Optimisation de temps d'exécution en R

Sophie Baillargeon, Université Laval

2019-04-01

Table des matières

Lorsque nous écrivons du code, notre but premier est évidemment que ce code fonctionne correctement. Après avoir vérifié que le code produit le résultat escompté et gère correctement les exceptions, nous devrions, selon les bonnes pratiques, nous assurer que le code est facile à maintenir (écriture, compréhension et réutilisation aisées). Si notre code rencontre toutes les conditions énumérées ci-dessus, mais que son temps d'exécution est long, nous devrions envisager de le rendre plus rapide. Cette amélioration pourrait faire la différence entre un programme peu ou pas utilisé et un programme fréquemment utilisé. Cependant, n'oublions pas qu'il est inutile d'optimiser le temps d'exécution d'un programme roulant déjà suffisamment rapidement.

Pour réduire le temps d'exécution d'un programme, il faut d'abord cerner la partie du programme responsable des lenteurs. Pour ce faire, il est conseillé d'utiliser des outils qui analysent la performance du code. Après avoir cerné les instructions problématiques, il faut les modifier de façon à effectuer le calcul plus rapidement. Certains outils d'analyse de performance sont présentés dans ce qui suit. Ensuite, nous verrons des stratégies d'optimisation de temps d'exécution.

Outils d'analyse de la performance d'un programme R

Une analyse de la performance d'un programme informatique est appelée profilage de code (en anglais *code profiling*). Il est possible de profiler l'utilisation du processeur et l'utilisation de la mémoire. Nous nous concentrerons ici sur le profilage de l'utilisation du processeur, qui vise principalement à évaluer le temps d'exécution d'un programme. Le R de base propose deux outils pour évaluer le temps d'exécution d'un programme R :

- la fonction `system.time`,
- les fonctions `Rprof` et `summaryRprof`.

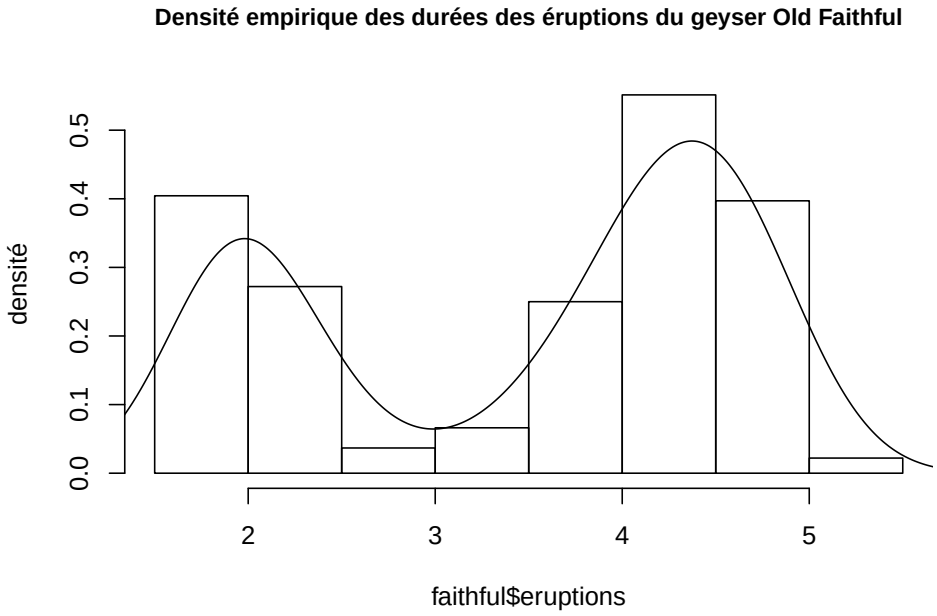
Cependant, ces fonctions ne sont parfois pas suffisantes. Les packages suivants nous seront aussi utiles :

- `microbenchmark` et
- `profvis`.

