

Formation R Perfectionnement

Drees – 16-17 avril 2018



Martin CHEVALIER (Insee)

Travailler efficacement sur des données avec R

Qu'est-ce que travailler efficacement avec R ?

Appliqué au travail sur des données, l'efficacité peut avoir au moins deux significations distinctes :

- ▶ efficacité **algorithmique** : minimisation du temps passé par la machine pour réaliser une série d'opérations ;
- ▶ **productivité** du programmeur : minimisation du temps passé à coder une série d'opération.

Travailler efficacement sur des données avec R

Qu'est-ce que travailler efficacement avec R ?

Appliqué au travail sur des données, l'efficacité peut avoir au moins deux significations distinctes :

- ▶ efficacité **algorithmique** : minimisation du temps passé par la machine pour réaliser une série d'opérations ;
- ▶ **productivité** du programmeur : minimisation du temps passé à coder une série d'opération.

En règle générale, on peut avoir l'idée que plus on souhaite être efficace algorithmiquement, plus la programmation risque d'être longue et difficile.

Travailler efficacement sur des données avec R

Qu'est-ce que travailler efficacement avec R ?

Appliqué au travail sur des données, l'efficacité peut avoir au moins deux significations distinctes :

- ▶ efficacité **algorithmique** : minimisation du temps passé par la machine pour réaliser une série d'opérations ;
- ▶ **productivité** du programmeur : minimisation du temps passé à coder une série d'opération.

En règle générale, on peut avoir l'idée que plus on souhaite être efficace algorithmiquement, plus la programmation risque d'être longue et difficile.

Ce n'est pas toujours vrai : on perd souvent beaucoup de temps à (ré)inventer une méthode peu efficace quand une beaucoup plus simple et rapide existe déjà.

Travailler efficacement sur des données avec R

Qu'est-ce que travailler efficacement avec R ?

Appliqué au travail sur des données, l'efficacité peut avoir au moins deux significations distinctes :

- ▶ efficacité **algorithmique** : minimisation du temps passé par la machine pour réaliser une série d'opérations ;
- ▶ **productivité** du programmeur : minimisation du temps passé à coder une série d'opération.

En règle générale, on peut avoir l'idée que plus on souhaite être efficace algorithmiquement, plus la programmation risque d'être longue et difficile.

Ce n'est pas toujours vrai : on perd souvent beaucoup de temps à (ré)inventer une méthode peu efficace quand une beaucoup plus simple et rapide existe déjà.

Référence GILLEPSIE C., LOVELACE R., *Efficient R programming* (disponible sur bookdown.org)

Travailler efficacement sur des données avec R

Mesure l'efficacité algorithmique

La fonction `system.time()` permet de mesurer la durée d'un traitement.

```
system.time(rnorm(1e6))  
##      user   system elapsed  
##      0.17      0.00      0.18
```

Travailler efficacement sur des données avec R

Mesure l'efficacité algorithmique

La fonction `system.time()` permet de mesurer la durée d'un traitement.

```
system.time(rnorm(1e6))  
##      user      system elapsed  
##      0.17       0.00       0.18
```

Néanmoins, elle est inadaptée aux traitements de très courte durée. Dans ces situations, privilégier la fonction `microbenchmark()` du package `microbenchmark`.

```
library(microbenchmark)  
microbenchmark(times = 10, rnorm(1e6))  
## Unit: milliseconds  
##           expr          min          lq          mean          median  
##  rnorm(1e+06) 98.33079 100.6687 101.3566 101.0949  
##           uq          max neval  
## 101.9453 104.6314      10
```

Travailler efficacement sur des données avec R

Mesurer la taille d'un objet en mémoire

R stocke l'ensemble des fichiers sur lesquels il travaille dans la mémoire vive.

Afin de loger les objets les plus gros mais aussi d'optimiser les performances, il est souvent utile de **limiter la taille des objets** sur lesquels portent les traitements.

Travailler efficacement sur des données avec R

Mesurer la taille d'un objet en mémoire

R stocke l'ensemble des fichiers sur lesquels il travaille dans la mémoire vive.

Afin de loger les objets les plus gros mais aussi d'optimiser les performances, il est souvent utile de **limiter la taille des objets** sur lesquels portent les traitements.

Pour mesurer la taille des objets, utiliser la fonction `object_size()` du *package* `pryr`.

```
library(pryr)
object_size(rnorm(1e6))
## 8 MB
```

Travailler efficacement sur des données avec R

Construire un exemple reproductible (MWE)

Lorsque l'on cherche à améliorer les performances d'un programme, il est important de pouvoir le tester sur des données **autonomes et reproductibles**.

Travailler efficacement sur des données avec R

Construire un exemple reproductible (MWE)

Lorsque l'on cherche à améliorer les performances d'un programme, il est important de pouvoir le tester sur des données **autonomes et reproductibles**.

Pour ce faire, les **fonctions de générations de nombres aléatoires** de R sont particulièrement utiles.

```
# Graine pour pouvoir reproduire l'aléa
set.seed(2018)

# Vecteur de nombres de taille 1 000
a <- rnorm(1000)

# Vecteur de lettres de taille 1 000
b <- letters[sample(1:26, 1000, replace = TRUE)]

# Matrice logique 1 000 x 100 avec 1 % de TRUE
c <- matrix(runif(100000) > 0.99, ncol = 100)
```

Travailler efficacement sur des données avec R

Plan de la partie

De l'importance des fonctions dans R

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Coder efficacement en base R

`dplyr` : une grammaire du traitement des données

`data.table` : un `data.frame` optimisé

Aller plus loin avec R

De l'importance des fonctions dans R

« Tout ce qui agit est un appel de fonction »

To understand computations in R, two slogans are helpful :

- ▶ *Everything that exists is an object.*
- ▶ *Everything that happens is a function call.*

John Chambers

De l'importance des fonctions dans R

« Tout ce qui agit est un appel de fonction »

To understand computations in R, two slogans are helpful :

- ▶ *Everything that exists is an object.*
- ▶ *Everything that happens is a function call.*

John Chambers

```
# ... même assigner une valeur
is.function(`<-`)
## [1] TRUE
`<-`(a, 10)

# ... même afficher la valeur d'un objet
a
## [1] 10
print(a)
## [1] 10
```

De l'importance des fonctions dans R

Définir une fonction dans R

Utilisé avec `<-`, `function()` définit une nouvelle fonction :

De l'importance des fonctions dans R

Définir une fonction dans R

Utilisé avec `<-`, `function()` définit une nouvelle fonction :

```
# Définition de la fonction monCalcul()
```

```
monCalcul <- function(a, b){  
  resultat <- 10 * a + b  
  return(resultat)  
}
```

```
# Code de monCalcul()
```

```
monCalcul  
## function(a, b){  
##   resultat <- 10 * a + b  
##   return(resultat)  
## }
```

```
# Appel de la fonction monCalcul()
```

```
monCalcul(2, 3)  
## [1] 23
```


De l'importance des fonctions dans R

Valeurs par défaut des paramètres

Des valeurs par défaut peuvent être renseignées pour les paramètres.

```
monCalcul <- function(a, b = 3) 10 * a + b  
monCalcul(8)  
## [1] 83
```

De l'importance des fonctions dans R

Valeurs par défaut des paramètres

Des valeurs par défaut peuvent être renseignées pour les paramètres.

```
monCalcul <- function(a, b = 3) 10 * a + b  
monCalcul(8)  
## [1] 83
```

Les valeurs par défaut peuvent dépendre des autres paramètres.

```
monCalcul <- function(a, b = a * 2) 10 * a + b  
monCalcul(2)  
## [1] 24
```

De l'importance des fonctions dans R

Valeurs par défaut des paramètres

Des valeurs par défaut peuvent être renseignées pour les paramètres.

```
monCalcul <- function(a, b = 3) 10 * a + b
monCalcul(8)
## [1] 83
```

Les valeurs par défaut peuvent dépendre des autres paramètres.

```
monCalcul <- function(a, b = a * 2) 10 * a + b
monCalcul(2)
## [1] 24
```

Remarque Ceci est la conséquence de la *lazy evaluation* des arguments dans R (cf. *Advanced R*).

De l'importance des fonctions dans R

Contrôle de la valeur des paramètres

Des structures conditionnelles `if()` permettent de contrôler la valeur des arguments.

