minimum a validé les d'ordre et de fréquence monobit mais il a échoué au test des runs. On a donc montré que ces générateur ne sont pas réellement aléatoire. On ne pourra donc pas toujours le considérer comme satisfaisant (tout dépend de la qualité d'aléatoire que l'on recherche).

Le générateur Mersienne Twister

Ce générateur a réussi chacun de nos trois tests. Nous n'avons donc toujours pas réussi à montrer que c'est un mauvais générateur. On ne peut affirmer pour autant que c'est un bon générateur mais nous pouvons penser pour l'instant qu'il est préférable de les utiliser plutôt que les générateur RANDU et Standard Minimum.

Simulations de loi de probabilités quelconques

Loi uniforme sur [0,1]

Pour cette partie on utilisera la fonction runif() qui permet de simuler une loi uniforme sur l'intervalle [0,1]. Cette fonction utilise le générateur Mersienne Twister qu'on a étudié précédemment. Notre but est de modéliser d'autres lois de probabilités à partir de celle-ci

Loi discrètes

Principe

Dans un premier temps nous nous intéressons à des lois discrètes. Pour obtenir une loi discrète à partir de la loi uniforme, une technique est d'associer à la probabilité p_X d'obtenir une valeur X à un intervalle $[a, a+p_X]$ de notre loi uniforme. Ainsi si on obtient un nombre appartenant à cette intervalle la loi retournera la valeur X. On note que pour une loi discrète la somme des p_X fait exactement 1. On peut donc avoir une répartition parfaite sur l'intervalle $[0, 1]$. Nous allons meintenant chercher à modéliser une loi binomiale grâce à cela.

Modélisation de la loi binomiale

Epreuve de Bernoulli

Une loi binomiale de paramètre (n, p) compte le nombre de succès parmis n épreuve de Bernoulli avec une probabilité de succés p. On implémente donc d'abord la loi de Bernoulli :

```
Bernoulli <- function(p)
{
  x <- runif(1)
  if(x<=p)
  {
```

```
    return(1)
  }
  else
  {
    return(0)
  }
}
```

