Compte rendu de TP Probas

Florian Rascoussier, Romain Gallé (B3208)

12/05/2021

Introduction

Ce document présente nos résultats aux parties demandés du TP de probabilités en R. Nous avons essayé d'expliquer notre code au maximum afin de lever les ambiguïtés. De même, nous avons parfois choisi de conserver dans le rapport des erreurs commises ainsi que d'expliquer celles-ci.

Remarques

Nous avons choisi de créer une version de VonNeumann et MersenneTwister avec 2 paramètres puisque le paramètre p ne sert presque jamais. De plus, cela s'accorde avec les fonctions RANDU et StandardMinimal.

Q1 - RANDU & StandardMinimal

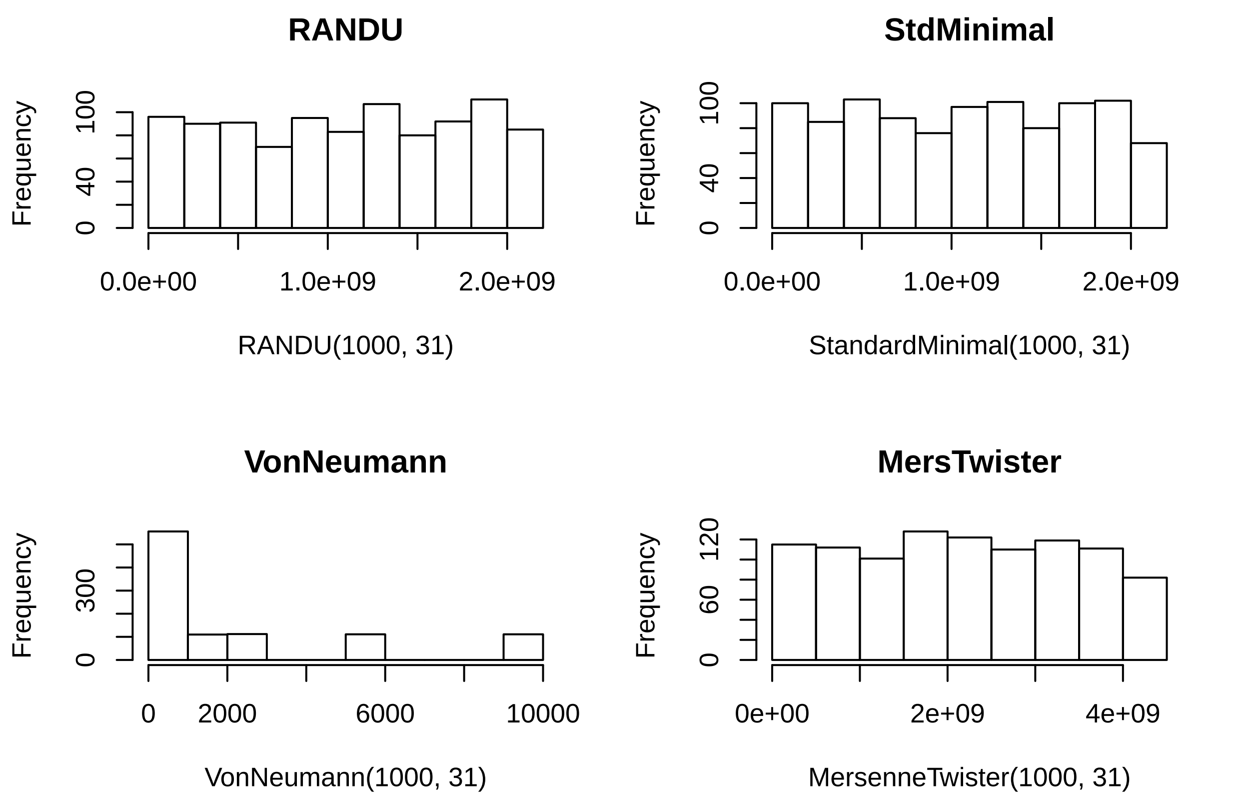
Création des fonctions RANDU et StandardMinimal dans le fichier `generateurs.R`. On notera simplement que l'index de départ est 1 au lieu de 0. Les deux fonctions sont similaires, seuls certains coefficients changent.

```r
RANDU <- function(k, graine) {
  x <-  rep(graine,k)
  # start index 1 (not 0), end index k (not k-1)
  # S0 (at index 1) is already initialized, start at S1 (idx 2)
  for(i in seq(2,k,1)) {
    x[i] <- ((65539*x[i-1]) %% (2^31))
  }
  return(x)
}

StandardMinimal <- function(k, graine) {
  x <-  rep(graine,k)
  for(i in seq(2,k,1)) {
    x[i] <- ((16807*x[i-1]) %% (2^31 - 1))
  }
  return(x)
}
```

Q2.1 - Histogrammes des générateurs

Création d'histogrammes pour les différents générateurs.

```r
# cut the window in 2 rows, 2 columns, graphs filled successively
par(mfrow=c(2,2))
## histogram of RANDU distribution for k=1000, seed=31
hist(RANDU(1000, 31), main="RANDU")
hist(StandardMinimal(1000, 31), main="StdMinimal")
hist(VonNeumann(1000, 31), main="VonNeumann")
hist(MersenneTwister(1000, 31), main="MersTwister")
```



