Fiche TD avec le logiciel : tdr317

Un peu de programmation autour de la loi de Wilcoxon

J.R. Lobry

1 Le nombre de combinaisons

Le nombre possible de sous-ensembles de k éléments pris dans un ensemble de n éléments est donné par :

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

La fonction pré-définie de R est `choose(n, k)`, pour *choose*, choisir k éléments parmi n :

```
choose(7, 3)
[1] 35
choose(7, 0:7)
[1]  1  7 21 35 35 21  7  1
```

On cherche ici à programmer nous même cette fonction en partant de la fonction factorielle, qui est également pré-définie sous R :

```
factorial(7)
[1] 5040
factorial(1:10)
 [1]       1       2       6      24     120     720    5040   40320  362880 3628800
```

Mais comme on veut partir de zéro on fait comme si cette dernière fonction n'existait pas non plus. La programmation de la fonction factorielle est détaillée ci-après, celle de la fonction qui donne le nombre de combinaisons le sera moins pour que vous puissiez vous exercer.

1.1 Programmation de $n!$

Une première solution est de partir de la relation de récurrence $n! = n(n-1)!$ avec $0! = 1$ et de définir la fonction de façon récursive :

```
facto1 <- function(n){
  if(n == 0) {
    return(1)
  } else {
    return(n*facto1(n - 1))
  }
}
facto1(7)
```

```
[1] 5040
```

Cette fonction n'est pas très robuste parce qu'elle ne contrôle pas la validité de son argument, essayez :

```
facto1(pi)
facto1(-1)
```

Une bonne idée consiste donc à contrôler que n est un entier positif ou nul. Attention, la fonction `is.integer()` ne doit pas être utilisée pour tester si une valeur est entière :

```
is.integer(1)
```

```
[1] FALSE
```

```
storage.mode(1)
```

```
[1] "double"
```

```
is.integer(1L)
```

```
[1] TRUE
```

```
storage.mode(1L)
```

```
[1] "integer"
```

En effet, R étant un langage orienté statistiques les constantes numériques sont toujours représentées par défaut avec un maximum de précision, c'est à dire un nombre à virgule de type `double`. Si vous voulez préciser qu'une constante est entière il faut lui accoller la lettre `L` à la fin. La documentation de la fonction `is.integer()` propose de définir une fonction `is.wholenumber()` pour tester si une valeur est entière, aux erreurs d'arrondi près. Nous nous en inspirons pour définir une petite fonction utilitaire pour tester si nous avons un entier positif ou nul :

```
is.entposnul <- function(x){
  if(abs(x - round(x)) >= .Machine$double.eps^0.5) return(FALSE) # pas entier
  if(x < 0) return(FALSE) # pas positif
  return(TRUE) # entier positif ou nul
}
is.entposnul(1)
```

```
[1] TRUE
```

```
is.entposnul(pi)
```

```
[1] FALSE
```

```
is.entposnul(-1)
```

```
[1] FALSE
```

C'est celle que nous utiliserons donc ici :

```
facto2 <- function(n){
  if(!is.entposnul(n)) stop("Entier positif ou nul attendu")
  if(n == 0) {
    return(1)
  } else {
    return(n*facto2(n - 1))
  }
}
```

Essayez :

```
facto2(pi)
facto2(-1)
```

Une autre approche consiste à partir de la relation :

$$n! = \prod_{i=1}^{n} i$$

valable pour $n \geq 0$ (avec par convention le produit vide égal à 1 pour que $0! = 1$) qui s'écrit assez directement sous R grâce à la fonction `prod()`. Notez que cette fonction respecte la convention qui veut que le produit des termes d'un vecteur de longueur nulle, ici `integer(0)` un vecteur d'entiers vide, vaut bien l'élement neutre pour la multiplication :

```
prod(1:7)
```
```
[1] 5040
```
```
prod(integer(0))
```
```
[1] 1
```

Mais nous allons tomber ici dans un petit piège classique de programmation sous R qu'il est important de détailler :

```
facto3 <- function(n){
  if(!is.entposnul(n)) stop("Entier positif ou nul attendu")
  return(prod(1:n))
}
```

En effet, notre fonction ne donne pas la valeur attendue pour 0! :

```
facto3(7)
```
```
[1] 5040
```
```
facto3(0)
```
```
[1] 0
```