Pour illustrer l'utilisation de ces fonctions, nous allons utiliser un exemple tiré de Peng et de Leeuw, (2002). Nous allons étudier les temps d'exécution de fonctions R ayant pour but d'estimer la fonction de densité d'une variable aléatoire par la méthode du noyau à partir d'observations de la variable aléatoire. De l'information sur cette méthode, appelée en anglais *Kernel density estimation*, peut être trouvée sur la page Wikipédia suivante : https://fr.wikipedia.org/wiki/Estimation_par_noyau.

Il existe en fait déjà une fonction dans le package `stats` pour faire de l'estimation de densité par noyau. Il s'agit de la fonction `density`. Voici un exemple de ce qu'il est possible de réaliser avec cette fonction.

```
dens <- density(faithful$eruptions)  # faithful est un jeu de données du package datasets
hist(faithful$eruptions, freq = FALSE, ylab = "densité", cex.main = 0.9,
     main = "Densité empirique des durées des éruptions du geyser Old Faithful")
lines(dens)
```



Un histogramme est aussi une méthode d'estimation de densité. Ici, nous avons superposé une courbe de densité estimée par la méthode du noyau (aussi appelée densité Kernel) à un histogramme.

Nous allons écrire une version moins puissante de la fonction `density`. L'estimation de densité par noyau au point x se fait par la formule suivante :

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

où x_i pour $i = 1, 2, \ldots, n$ sont les observations, K est une fonction noyau (en anglais *kernel*) à définir et h est un paramètre de lissage (parfois appelée fenêtre). Plus la valeur de h est grande, plus la courbe obtenue est lisse.

La fonction `density` permet l'utilisation de plusieurs fonctions noyau via l'argument `kernel`. Nous allons plutôt nous restreindre au noyau gaussien, qui est en fait la fonction de densité d'une distribution normale standard. Nous allons donc utiliser la fonction `dnorm` pour évaluer la fonction K dans la formule ci-dessus.

Voici la première fonction proposée.

```
#' Estimation de densité par noyau gaussien
#'
#' Version 1 : utilisation de 2 boucles imbriquées
#'
#' @param x vecteur numérique contenant les observations
#' @param xpts vecteur numérique contenant les points en lesquels l'estimation de la
#'             densité doit être effectuée
#' @param h nombre réel > 0 : la valeur du paramètre de lissage
#'
#' @return vecteur numérique contenant la densité estimée en tous les points de xpts
#'
ksmooth1 <- function(x, xpts, h)
{
  dens <- double(length(xpts))
  n <- length(x)
  for(i in 1:length(xpts)) {
    ksum <- 0
    for(j in 1:length(x)) {
      d <- xpts[i] - x[j]
      ksum <- ksum + dnorm(d / h)
    }
    dens[i] <- ksum / (n * h)
  }
  dens
}
```

Des tags `roxygen2` sont utilisés pour documenter la fonction. Cependant, les commentaires `roxygen2` écrits ici ne formeraient pas une fiche d'aide complète. Ce n'est pas un problème puisque nous n'avons pas l'intention de créer un package avec cette fonction.

Dans la fonction `ksmooth1`, le premier argument, nommé `x`, n'est pas équivalent au x de la formule. Le x de la formule représente un point en lequel nous souhaitons faire l'estimation. Son équivalent dans la fonction `ksmooth1` est donc un élément du vecteur `xpts`. Ce sont les x_i de la formule que nous retrouvons dans le vecteur `x`. Dans la boucle, ce vecteur `x` est parcouru en utilisant l'indice `j`. Alors, en fait, `x[j]` dans le corps de la fonction représente un x_i dans la formule. Le code aurait pu coller davantage à la notation dans la formule pour être encore plus clair, mais j'ai choisi de le conserver tel qu'il a été proposé dans Peng et de Leeuw, (2002).

