compte_rendu_tp_R

Florian Rascoussier

7/26/2020

Compte rendu TP R

Q1

Créations 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.

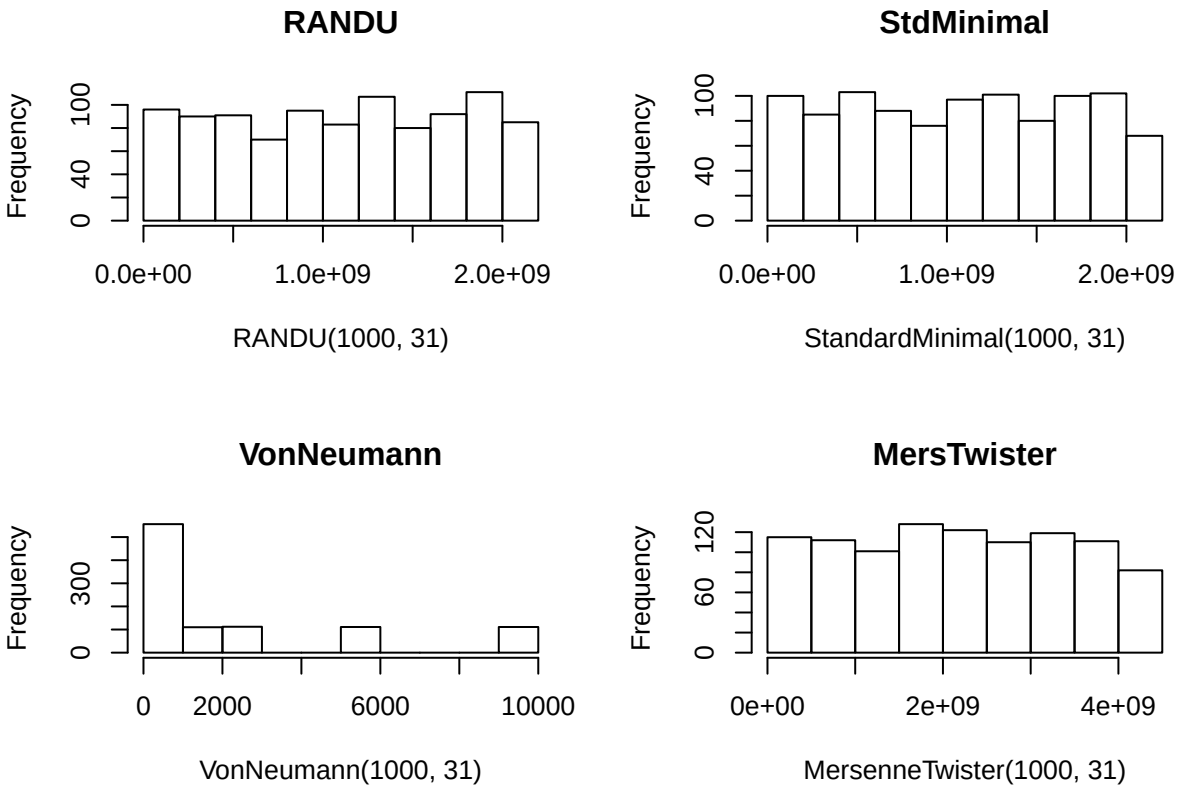
```
RANDU <- function(k, graine) {
  x <-  rep(graine,k)
  # start index 1 (not 0), end index k (not k-1)
  # S0 (at index 1) is already initialized, start at S1 (idx 2)
  for(i in seq(2,k,1)) {
    x[i] <- as.numeric((65539*x[i-1]) %% (2^31))
  }
  return(x)
}

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

Q2.1

Création d'histogrammes pour les différents générateurs.

```
# 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")
```

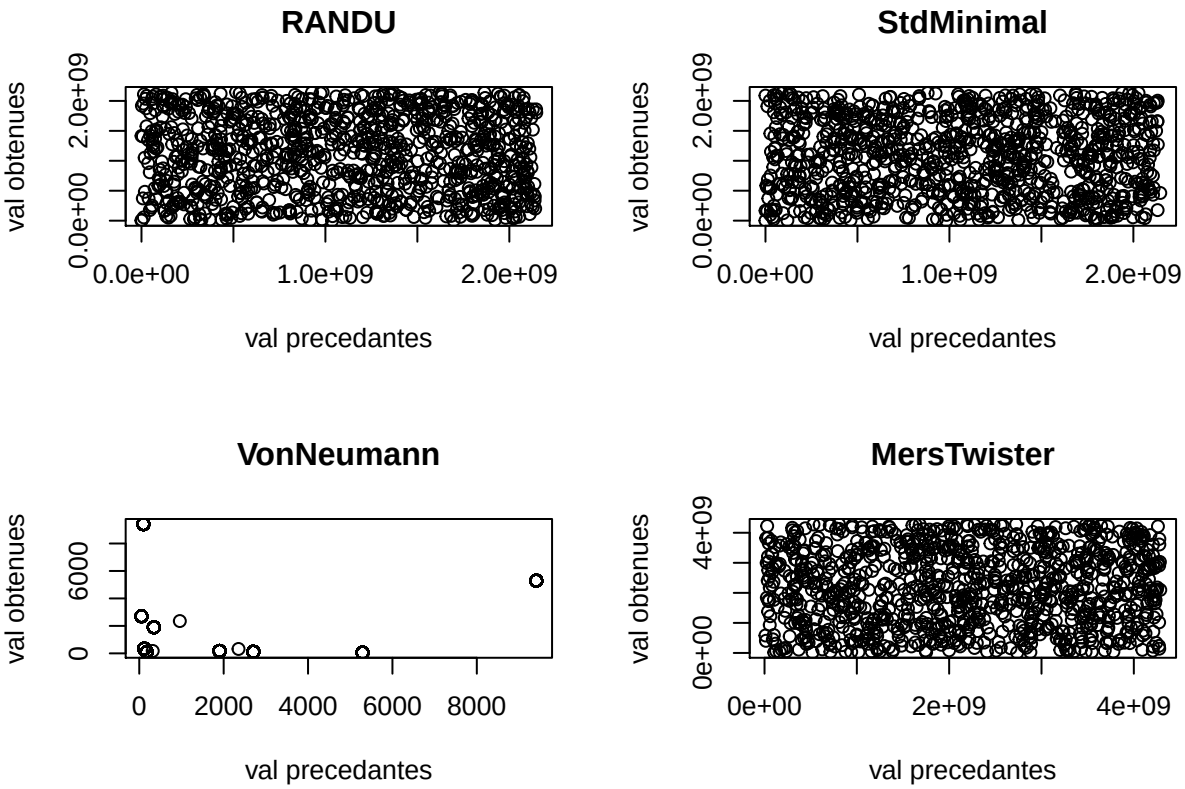


On constate que la répartition des valeurs aléatoire ne respecte pas parfaitement une loi uniforme. En particulier, VonNeumann génère beaucoup de valeurs proches et parfois plus aucune selon l'intervalle. Ce générateur est donc assez peu satisfaisant. StandardMinimal et Randu sont assez proches dans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

Q2.2

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