De l'importance des fonctions dans R

Contrôle de la valeur des paramètres

Des structures conditionnelles `if()` permettent de contrôler la valeur des arguments.

```
monCalcul <- function(a = NULL, b = NULL){  
  if(is.null(a)) stop("a n'est pas renseigné.")  
  if(is.null(b)){  
    b <- a * 2  
    warning("b n'est pas renseigné.")  
  }  
  return(10 * a + b)  
}
```

```
monCalcul(b = 3)  
## Error in monCalcul(b = 3): a n'est pas renseigné.  
monCalcul(a = 1)  
## Warning in monCalcul(a = 1): b n'est pas renseigné.  
## [1] 12
```

De l'importance des fonctions dans R

Portée des variables et environnements (1)

Dans R **chaque objet est repéré par son nom et son environnement** : cela permet d'éviter les conflits de noms.

De l'importance des fonctions dans R

Portée des variables et environnements (1)

Dans R **chaque objet est repéré par son nom et son environnement** : cela permet d'éviter les conflits de noms.

```
# Création d'une fonction sum() un peu absurde
sum <- function(...) "Ma super somme !"
sum(2, 3)
## [1] "Ma super somme !"

# Cette fonction est rattachée à l'environnement global
ls()
## [1] "a"          "b"          "c"          "file"
## [5] "monCalcul" "sum"

# Mais on peut toujours accéder à la fonction
# de base en utilisant ::
base::sum(2, 3)
## [1] 5
```

De l'importance des fonctions dans R

Portée des variables et environnements (2)

À chaque appel d'une fonction, un **environnement d'exécution** est créé.

```
maFun <- function() environment()  
maFun()  
## <environment: 0x00000000d466ec8>  
maFun()  
## <environment: 0x00000000d400448>
```


De l'importance des fonctions dans R

Portée des variables et environnements (2)

À chaque appel d'une fonction, un **environnement d'exécution** est créé.

```
maFun <- function() environment()  
maFun()  
## <environment: 0x00000000d466ec8>  
maFun()  
## <environment: 0x00000000d400448>
```

En conséquence, les instructions exécutées à l'intérieur d'une fonction **ne modifient pas l'environnement global**.

```
a <- 10  
maFonction3 <- function(){  
  a <- 5  
}  
maFonction3()  
a  
## [1] 10
```

De l'importance des fonctions dans R

Portée des variables et environnements (3)

En revanche, les objets définis dans l'environnement global sont accessibles au sein d'une fonction.

```
a <- 10
maFonction4 <- function(){
  a + 5
}
maFonction4()
## [1] 15
```

De l'importance des fonctions dans R

Portée des variables et environnements (3)

En revanche, les objets définis dans l'environnement global sont accessibles au sein d'une fonction.

```
a <- 10
maFonction4 <- function(){
  a + 5
}
maFonction4()
## [1] 15
```

Ceci est dû au fait que les environnements dans lequel R recherche des objets sont **emboîtés les uns dans les autres** (cf. la fonction `search()`).

Pour en savoir plus [Advanced R](#), obeautifulcode.com

De l'importance des fonctions dans R

Valeur de retour d'une fonction

La fonction `return()` spécifie la valeur à renvoyer. Pour renvoyer plusieurs valeurs, utiliser une liste.

De l'importance des fonctions dans R

Valeur de retour d'une fonction

La fonction `return()` spécifie la valeur à renvoyer. Pour renvoyer plusieurs valeurs, utiliser une liste.

```
maFonction1 <- function(){  
  a <- 1:5; b <- 6:10; return(a)  
}
```

```
maFonction1()  
## [1] 1 2 3 4 5
```

```
maFonction2 <- function(){  
  a <- 1:5; b <- 6:10; return(list(a = a, b = b))  
}
```

```
maFonction2()  
## $a  
## [1] 1 2 3 4 5  
##  
## $b  
## [1] 6 7 8 9 10
```

De l'importance des fonctions dans R

Effets de bord et programmation fonctionnelle

Par défaut, les fonctions dans R :

- ▶ ne modifient pas l'environnement d'origine (il n'y a **pas d'effets de bord**);
- ▶ peuvent être utilisées en lieu et place des valeurs qu'elles retournent.

```
monCalcul <- function(a, b) 10 * a + b
monCalcul(2, 3) + 5
## [1] 28
```

De l'importance des fonctions dans R

Effets de bord et programmation fonctionnelle

Par défaut, les fonctions dans R :

- ▶ ne modifient pas l'environnement d'origine (il n'y a **pas d'effets de bord**) ;
- ▶ peuvent être utilisées en lieu et place des valeurs qu'elles retournent.

```
monCalcul <- function(a, b) 10 * a + b
monCalcul(2, 3) + 5
## [1] 28
```

Ces éléments font de R un **langage particulièrement adapté à la programmation fonctionnelle**.

De l'importance des fonctions dans R

Quelques principes de la programmation fonctionnelle

1. **Ne jamais créer d'effets de bord** Toute modification apportée à l'environnement par une fonction passe par sa valeur de sortie.

Quelques principes de la programmation fonctionnelle

1. **Ne jamais créer d'effets de bord** Toute modification apportée à l'environnement par une fonction passe par sa valeur de sortie.
2. **Vectoriser *i.e.* appliquer des fonctions systématiquement à un ensemble d'éléments**
Fonctions `*apply()`, `Reduce()`, `do.call()`.

Quelques principes de la programmation fonctionnelle

1. **Ne jamais créer d'effets de bord** Toute modification apportée à l'environnement par une fonction passe par sa valeur de sortie.
2. **Vectoriser *i.e.* appliquer des fonctions systématiquement à un ensemble d'éléments**
Fonctions `*apply()`, `Reduce()`, `do.call()`.
3. **Structurer les traitements à l'aide de fonctions courtes et explicites** Faciliter la relecture, la maintenance et la modularisation.

Quelques principes de la programmation fonctionnelle

1. **Ne jamais créer d'effets de bord** Toute modification apportée à l'environnement par une fonction passe par sa valeur de sortie.
2. **Vectoriser *i.e.* appliquer des fonctions systématiquement à un ensemble d'éléments**
Fonctions `*apply()`, `Reduce()`, `do.call()`.
3. **Structurer les traitements à l'aide de fonctions courtes et explicites** Faciliter la relecture, la maintenance et la modularisation.

Pour en savoir plus [Wikipedia](#), maryrosecook.com.

Vectoriser : *`apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque indépendamment : `apply()`

La fonction `apply(X, MARGIN, FUN)` applique la fonction `FUN` à la **matrice** `X` selon la dimension `MARGIN`.

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur chaque indépendamment : apply()

La fonction apply(X, MARGIN, FUN) applique la fonction FUN à la **matrice** X selon la dimension MARGIN.

```
# Définition et affichage de la matrice m
```

```
m <- matrix(1:6, ncol = 3)
```

```
m
```

```
##      [,1] [,2] [,3]
```

```
## [1,]    1    3    5
```

```
## [2,]    2    4    6
```

```
# Application de la fonction sum() selon les lignes
```

```
apply(m, 1, sum)
```

```
## [1]  9 12
```

```
# Application de la fonction sum() selon les colonnes
```

```
apply(m, 2, sum)
```

```
## [1]  3  7 11
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque indépendamment : `lapply()`

La fonction `lapply(X, FUN)` applique la fonction `FUN` au **vecteur** ou à la **liste** `X`.

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur chaque indépendamment : lapply()

La fonction lapply(X, FUN) applique la fonction FUN au **vecteur** ou à la **liste** X.

```
l <- list(1:5, c(6:9, NA))
```

```
l
```

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

```
## [1] 1 2 3 4 5
```

```
##
```

```
## [[2]]
```

```
## [1] 6 7 8 9 NA
```

```
lapply(l, sum)
```

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

```
## [1] 15
```

```
##
```

```
## [[2]]
```

```
## [1] NA
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque indépendamment : `lapply()`

La fonction `lapply(X, FUN)` applique la fonction `FUN` au **vecteur** ou à la **liste** `X`.

```
l <- list(1:5, c(6:9, NA))
l
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 6 7 8 9 NA
lapply(l, sum)
## [[1]]
## [1] 15
##
## [[2]]
## [1] NA
```

Exemple d'utilisation Appliquer une fonction à toutes les variables d'une table.

