Mise à niveau R 3/3

Eric Marcon

31 January 2024

Vectoriser

Fonctions vectorielles

La plupart des fonctions de R sont vectorielles :

```
x1 <- runif(3)
x2 <- runif(3)
sqrt(x1)
```

```
## [1] 0.2196613 0.4676215 0.7086498
```

```
x1 + x2
```

```
## [1] 0.4535659 0.6173199 1.4497125
```

Raisonner en termes de vecteurs plutôt que de scalaires.

Fonctions vectorielles

Ecrire des fonctions vectorielles sur leur premier argument :

```
entropart::lnq
```

```
## function (x, q)
## {
##     if (q == 1) {
##         return(log(x))
##     }
##     else {
##         Log <- (x^(1 - q) - 1)/(1 - q)
##         Log[x < 0] <- NA
##         return(Log)
##     }
## }
## <bytecode: 0x7fba0fde8e30>
## <environment: namespace:entropart>
```

Fonctions de résumé

Exceptions à la règle : fonctions d'un vecteur, résultat scalaire.

```
sum(x1)
```

```
## [1] 0.7691056
```

Fonctions non vectorielles

`sapply()` applique une fonction à chaque élément d'un vecteur ou d'une liste.

```
x1 <- runif(1000)
identical(
  sqrt(x1),
  sapply(x1, FUN = sqrt)
)
```

```
## [1] TRUE
```

On utilise donc `sapply()` quand on ne dispose pas d'une fonction vectorielle.

On n'utilise donc jamais `sapply()` avec `FUN = sqrt`.

Fonctions non vectorielles

Fonctions similaires :

```
library("microbenchmark")
mb <- microbenchmark(sqrt(x1), sapply(x1, FUN = sqrt),
    lapply(x1, sqrt), vapply(x1, sqrt, FUN.VALUE = 0))
summary(mb)[, c("expr", "median")]
```

```
##                                 expr   median
## 1                           sqrt(x1)  10.0705
## 2            sapply(x1, FUN = sqrt) 487.8405
## 3                  lapply(x1, sqrt) 404.5835
## 4 vapply(x1, sqrt, FUN.VALUE = 0) 435.9040
```

Infiniment plus lent qu'une fonction vectorielle.

- `lapply()` renvoie une liste (économise le temps de `simplify2array()`) - `vapply()` économise le temps de détermination du type du vecteur.

Boucles

Les boucles sont plus rapides !

```
boucle <- function(x) {
  racine <- numeric(length(x))
  for(i in 1:length(x)) racine[i] <- sqrt(x[i])
  return(racine)
}
vapply_sqrt <- function(x) vapply(x, FUN = sqrt, 0)
mb <- microbenchmark(vapply_sqrt(x1), boucle(x1))
summary(mb)[, c("expr", "median")]
```

```
##              expr   median
## 1 vapply_sqrt(x1) 379.7635
## 2      boucle(x1) 103.1430
```

Boucles

Les boucles longues permettent un suivi :

```
boucle <- function(x) {
  pgb <- txtProgressBar(min = 0, max = length(x))
  racine <- numeric(length(x))
  for(i in 1:length(x)) {
    racine[i] <- sqrt(x[i])
    setTxtProgressBar(pgb, i)
  }
  return(racine)
}
racine_x1 <- boucle(x1)
```

```
## ================================================
```

Mais le package *pbapply* aussi.

replicate et vectorize

`replicate()` répète une instruction.

```
replicate(3, runif(1))
```

```
## [1] 0.1349295 0.7730813 0.7825757
```

est équivalent à `runif(3)`. A utiliser avec des fonctions non vectorielles.

`vectorize()` rend vectorielle une fonction qui ne l'est pas par des boucles. Ecrire plutôt les boucles.

Pratique

Vectoriser un problème

Données : inventaire d'une parcelle de Paracou, 4 carrés distincts.

Objectif : calculer le nombre d'arbres par espèce, le nombre d'arbres par carré, la biodiversité par carré.

Technique : utiliser les fonctions vectorielles, les fonctions de type `apply`, éventuellement des boucles.