```
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 precedantes', ylab='val obtenues', main="RANDU")
plot(u2[1:(n-1)], u2[2:n], xlab='val precedantes', ylab='val obtenues', main="StdMinimal")
plot(u3[1:(n-1)], u3[2:n], xlab='val precedantes', ylab='val obtenues', main="VonNeumann")
plot(u4[1:(n-1)], u4[2:n], xlab='val precedantes', ylab='val obtenues', main="MersTwister")
```



On remarque tout de suite pour VonNeumann que les toutes les valeurs sont regroupées en quelques points, ce qui indique que le générateur a tendance à toujours donner certaines valeurs. Il n'est pas très satisfaisant. Ensuite, RANDU et StandardMinimal donnent de meilleurs résultats, avec plus de points mieux répartient indiquant un générateur aléatoire de meilleur qualité. Enfin, MersenneTwister donne le résultat le plus uniforme et réparti. La distinction entre les 3 derniers algorithmes est moins visible qu'avec un histogramme.

Q3

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 effets, tous les nombres générés ne nécessitent pas forcément 32 bits pour être converti du décimal au binaire et on ne veut pas considérer les 0 en trop de la convertion renvoyée par la fonction binary. L'example ci-dessous montre ce phénomène de 0 en trop. Seuls les 0 après un bit 1 de poids le plus fort sont utiles.

```
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
```

La fonction Frequency, renvoie la pValeur pour une série de nombres supposée aléatoire :

```
## 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)
    ## 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)
  }
  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 :

```
## get the number of bits to use to convert a decimal to binary
bitsNecessary <- function(decimalNumber) {
  nbBits <- 0
  while(decimalNumber/2^nbBits > 1.0) {
    nbBits <- nbBits + 1
  }
  return(nbBits)
}
```

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

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

```
## average pValeur VonNeumann :  NA
```

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

```
## average pValeur RANDU :  0.1195535
```

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

```
## average pValeur StandardMinimal :  7.139489e-05
```

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

```
## average pValeur MersenneTwister :  5.976355e-06
```

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

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

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 :

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

```
## seeds :  4863
##  single pValeur VonNeumann :  NA
## seeds :  4398
##  single pValeur VonNeumann :  NA
## seeds :  5038
##  single pValeur VonNeumann :  NA
## seeds :  8116
##  single pValeur VonNeumann :  3.392356e-08
## seeds :  7874
##  single pValeur VonNeumann :  NA
## seeds :  7991
##  single pValeur VonNeumann :  1.542217e-07
## seeds :  5132
##  single pValeur VonNeumann :  1.16392e-07
## seeds :  9362
##  single pValeur VonNeumann :  0
```

En regardant de plus prêt, 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.

Q4

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

```
## 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
      bitOor1 <- seq32Bits[32+1 - j]
      ## error with VonNeumann required fix
      if(is.na(bitOor1)) {
        bitOor1 <- 0
      }
      ## pre-compute proportionOf1
      sumOf1 <- sumOf1 + bitOor1
      ## compute vObs
      if(lastBit != bitOor1) {
        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
  pValeur = 2*(1 - pnorm((abs(vObs - 2*allConsideredBits*proportionOf1*(1 - proportionOf1)))/(2*sqrt(al
  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),
```

```
## average pValeur (runs) VonNeumann :  0.1936601
```

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

```
## average pValeur (runs) RANDU :  0.04240933
```

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

```
## average pValeur (runs) StandardMinimal :  3.239597e-06
```

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

```
## average pValeur (runs) MersenneTwister :  3.893676e-06
```

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

Q5

Cette question est assez rapide puisque la fonction de test est fourni 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. Attention a bien utiliser ici le as.numeric() pour que u soit considéré comme Values, et pas comme Data sur lequel le test ne fonctionne pas.

```
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 :

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

```
## average pValeur (order) VonNeumann :  1.5e-39
```

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

```
## average pValeur (order) RANDU :  0.48
```

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

```
## average pValeur (order) StandardMinimal :  0.49
```

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

```
## average pValeur (order) MersenneTwister :  0.49
```

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.
- pValeur moyenne (runs) VonNeumann $<$ 0,01 donc VonNeumann n'est pas un générateur de séquences aléatoires.
- 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.

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