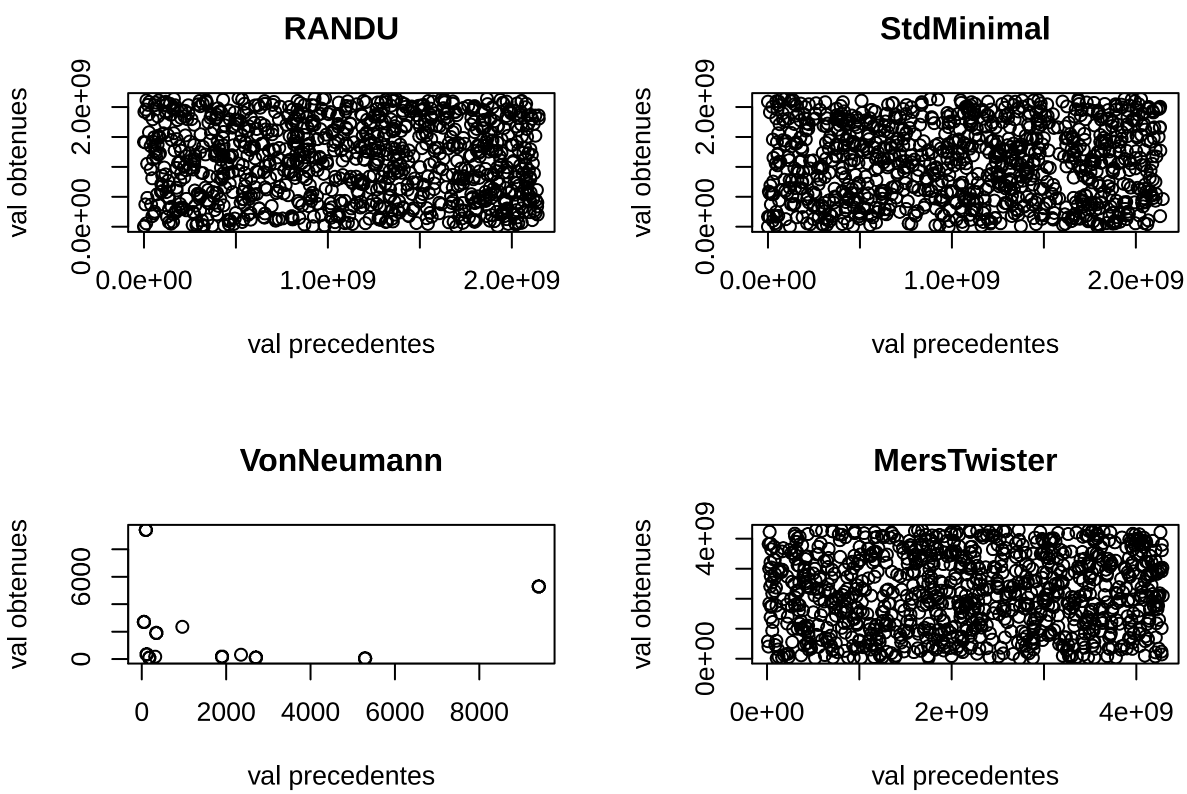
On constate que la répartition des valeurs aléatoires ne respecte pas parfaitement une loi uniforme. En particulier, VonNeumann génère beaucoup de valeurs proches et parfois plus aucune selon l'intervalle. Ce générateur est donc peu satisfaisant. StandardMinimal et RANDU sont assez proches dans leurs répartitions des valeurs, ce qui correspond au fait que ces générateurs sont très similaires. Enfin, MersenneTwister est le plus satisfaisant avec une meilleure répartition des valeurs générées. Les écarts entre les différentes colonnes de l'histogramme sont les plus faibles.

Q2.2 - Valeurs en fonction des prédécesseurs

On trace les valeurs obtenues en fonctions des valeurs précédentes pour chaque algorithme.

```r
par(mfrow=c(2,2))
n <- 1000 # size of vector
u1 <- RANDU(n, 31)
u2 <- StandardMinimal(n, 31)
u3 <- VonNeumann(n, 31)
u4 <- MersenneTwister(n, 31)
plot(u1[1:(n-1)], u1[2:n], xlab='val precedentes', ylab='val obtenues', main="RANDU")
plot(u2[1:(n-1)], u2[2:n], xlab='val precedentes', ylab='val obtenues', main="StdMinimal")
```

```r
plot(u3[1:(n-1)], u3[2:n], xlab='val precedentes', ylab='val obtenues', main="VonNeumann")
plot(u4[1:(n-1)], u4[2:n], xlab='val precedentes', ylab='val obtenues', main="MersTwister")
```



On remarque tout de suite pour VonNeumann que toutes les valeurs sont regroupées en quelques points, ce qui indique que le générateur a tendance à toujours donner les mêmes suites de valeurs. Il n'est pas très satisfaisant. Ensuite, RANDU et StandardMinimal donnent de meilleurs résultats, avec plus de points mieux répartis indiquant un générateur aléatoire de meilleure qualité. Enfin, il semble que MersenneTwister donne le résultat le plus uniforme et réparti, cependant, la différence observée reste faible. La distinction entre les 3 derniers algorithmes est faible (presque négligeable).