Le piège vient de ce que la construction `from:to` ne génère pas nécessairement une suite croissante.

```
1:10
```
```
 [1]  1  2  3  4  5  6  7  8  9 10
```
```
10:1
```
```
 [1] 10  9  8  7  6  5  4  3  2  1
```
```
1:0
```
```
[1] 1 0
```

Dans notre fonction quand $n = 0$ on aura donc un vecteur de deux entiers au lieu d'un vecteur de longueur nulle. C'est une erreur assez naturelle : on pense désigner les n premiers entiers naturels par `1:n`, jusqu'au jour où n est inférieur à 1, et là, bing ! On y fait parfois référence comme étant le piège du `1:0` :

```
library(fortunes)
fortune("1:0")
```
```
Brian D. Ripley: Add to package utils in R-devel, after correction. I was surprised
you had fallen into the 1:0 trap.
Patrick Burns: I'm surprised too -- good catch.
   -- Brian D. Ripley and Patrick Burns (after adding Patrick Burns' head() to the
      utils package)
      R-devel (January 2004)
```

Pour pallier cet inconvénient une solution est d'utiliser la fonction `seq_len()` :

```
 seq_len(7)
[1] 1 2 3 4 5 6 7
 seq_len(0)
integer(0)
 facto4 <- function(n){
   if(!is.entposnul(n)) stop("Entier positif ou nul attendu")
   return(prod(seq_len(n)))
 }
 facto4(7)
[1] 5040
 facto4(0)
[1] 1
```

Cette fonction se comporte bien comme voulu sauf que sous R on a l'habitude de manipuler des fonctions dites vectoriées qui acceptent en argument non pas une simple valeur mais tout un vecteur d'un coup, par exemple pour la fonction factorielle pré-définie `factorial()` :

```
 factorial(0:7)
[1]    1    1    2    6   24  120  720 5040
```

Avant de passer à la vectorisation de notre fonction ouvrons une parenthèse pour voir comment la fonction factorielle est définie dans R. La fonction `body()` nous permet de voir directement le corps d'une fonction :

```
 body(facto4)
{
    if (!is.entposnul(n))
        stop("Entier positif ou nul attendu")
    return(prod(seq_len(n)))
}
 body(factorial)
gamma(x + 1)
```

La fonction factorielle de R est donc définie de façon particulièrement succincte à partir de la fonction `gamma(x)`[1] :

$$\Gamma(x) = \int_0^{+\infty} t^{x-1}\, e^{-t}\, \mathrm{d}t$$

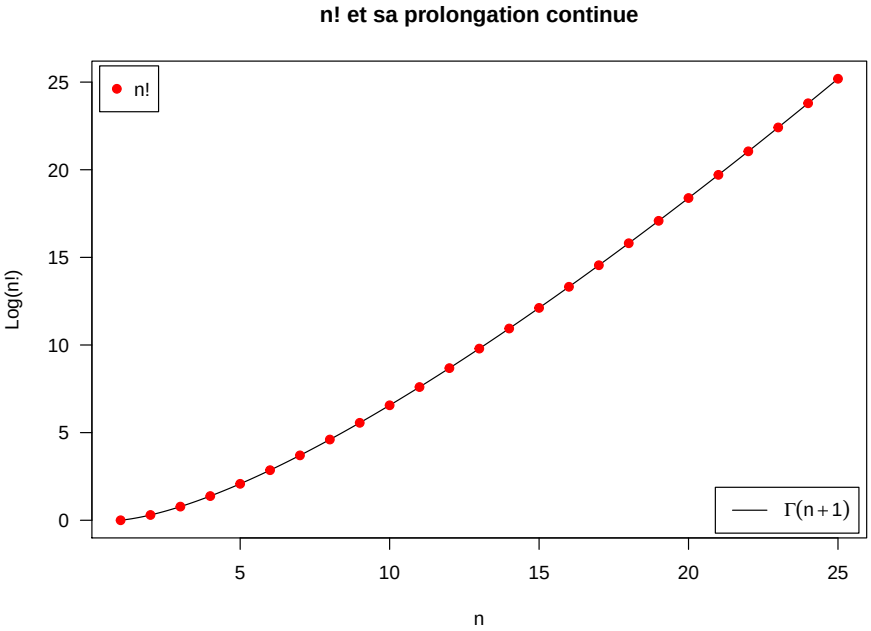
qui est une généralisation de la fonction factorielle pour pouvoir traiter des nombres réels et qui est telle que :