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur chaque indépendamment : sapply()

La fonction sapply() est analogue à la fonction lapply(), mais simplifie le résultat produit quand c'est possible.

```
sapply(1, sum)
## [1] 15 NA
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque indépendamment : `sapply()`

La fonction `sapply()` est analogue à la fonction `lapply()`, mais simplifie le résultat produit quand c'est possible.

```
sapply(1, sum)
## [1] 15 NA
```

Les arguments optionnels de la fonction utilisée peuvent être ajoutés à la suite dans toutes les fonctions `*apply()`.

```
sapply(1, sum, na.rm = TRUE)
## [1] 15 30
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque indépendamment : `sapply()`

La fonction `sapply()` est analogue à la fonction `lapply()`, mais simplifie le résultat produit quand c'est possible.

```
sapply(1, sum)
## [1] 15 NA
```

Les arguments optionnels de la fonction utilisée peuvent être ajoutés à la suite dans toutes les fonctions `*apply()`.

```
sapply(1, sum, na.rm = TRUE)
## [1] 15 30
```

Exemple d'utilisation Calcul de statistiques sur toutes les variables d'une table.

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Définir une fonction à la volée dans `*apply()`

Il est fréquent que l'opération que l'on souhaite appliquer ne corresponde pas exactement à une fonction pré-existante.

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Définir une fonction à la volée dans `*apply()`

Il est fréquent que l'opération que l'on souhaite appliquer ne corresponde pas exactement à une fonction pré-existante.

Dans ce cas, on peut définir une **fonction à la volée** dans la fonction `*apply()`.

```
# On souhaite sélectionner le second élément de
# de chaque vecteur de la liste l
l
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 6 7 8 9 NA

# On définit une fonction dans sapply()
sapply(l, function(x) x[2])
## [1] 2 7
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque par groupe : `tapply()`

La fonction `tapply(X, INDEX, FUN)` applique la fonction `FUN`, à l'objet `X` ventilé selon les modalités de `INDEX`.

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque par groupe : `tapply()`

La fonction `tapply(X, INDEX, FUN)` applique la fonction `FUN`, à l'objet `X` ventilé selon les modalités de `INDEX`.

```
# Variables d'âge et de sexe
```

```
age <- c(45, 50, 35, 20)
```

```
sexe <- c("H", "F", "F", "H")
```

```
# Âge moyen par sexe
```

```
tapply(age, sexe, mean)
```

```
##      F      H
```

```
## 42.5 32.5
```

```
# Même résultat avec une combinaison de sapply() et de split()
```

```
sapply(split(age, sexe), mean)
```

```
##      F      H
```

```
## 42.5 32.5
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur chaque par groupe : `tapply()`

La fonction `tapply(X, INDEX, FUN)` applique la fonction `FUN`, à l'objet `X` ventilé selon les modalités de `INDEX`.

```
# Variables d'âge et de sexe
```

```
age <- c(45, 50, 35, 20)
```

```
sexe <- c("H", "F", "F", "H")
```

```
# Âge moyen par sexe
```

```
tapply(age, sexe, mean)
```

```
##      F      H
```

```
## 42.5 32.5
```

```
# Même résultat avec une combinaison de sapply() et de split()
```

```
sapply(split(age, sexe), mean)
```

```
##      F      H
```

```
## 42.5 32.5
```

Exemple d'utilisation Calcul de statistiques agrégées par catégories.

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous : `do.call()`

La fonction `do.call(what, args)` permet d'appliquer la fonction `what()` à un **ensemble** d'arguments `args` spécifié comme une liste (alors que les fonctions `*apply()` appliqueraient `what()` à **chaque** élément de `args`).

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous : `do.call()`

La fonction `do.call(what, args)` permet d'appliquer la fonction `what()` à un **ensemble** d'arguments `args` spécifié comme une liste (alors que les fonctions `*apply()` appliqueraient `what()` à **chaque** élément de `args`).

```
# Concaténation des vecteurs de 1
do.call(base::c, 1)
## [1] 1 2 3 4 5 6 7 8 9 NA

# Equivalent à
base::c(1[[1]], 1[[2]])
## [1] 1 2 3 4 5 6 7 8 9 NA
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous : `do.call()`

La fonction `do.call(what, args)` permet d'appliquer la fonction `what()` à un **ensemble** d'arguments `args` spécifié comme une liste (alors que les fonctions `*apply()` appliqueraient `what()` à **chaque** élément de `args`).

```
# Concaténation des vecteurs de 1
do.call(base::c, 1)
## [1] 1 2 3 4 5 6 7 8 9 NA

# Equivalent à
base::c(1[[1]], 1[[2]])
## [1] 1 2 3 4 5 6 7 8 9 NA
```

Exemple d'utilisation Concaténer de nombreuses tables avec `rbind()` ou `cbind()`.

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous successivement : `Reduce()`

La fonction `Reduce(f, x)` permet d'appliquer la fonction `f()` **successivement** à l'ensemble des éléments de `x` (alors que `do.call()` applique `f` **simultanément**).

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous successivement : `Reduce()`

La fonction `Reduce(f, x)` permet d'appliquer la fonction `f()` **successivement** à l'ensemble des éléments de `x` (alors que `do.call()` applique `f` **simultanément**).

```
# Application successive de la division au vecteur 1:4
```

```
Reduce(`/`, 1:4)
```

```
## [1] 0.04166667
```

```
# Equivalent à
```

```
((1/2)/3)/4
```

```
## [1] 0.04166667
```

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Appliquer sur tous successivement : `Reduce()`

La fonction `Reduce(f, x)` permet d'appliquer la fonction `f()` **successivement** à l'ensemble des éléments de `x` (alors que `do.call()` applique `f` **simultanément**).

```
# Application successive de la division au vecteur 1:4
Reduce(`/`, 1:4)
## [1] 0.04166667

# Equivalent à
((1/2)/3)/4
## [1] 0.04166667
```

Exemple d'utilisation Fusionner de nombreuses tables avec `merge()` (sur les mêmes identifiants).

Coder efficacement en base R

L'idée : En faire faire le moins possible à R

R est un langage dit « de haut niveau » : les objets qui le composent sont relativement faciles d'utilisation, au prix de performances limitées.

À l'inverse, des langages dits de « bas niveau » (par exemple C++) sont plus difficiles à utiliser mais aussi plus efficaces.

Coder efficacement en base R

L'idée : En faire faire le moins possible à R

R est un langage dit « de haut niveau » : les objets qui le composent sont relativement faciles d'utilisation, au prix de performances limitées.

À l'inverse, des langages dits de « bas niveau » (par exemple C++) sont plus difficiles à utiliser mais aussi plus efficaces.

La plupart des fonctions fondamentales de R font appel à des fonctions compilées à partir d'un langage de plus bas niveau.

D'où le principe : **limiter au maximum la surcharge liée à R** pour retomber au plus vite sur des fonctions pré-compilées.

Coder efficacement en base R

L'idée : En faire faire le moins possible à R

R est un langage dit « de haut niveau » : les objets qui le composent sont relativement faciles d'utilisation, au prix de performances limitées.

À l'inverse, des langages dits de « bas niveau » (par exemple C++) sont plus difficiles à utiliser mais aussi plus efficaces.

La plupart des fonctions fondamentales de R font appel à des fonctions compilées à partir d'un langage de plus bas niveau.

D'où le principe : **limiter au maximum la surcharge liée à R** pour retomber au plus vite sur des fonctions pré-compilées.

Remarque Il est très facile en pratique d'utiliser R comme une interface vers des langages de plus bas niveau, cf. *infra* à propos de Rcpp.

Utiliser les boucles avec parcimonie (1)

Comme la plupart des langages de programmation, R dispose de **structures de contrôles** permettant de réaliser des boucles.

```
boucle <- function(x){  
  cumul <- rep(NA, length(x))  
  for(i in seq_along(x))  
    cumul[i] <- if(i == 1) x[i] else cumul[i - 1] + x[i]  
  return(cumul)  
}  
boucle(1:5)  
## [1] 1 3 6 10 15
```

Utiliser les boucles avec parcimonie (1)

Comme la plupart des langages de programmation, R dispose de **structures de contrôles** permettant de réaliser des boucles.