Fonction `system.time`

Si nous donnons en entrée à la fonction `ksmooth1` un vecteur d'observations `x` de longueur 10000, mesurons combien de temps prendra l'évaluation d'un appel à la fonction. Pour ce faire, utilisons d'abord la fonction `system.time`.

```
# Simulation des observations
x <- rnorm(10000)
# Points pour lesquels nous souhaitons estimer la densité
xpts <- seq(from = -4, to = 4, length.out = 17)
xpts
```

```
##  [1] -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0
```

```r
# Estimation de la densité
temps1 <- system.time(dens1 <- ksmooth1(x = x, xpts = xpts, h = 1))
```

Il faut donner en entrée à `system.time` une instruction R, contenant ou non une affectation, ou encore des instructions R encadrées d'accolades. Cet ensemble d'instruction(s) est appelé *expression*. La fonction retourne le temps, en secondes, d'utilisation du CPU (*Central Processing Unit*) pour l'évaluation de l'expression. Le terme « évaluation » a ici la même signification que « exécution ».

Ici, j'obtiens les temps suivants :

```r
temps1
```

```
##    user  system elapsed
##    0.37    0.01    0.42
```

Les trois temps dans une sortie produite par `system.time` peuvent être définis ainsi :

- `user` : temps écoulé par le logiciel R (*calling process*) pour évaluer l'expression ;
- `system` : temps écoulé par le système d'exploitation de notre ordinateur, mais pour le compte du logiciel R, pendant l'évaluation de l'expression ;
- `elapsed` : temps total écoulé entre la soumission de l'expression et le retour du résultat (peut être plus grand que la somme des deux autres temps, car le système a peut-être dû accorder du temps à d'autres processus actifs sur l'ordinateur pendant l'évaluation de l'expression)

Les temps d'exécution obtenus dépendent des spécifications de l'ordinateur utilisé, en particulier de la puissance de son CPU. De plus, nous n'obtiendrons probablement pas exactement les mêmes si nous évaluons à nouveau l'expression. Il y a une petite variation normale du temps d'exécution, causée notamment par les autres processus utilisant le CPU de notre ordinateur au moment où la commande est lancée.

Package `microbenchmark`

Pour évaluer plus précisément le temps d'évaluation d'une expression, il est préférable de l'évaluer plusieurs fois, puis de calculer le temps moyen ou médian d'évaluation. La médiane est la statistique que je recommande d'utiliser ici, car elle est robuste aux valeurs extrêmes.

Le package `microbenchmark` s'utilise comme suit :

```r
library(microbenchmark)
microbenchmark(dens1 <- ksmooth1(x = x, xpts = xpts, h = 1), unit = "ms")
```

```
## Unit: milliseconds
##                                        expr      min       lq     mean   median
##  dens1 <- ksmooth1(x = x, xpts = xpts, h = 1) 146.1724 157.1413 170.9591 159.5962
##        uq    max neval
##  165.7131 473.76   100
```

Ici, la fonction `microbenchmark` du package du même nom a été utilisée. Par défaut, cette fonction évalue 100 fois l'expression. Dans la commande précédente, nous avons fourni la valeur `"ms"` à l'argument `unit`. Le temps a donc été mesuré en millisecondes. Nous aurions pu aller jusqu'à des unités aussi petites que des nanosecondes, ce qui est impossible avec la fonction `system.time`.

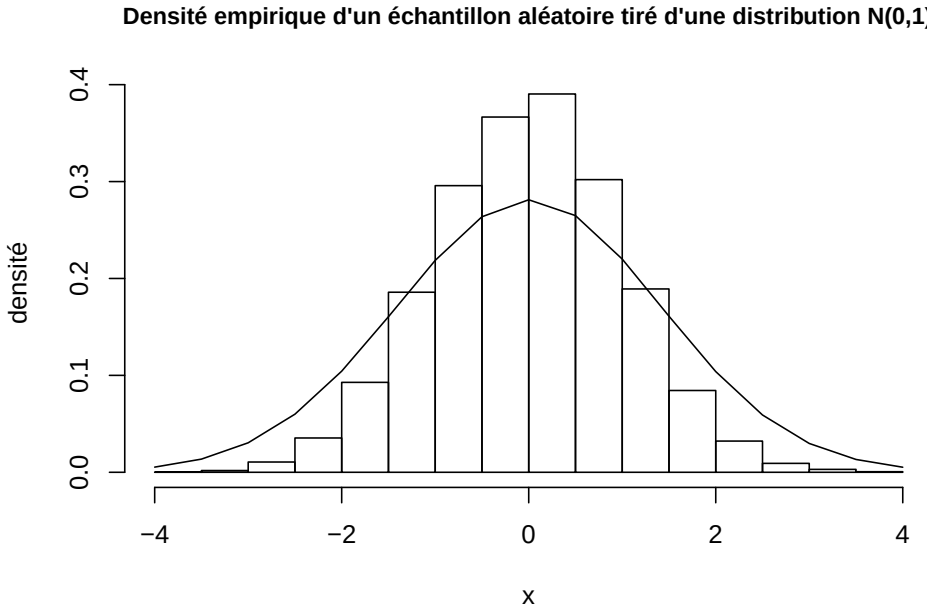
Le premier argument fourni dans l'appel à la fonction `microbenchmark` précédent est l'expression dont le temps d'évaluation est à mesurer, comme dans l'appel à la fonction `system.time`. Cependant, nous verrons plus loin que la fonction `microbenchmark` accepte en entrée plusieurs expressions, dont les temps d'exécution sont à comparer, alors que la fonction `system.time` est limitée au chronométrage d'une expression à la fois.