Q3 - Test de fréquence monobit

On crée la fonction `Frequency`, qui prend en entrée un vecteur x contenant une série de nombres issus de nos générateurs et pour lesquels une autre suite nb indique le nombre de bits à considérer. En effet, tous les nombres générés ne nécessitent pas forcément 32 bits pour être convertis du décimal au binaire et on ne veut pas considérer les 0 en trop de la conversion renvoyée par la fonction `binary`. L'exemple ci-dessous montre ce phénomène de 0 en trop. Seuls les 0 après un bit 1 de poids le plus fort sont utiles. Il faut donc bien considérer les bits réellements générés, et non pas tous les bits possiblement utilisés jusqu'à la borne supérieure. Une analogie en décimal serait, par exemple, pour le nombre généré 0000035 de ne pas comptabiliser les zéros de tête.

```r
cat(binary(2^31), '\n')
```

```
## 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
cat(binary(12), '\n')
```

```
## 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
```

L'implémentation de la fonction `Frequency` est assez directe. Son rôle est d'effectuer les opérations demandées, à savoir, le calcul de la somme des bits convertis (1 devient +1 et 0 devient -1), puis *sObs* et enfin *pValeur*.

```
## x - observed vector of numbers, nb - number of bits to consider
## WARN : starts by the bit of lowest weight !
## WARN : test at least 100 bit
Frequency <- function(x, nb) {
  sumConvertedBits <- 0
  allConsideredBits <- 0
  # convert all numbers into a sum of converted bits
  for(i in 1:length(x)) {
    seq32Bits <- binary(x[i])
    nbBitsToConsider <- nb[i]
    allConsideredBits <- allConsideredBits + nbBitsToConsider
    for(j in 1:nbBitsToConsider) {
      # conversion in +1 or -1 of a number's bits
      # start by the bit of lowest weight, j in [1:32] or less
      bit0or1 <- seq32Bits[32+1 - j]
      sumConvertedBits <- sumConvertedBits + (2*bit0or1 - 1)
    }
  }
  # computation of sObs of all numbers
  sObs <- abs(sumConvertedBits)/sqrt(allConsideredBits)
  # computation of pValeur of all numbers
  pValeur <- 2*(1 - pnorm(sObs))
  return(pValeur)
}
```

La fonction `Frequency` renvoie donc la pValeur pour une série de nombres supposée aléatoire. On va maintenant l'utiliser afin de tester la qualité de nos générateurs aléatoires. :

```
## compute the average pValeur of a generator
## specify the number of numbers generated for each sequence on which to compute a pValeur
## specify how many time to repeat the test (how much pValeur to compute)
computeAvgPValeur <- function(generator, lengthSeq, repetition,
                              maxSeed=100000000, printSeeds=FALSE) {
  seeds <- sample.int(maxSeed,repetition)
  if(printSeeds) {
    cat('seeds : ', seeds, '\n')
  }
  ## generate a vector x of 1000 numbers by our generators
  sumPValeur <- 0
  for (i in 1:repetition) {
    ## generate a vector x of 1000 numbers by our generators
    x <- generator(lengthSeq, seeds[i])
    ## initialize nb, the vector of bits to consider for every number of x
    nb <- rep(0,lengthSeq)
```

```r
    ## for every number of x, determine how many bits are to be considered
    for(j in 1:lengthSeq) {
      nb[j] <- bitsNecessary(x[j])
    }
    ## determine the pValeur for the sequence x and sum it with the others
    sumPValeur <- sumPValeur + Frequency(x, nb)
    if(is.na(Frequency(x, nb))) {
      cat("x = ", x, ", nb = ", nb)
    }
  }
  avgPValeur <- sumPValeur/repetition
  return(avgPValeur)
}
```

Afin de déterminer, pour chaque nombre contenu dans x, le nombre de bits à considérer, on utilise la fonction `bitsNecessary` :

```r
## get the number of bits to use to convert a decimal to binary
bitsNecessary <- function(decimalNumber) {
  if(decimalNumber == 0) return(1);
  return(log2(decimalNumber) + 1);
}
```

Enfin, pour chaque générateur, on réitère 100 fois le calcul de la *pValeur* pour des seeds différentes et on calcule la *pValeur* moyenne sur ces 100 itérations. On obtient :

```r
cat('average pValeur VonNeumann : ', computeAvgPValeur(VonNeumann, 1000, 100, 9999, FALSE), '\n')
```

```
## average pValeur VonNeumann :  1.534512e-07
```

```r
cat('average pValeur RANDU : ', computeAvgPValeur(RANDU, 1000, 100), '\n')
```

```
## average pValeur RANDU :  0.1200994
```

```r
cat('average pValeur StandardMinimal : ', computeAvgPValeur(StandardMinimal, 1000, 100), '\n')
```

```
## average pValeur StandardMinimal :  7.708333e-05
```

```r
cat('average pValeur MersenneTwister : ', computeAvgPValeur(MersenneTwister, 1000, 100), '\n')
```

```
## average pValeur MersenneTwister :  6.980805e-06
```

On remaque donc que d'après la règle de décision à 1%, que :

- *pValeur* moyenne de VonNeumann $<$ 0,01, donc VonNeumann n'est pas un générateur de séquences aléatoires au sens de ce test.
- *pValeur* moyenne RANDU $>$ 0,01 donc RANDU est aléatoire au sens de ce test.
- *pValeur* moyenne StandardMinimal $<$ 0,01 donc, contrairement à RANDU, StandardMinimal n'est pas un générateur de séquences aléatoires.

- *pValeur* moyenne MersenneTwister $< 0{,}01$ donc MersenneTwister n'est pas un générateur de séquences aléatoires.

> La prochaine partie résulte d'une erreur dans le code, nous avons tout de même choisi de la laisser pour le moment où nous relirons ce rapport. De plus, elle nous a permis de nous intéresser un peu plus au générateur de Von Neumann.