```
boucle <- function(x){  
  cumul <- rep(NA, length(x))  
  for(i in seq_along(x))  
    cumul[i] <- if(i == 1) x[i] else cumul[i - 1] + x[i]  
  return(cumul)  
}  
boucle(1:5)  
## [1] 1 3 6 10 15
```

Ces opérations présentent plusieurs inconvénients :

1. Elles sont longues à écrire et assez peu claires ;
2. Elles reposent sur des effets de bord ;
3. Elles sont en général très peu **efficaces algorithmiquement**.

Utiliser les boucles avec parcimonie (2)

Les méthodes de vectorisation sont en général beaucoup plus efficaces que les boucles en R :

- ▶ vectorisation de haut niveau (*cf. supra*) ;
- ▶ vectorisation de bas niveau : la vectorisation est opérée par le langage de bas niveau auquel fait appel R.

Utiliser les boucles avec parcimonie (2)

Les méthodes de vectorisation sont en général beaucoup plus efficaces que les boucles en R :

- ▶ vectorisation de haut niveau (*cf. supra*) ;
- ▶ vectorisation de bas niveau : la vectorisation est opérée par le langage de bas niveau auquel fait appel R.

```
summary(microbenchmark(times = 10L
  , boucle = boucle(1:1e4)
  , Reduce = Reduce(`+`, 1:1e4, accumulate = TRUE)
  , cumsum = cumsum(1:1e4)
))[, 1:4]
```

##	expr	min	lq	mean
## 1	boucle	23045.637	23332.998	24138.5132
## 2	Reduce	6811.773	6955.059	7425.0581
## 3	cumsum	41.052	45.789	61.8935

Coder efficacement en base R

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

Coder efficacement en base R

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

- ▶ les langages **compilés** (C, C++) : l'ensemble du code est transformé en langage machine par un *compilateur* puis soumis par le système d'exploitation ;

Coder efficacement en base R

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

- ▶ les langages **compilés** (C, C++) : l'ensemble du code est transformé en langage machine par un *compilateur* puis soumis par le système d'exploitation ;
- ▶ les langages **interprétés** (R, Python) : les instructions du code sont soumises les unes après les autres par un *interpréteur*, ce qui est moins efficace (*cf.* boucles en R).

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

- ▶ les langages **compilés** (C, C++) : l'ensemble du code est transformé en langage machine par un *compilateur* puis soumis par le système d'exploitation ;
- ▶ les langages **interprétés** (R, Python) : les instructions du code sont soumises les unes après les autres par un *interpréteur*, ce qui est moins efficace (*cf.* boucles en R).

La fonction `compiler::cmpfun()` permet néanmoins de **compiler** des fonctions R avant utilisation.

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

- ▶ les langages **compilés** (C, C++) : l'ensemble du code est transformé en langage machine par un *compilateur* puis soumis par le système d'exploitation ;
- ▶ les langages **interprétés** (R, Python) : les instructions du code sont soumises les unes après les autres par un *interpréteur*, ce qui est moins efficace (cf. boucles en R).

La fonction `compiler::cmpfun()` permet néanmoins de **compiler** des fonctions R avant utilisation.

```
# Compilation de la fonction boucle()
```

```
boucle_compil <- compiler::cmpfun(boucle)
```

```
microbenchmark(boucle(1:1e4), boucle_compil(1:1e4))
```

```
## Unit: milliseconds
```

```
##           expr           min           lq           mean
```

```
##   boucle(1:10000) 21.755676 22.578677 23.45106
```

```
##  boucle_compil(1:10000)  1.582848  1.592914  1.61584
```

Coder efficacement en base R

Tirer le meilleur parti de la compilation (2)

Une autre fonctionnalité du *package* `compiler` est la compilation « juste-à-temps » (ou *just-in-time*, JIT) : le code n'est plus interprété mais **compilé au fur et à mesure**.

Coder efficacement en base R

Tirer le meilleur parti de la compilation (2)

Une autre fonctionnalité du *package* `compiler` est la compilation « juste-à-temps » (ou *just-in-time*, JIT) : le code n'est plus interprété mais **compilé au fur et à mesure**.

Dans R, on active le mode JIT pour une session grâce à la fonction `compiler::enableJIT()` en spécifiant le niveau de compilation JIT (de 0 à 3).

Tirer le meilleur parti de la compilation (2)

Une autre fonctionnalité du *package* `compiler` est la compilation « juste-à-temps » (ou *just-in-time*, JIT) : le code n'est plus interprété mais **compilé au fur et à mesure**.

Dans R, on active le mode JIT pour une session grâce à la fonction `compiler::enableJIT()` en spécifiant le niveau de compilation JIT (de 0 à 3).

```
# Passage au niveau maximal de compilation JIT
```

```
compiler::enableJIT(3)
```

```
## [1] 0
```

```
summary(microbenchmark(boucle(1:1e4), boucle_compil(1:1e4)))[, 1
```

```
##          expr          min          lq          mean
```

```
## 1      boucle(1:10000) 1.576533 1.584626 1.826098
```

```
## 2 boucle_compil(1:10000) 1.575349 1.585217 1.730038
```

Tirer le meilleur parti de la compilation (2)

Une autre fonctionnalité du *package* `compiler` est la compilation « juste-à-temps » (ou *just-in-time*, JIT) : le code n'est plus interprété mais **compilé au fur et à mesure**.

Dans R, on active le mode JIT pour une session grâce à la fonction `compiler::enableJIT()` en spécifiant le niveau de compilation JIT (de 0 à 3).

```
# Passage au niveau maximal de compilation JIT
```

```
compiler::enableJIT(3)
```

```
## [1] 0
```

```
summary(microbenchmark(boucle(1:1e4), boucle_compil(1:1e4)))[, 1]
```

```
##          expr          min          lq          mean
```

```
## 1      boucle(1:10000) 1.576533 1.584626 1.826098
```

```
## 2 boucle_compil(1:10000) 1.575349 1.585217 1.730038
```

Remarque Depuis R 3.4.0, `enableJIT()` vaut 3 par défaut.

Coder efficacement en base R

Utiliser l'opérateur [au lieu de ifelse()

Lorsqu'on crée une variable en faisant intervenir une condition, il est fréquent d'utiliser la fonction `ifelse()` :

```
notes <- runif(n = 100000, min = 0, max = 20)
mavar <- ifelse(notes >= 10, "Reçu", "Recalé")
```

Coder efficacement en base R

Utiliser l'opérateur [au lieu de ifelse()

Lorsqu'on crée une variable en faisant intervenir une condition, il est fréquent d'utiliser la fonction `ifelse()` :

```
notes <- runif(n = 100000, min = 0, max = 20)
mavar <- ifelse(notes >= 10, "Reçu", "Recalé")
```

Il est néanmoins beaucoup plus efficace d'utiliser l'opérateur `[`.

```
microbenchmark(times = 10L
  , ifelse = ifelse(notes >= 10, "Reçu", "Recalé")
  , "[" = {
    mavar <- rep("Recalé", length(notes))
    mavar[notes >= 10] <- "Reçu"
  }
)

## Unit: milliseconds
##      expr      min       lq      mean  median
##  ifelse 47.746751 50.191284 63.255584 51.28902
##      [   1.976785   2.041915   2.816405   2.45381
##      uq      max neval
```


Coder efficacement en base R

Simplifier les données : le type factor

On utilise souvent des chaînes de caractère pour coder une variable de nature catégorielle.

Le type `factor` permet de remplacer chaque valeur distincte par un entier en sauvegardant la table de correspondance. Il est **beaucoup plus léger**.

Simplifier les données : le type factor

On utilise souvent des chaînes de caractère pour coder une variable de nature catégorielle.

Le type factor permet de remplacer chaque valeur distincte par un entier en sauvegardant la table de correspondance. Il est **beaucoup plus léger**.