Notons que nous pouvons considérer ici que la fonction `ksmooth1` a d'abord été testée. Nous supposons donc qu'elle retourne un résultat valide. Voici d'ailleurs ce qu'elle retourne pour l'exemple précédent.

```r
# Résultat obtenu
dens1
```

```
##  [1] 0.005315167 0.013579675 0.030418267 0.059964210 0.104331243 0.160531643
##  [7] 0.218665414 0.263679673 0.281221960 0.264888188 0.220039533 0.161064723
## [13] 0.103894124 0.059117471 0.029725414 0.013232270 0.005219998
```

```r
# Graphique
hist(x, freq=FALSE, ylab = "densité", cex.main = 0.9,
     main = "Densité empirique d'un échantillon aléatoire tiré d'une distribution N(0,1)")
lines(xpts, dens1)
```



Fonctions `Rprof` et `summaryRprof`

Les fonctions `system.time` et `microbenchmark` sont bien pratiques pour évaluer un temps global d'exécution. Cependant, elles ne nous aident pas à identifier les parties du corps de la fonction `ksmooth1` qui sont les plus lentes. Pour ce faire, nous pouvons utiliser les fonctions `Rprof` et `summaryRprof`.

Pour utiliser ces fonctions, il faut ajouter la commande `Rprof()` avant le bout de code à minuter et ajouter `Rprof(NULL)` après le bout de code.

Pour faire un exemple avec la fonction `ksmooth1`, nous allons d'abord créer un nouveau vecteur d'observations et un nouveau vecteur de points en lesquels effectuer une estimation.

```r
x_long <- rnorm(250000)
xpts_long <- seq(from = -4, to = 4,length.out = 101)
```

Ces vecteurs sont plus longs que les anciens afin que l'appel à la fonction prenne plus de temps à être évalué, ce qui permettra à `Rprof` de mieux analyser la performance du code.

```r
Rprof(interval = 0.01)
dens1 <- ksmooth1(x = x_long, xpts = xpts_long, h = 1)
Rprof(NULL)
```

Un fichier a est créé dans notre répertoire de travail. Il se nomme par défaut *Rprof.out*, mais nous pouvons changer ce nom avec l'argument `filename` de la fonction `Rprof`. Toutes les 0.01 seconde (argument `interval`), R a écrit dans ce fichier le nom de la fonction ou des fonctions dont un appel est en cours d'évaluation.