Lecture et organisation des données

Lecture des arbres de la parcelle 6 de Paracou

```
# Lecture des arbres de la parcelle 6 de Paracou
paracou6 <- read.csv2("data/Paracou6.csv")
```

Création d'un tableau croisé :

```
paracou6_x <- as.data.frame.matrix(xtabs(
  ~paste(Family, Genus, Species) + SubPlot, data = paracou6
))
paracou6_x[1:2, ]
```

```
##                                        1 2 3 4
## Anacardiaceae Anacardium spruceanum 0 1 0 2
## Anacardiaceae Tapirira guianensis   1 2 0 0
```

`as.data.frame.matrix` est la méthode de conversion des matrices en dataframes...

Tableau croisé dans le tidyverse


```
library("tidyverse")
read.csv2("data/Paracou6.csv") |>
  # Nouvelle colonne
  unite(col = spName, Family, Genus, Species, sep = " ") |>
  # Regrouper et résumer
  group_by(spName, SubPlot) |>
  summarise(abundance = n()) |>
  # Voir l'aide de la fonction pivot_wider
  pivot_wider(names_from = SubPlot, values_from = abundance,
              names_sort = TRUE, values_fill = 0) ->
  paracou6_pw
```

Syntaxe plus verbeuse mais n'importe quelle statistique est possible, pas seulement le comptage.

Statistiques marginales

`apply()` applique une fonction aux lignes ou colonnes d'un objet 2D.

`colSums()` et semblables (`colMeans()`, `rowMeans()`) sont optimisées.

```
paracou6_x <- as.matrix(paracou6_pw[, -1])
mb <- microbenchmark(
  apply(paracou6_x, MARGIN = 2, FUN = sum),
  colSums(paracou6_x)
)
summary(mb)[, c("expr", "median")]
```

```
##                                       expr median
## 1 apply(paracou6_x, MARGIN = 2, FUN = sum) 38.789
## 2                      colSums(paracou6_x)  6.576
```

```
colSums(paracou6_x)
```

```
##   1   2   3   4
## 942 872 929 798
```

Comptage du nombre d'espèces

```
mb <- microbenchmark(
  apply(paracou6_x, 2, function(x) sum(x > 0)),
  colSums(paracou6_x > 0)
)
summary(mb)[, c("expr", "median")]
```

```
##                                          expr
## 1 apply(paracou6_x, 2, function(x) sum(x > 0))
## 2                      colSums(paracou6_x > 0)
##    median
## 1 46.9625
## 2 11.0160
```

```
colSums(paracou6_x > 0)
```

```
##   1   2   3   4
## 189 200 197 177
```

Remarquer :

- le comptage d'un résultat de test (`TRUE` vaut 1, `FALSE` vaut 0)
- la définition d'une fonction sans nom, appelée "fonction

Fonctions complexes

Estimation de la richesse spécifique avec *entropart*

```
library("entropart")
apply(paracou6_x, MARGIN = 2, FUN = Richness)
```

```
##   1   2   3   4
## 355 348 315 296
```

Exercice

Estimer par simulation l'espérance et la variance d'une loi binomiale.

1. Effectuer 10 tirages dans une loi uniforme ;
2. Compter le nombre de succès = résultats inférieurs à la probabilité `p` choisie ;
3. Effectuer 5 fois 10 tirages ;
4. Calculer la moyenne et l'espérance des 5 nombres de succès.
5. Passer à l'échelle : 10000 tirages, 1000 répétitions.

Aide pour l'exercice

1. `runif()`, paramètre `tirages_n`
2. Somme des valeurs vraies
3. `replicate` fournit une matrice ; paramètre `repetitions_n`.
4. `colSums`
5. Changer les paramètres.

Solution de l'exercice


```
p <- 0.5
tirages_n <- 10000
repetitions_n <- 1000
succes_n <- colSums(replicate(repetitions_n, runif(tirages_n)) < p)
mean(succes_n)
```

```
## [1] 4998.148
```

```
sd(succes_n)
```

```
## [1] 51.40047
```

Conclusion

Vecteurs ou tidy

Deux approches différentes :

- R classique : sélection par `[ ]`, fonctions *x*`apply()` ;
- Tidyverse : pipelines, données rectangulaires.

Le tidyverse est très efficace pour la bagarre avec les données, les *x*`apply()` pour appliquer les mêmes fonctions à plusieurs vecteurs de données.

AgroParisTech

Mise à niveau R 3/3

Eric Marcon