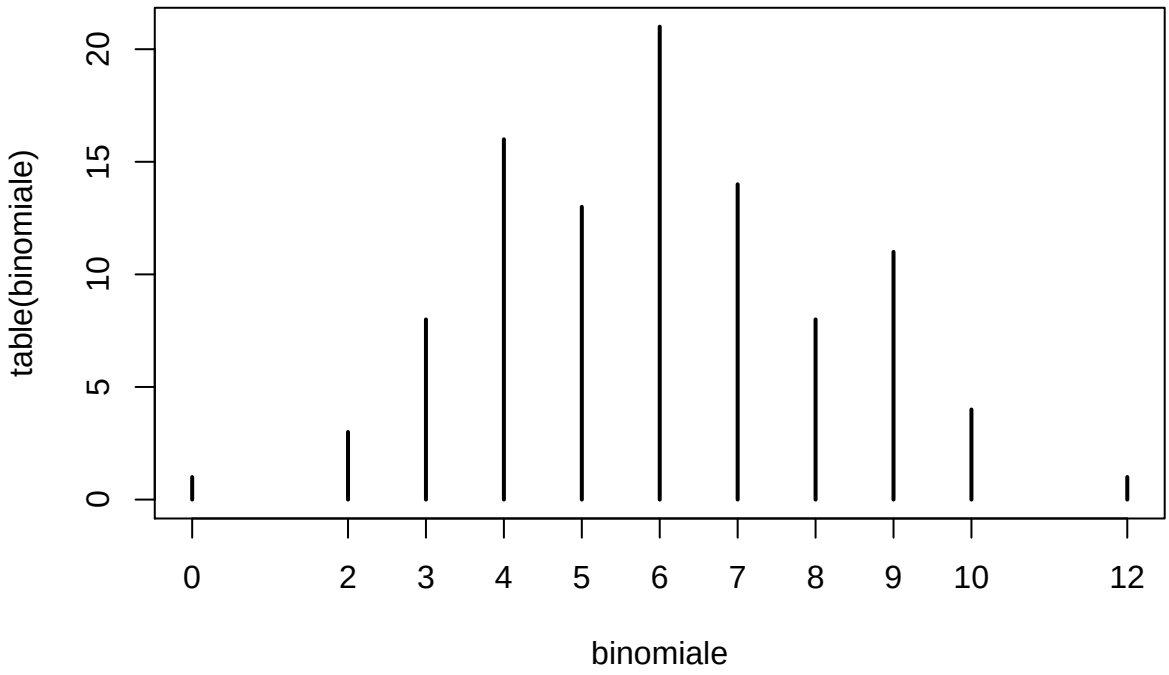
Loi binomiale

Pour réaliser la loi binomiale on a plus qu'à faire la somme de n épreuve de Bernoulli

```
Binomiale <- function(n,p)
{
  somme = 0;
  for(i in 1:n)
  {
    somme <- somme + Bernoulli(p)
  }
  return(somme)
}
```

Graphe de la loi de Bernoulli

```
grapheBinomiale <- function(nBinom,p,nbTirage)
{
  binomiale <- c()
  for(i in 1:nbTirage)
  {
    binomiale <- c(binomiale,Binomiale(nBinom,p))
  }
  plot(table(binomiale))
}
grapheBinomiale(20,0.3,100)
```



Loi continue

Principe

Les lois continues ne sont pas simulées directement, pour les simuler on utilise d'autres lois plus simples à simuler. Elles sont donc obtenues par changement de variable à partir d'autres lois.

Simulation d'une gaussienne

On peut simuler une variable X suivant une loi gaussienne à partir de deux variables U et V suivant des lois uniformes sur [0,1] grâce à la formule suivante

$$\sqrt{-\ln(U)}\cos(2\pi V)$$

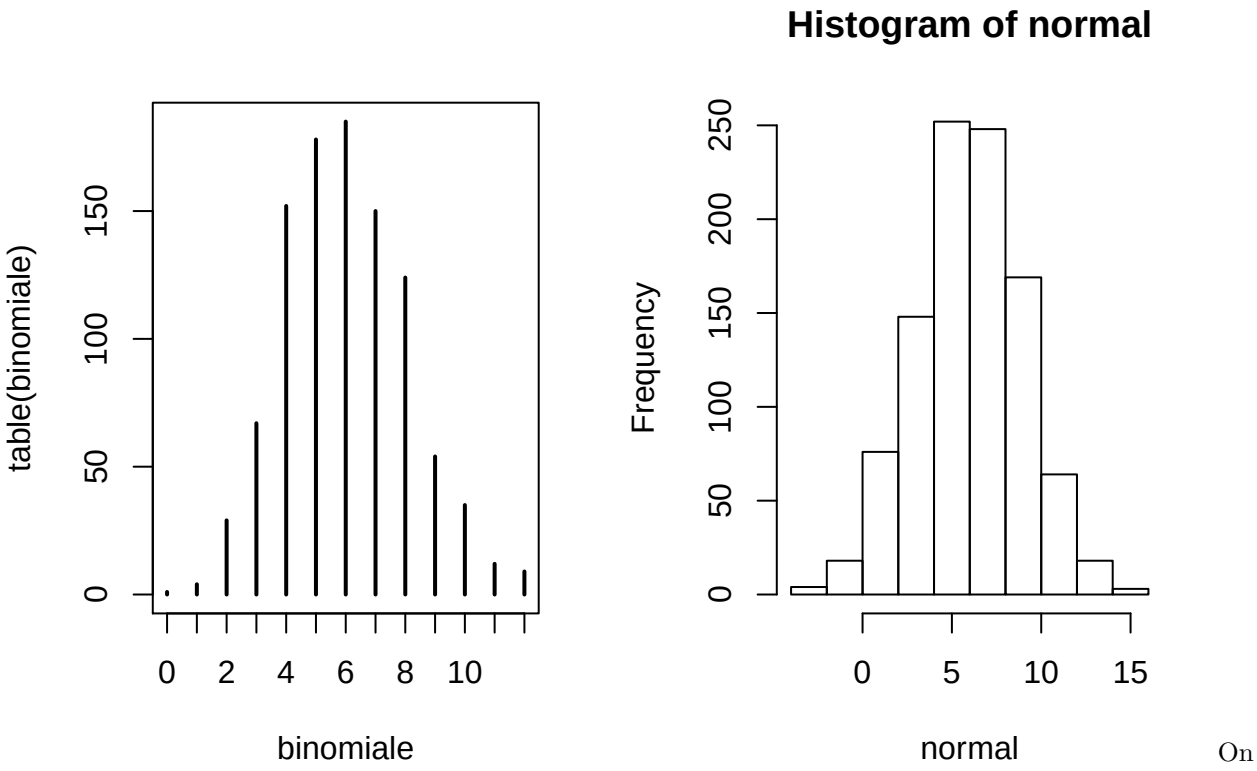
```
LoiNormale01 <- function()
{
  u <- runif(1)
  v <- runif(1)
  return(sqrt(-log(u))*cos(2*pi*v))
}
```