Pour le dernier test VonNeumann, on obtiendra dans l'immense majorité du temps NA. Cependant on peut parvenir à obtenir des *pValeurs*. Dans ces cas, on aura *pValeur* $< 0{,}01$ et donc donc VonNeumann n'est pas un générateur de séquences aléatoires au sens de ce test :

```
for(i in 1:8) {
  cat(' single pValeur VonNeumann : ', computeAvgPValeur(VonNeumann, 1000, 1, 9999, TRUE), '\n')
}
```

```
## seeds :  4863
##  single pValeur VonNeumann :  0
## seeds :  4398
##  single pValeur VonNeumann :  0
## seeds :  5038
##  single pValeur VonNeumann :  0
## seeds :  8116
##  single pValeur VonNeumann :  7.847929e-08
## seeds :  7874
##  single pValeur VonNeumann :  0
## seeds :  7991
##  single pValeur VonNeumann :  3.291319e-07
## seeds :  5132
##  single pValeur VonNeumann :  2.514968e-07
## seeds :  9362
##  single pValeur VonNeumann :  0
```

En regardant de plus près, on se rend compte qu'en fonction de la seed choisie, on aura soit NA soit une *pValeur* numérique. En creusant la piste des seeds, on finit par réaliser qu'en certains cas, le nombre 0 est généré ce qui conduit la suite de la séquence à être des 0 !

```
cat(VonNeumann(100, 33), '\n')
```

```
## 1089  859  3788  3489  1731  963  2736  856  3273  7125  7656  6143  7364  2284  166
## 755  7002  280  840  560  1360  496  4601  1692  628  9438  758  7456  5919  345
## 1902  176  97  9409  5292  52  2704  116  345  1902  176  97  9409  5292  52
## 2704  116  345  1902  176  97  9409  5292  52  2704  116  345  1902  176  97
## 9409  5292  52  2704  116  345  1902  176  97  9409  5292  52  2704  116  345
## 1902  176  97  9409  5292  52  2704  116  345  1902  176  97  9409  5292  52
## 2704  116  345  1902  176  97  9409  5292  52  2704
```

```
cat(VonNeumann(100, 94), '\n')
```

```
## 8836   748   5950   4025   2006   240   760   7760   2176   349   2180   524   7457   6068   8206
## 3384   4514   3761   1451   54   2916   30   900   1000   0   0   0   0   0   0
## 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## 0   0   0   0   0   0   0   0   0   0
```

L'origine du NA vient donc de l'algorithme VonNeumann lui-même.

Après après avoir continué les recherches, nous nous sommes rendu compte que le problème venait du fait que la fonction qui calculait le nombre de bits à considérer pour un nombre donné renvoyait 0 au lieu de 1 pour le nombre 0.

En voici une version corrigée :

On trouve alors les résultats corrigés suivants :

```r
cat('average pValeur VonNeumann : ', computeAvgPValeur(VonNeumann, 1000, 100, 9999, FALSE), '\n')
```

```
## average pValeur VonNeumann :  1.268276e-07
```

```r
cat('average pValeur RANDU : ', computeAvgPValeur(RANDU, 1000, 100), '\n')
```

```
## average pValeur RANDU :  0.04269378
```

```r
cat('average pValeur StandardMinimal : ', computeAvgPValeur(StandardMinimal, 1000, 100), '\n')
```

```
## average pValeur StandardMinimal :  5.805504e-06
```

```r
cat('average pValeur MersenneTwister : ', computeAvgPValeur(MersenneTwister, 1000, 100), '\n')
```

```
## average pValeur MersenneTwister :  7.982065e-06
```

Fin de la partie "erreur"

Q4 - Test des runs

La fonction à implémenter est assez similaire à celle de la question précédente.

```r
## consider a sequence of decimal numbers in a single seq of bits
## loop once on every number of x
## compute proportionOf1 and vObs at the same time
## then evaluate the test and continue if necessary to get pValue
runs <- function(x, nb) {
  allConsideredBits <- 0
  sumOf1 <- 0
  vObs <- 0
  ## init the first bit of first number to be considered
```

```
  lastBit <- binary(x[1])[1]
  ## pre-compute proportionOf1 and compute vObs
  for(i in 1:length(x)) {
    seq32Bits <- binary(x[i])
    nbBitsToConsider <- nb[i]
    allConsideredBits <- allConsideredBits + nbBitsToConsider
    for(j in 1:nbBitsToConsider) {
      # start by the bit of lowest weight, j in [1:32] or less
      bit0or1 <- seq32Bits[32+1 - j]
      ## pre-compute proportionOf1
      sumOf1 <- sumOf1 + bit0or1
      ## compute vObs
      if(lastBit != bit0or1) {
        vObs <- vObs + 1
      }
    }
  }
  ## compute proportionOf1 and do the test
  proportionOf1 <- sumOf1/allConsideredBits
  if(abs(proportionOf1 - (1/2)) >= (2/sqrt(allConsideredBits))) {
    return(0.0)
  }
  ## compute pValeur
  a = abs(vObs - 2*allConsideredBits*proportionOf1*(1 - proportionOf1))
  b = 2*sqrt(allConsideredBits)*proportionOf1*(1 - proportionOf1)

  pValeur = 2*(1 - pnorm(a/b))
  return(pValeur)
}
```

On réutilisera également une version de `computeAvgPValeur` modifiée pour utiliser `runs` afin de calculer des *pValeur* moyennes sur 100 itérations.

```
computeAvgPValeurRuns <- function(generator, lengthSeq, repetition,
                                  maxSeed=100000000, printSeeds=FALSE) {
  seeds <- sample.int(maxSeed,repetition)
  if(printSeeds) {
    cat('seeds : ', seeds, '\n')
  }
  ## generate a vector x of 1000 numbers by our generators
  sumPValeur <- 0
  for (i in 1:repetition) {
    ## generate a vector x of 1000 numbers by our generators
    x <- generator(lengthSeq, seeds[i])
    ## initialize nb, the vector of bits to consider for every number of x
    nb <- rep(0,lengthSeq)
    ## for every number of x, determine how many bits are to be considered
    for(j in 1:lengthSeq) {
      nb[j] <- bitsNecessary(x[j])
    }
    ## determine the pValeur for the sequence x and sum it with the others
    sumPValeur <- sumPValeur + runs(x, nb)
  }
```

```
  avgPValeur <- sumPValeur/repetition
  return(avgPValeur)
}
```