$$n! = \Gamma(n+1)$$

On peut illustrer cette propriété avec le graphique suivant :

```
 nmax <- 25
 x <- seq(2, nmax + 1, length = 200)
 plot(x - 1, log10(gamma(x)), xlab = "n", las = 1, lty = 1, type = "l",
      ylab = "Log(n!)", main = "n! et sa prolongation continue")
 points(1:nmax, log10(factorial(1:nmax)), pch = 19, col = "red")
 legend("bottomright", inset = 0.01, legend = expression(Gamma(n + 1)), lty = 1)
 legend("topleft", inset = 0.01, legend = "n!", pch = 19, col = "red")
```

[1] Voir `?gamma` pour la référence précise vers la page de l'ouvrage de référence d'Abramowitz et Stegun (1972) *Handbook of Mathematical Functions*, ne pas confondre avec la **loi** de probabilité `dgamma()`, *cf* la fiche sur les lois de probabilité `http://pbil.univ-lyon1.fr/R/pdf/tdr21.pdf`



Fermons la parenthèse et revenons au problème de la vectorisation. Une première solution est d'encapsuler notre fonction dans une fonction qui s'occupe de la vectorisation. On utilise ici la fonction `sapply()` qui fonctionne comme `lapply()` mais cherche à simplifier son résultat en un vecteur.

```
facto5 <- function(n){
  f <- function(n){
    if(!is.entposnul(n)) stop("Entier positif ou nul attendu")
    return(prod(seq_len(n)))
  }
  return(sapply(n, f))
}
facto5(0:7)
```

```
[1]    1    1    2    6   24  120  720 5040
```

Un deuxième solution consiste à utiliser la fonction `Vectorize()` qui permet d'obtenir le même résultat sans modifier le corps de la fonction :

```
facto6 <- Vectorize(facto4, "n")
facto6(0:7)
```

```
[1]    1    1    2    6   24  120  720 5040
```

1.2 Programmation du nombre de combinaisons

Définir une fonction `comb1(n, k)` qui calcule le nombre de combinaisons de façon récursive en se basant sur la relation :

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right)$$

et comme condition d'arrêt $\left(\begin{array}{c} n \\ 0 \end{array} \right) = 1$ et $\left(\begin{array}{c} n \\ n \end{array} \right) = 1$. Pensez à tester que `n` et `k` sont positifs ou nuls. Utilisez la fonction `all.equal()` pour tester l'égalité

numérique de deux valeurs. Vectorisez alors votre fonction et vérifiez qu'elle donne bien les mêmes résultats que `choose()` :

```
comb2 <- Vectorize(comb1, "k")
comb2(10, 0:10)
 [1]   1  10  45 120 210 252 210 120  45  10   1
all.equal(comb2(10,0:10), choose(10,0:10))
[1] TRUE
```

Définir maintenant une fonction `comb3(n, k)` à partir de la fonction pré-définie `factorial()` et de la relation :

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

Vectorisez votre fonction et vérifiez qu'elle donne bien les mêmes résultats que `choose()` :

```
comb4 <- Vectorize(comb3, "k")
comb4(10, 0:10)
 [1]   1  10  45 120 210 252 210 120  45  10   1
all.equal(comb4(10,0:10), choose(10, 0:10))
[1] TRUE
```

2 Examen des combinaisons possibles

Nous sommes intéréssés jusqu'ici qu'au nombre total de combinaisons possibles. Nous voulons maintenant aller un peu plus dans le détail et examiner toutes les combinaisons possibles. Nous utilisons pour cela la fonction pré-définie `combn(x, m)` qui permet de lister toutes les combinaisons de `m` objets parmi `x` :

```
combn(7, 3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
[1,]    1    1    1    1    1    1    1    1    1     1     1     1     1     1
[2,]    2    2    2    2    2    3    3    3    3     4     4     4     5     5
[3,]    3    4    5    6    7    4    5    6    7     5     6     7     6     7
     [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27]
[1,]     1     2     2     2     2     2     2     2     2     2     2     3     3
[2,]     6     3     3     3     3     4     4     4     5     5     6     4     4
[3,]     7     4     5     6     7     5     6     7     6     7     7     5     6
     [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
[1,]     3     3     3     3     4     4     4     5
[2,]     4     5     5     6     5     5     6     6
[3,]     7     6     7     7     6     7     7     7
```