Simulation d'une loi normale

On cherche maintenant à obtenir une loi normale défini par les paramètre (n, p) de moyenne np et d'écart-type $np(1-p)$, on compare ensuite le résultat avec la loi binomiale

```
LoiNormale <- function(n,p)
{
  return(n*p+n*p*(1-p)*LoiNormale01())
}
```

```
grapheBinNormal <- function(n,p, nbTirage)
{
  par(mfrow=c(1,2))
  grapheBinomiale(n,p,nbTirage)

  normal <- c()
  for(i in 1:nbTirage)
  {
    normal <- c(normal,LoiNormale(n,p))
  }
  hist(normal)
}

grapheBinNormal(20,0.3,1000)
```



On trouve logiquement des courbes similaires avec les deux lois.

Application aux files d'attentes, la file M/M/1

Implémentation de la file

On suppose une file d'attente pour accéder à un serveur. La probabilité que le client pris en charge ait fini suit une loi exponentielle de paramètre λ et la probabilité qu'un client arrive suit une loi exponentielle de paramètre μ.

```
FileMM1 <- function(lambda,mu,D)
{
  nbArrivee<-0
```

```
  d <- 0
  depart <- c()
  arrivee <- c()

  d<-rexp(1,lambda)
  while(d<D)
  {
    nbArrivee <- nbArrivee+1
    arrivee <- c(arrivee,d)
    d<- d+rexp(1,lambda)
  }

  if(nbArrivee>0)
  {
    nDepart <- 1
    d<-arrivee[nDepart]
    d<- d+rexp(1,mu)

    while(d<D && nDepart<=nbArrivee)
    {
      depart <- c(depart,d)
      nDepart <- nDepart+1
      if(nDepart<=nbArrivee)
      {
        if(d<arrivee[nDepart])
        {
          d<-arrivee[nDepart]
        }
      }
      d<- d+rexp(1,mu)
    }
  }
  return(list(arrive=arrivee,depart=depart))
}
```