Finalement, on obtient les résultats suivants :

```
cat('average pValeur (runs) VonNeumann : ',
    computeAvgPValeurRuns(VonNeumann, 1000, 100, 9999, FALSE), '\n')
```

```
## average pValeur (runs) VonNeumann :  0
```

```
cat('average pValeur (runs) RANDU : ',
    computeAvgPValeurRuns(RANDU, 1000, 100), '\n')
```

```
## average pValeur (runs) RANDU :  0.01390213
```

```
cat('average pValeur (runs) StandardMinimal : ',
    computeAvgPValeurRuns(StandardMinimal, 1000, 100), '\n')
```

```
## average pValeur (runs) StandardMinimal :  0.06500146
```

```
cat('average pValeur (runs) MersenneTwister : ',
    computeAvgPValeurRuns(MersenneTwister, 1000, 100), '\n')
```

```
## average pValeur (runs) MersenneTwister :  0.02214524
```

On a donc, avec la règle de décision à 1% :

- $pValeur$ moyenne (runs) RANDU $>$ 0,01 donc RANDU est aléatoire au sens de ce test (même si ça reste souvent assez proche de 0,01).
- $pValeur$ moyenne (runs) VonNeumann $<$ 0,01 donc VonNeumann n'est pas un générateur de séquences aléatoires au sens de ce test.
- $pValeur$ moyenne (runs) StandardMinimal $>$ 0,01 donc StandardMinimal est aléatoire au sens de ce test.
- $pValeur$ moyenne (runs) MersenneTwister $>$ 0,01 donc MersenneTwister est aléatoire au sens de ce test.

Q5 - Test d'ordre

Cette question est assez rapide puisque la fonction de test est fournie dans le paquet `randtoolbox`. Ici, v est une séquence de 1000 nombres issue du générateur à étudier.

```
v <- as.numeric(MersenneTwister(1000, 32))
cat(order.test(v, d=4, echo=FALSE)$p.value)
```

On n'a plus qu'à réitérer le test 100 fois pour chaque générateur afin de calculer la $pValeur$ moyenne du test. Pour cela on utilise la fonction suivante. Ici, le `as.numeric` est essentiel car les générateurs renvoient des listes dont chaque élément est une liste à un seul élément. Le `as.numeric` permet "d'unpack" ces listes en un vecteur simple. Ainsi, il n'est pas nécessaire d'utiliser la fonction de transposition.

```r
computeAvgPValeurOrder <- function(generator, lengthSeq, repetition, maxSeed=100000000) {
  seeds <- sample.int(maxSeed,repetition)
  sumPValeur <- 0
  for (i in 1:repetition) {
    u <- as.numeric(generator(lengthSeq, seeds[i]))
    sumPValeur <- sumPValeur + order.test(u, d=4, echo=FALSE)$p.value
  }
  avgPValeur <- sumPValeur/repetition
  return(avgPValeur)
}
```

On obtient les résultats suivants :

```r
cat('average pValeur (order) VonNeumann : ',
    computeAvgPValeurOrder(VonNeumann, 1000, 100, 9999), '\n')
```

```
## average pValeur (order) VonNeumann :  1.1e-37
```

```r
cat('average pValeur (order) RANDU : ',
    computeAvgPValeurOrder(RANDU, 1000, 100), '\n')
```

```
## average pValeur (order) RANDU :  0.53
```

```r
cat('average pValeur (order) StandardMinimal : ',
    computeAvgPValeurOrder(StandardMinimal, 1000, 100), '\n')
```

```
## average pValeur (order) StandardMinimal :  0.47
```

```r
cat('average pValeur (order) MersenneTwister : ',
    computeAvgPValeurOrder(MersenneTwister, 1000, 100), '\n')
```

```
## average pValeur (order) MersenneTwister :  0.54
```

On a donc, avec la règle de décision à 1% :

- $pValeur$ moyenne (order test) VonNeumann $<$ 0,01 donc VonNeumann n'est pas un générateur de séquences aléatoires.
- $pValeur$ moyenne (order test) RANDU $>$ 0,01 donc RANDU est aléatoire au sens de ce test.
- $pValeur$ moyenne (order test) StandardMinimal $>$ 0,01 donc StandardMinimal est aléatoire au sens de ce test.
- $pValeur$ moyenne (order test) MersenneTwister $>$ 0,01 donc MersenneTwister est aléatoire au sens de ce test.

Nous venons de répondre à toutes les questions obligatoires de la partie 1. Commençons maintenant la partie 2.