Les éléments de l'ensemble `x` sont par défaut notés de 1 à n, mais ce n'est pas une obligation :

```
LETTERS[1:7]
[1] "A" "B" "C" "D" "E" "F" "G"
combn(LETTERS[1:7], 3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
[1,] "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"   "A"   "A"   "A"   "A"
[2,] "B"  "B"  "B"  "B"  "B"  "C"  "C"  "C"  "C"  "D"   "D"   "D"   "E"   "E"
[3,] "C"  "D"  "E"  "F"  "G"  "D"  "E"  "F"  "G"  "E"   "F"   "G"   "F"   "G"
     [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27]
[1,] "A"   "B"   "B"   "B"   "B"   "B"   "B"   "B"   "B"   "B"   "B"   "C"   "C"
[2,] "F"   "C"   "C"   "C"   "C"   "D"   "D"   "D"   "E"   "E"   "F"   "D"   "D"
[3,] "G"   "D"   "E"   "F"   "G"   "E"   "F"   "G"   "F"   "G"   "G"   "E"   "F"
     [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
[1,] "C"   "C"   "C"   "C"   "D"   "D"   "D"   "E"
[2,] "D"   "E"   "E"   "F"   "E"   "E"   "F"   "F"
[3,] "G"   "F"   "G"   "G"   "F"   "G"   "G"   "G"
```

La fonction `combn()` accepte également un argument `FUN` qui est une fonction à appliquer à toutes les combinaisons. On utilise ici par exemple la fonction `tabulate()` pour avoir une représentation spatiale des combinaisons plutôt que la simple liste des rangs. On transpose également le tout pour faciliter la lecture en ligne :

```
combntab <- function(n, k){
  res <- combn(n, k, tabulate, nbins = n)
  return(t(res))
}
combntab(7, 3)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
 [1,]    1    1    1    0    0    0    0
 [2,]    1    1    0    1    0    0    0
 [3,]    1    1    0    0    1    0    0
 [4,]    1    1    0    0    0    1    0
 [5,]    1    1    0    0    0    0    1
 [6,]    1    0    1    1    0    0    0
 [7,]    1    0    1    0    1    0    0
 [8,]    1    0    1    0    0    1    0
 [9,]    1    0    1    0    0    0    1
[10,]    1    0    0    1    1    0    0
[11,]    1    0    0    1    0    1    0
[12,]    1    0    0    1    0    0    1
[13,]    1    0    0    0    1    1    0
[14,]    1    0    0    0    1    0    1
[15,]    1    0    0    0    0    1    1
[16,]    0    1    1    1    0    0    0
[17,]    0    1    1    0    1    0    0
[18,]    0    1    1    0    0    1    0
[19,]    0    1    1    0    0    0    1
[20,]    0    1    0    1    1    0    0
[21,]    0    1    0    1    0    1    0
[22,]    0    1    0    1    0    0    1
[23,]    0    1    0    0    1    1    0
[24,]    0    1    0    0    1    0    1
[25,]    0    1    0    0    0    1    1
[26,]    0    0    1    1    1    0    0
[27,]    0    0    1    1    0    1    0
[28,]    0    0    1    1    0    0    1
[29,]    0    0    1    0    1    1    0
[30,]    0    0    1    0    1    0    1
[31,]    0    0    1    0    0    1    1
[32,]    0    0    0    1    1    1    0
[33,]    0    0    0    1    1    0    1
[34,]    0    0    0    1    0    1    1
[35,]    0    0    0    0    1    1    1
```

En vous inspirant de cette fonction définissez une fonction `sdr(n1, n2)` qui pour deux échantillons de taille `n1` et `n2` liste toutes les possibilités et donne la statistique de la somme des rangs :