```
# Variable à deux modalités codées en caractères
sexe <- sample(c("H", "F"), 120000, replace = TRUE)
object_size(sexe)
## 960 kB

# Conversion en facteur
f.sexe <- factor(sexe)
str(f.sexe)
## Factor w/ 2 levels "F","H": 1 2 1 2 1 2 2 2 1 1 ...
object_size(f.sexe)
## 481 kB
```

Coder efficacement en base R

Utiliser les noms à bon escient (1)

La plupart des objets manipulés couramment dans R peuvent être **nommés** : vecteurs, matrices, listes, `data.frame`.

Utiliser des noms est une méthode souvent **très rapide** pour **accéder aux éléments** qui composent ces objets.

Utiliser les noms à bon escient (1)

La plupart des objets manipulés couramment dans R peuvent être **nommés** : vecteurs, matrices, listes, `data.frame`.

Utiliser des noms est une méthode souvent **très rapide** pour **accéder aux éléments** qui composent ces objets.

Exemple On cherche à extraire les observations d'une table *via* leur identifiant `id`. On compare l'utilisation des noms à une fusion réalisée avec `merge()`.

```
# Création de la table df
id <- as.character(sample(1e5))
sexe <- sample(1:2, 1e5, replace = TRUE)
df <- data.frame(id, sexe)
```

Coder efficacement en base R

Utiliser les noms à bon escient (2)

```
# Affectation de noms à df
row.names(df) <- id

# Liste des identifiants à extraire
extract <- c("234", "12", "7890")

# Comparaison
microbenchmark(times = 10L
  , merge = merge(data.frame(id = extract), df, sort = FALSE)
  , names = df[extract, ]
)

## Unit: milliseconds
##      expr      min       lq      mean     median
##  merge 18.789710 19.017861 20.620011 19.230421
##  names  3.080831  3.117935  3.848769  3.211682
##           uq      max neval
## 19.727183 26.619084    10
##  3.387532  9.399197    10
```

Coder efficacement en base R

À propos des matrices (1)

Quand c'est possible, **travailler sur des matrices** (plutôt que des `data.frame`) est souvent source d'efficacité :

Coder efficacement en base R

À propos des matrices (1)

Quand c'est possible, **travailler sur des matrices** (plutôt que des `data.frame`) est souvent source d'efficacité :

- ▶ de nombreuses opérations sont **vectorisées** pour les matrices : sommes en lignes et en colonnes (`rowSums()` et `colSums()`), etc. ;

Coder efficacement en base R

À propos des matrices (1)

Quand c'est possible, **travailler sur des matrices** (plutôt que des `data.frame`) est souvent source d'efficacité :

- ▶ de nombreuses opérations sont **vectorisées** pour les matrices : sommes en lignes et en colonnes (`rowSums()` et `colSums()`), etc. ;
- ▶ l'**algèbre matricielle** (le produit matriciel notamment) est très bien optimisée ;

Coder efficacement en base R

À propos des matrices (1)

Quand c'est possible, **travailler sur des matrices** (plutôt que des `data.frame`) est souvent source d'efficacité :

- ▶ de nombreuses opérations sont **vectorisées** pour les matrices : sommes en lignes et en colonnes (`rowSums()` et `colSums()`), etc. ;
- ▶ l'**algèbre matricielle** (le produit matriciel notamment) est très bien optimisée ;
- ▶ selon la nature du problème, l'utilisation de **matrices lacunaires** (*sparse*) peut faire gagner en empreinte mémoire et en temps de calcul (*cf.* le *package* `Matrix`).

Coder efficacement en base R

À propos des matrices (2)

```
# Création d'une matrice m avec 99 % de 0
v <- rep(0, 1e6); v[sample(1e6, 1e4)] <- rnorm(1e4)
m <- matrix(v, ncol = 100)

# Transformation en matrice lacunaire
library(Matrix)
M <- Matrix(m)

# Gain en espace (en ko)
c(object_size(m), object_size(M))
## [1] 8000200 121824

# Gain de performances pour la fonction colSums()
microbenchmark(dense = colSums(m), sparse = colSums(M))
## Unit: microseconds
##      expr      min       lq      mean    median      uq
##  dense 1288.384 1294.700 1362.1935 1312.067 1323.514
##  sparse   69.472   71.051  111.5303  107.366  109.340
##           max neval
```

dplyr : une grammaire du traitement des données

Philosophie de dplyr

dplyr est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

dplyr : une grammaire du traitement des données

Philosophie de dplyr

dplyr est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

- ▶ il fournit un ensemble de **fonctions élémentaires** (les « verbes ») pour effectuer les manipulations de données ;

dplyr : une grammaire du traitement des données

Philosophie de dplyr

dplyr est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

- ▶ il fournit un ensemble de **fonctions élémentaires** (les « verbes ») pour effectuer les manipulations de données ;
- ▶ plusieurs verbes peuvent facilement être **combinés en utilisant l'opérateur %>%** (*pipe*) ;

dplyr : une grammaire du traitement des données

Philosophie de dplyr

dplyr est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

- ▶ il fournit un ensemble de **fonctions élémentaires** (les « verbes ») pour effectuer les manipulations de données ;
- ▶ plusieurs verbes peuvent facilement être **combinés en utilisant l'opérateur %>%** (*pipe*) ;
- ▶ toutes les opérations sont optimisées par du **code de bas niveau**.

```
library(dplyr)
```

dplyr : une grammaire du traitement des données

Philosophie de dplyr

dplyr est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

- ▶ il fournit un ensemble de **fonctions élémentaires** (les « verbes ») pour effectuer les manipulations de données ;
- ▶ plusieurs verbes peuvent facilement être **combinés en utilisant l'opérateur %>%** (*pipe*) ;
- ▶ toutes les opérations sont optimisées par du **code de bas niveau**.

```
library(dplyr)
```

Pour en savoir plus De nombreuses vignettes très pédagogiques sont disponibles sur la page du package. Un aide-mémoire est également disponible sur le site de RStudio.

dplyr : une grammaire du traitement des données

Données d'exemple : table `flights` de `nycflights13`

Les exemples relatifs aux *packages* `dplyr` et `data.table` s'appuient sur les données du *package* `nycflights13`.

```
library(nycflights13)
```


dplyr : une grammaire du traitement des données

Données d'exemple : table `flights` de `nycflights13`

Les exemples relatifs aux *packages* `dplyr` et `data.table` s'appuient sur les données du *package* `nycflights13`.

```
library(nycflights13)
```

Ce *package* contient des données sur tous les vols au départ de la ville de New-York en 2013.

```
data(package = "nycflights13")
```

```
dim(flights)
```

```
## [1] 336776      19
```

```
names(flights)[1:9]
```

```
## [1] "year"           "month"          "day"
## [4] "dep_time"       "sched_dep_time" "dep_delay"
## [7] "arr_time"       "sched_arr_time" "arr_delay"
```

dplyr : une grammaire du traitement des données

Simplifier des opérations de base R

dplyr propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

dplyr : une grammaire du traitement des données

Simplifier des opérations de base R

dplyr propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

- ▶ `filter()` sélectionne des observations selon une ou plusieurs conditions :

```
filter(flights, month == 7, day == 4)
```

dplyr : une grammaire du traitement des données

Simplifier des opérations de base R

dplyr propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

- ▶ `filter()` sélectionne des observations selon une ou plusieurs conditions :

```
filter(flights, month == 7, day == 4)
```

- ▶ `arrange()` trie le fichier selon une ou plusieurs variables :

```
arrange(flights, month, desc(distance))
```

dplyr : une grammaire du traitement des données

Simplifier des opérations de base R

dplyr propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

- ▶ `filter()` sélectionne des observations selon une ou plusieurs conditions :

```
filter(flights, month == 7, day == 4)
```

- ▶ `arrange()` trie le fichier selon une ou plusieurs variables :

```
arrange(flights, month, desc(distance))
```

- ▶ `select()` sélectionne des variables par leur noms :

```
select(flights, year:arr_delay)
```

dplyr : une grammaire du traitement des données

Simplifier des opérations de base R

dplyr propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

- ▶ `filter()` sélectionne des observations selon une ou plusieurs conditions :

```
filter(flights, month == 7, day == 4)
```

- ▶ `arrange()` trie le fichier selon une ou plusieurs variables :

```
arrange(flights, month, desc(distance))
```

- ▶ `select()` sélectionne des variables par leur noms :

```
select(flights, year:arr_delay)
```

- ▶ `rename()` renomme des variables.