Modélisation de la file d'attente

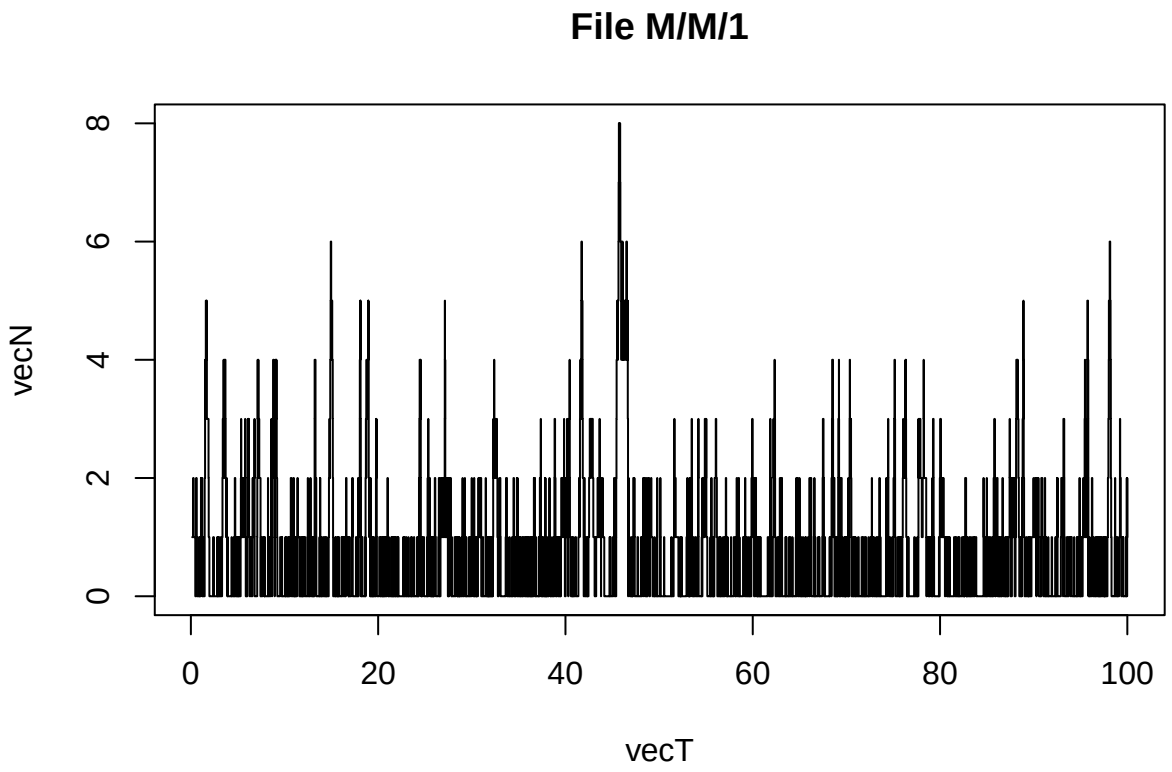
Pour afficher clairement le résultat obtenu par la fonction ci-dessus on dessine un histogramme qui décrit l'évolution du nombre de client en attente au cours du temps. On implémente donc une fonction de modélisation de la façon suivante :

```
Modelisation <- function(liste,max)
{
  arrive <- c(liste$arrive,max)
  depart <- c(liste$depart,max)
  nArrivee <- length(arrive)
  nDepart <- length(depart)
  if(nArrivee > 1)
  {
    currentArrivee <- 1
    currentDepart <- 1

    vecN <- c()
```

```
    vecT <- c()
    n<-0

    while (currentArrivee<nArrivee || currentDepart<nDepart) {
      if(arrive[currentArrivee] < depart[currentDepart])
      {
        n<-n+1
        vecT <- c(vecT,arrive[currentArrivee])
        currentArrivee <- currentArrivee+1
      }
      else
      {
        n<-n-1
        vecT <- c(vecT,depart[currentDepart])
        currentDepart <- currentDepart+1
      }
      vecN <- c(vecN,n)
    }
    #plot(vecN)
    plot(vecT, vecN, type = "s", main='File M/M/1')
    #return(vec)
  }
  else
  {
    print(liste)
    print("Aucune arrivée")
  }
}
```