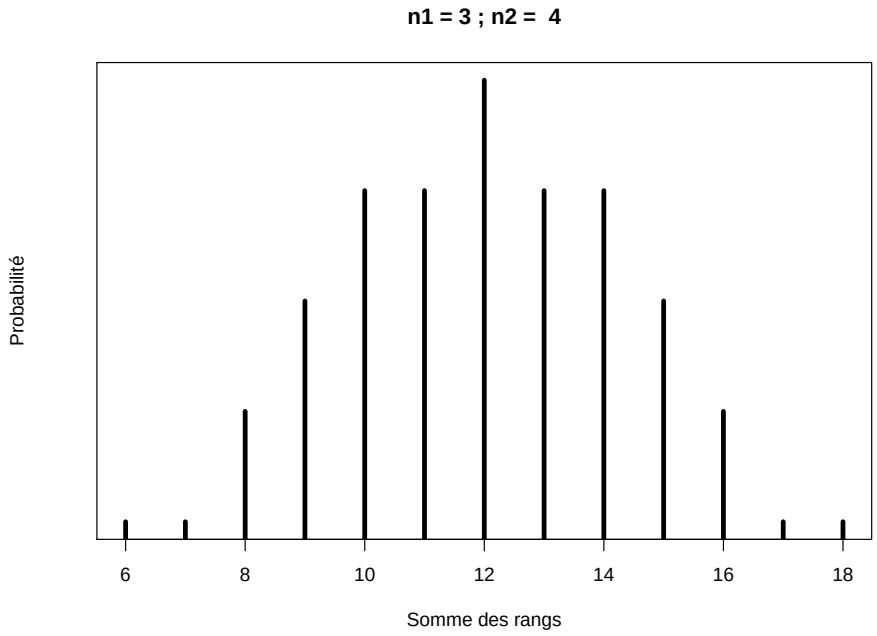
```
sdr(3, 4)
```

```
      1 2 3 4 5 6 7  W
 [1,] 1 1 1 0 0 0 0  6
 [2,] 1 1 0 1 0 0 0  7
 [3,] 1 1 0 0 1 0 0  8
 [4,] 1 1 0 0 0 1 0  9
 [5,] 1 1 0 0 0 0 1 10
 [6,] 1 0 1 1 0 0 0  8
 [7,] 1 0 1 0 1 0 0  9
 [8,] 1 0 1 0 0 1 0 10
 [9,] 1 0 1 0 0 0 1 11
[10,] 1 0 0 1 1 0 0 10
[11,] 1 0 0 1 0 1 0 11
[12,] 1 0 0 1 0 0 1 12
[13,] 1 0 0 0 1 1 0 12
[14,] 1 0 0 0 1 0 1 13
[15,] 1 0 0 0 0 1 1 14
[16,] 0 1 1 1 0 0 0  9
[17,] 0 1 1 0 1 0 0 10
[18,] 0 1 1 0 0 1 0 11
[19,] 0 1 1 0 0 0 1 12
[20,] 0 1 0 1 1 0 0 11
[21,] 0 1 0 1 0 1 0 12
[22,] 0 1 0 1 0 0 1 13
```

```
[23,] 0 1 0 0 1 1 0 13
[24,] 0 1 0 0 1 0 1 14
[25,] 0 1 0 0 0 1 1 15
[26,] 0 0 1 1 1 0 0 12
[27,] 0 0 1 1 0 1 0 13
[28,] 0 0 1 1 0 0 1 14
[29,] 0 0 1 0 1 1 0 14
[30,] 0 0 1 0 1 0 1 15
[31,] 0 0 1 0 0 1 1 16
[32,] 0 0 0 1 1 1 0 15
[33,] 0 0 0 1 1 0 1 16
[34,] 0 0 0 1 0 1 1 17
[35,] 0 0 0 0 1 1 1 18
```

Définir une fonction pour représenter la probabilité d'obtenir chaque somme des rangs possible :

```
plot.sdr <- function(n1, n2, ...){
  W <- combn(n1 + n2, n1, sum)
  Wn <- table(W)
  x <- min(W):max(W)
  plot(x, Wn/sum(Wn), xlab = "Somme des rangs", ylab = "Probabilité",
          main = paste("n1 =", n1, "; n2 = ", n2), type = "h", ...)
}
plot.sdr(3, 4, lwd = 4)
```



Explorez différentes valeurs de `n1` et `n2`, par exemple :

```
plot.sdr(4, 5)
plot.sdr(10, 5)
plot.sdr(10, 3)
plot.sdr(15, 8)
```

Modifier le code de la fonction `plot.sdr()` pour illustrer le fait que l'on a bien retrouvé la densité de probabilité de la loi de Wilcoxon. Attention : `dwilcox()` utilise la statistique translatée de Mann-Whitney dont les valeurs possibles sont `0:(n1*n2)`.

```
plot.sdr2(10, 4, lwd = 4)
```