Typiquement, nous n'allons pas voir directement le contenu de ce fichier. Nous en affichons plutôt un résumé avec la fonction `summaryRprof` comme suit :

```
summaryRprof("Rprof.out")
```

```
## $by.self
##            self.time self.pct total.time total.pct
## "ksmooth1"     12.26    50.12      24.46    100.00
## "dnorm"        12.20    49.88      12.20     49.88
##
## $by.total
##            total.time total.pct self.time self.pct
## "ksmooth1"      24.46    100.00     12.26    50.12
## "dnorm"         12.20     49.88     12.20    49.88
##
## $sample.interval
## [1] 0.01
##
## $sampling.time
## [1] 24.46
```

Dans cette sortie, les éléments `by.self` et `by.total` contiennent les mêmes valeurs, mais pas dans le même ordre (colonnes interchangées). Les colonnes `total.time` et `total.pct` réfèrent au temps total passé à l'évaluation de l'appel à une fonction. Pour les colonnes `self.time` et `self.pct`, le temps d'évaluation des appels de fonctions imbriqués dans l'appel de la fonction en question est retiré du temps total.

Dans l'exemple, la commande `dens1 <- ksmooth1(x = x_long, xpts = xpts_long, h = 1)` prend un total de 24.46 secondes à être évaluée. Cette commande comprend une assignation et un appel à la fonction `ksmooth1`. Les assignations simples ne sont pas retracées par `Rprof`, car il s'agit d'appels à un opérateur interne. Elles n'apparaissent donc pas dans la sortie de `summaryRprof`. Ici, les 24.46 secondes du temps total d'exécution sont 24.46 secondes à évaluer un appel à la fonction `ksmooth1`.

Évaluer un appel à la fonction `ksmooth1` signifie évaluer le corps de la fonction avec les valeurs d'arguments fournis en entrée. De l'évaluation du corps de la fonction `ksmooth1`, seul l'appel à la fonction `dnorm` apparaît dans la sortie de `summaryRprof`. Les appels aux autres fonctions ou opérateurs sont ici tellement rapides qu'ils n'ont pas été détectés par `Rprof`. Le temps passé à évaluer les appels à la fonction `dnorm` est de 12.20 secondes. Ainsi, le `self.time` de `ksmooth1` est 24.46 - 12.20 = 12.26 secondes.

Afin de mieux expliquer l'interprétation de la sortie de `summaryRprof`, voyons aussi un autre exemple qui produit une sortie un peu plus longue.

```
# Facteur généré aléatoirement pour l'exemple
fac_long <- sample(x = 1:10, size = length(x_long), replace = TRUE)

# Profilage du temps d'exécution d'un appel à aggregate avec les données simulées
Rprof(interval = 0.01)
res <- aggregate(x = x_long, by = list(fac_long), FUN = median)
Rprof(NULL)

# Résumé de la sortie de Rprof
summaryRprof("Rprof.out")
```

```
## $by.self
##                         self.time self.pct total.time total.pct
## ".row_names_info"            0.07    29.17       0.07     29.17
## "anyDuplicated.default"      0.07    29.17       0.07     29.17
## "as.character.factor"        0.04    16.67       0.04     16.67
## "aggregate.data.frame"       0.01     4.17       0.24    100.00
## "[.data.frame"               0.01     4.17       0.09     37.50
## "<Anonymous>"                0.01     4.17       0.01      4.17
## "factor"                     0.01     4.17       0.01      4.17
## "levels<-.factor"            0.01     4.17       0.01      4.17
## "unique.default"             0.01     4.17       0.01      4.17
##
## $by.total
##                         total.time total.pct self.time self.pct
## "aggregate.data.frame"        0.24    100.00      0.01     4.17
## "aggregate"                   0.24    100.00      0.00     0.00
## "aggregate.default"           0.24    100.00      0.00     0.00
## "[.data.frame"                0.09     37.50      0.01     4.17
## "["                           0.09     37.50      0.00     0.00
## ".row_names_info"             0.07     29.17      0.07    29.17
## "anyDuplicated.default"       0.07     29.17      0.07    29.17
## "anyDuplicated"               0.07     29.17      0.00     0.00
## "dim"                         0.07     29.17      0.00     0.00
## "dim.data.frame"              0.07     29.17      0.00     0.00
## "ncol"                        0.07     29.17      0.00     0.00
## "FUN"                         0.06     25.00      0.00     0.00
## "lapply"                      0.06     25.00      0.00     0.00
## "as.character.factor"         0.04     16.67      0.04    16.67
## "as.character"                0.04     16.67      0.00     0.00
## "<Anonymous>"                 0.01      4.17      0.01     4.17
## "factor"                      0.01      4.17      0.01     4.17
## "levels<-.factor"             0.01      4.17      0.01     4.17
## "unique.default"              0.01      4.17      0.01     4.17
## "as.factor"                   0.01      4.17      0.00     0.00
## "do.call"                     0.01      4.17      0.00     0.00
## "levels<-"                    0.01      4.17      0.00     0.00
## "match"                       0.01      4.17      0.00     0.00
## "sort"                        0.01      4.17      0.00     0.00
## "split"                       0.01      4.17      0.00     0.00
## "split.default"               0.01      4.17      0.00     0.00
## "unique"                      0.01      4.17      0.00     0.00
##
## $sample.interval
## [1] 0.01
##
## $sampling.time
## [1] 0.24
```