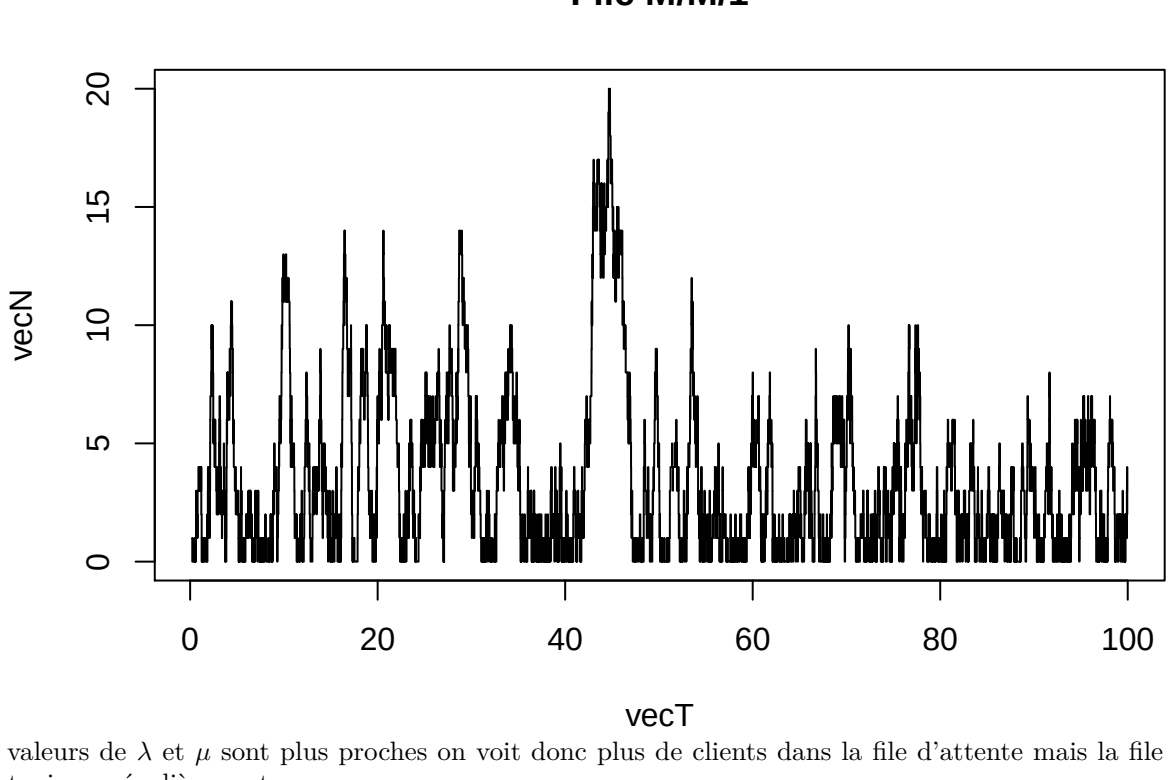
On affiche le résultat avec différentes valeurs de λ

```
Modelisation(FileMM1(8,20,100),100)
```



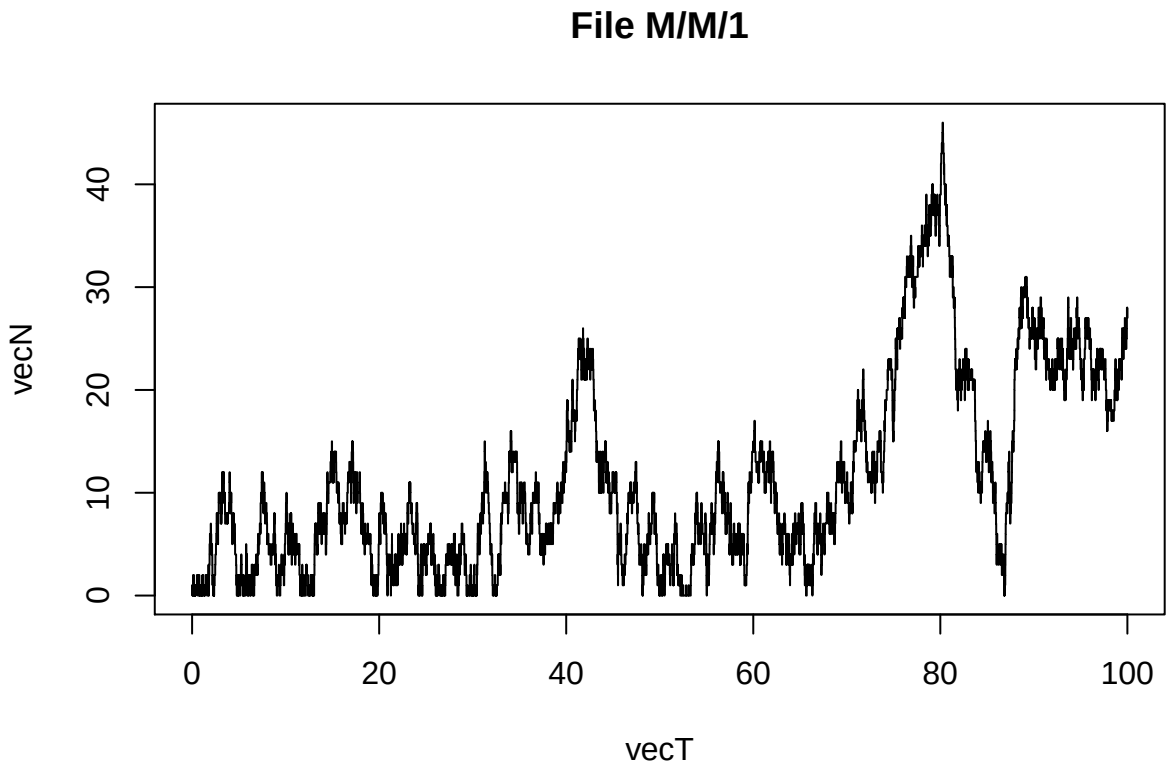
Dans cette configuration les demandes sont traités bien plus vites qu'elles n'arrivent il y a donc peu souvent plus d'un client dans la file d'attente.

```
Modelisation(FileMM1(16,20,100),100)
```