Q6 - Génération de files d'attente

La fonction suivante permet de générer deux listes correspondant aux temps d'arrivée et de départ des clients, tel qu'indiqué dans le sujet :

```r
FileMM1 <- function(lambda, mu, D) {
  result <- list(departs=c(), arrivees=c())
  arrivees = c()
  departs = c()

  temps = 0
  client = 1
  while(temps < D)
  {
    d_ = rexp(1, rate=mu)
    t_ = rexp(1, rate=lambda)

    arrivees[client] = temps
    departs[client] = arrivees[client] + d_

    # si le client précédent n'a pas fini quand on arrive
    if(client > 1 && departs[client - 1] > arrivees[client])
    {
      departs[client] = departs[client] + (departs[client - 1] - arrivees[client])
    }

    temps = temps + t_
    client = client + 1
  }

  departs = departs[departs <= D]

  result$departs = departs
  result$arrivees = arrivees

  return(result)
}
```

Q7 - Visualisation du comportement de la file

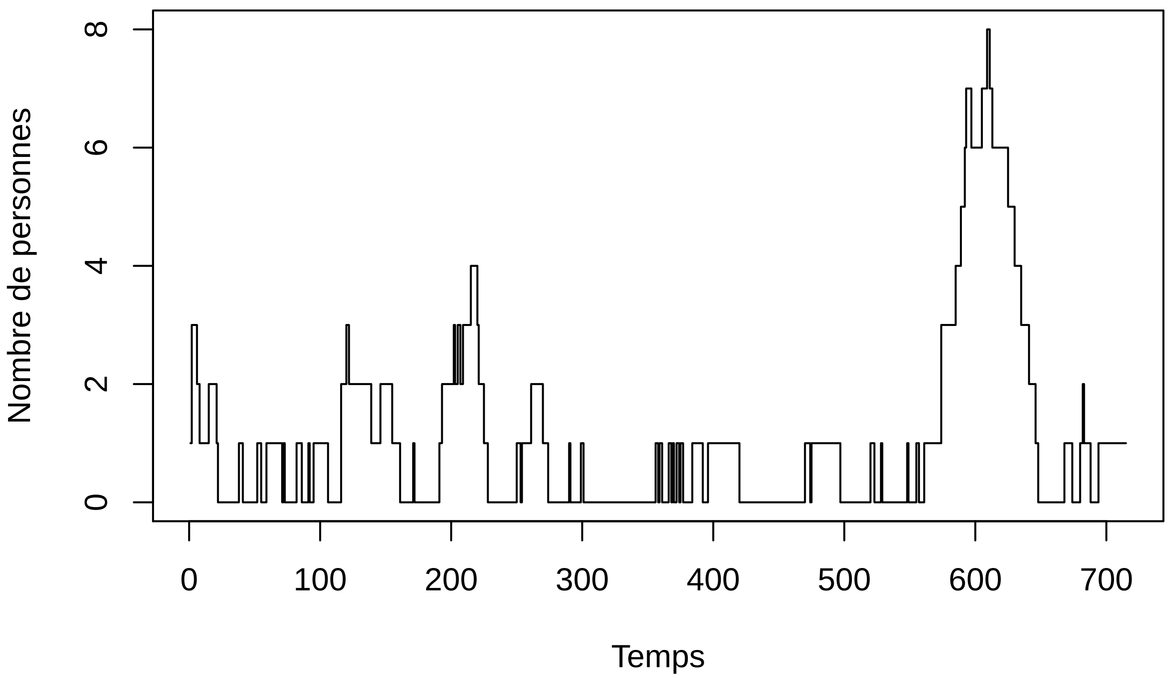
La fonction suivante permet d'obtenir deux vecteurs T et N représentant les instants de temps et le nombre de personnes dans le système :

```r
VisualisationComportement <- function(arrivees, departs) {
  ts = c()
  ns = c()

  maxTime = departs[length(departs)]
  for(i in 0:maxTime) {
    ts[i] = i

    nb_arrives = sum(arrivees <= i)
```

```
    nb_partis = sum(departs <= i)

    ns[i] = nb_arrives - nb_partis
  }

  return(list(t=ts, n=ns))
}
```

Voyons maintenant une application pour laquelle 6 clients arrivent par heure en moyenne, alors que 11 repartent par heure en moyenne : on a donc lambda = 6 et mu = 11. La simulation dure 12 heures, donc D = 12. On aura :

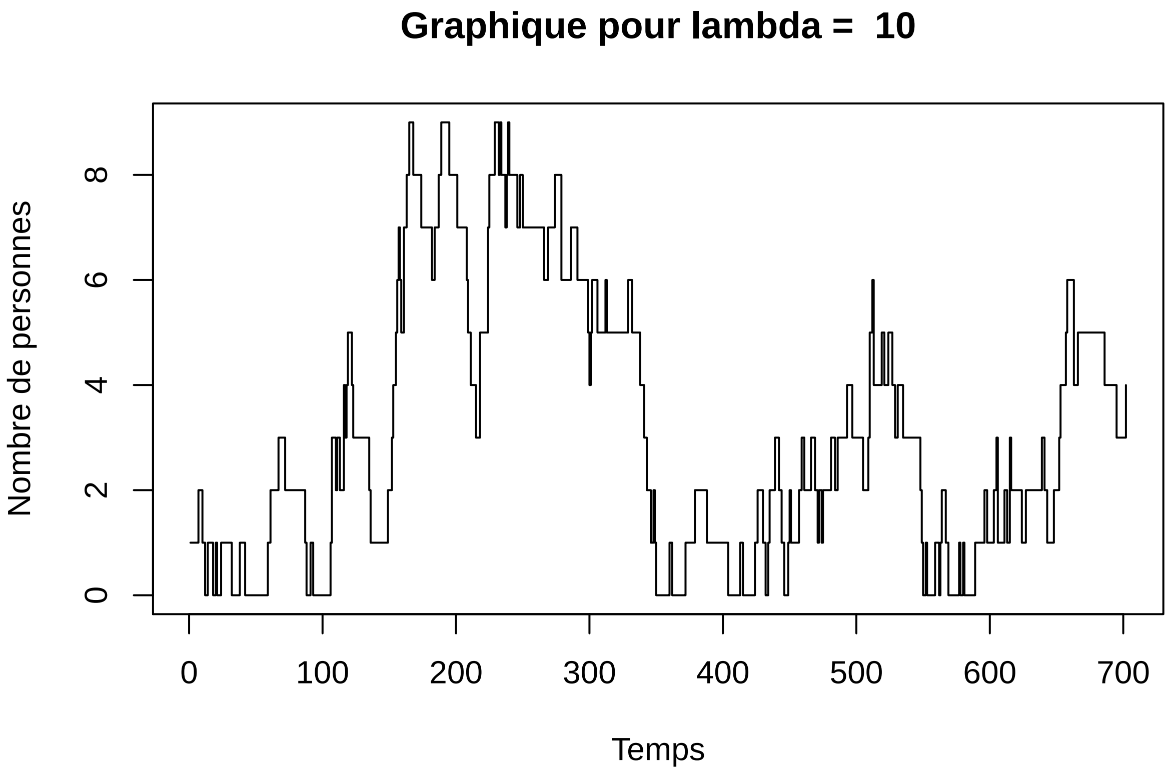
```
files_attente <- FileMM1(lambda = 6/60, mu = 11/60, D = 12*60)

nb_personnes_temps <- VisualisationComportement(files_attente$arrivees, files_attente$departs)

plot(nb_personnes_temps$t, nb_personnes_temps$n, xlab="Temps", ylab="Nombre de personnes", type="s")
```