Dans cet exemple, les éléments `by.self` et `by.total` ne contiennent pas les mêmes lignes. Les fonctions dont les `self.time` sont nuls n'apparaissent pas dans l'élément `by.self`. Nous voyons aussi que les lignes sont ordonnées en ordre décroissant de `self.time` dans l'élément `by.self` et en ordre décroissant de `total.time` dans l'élément `by.total`.

Nous constatons que le code de la méthode `aggregate.data.frame` fait appel à un grand nombre de fonctions. Nous n'analyserons pas cette sortie davantage ici. Mentionnons seulement que les méthodes de la fonction

générique `aggregate` ne sont pas vraiment conçues pour être rapides.

Package `profvis`

Pour identifier encore plus facilement les lignes les plus lentes de notre code, nous allons utiliser le package `profvis`. Voici un exemple de son utilisation.

```
library(profvis)
profvis({
  ksmooth1 <- function(x, xpts, h)
  {
    dens <- double(length(xpts))
    n <- length(x)
    for(i in 1:length(xpts)) {
      ksum <- 0
      for(j in 1:length(x)) {
        d <- xpts[i] - x[j]
        ksum <- ksum + dnorm(d / h)
      }
      dens[i] <- ksum / (n * h)
    }
    dens
  }
  dens1 <- ksmooth1(x = x_long, xpts = xpts_long, h = 1)
})
```

Pour obtenir le détail du temps d'exécution par ligne du corps d'une de nos fonctions, il faut fournir, dans l'appel à la fonction `profvis`, le code définissant la fonction en plus de l'instruction appelant la fonction. Remarquez les accolades nécessaires lorsque l'expression à profiler s'étend sur plus d'une ligne. Le résultat obtenu est ouvert dans une fenêtre indépendante, dont voici une copie :