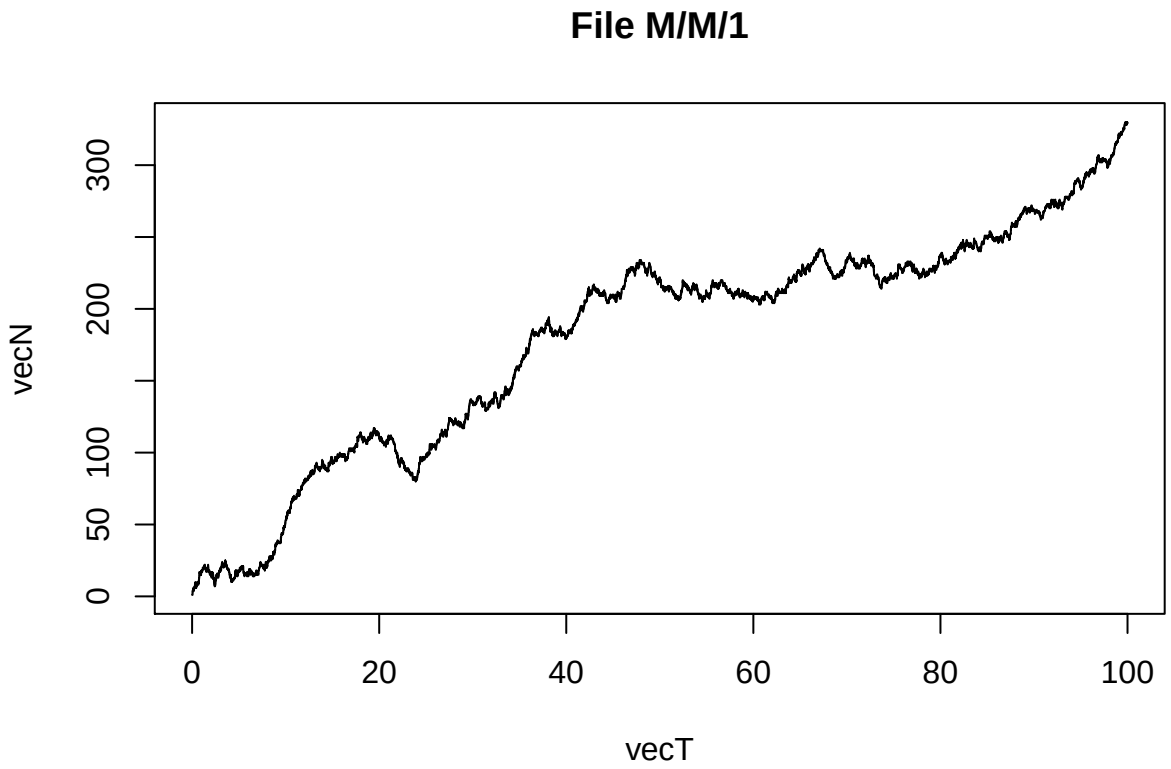
Les valeurs de λ et μ sont plus proches on voit donc plus de clients dans la file d'attente mais la file se vide toujours régulièrement

```
Modelisation(FileMM1(20,20,100),100)
```