```
rename(flights, annee = year)
```

dplyr : une grammaire du traitement des données

Calculer des statistiques avec `summarise()`

La fonction `summarise()` permet de facilement calculer des statistiques sur des données.

dplyr : une grammaire du traitement des données

Calculer des statistiques avec summarise()

La fonction summarise() permet de facilement calculer des statistiques sur des données.

```
summarise(flights  
  , distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)
```

```
##    distance_moyenne retard_max  
## 1           1039.913       1272
```


dplyr : une grammaire du traitement des données

Calculer des statistiques avec summarise()

La fonction summarise() permet de facilement calculer des statistiques sur des données.

```
summarise(flights  
  , distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)
```

```
##    distance_moyenne retard_max  
## 1           1039.913       1272
```

Remarque Comme toutes les fonctions de dplyr, summarise() prend un data.frame en entrée et produit un data.frame en sortie.

dplyr : une grammaire du traitement des données

Ventiler des traitements avec group_by()

Appliqué au préalable à un data.frame, group_by() ventile tous les traitements ultérieurs selon les modalités d'une ou plusieurs variables.

```
flights_bymonth <- group_by(flights, month)
summarise(flights_bymonth
  , distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
)[1:3, ]
```

##	month	distance_moyenne	retard_max
## 1	1	1006.844	1272
## 2	2	1000.982	834
## 3	3	1011.987	915

dplyr : une grammaire du traitement des données

Enchaîner des opérations avec %>%

L'utilisation des verbes de dplyr ne prend tout son intérêt que quand ils sont enchaînés en utilisant l'opérateur *pipe* %>%.

`maTable %>% maFonction(param1, param2)` est équivalent à `maFonction(maTable, param1, param2)`.

dplyr : une grammaire du traitement des données

Enchaîner des opérations avec %>%

L'utilisation des verbes de dplyr ne prend tout son intérêt que quand ils sont enchaînés en utilisant l'opérateur *pipe* %>%.

`maTable %>% maFonction(param1, param2)` est équivalent à `maFonction(maTable, param1, param2)`.

Ainsi, l'**enchaînement de nombreuses opérations** devient beaucoup plus facile à mettre en œuvre et à comprendre.

dplyr : une grammaire du traitement des données

Enchaîner des opérations avec %>%

L'utilisation des verbes de dplyr ne prend tout son intérêt que quand ils sont enchaînés en utilisant l'opérateur *pipe* %>%.

maTable %>% maFonction(param1, param2) est équivalent à maFonction(maTable, param1, param2).

Ainsi, l'**enchaînement de nombreuses opérations** devient beaucoup plus facile à mettre en œuvre et à comprendre.

```
flights %>%  
  group_by(year, month, day) %>%  
  summarise(  
    retard_arrivee = mean(arr_delay, na.rm = TRUE),  
    retard_depart = mean(dep_delay, na.rm = TRUE)  
  ) %>%  
  filter(retard_arrivee > 30 | retard_depart > 30)
```

dplyr : une grammaire du traitement des données

Fusionner des tables avec `*_join()`

dplyr dispose de nombreuses fonctions très utiles pour fusionner une ou plusieurs tables ensemble, qui **s'inspirent très fortement de SQL** :

- ▶ `a %>% left_join(b, by = "id")` : fusionne a et b en conservant toutes les observations de a ;
- ▶ `a %>% right_join(b, by = "id")` : fusionne a et b en conservant toutes les observations de b ;
- ▶ `a %>% inner_join(b, by = "id")` : fusionne a et b en ne conservant que les observations dans a et b ;
- ▶ `a %>% full_join(b, by = "id")` : fusionne a et b en conservant toutes les observations.

Pour en savoir plus Une vignette est consacrée à la présentation des fonctions de dplyr portant sur deux tables.

dplyr : une grammaire du traitement des données

Comparaison de base R et de dplyr

dplyr est particulièrement intéressant pour travailler sur des données par groupe. On compare donc l'utilisation de `tapply()` de base R avec `group_by()` de dplyr.

dplyr : une grammaire du traitement des données

Comparaison de base R et de dplyr

dplyr est particulièrement intéressant pour travailler sur des données par groupe. On compare donc l'utilisation de `tapply()` de base R avec `group_by()` de dplyr.

```
df <- data.frame(  
  x = rnorm(1e6)  
  , by = sample(1e3, 1e6, replace = TRUE)  
)  
  
microbenchmark(times = 10L  
  , base = tapply(df$x, df$by, sum)  
  , dplyr = df %>% group_by(by) %>% summarise(sum(x))  
)  
  
## Note: no visible binding for global variable 'x'  
## Note: no visible binding for global variable 'x'  
## Note: no visible binding for global variable 'x'  
## Note: no visible binding for global variable 'x'  
## Note: no visible binding for global variable 'x'  
## Note: no visible binding for global variable 'x'
```


`data.table` : un `data.frame` optimisé

Philosophie de `data.table`

Contrairement à `dplyr`, `data.table` ne cherche pas à se substituer à base R mais à le compléter.

`data.table` : un `data.frame` optimisé

Philosophie de `data.table`

Contrairement à `dplyr`, `data.table` ne cherche pas à se substituer à base R mais à le compléter.

Il introduit un nouveau type d'objet, le `data.table`, qui **hérite** du `data.frame` (tout `data.table` est un `data.frame`).

data.table : un data.frame optimisé

Philosophie de data.table

Contrairement à dplyr, data.table ne cherche pas à se substituer à base R mais à le compléter.

Il introduit un nouveau type d'objet, le data.table, qui **hérite** du data.frame (tout data.table est un data.frame).

Appliqué à un data.table, l'opérateur [est **enrichi et optimisé**.

```
library(data.table)
flights_DT <- data.table(flights)
```

data.table : un data.frame optimisé

Philosophie de data.table

Contrairement à dplyr, data.table ne cherche pas à se substituer à base R mais à le compléter.

Il introduit un nouveau type d'objet, le data.table, qui **hérite** du data.frame (tout data.table est un data.frame).

Appliqué à un data.table, l'opérateur [est **enrichi et optimisé**.

```
library(data.table)
flights_DT <- data.table(flights)
```

Pour en savoir plus Là encore des vignettes très pédagogiques sont disponibles sur la [page du package](#).

`data.table` : un `data.frame` optimisé

L'opérateur `[` du `data.table` : `i`, `j` et `by`

La syntaxe de l'opérateur `[` appliqué à un `data.table` est la suivante (DT représente le `data.table`) :

data.table : un data.frame optimisé

L'opérateur [du data.table : i, j et by

La syntaxe de l'opérateur [appliqué à un data.table est la suivante (DT représente le data.table) :

DT[i, j, by]

- ▶ i : sélectionner des observations selon une condition ;
- ▶ j : sélectionner ou **créer** une ou plusieurs variables ;
- ▶ by : ventiler les traitements selon les modalités d'une ou plusieurs variables.

data.table : un data.frame optimisé

L'opérateur [du data.table : i, j et by

La syntaxe de l'opérateur [appliqué à un data.table est la suivante (DT représente le data.table) :

DT[i, j, by]

- ▶ i : sélectionner des observations selon une condition ;
- ▶ j : sélectionner ou **créer** une ou plusieurs variables ;
- ▶ by : ventiler les traitements selon les modalités d'une ou plusieurs variables.

data.table : un data.frame optimisé

L'opérateur [du data.table : i, j et by

La syntaxe de l'opérateur [appliqué à un data.table est la suivante (DT représente le data.table) :

DT[i, j, by]

- ▶ i : sélectionner des observations selon une condition ;
- ▶ j : sélectionner ou **créer** une ou plusieurs variables ;
- ▶ by : ventiler les traitements selon les modalités d'une ou plusieurs variables.

Exemple Retard quotidien maximal au mois de janvier.

```
flights_DT[  
  month == 1, max(arr_delay, na.rm = TRUE), by = day  
]
```


`data.table` : un `data.frame` optimisé

Sélectionner des observations avec `i`

Il est beaucoup plus simple et efficace de sélectionner des observations dans un `data.table` que dans un `data.frame` :

data.table : un data.frame optimisé

Sélectionner des observations avec i

Il est beaucoup plus simple et efficace de sélectionner des observations dans un data.table que dans un data.frame :

- ▶ il n'y a pas à répéter le nom du data.frame dans [;

data.table : un data.frame optimisé

Sélectionner des observations avec i

Il est beaucoup plus simple et efficace de sélectionner des observations dans un data.table que dans un data.frame :

- ▶ il n'y a pas à répéter le nom du data.frame dans [;
- ▶ il est possible d'indexer un data.table par une ou plusieurs « clés » permettant une recherche souvent plus rapide.