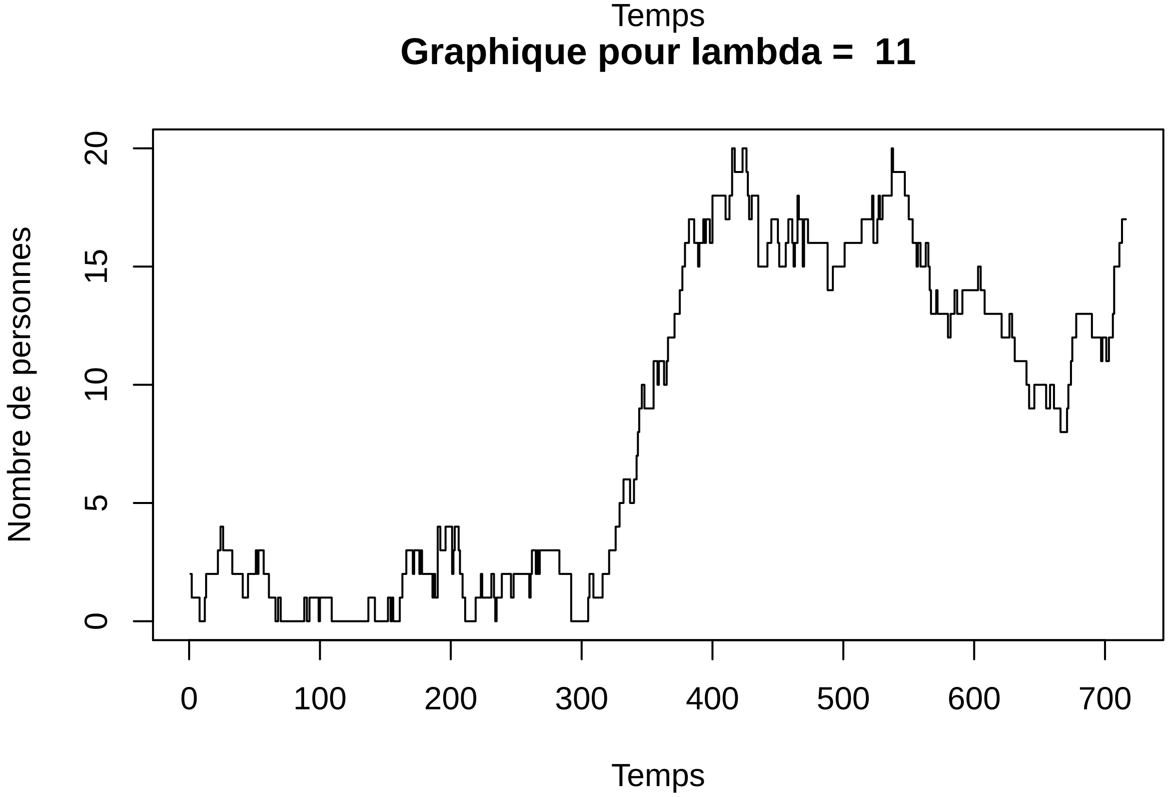
Nous avons choisi la minute comme unité de temps afin d'avoir plus de points dans notre graphique.

On répète ensuite l'expérience avec lambda = 10, puis 11, et enfin 15 :