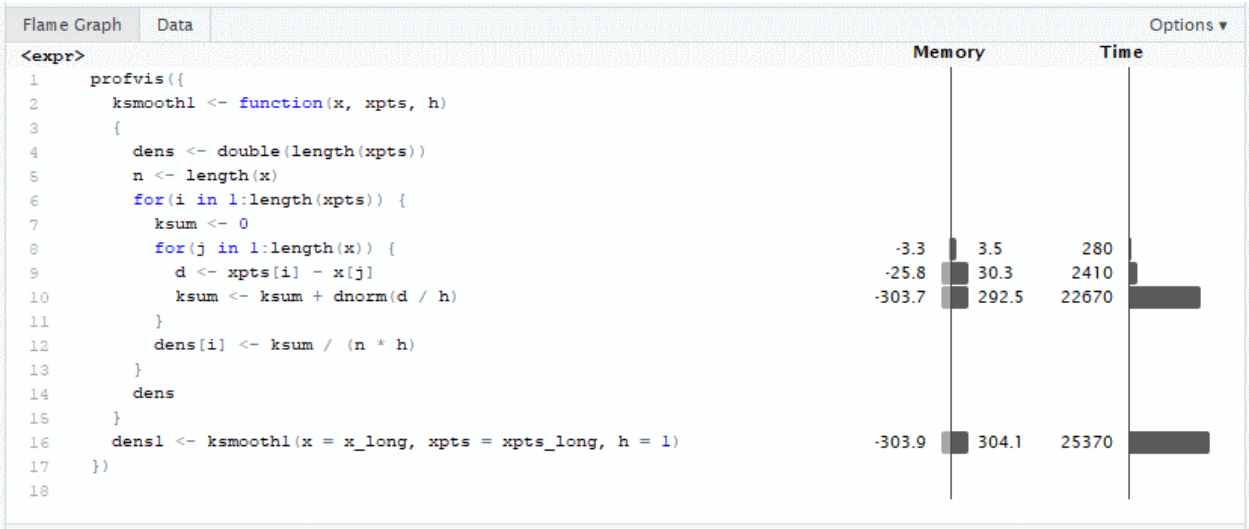


FIGURE 1 – Fenêtre de profilage ouverte par l'exemple d'appel à la fonction profvis précédent

RStudio intègre particulièrement bien la fenêtre affichant les résultats d'un appel à la fonction `profvis`. La figure précédente ne montre qu'un des deux onglets de cette fenêtre, soit l'onglet `Flame Graph`. La fonction `profvis` profile à la fois l'utilisation de la mémoire (colonne `Memory`) et du temps d'exécution (colonne `Time`). Nous nous intéressons ici seulement à la colonne `Time`.

Dans cet exemple, c'est la deuxième boucle, celle imbriquée dans la première, qui prend du temps à être évaluée. Et dans cette boucle, les appels à la fonction `dnorm` sont particulièrement lents. N'oublions pas qu'ici `dnorm` est appelé au total `length(xpts_long)` $\times$ `length(x_long)` fois ($101 \times 250000 = 25\ 250\ 000$ fois dans l'exemple).

Le site web suivant documente l'utilisation du package `profvis` : http://rstudio.github.io/profvis/index.html.

Comparaison de temps d'exécution avec la fonction `microbenchmark`

Étudions maintenant la performance d'une autre fonction d'estimation de densité par noyau gaussien tirée de Peng et de Leeuw, (2002). Le corps de cette fonction ne contient aucune boucle. Il fait plutôt du calcul vectoriel en utilisant, notamment, la fonction `outer`.

```
#' Estimation de densité par noyau gaussien
#'
#' Version 2 : utilisation de calcul vectoriel seulement
#'
#' mêmes arguments et même sortie que ksmooth1
#'
ksmooth2 <- function(x, xpts, h)
{
  n <- length(x)
  D <- outer(x, xpts, "-")
  K <- dnorm(D / h)
  dens <- colSums(K) / (h * n)
}
```

Assurons-nous d'abord que cette fonction retourne exactement les mêmes valeurs que `ksmooth1`.

```
dens1 <- ksmooth1(x = x, xpts = xpts, h = 1)
dens2 <- ksmooth2(x = x, xpts = xpts, h = 1)
all.equal(dens1, dens2)
```

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

C'est bien le cas.

Comparons maintenant le temps d'exécution de cette fonction à celui de `ksmooth1`.

```
microbenchmark(dens1 = ksmooth1(x = x, xpts = xpts, h = 1),
               dens2 = ksmooth2(x = x, xpts = xpts, h = 1),
               unit = "ms")
```

```
## Unit: milliseconds
##   expr      min       lq      mean    median       uq      max neval
##  dens1 152.5242 158.9074 188.75958 163.78575 190.2054 489.2212   100
##  dens2  13.1774  13.6081  14.86816  13.97505  15.0447  34.8146   100
```

Dans cet appel à la fonction `microbenchmark`, les deux instructions à comparer sont fournies en deux arguments distincts. Grâce à l'argument `...` de la fonction `microbenchmark`, cette fonction peut prendre autant d'expressions à chronométrer que souhaité.

Débuter ces expressions par une assignation avec l'opérateur `=` aide à alléger l'affichage des résultats. Le nom fourni à gauche de l'opérateur `=` pour une expression est celui utilisé pour identifier l'expression dans la sortie.