```
setkey(flights_DT, origin)
microbenchmark(times = 100L
  , base = flights[flights$origin == "JFK",]
  , dt1 = flights_DT[origin == "JFK"]
  , dt2 = flights_DT[list("JFK")]
)
## Unit: milliseconds
##  expr      min       lq     mean  median      uq
##  base 55.48376 57.77632 69.19374 67.23236 69.21842
##  dt1  17.67461 18.04190 22.23228 18.29374 18.86352
##  dt2  16.80345 17.39613 22.39986 17.70816 25.35833
```

data.table : un data.frame optimisé

Calculer des statistiques avec j

L'argument j permet de calculer des statistiques agrégées.

data.table : un data.frame optimisé

Calculer des statistiques avec j

L'argument j permet de calculer des statistiques agrégées.

```
flights_DT[, j = list(  
  distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)]  
##      distance_moyenne retard_max  
## 1:           1039.913         1272
```

data.table : un data.frame optimisé

Calculer des statistiques avec j

L'argument j permet de calculer des statistiques agrégées.

```
flights_DT[, j = list(  
  distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)]  
  
##      distance_moyenne retard_max  
## 1:           1039.913         1272
```

Utilisé avec := il permet de les refusionner automatiquement avec les données d'origine.

data.table : un data.frame optimisé

Calculer des statistiques avec j

L'argument j permet de calculer des statistiques agrégées.

```
flights_DT[, j = list(  
  distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)]  
  
##      distance_moyenne retard_max  
## 1:           1039.913         1272
```

Utilisé avec := il permet de les refusionner automatiquement avec les données d'origine.

```
flights_DT <- flights_DT[, j := list(  
  distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
)]
```

data.table : un data.frame optimisé

Ventiler des traitements avec by et keyby

L'argument by de [ventile tous les traitements renseignés dans j selon les modalités d'une ou plusieurs variables.

data.table : un data.frame optimisé

Ventiler des traitements avec by et keyby

L'argument by de [ventile tous les traitements renseignés dans j selon les modalités d'une ou plusieurs variables.

```
flights_DT[, j = list(  
  distance_moyenne = mean(distance)  
  , retard_max = max(arr_delay, na.rm = TRUE)  
) , by = month][1:3,]  
##      month distance_moyenne retard_max  
## 1:      1          1006.844         1272  
## 2:     10          1038.876          688  
## 3:     11          1050.305          796
```

data.table : un data.frame optimisé

Ventiler des traitements avec by et keyby

L'argument by de [ventile tous les traitements renseignés dans j selon les modalités d'une ou plusieurs variables.

```
flights_DT[, j = list(
  distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
), by = month][1:3,]
##      month distance_moyenne retard_max
## 1:      1          1006.844         1272
## 2:     10          1038.876          688
## 3:     11          1050.305          796
```

Remarque Par défaut, by ordonne les résultats dans l'ordre des groupes dans le data.table. keyby trie les données selon la variable d'agrégation (comme group_by de dplyr).

data.table : un data.frame optimisé

Chaîner les opérations dans un data.table

Il est très facile de chaîner les opérations sur un data.table en enchaînant les [.

data.table : un data.frame optimisé

Chaîner les opérations dans un data.table

Il est très facile de chaîner les opérations sur un data.table en enchaînant les [.

```
flights_DT[
  , j = list(
    retard_arrivee = mean(arr_delay, na.rm = TRUE)
    , retard_depart = mean(dep_delay, na.rm = TRUE)
  )
  , keyby = list(year, month, day)
][retard_arrivee > 30 | retard_depart > 30]
```

data.table : un data.frame optimisé

Chaîner les opérations dans un data.table

Il est très facile de chaîner les opérations sur un data.table en enchaînant les [.

```
flights_DT[
  , j = list(
    retard_arrivee = mean(arr_delay, na.rm = TRUE)
    , retard_depart = mean(dep_delay, na.rm = TRUE)
  )
  , keyby = list(year, month, day)
][retard_arrivee > 30 | retard_depart > 30]
```

Remarque Ces chaînages sont possibles avec un data.table mais pas avec un data.frame.

data.table : un data.frame optimisé

Comparaison de base R, dplyr et data.table

```
# Conversion de la table de test en data.table  
dt <- data.table(df)
```

```
microbenchmark(times = 10L  
  , base = tapply(df$x, df$by, sum)  
  , dplyr = df %>% group_by(by) %>% summarise(sum(x))  
  , data.table = dt[, sum(x), keyby = by]  
)
```

data.table : un data.frame optimisé

Comparaison de base R, dplyr et data.table

```
# Conversion de la table de test en data.table
```

```
dt <- data.table(df)
```

```
microbenchmark(times = 10L
```

```
  , base = tapply(df$x, df$by, sum)
```

```
  , dplyr = df %>% group_by(by) %>% summarise(sum(x))
```

```
  , data.table = dt[, sum(x), keyby = by]
```

```
)
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
##      expr      lg      mean      ug
```

data.table : un data.frame optimisé

Comparaison de base R, dplyr et data.table

```
# Conversion de la table de test en data.table
```

```
dt <- data.table(df)
```

```
microbenchmark(times = 10L
```

```
  , base = tapply(df$x, df$by, sum)
```

```
  , dplyr = df %>% group_by(by) %>% summarise(sum(x))
```

```
  , data.table = dt[, sum(x), keyby = by]
```

```
)
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
## Note: no visible binding for global variable 'x'
```

```
##      expr      lg      mean      ug
```


data.table : un data.frame optimisé

Parenthèse : lire et écrire rapidement des fichiers plats avec data.table

Le *package* data.table fournit par ailleurs deux fonctions extrêmement utiles en pratique pour **lire et écrire** des fichiers plats (.csv, .txt, .d1m) :

data.table : un data.frame optimisé

Parenthèse : lire et écrire rapidement des fichiers plats avec data.table

Le *package* data.table fournit par ailleurs deux fonctions extrêmement utiles en pratique pour **lire et écrire** des fichiers plats (.csv, .txt, .d1m) :

- ▶ fread() : beaucoup plus rapide et plus ergonomique que read.table() et ses alias (read.csv(), read.delim()), produit directement un data.table ;

data.table : un data.frame optimisé

Parenthèse : lire et écrire rapidement des fichiers plats avec data.table

Le *package* data.table fournit par ailleurs deux fonctions extrêmement utiles en pratique pour **lire et écrire** des fichiers plats (.csv, .txt, .d1m) :

- ▶ fread() : beaucoup plus rapide et plus ergonomique que read.table() et ses alias (read.csv(), read.delim()), produit directement un data.table ;
- ▶ fwrite() : beaucoup plus rapide que write.table().

data.table : un data.frame optimisé

Parenthèse : lire et écrire rapidement des fichiers plats avec data.table

Le *package* data.table fournit par ailleurs deux fonctions extrêmement utiles en pratique pour **lire et écrire** des fichiers plats (.csv, .txt, .dml) :

- ▶ fread() : beaucoup plus rapide et plus ergonomique que read.table() et ses alias (read.csv(), read.delim()), produit directement un data.table;
- ▶ fwrite() : beaucoup plus rapide que write.table().