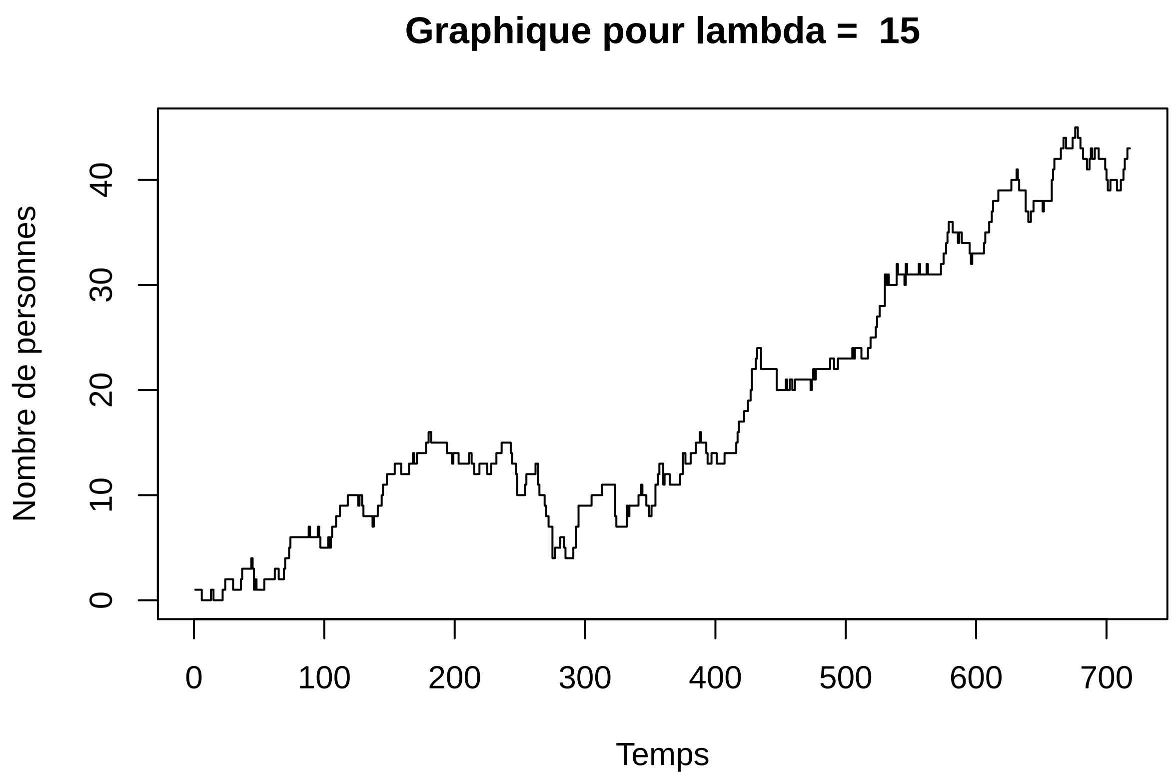
```
for(i in c(10, 11, 15)) {
  file <- FileMM1(lambda = i/60, mu = 11/60, D = 12*60)
  nb_pers <- VisualisationComportement(file$arrivees, file$departs)
  plot(nb_pers$t, nb_pers$n, xlab="Temps", ylab="Nombre de personnes", type="s")
  title=
  title(main = paste("Graphique pour lambda = ", i))
}
```



Graphique pour lambda = 10
Nombre de personnes
0
2
4
6
8
0
100
200
300
400
500
600
700
Temps



Graphique pour lambda = 11
Nombre de personnes
0
5
10
15
20
0
100
200
300
400
500
600
700
Temps



Q8 - Estimations

On cherche à vérifier la loi de Little. Pour cela, on va calculer de façon indépendante le terme de droite et de gauche de l'égalité et constater si elle est vérifiée ou non.

Loi de Little : E(N) = lambda*E(W), où :

- terme_gauche = E(N) = espérance de N (nombre de clients dans la système)
- terme_droite = lambda*E(W) = lambda* espérance de W, le temps durant lequel un client reste dans le système = somme des temps passés dans la file pour chaque client divisé par le nombre de clients, multiplié par lambda

```
CalculNombreMoyenPersonnes <- function(valeursN) {
  return(mean(valeursN))
}

CalculAttenteMoyenne <- function(arrivees, departs) {
  attentes = c()
  for(i in 1:length(departs))
  {
    attentes[i] = departs[i] - arrivees[i]
  }
  return(mean(attentes))
}

SommeTempsPasseFile <- function(arrivees, departs) {
  somme_attentes = 0
```

```
  for(i in 1:length(departs))
  {
    somme_attentes = somme_attentes + (departs[i] - arrivees[i])
  }
  return(somme_attentes)
}
```

On effectue les calculs demandé pour différentes valeurs de lambda et mu et on calcul à chaque fois la valeur du terme de droite et gauche de l'égalité de Little.

```
for(i in c(6, 10, 11, 15)) {
  lambda <- i/60
  mu <- 11/60
  alpha <- lambda / mu

  file <- FileMM1(lambda, mu, D = 12*60)
  nb_pers <- VisualisationComportement(file$arrivees, file$departs)

  moy_pers <- CalculNombreMoyenPersonnes(nb_pers$n)
  attente_moy <- CalculAttenteMoyenne(file$arrivees, file$departs)

  # calcul terme de droite
  terme_droite <- attente_moy*lambda

  cat("Pour lambda =", i, "alpha =", alpha, ": moyenne des personnes =", moy_pers, ", attente moyenne =
}
```

```
## Pour lambda = 6 alpha = 0.55 : moyenne des personnes = 0.99 , attente moyenne = 10
##  Terme droite = 1
## Pour lambda = 10 alpha = 0.91 : moyenne des personnes = 4.6 , attente moyenne = 28
##  Terme droite = 4.6
## Pour lambda = 11 alpha = 1 : moyenne des personnes = 8.7 , attente moyenne = 48
##  Terme droite = 8.8
## Pour lambda = 15 alpha = 1.4 : moyenne des personnes = 37 , attente moyenne = 153
##  Terme droite = 38
```

On remarque très bien que quand alpha <= 1, on a des valeurs de "moyenne des personnes" (terme de gauche) très proches de celles de "Terme droite". Ainsi, dans ces cas, on se trouve donc dans le cas de systèmes dit stabilisés, pour lesquels la formule de Little est bien vérifié. En revanche, dans le cas où alpha > 1, le système n'est plus stabilisé et la formule de Little n'est plus vérifiée.