Les valeurs de λ et μ sont égales, cette courbe reste donc aléatoire, son comportement asymptotique est indéterminé

```
Modelisation(FileMM1(24,20,100),100)
```



La

valeur de λ est supérieure à μ le nombre de clients dans la file d'attente augmente inéxorablement au cours du temps. Asymptotiquement la courbe tends vers l'infini. ## Vérification de la formule de Little ### Calcul du nombre de clients restants moyen On calcule le nombre clients restants moyen à la fin de l'observation avec la fonction suivante :

```
MoyenneClientReste <- function(lambda, mu, D, N)
{
  vec <- c()
  for (i in 1:N) {
    liste <- FileMM1(lambda, mu, D)
    n <- length(liste$arrive)-length(liste$depart)
    vec <- c(vec,n)
  }
  return(mean(vec))
}
```

Calcul du temps moyen passé par client

On calcule le temps moyen passé par client au cours d'un grand nombre d'observations (on ne prend en compte que les clients arrivés et partis durant l'observation)

```
MoyenneTempsPasse <- function(lambda, mu, D, N)
{
  vec <- c()
  for (i in 1:N) {
    liste <- FileMM1(lambda, mu, D)
    for (j in 1:length(liste$depart)) {
      n <- liste$depart[j] - liste$arrive[j]
      vec <- c(vec,n)
    }

  }
  return(mean(vec))
}
```

Vérification de la formule de Little

On implémente une fonction permettant de calculer d'un seul coup toutes les valeurs impliquées dans la formule de Little

```
VerifFormLittle <- function(lambda,mu,D,N)
{
  moyClientRestant <- MoyenneClientReste(lambda,mu,D,N)
  moyTempsPasse <- MoyenneTempsPasse(lambda,mu,D,N)
  print("Moyenne des Clients restants : E(N)")
  print(moyClientRestant)
  print("Moyenne du temps passés dans la file : E(W)")
  print(moyTempsPasse)
  print("Moyenne du temps passés fois lambda : lambda*E(W)")
  print(lambda*moyTempsPasse)
}
```

On effectue les tests sur les valeurs qu'on a prises précédemment :

```
VerifFormLittle(8,20,100,20)
```

```
## [1] "Moyenne des Clients restants : E(N)"
## [1] 0.65
## [1] "Moyenne du temps passés dans la file : E(W)"
## [1] 0.084
## [1] "Moyenne du temps passés fois lambda : lambda*E(W)"
## [1] 0.67
```

```
VerifFormLittle(16,20,100,20)
```

```
## [1] "Moyenne des Clients restants : E(N)"
## [1] 5.6
## [1] "Moyenne du temps passés dans la file : E(W)"
## [1] 0.24
## [1] "Moyenne du temps passés fois lambda : lambda*E(W)"
## [1] 3.8
```

```
VerifFormLittle(20,20,100,20)
```

```
## [1] "Moyenne des Clients restants : E(N)"
## [1] 36
## [1] "Moyenne du temps passés dans la file : E(W)"
## [1] 1.9
## [1] "Moyenne du temps passés fois lambda : lambda*E(W)"
## [1] 38
```

```
VerifFormLittle(24,20,100,20)
```

```
## [1] "Moyenne des Clients restants : E(N)"
## [1] 395
## [1] "Moyenne du temps passés dans la file : E(W)"
## [1] 8.2
## [1] "Moyenne du temps passés fois lambda : lambda*E(W)"
## [1] 196
```

La formule de Little n'est pas tout à fait vérifiée mais les valeurs reste proche. Pour obtenir une véritable égalité il faudrait probablement calculer la moyenne sur un plus grand nombre de valeur et augmenter la durée d'observation.