```
# Lecture avec fread()
```

```
mon_fichier <- fread(file = "mon_fichier.csv", sep = ",")
```

```
# Ecriture avec fwrite()
```

```
fwrite(mon_fichier, file = "mon_fichier.txt", sep = "\t")
```

Aller plus loin avec R

Les limites du logiciel

Les outils présentés jusqu'à présent correspondent à une utilisation « classique » de R : production d'une enquête, redressements, études.

Les limites du logiciel

Les outils présentés jusqu'à présent correspondent à une utilisation « classique » de R : production d'une enquête, redressements, études.

Il arrive néanmoins que certains traitements soient rendus **difficiles par les caractéristiques du logiciel** :

- ▶ travail sur des volumes de données impossibles à loger en mémoire ;
- ▶ temps de calcul trop longs et impossibles à réduire.

Les limites du logiciel

Les outils présentés jusqu'à présent correspondent à une utilisation « classique » de R : production d'une enquête, redressements, études.

Il arrive néanmoins que certains traitements soient rendus **difficiles par les caractéristiques du logiciel** :

- ▶ travail sur des volumes de données impossibles à loger en mémoire ;
- ▶ temps de calcul trop longs et impossibles à réduire.

Dans ce genre de situations, la solution consiste en général à utiliser R comme une **interface** vers des techniques ou langages susceptibles de répondre au problème posé.

Aller plus loin avec R

Se connecter à des bases de données

Une autre solution pour exploiter de grands volumes de données dans R est de l'utiliser pour **interroger des bases de données**, *via* par exemple le *package* RPostgreSQL.

Aller plus loin avec R

Se connecter à des bases de données

Une autre solution pour exploiter de grands volumes de données dans R est de l'utiliser pour **interroger des bases de données**, *via* par exemple le *package* RPostgreSQL.

```
library(RPostgreSQL)

# Connexion à la base de données maBdd
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "maBdd"
  , host = "localhost", port = 5432
  , user = "utilisateur", password = "motDePasse"
)

# Requête SQL sur la table maTable
dbGetQuery(con, "SELECT COUNT(*) FROM maTable")
```

Aller plus loin avec R

Se connecter à des bases de données

Une autre solution pour exploiter de grands volumes de données dans R est de l'utiliser pour **interroger des bases de données**, *via* par exemple le *package* RPostgreSQL.

```
library(RPostgreSQL)

# Connexion à la base de données maBdd
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "maBdd"
  , host = "localhost", port = 5432
  , user = "utilisateur", password = "motDePasse"
)

# Requête SQL sur la table maTable
dbGetQuery(con, "SELECT COUNT(*) FROM maTable")
```

Remarque Différents *packages* permettent de se connecter à différents types de base de données : RMySQL pour MySQL, etc.

Aller plus loin avec R

Se connecter à des bases de données avec dplyr

dplyr a la particularité de pouvoir fonctionner de façon totalement transparente sur des bases de données de différents types.

Aller plus loin avec R

Se connecter à des bases de données avec dplyr

dplyr a la particularité de pouvoir fonctionner de façon totalement transparente sur des bases de données de différents types.

```
library(dplyr)

# Connexion à la base de données maBdd
con <- src_postgres(
  dbname = "maBdd", host = "localhost", port = 5432
  , user = "utilisateur", password = "motDePasse"
)

# Requête SQL sur la table maTable...
tbl(con, "SELECT COUNT(*) FROM maTable")

# ... ou utilisation des verbes de dplyr
tbl(con) %>% summarise(n())
```

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'exploite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'exploite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Ce type d'opérations est composé de plusieurs étapes :

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'exploite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Ce type d'opérations est composé de plusieurs étapes :

1. Création et paramétrage du « *cluster* » de cœurs à utiliser (chargement des fonctions et *packages* nécessaires sur chaque cœur) ;

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'exploite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Ce type d'opérations est composé de plusieurs étapes :

1. Création et paramétrage du « *cluster* » de cœurs à utiliser (chargement des fonctions et *packages* nécessaires sur chaque cœur) ;
2. Lancement du traitement parallélisé avec `parLapply()` ;

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'exploite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Ce type d'opérations est composé de plusieurs étapes :

1. Création et paramétrage du « *cluster* » de cœurs à utiliser (chargement des fonctions et *packages* nécessaires sur chaque cœur) ;
2. Lancement du traitement parallélisé avec `parLapply()` ;
3. Arrêt des processus du *cluster* avec `stopCluster()`.

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (2)

Dans cet exemple, on cherche à appliquer la fonction f à chaque matrice de la liste `l`.

Aller plus loin avec R

Paralléliser des traitements avec parallel (2)

Dans cet exemple, on cherche à appliquer la fonction `f` à chaque matrice de la liste `l`.

```
library(MASS)
f <- function(x) rowSums(ginv(x))
l <- lapply(1:100, function(x) matrix(runif(1e4), ncol = 1e2))

# Création et paramétrage du cluster
library(parallel)
cl <- makeCluster(4)
clusterEvalQ(cl, library(MASS))
clusterExport(cl, "f")

# Lancement du calcul parallélisé
parLapply(cl, l, f)

# Arrêt des processus du cluster
stopCluster(cl)
```

Aller plus loin avec R

Paralléliser des traitements avec `parallel` (3)

Aller plus loin avec R

Paralléliser des traitements avec parallel (3)

```
microbenchmark(times = 10
  , lapply(1, f)
  , parLapply(cl, 1, f)
)
## Unit: milliseconds
##              expr      min       lq      mean
##      lapply(1, f) 767.2479 767.8207 782.6532
## parLapply(cl, 1, f) 455.8512 473.0162 518.5828
##      median      uq      max neval
## 769.1353 782.2881 867.1639     10
## 515.9757 540.7917 611.7799     10
```

Aller plus loin avec R

Paralléliser des traitements avec parallel (3)

```
microbenchmark(times = 10
  , lapply(1, f)
  , parLapply(cl, 1, f)
)
## Unit: milliseconds
##              expr      min       lq      mean
##      lapply(1, f) 767.2479 767.8207 782.6532
## parLapply(cl, 1, f) 455.8512 473.0162 518.5828
##      median      uq      max neval
## 769.1353 782.2881 867.1639     10
## 515.9757 540.7917 611.7799     10
```

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (1)

Le *package* Rcpp permet d'intégrer facilement des fonctions codées en C++ dans un programme R.

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (1)

Le *package* Rcpp permet d'intégrer facilement des fonctions codées en C++ dans un programme R.

```
library(Rcpp)
cppFunction('int add(int x, int y) {
  int result = x + y;
  return result;
}')
```



```
add(1, 2)
## [1] 3
```

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (1)

Le *package* Rcpp permet d'intégrer facilement des fonctions codées en C++ dans un programme R.

```
library(Rcpp)
cppFunction('int add(int x, int y) {
  int result = x + y;
  return result;
}')
```



```
add(1, 2)
## [1] 3
```

Remarque Il est également possible de soumettre un fichier contenant des fonctions C++ écrit par ailleurs à l'aide de la fonction `sourceCpp()`.

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (1)

Le *package* Rcpp permet d'intégrer facilement des fonctions codées en C++ dans un programme R.

```
library(Rcpp)
cppFunction('int add(int x, int y) {
  int result = x + y;
  return result;
}')
```



```
add(1, 2)
## [1] 3
```

Remarque Il est également possible de soumettre un fichier contenant des fonctions C++ écrit par ailleurs à l'aide de la fonction `sourceCpp()`.

Pour en savoir plus *Advanced R*

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (2)

Contrairement à R, C++ est un langage de bas niveau : les boucles y sont en particulier extrêmement rapides.

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (2)

Contrairement à R, C++ est un langage de bas niveau : les boucles y sont en particulier extrêmement rapides.

Exemple Somme cumulée par colonne

```
# Fonction C++
cppFunction('NumericMatrix cumColSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericMatrix out(nrow, ncol);
  for (int j = 0; j < ncol; j++) {
    double acc = 0;
    for(int i = 0; i < nrow; i++){
      acc += x(i, j);
      out(i, j) = acc;
    }
  }
  return out;
}')
```

Aller plus loin avec R

Rcpp : un package R pour utiliser C++ (3)

```
# Fonction R
cumColSumsR <- function(x){
  apply(x, 2, cumsum)
}

# Les deux fonctions produisent les mêmes résultats...
x <- matrix(rnorm(1e6), ncol = 1e2)
all.equal(cumColSumsR(x), cumColSumsC(x))
## [1] TRUE

# ... mais cumColSumsC() est beaucoup plus rapide !
summary(microbenchmark(times = 10
  , cumColSumsR(x)
  , cumColSumsC(x)
))[, c("expr", "lq", "mean", "uq")]
##           expr           lq       mean       uq
## 1 cumColSumsR(x) 23.928243 26.555889 28.740733
## 2 cumColSumsC(x)  4.380266  5.562942  4.626575
```