La fonction `ksmooth2` est environ 12 fois plus rapide que la fonction `ksmooth1`. Cependant, elle utilise beaucoup plus de mémoire. Une matrice de dimension `length(x)` par `length(xpts)` est créée par la fonction. R impose une limite sur la taille des objets créés (pour plus de détails voir la fiche d'aide `help("Memory-limits")`). Ainsi, la fonction `ksmooth2` retourne une erreur sur mon ordinateur si je lui donne en entrée des arguments `x` et `xpts` trop grands, par exemple :

```
x_test <- rnorm(1000000)
xpts_test <- seq(from = -4, to = 4, length.out = 1000000)
test_memoire <- ksmooth2(x = x_test, xpts = xpts_test, h = 1)
```

```
## Error: cannot allocate vector of size 7450.6 Gb
```

alors que la fonction `ksmooth1` est capable de traiter ces vecteurs.

```
# Attention, long à exécuter
test_memoire <- ksmooth1(x = x_test, xpts = xpts_test, h = 1)
```

Il y donc un compromis à gérer entre le temps d'exécution et la quantité de mémoire utilisée pour faire des calculs. Un programme peut être très rapide, mais créer un objet potentiellement de taille trop grande pour être stocké en mémoire. Étant donné que notre priorité est un code fonctionnel, il faut s'assurer de ne pas aller au-delà des limites de la mémoire de notre ordinateur. Alors, dans l'optimisation du temps d'exécution, il ne faut pas oublier de garder le contrôle sur la taille des objets créés par notre programme.

Stratégies d'optimisation du temps d'exécution

Il y a différentes stratégies utiles à connaître pour écrire du code R rapide. Voici une énumération de ces stratégies, qui sont présentées dans les sous-sections suivantes.

1. Utiliser des fonctions optimisées
2. Faire seulement ce qui est nécessaire
3. Exploiter les calculs matriciels et vectoriels
4. Éviter les allocations de mémoire inutiles
5. Faire du calcul en parallèle
6. Reprogrammer en C ou C++ les bouts de code les plus lents

Truc 1 : Utiliser des fonctions optimisées

Lorsque nous devons effectuer une tâche pour laquelle une fonction optimisée en temps de calcul existe déjà, il est préférable d'utiliser cette fonction. R est un logiciel libre. Le partage de code fait partie de la philosophie première du logiciel. Et cette réutilisation peut nous faire économiser beaucoup de temps.

Par exemple, R possède déjà une fonction pour l'estimation de densité par noyau. Est-ce que la fonction `density`, provenant du package `stats`, est plus rapide que la fonction `ksmooth1` ?

Premièrement, convainquons-nous que les deux fonctions peuvent effectuer le même calcul. La commande suivante :

```
dens1 <- ksmooth1(x = x, xpts = xpts, h = 1)
```

lance pratiquement le même calcul que cette commande :

```
densd <- density(x = x, bw = 1, kernel = "gaussian",
                 from = -4, to = 4, n = 17)
```

Rappelons que `xpts` avait été défini comme suit :

```
xpts <- seq(from = -4, to = 4, length.out = 17)
```

d'où le choix des valeurs fournies aux arguments `from`, `to` et `n` de `density`.

Comparons maintenant les valeurs obtenues.

```
all.equal(dens1, densd$y)
```

```
## [1] "Mean relative difference: 0.0009595629"
```

Il y a de très petites différences ente les valeurs, parce que le paramètre de lissage `h` de `ksmooth1` n'est pas tout à fait définit comme le paramètre `bw` de la fonction `density`. Cependant, ces différences sont tellement petites que nous pouvons tout de même considérer que les deux fonctions effectuent le même calcul.

Comparons les temps d'exécution des deux fonctions.

```
microbenchmark(densd = density(x = x, bw = 1, kernel = "gaussian", from = -4, to = 4, n = 17),
               dens1 = ksmooth1(x = x, xpts = xpts, h = 1),
               unit = "ms")
```

```
## Unit: milliseconds
##   expr      min       lq       mean    median        uq      max neval
##  densd   0.5538   0.6013   0.808639   0.73375   0.92695   1.8867   100
##  dens1 152.9800 158.6999 182.458223 162.05225 169.70330 433.7155   100
```

La fonction `density` retourne presque instantanément le résultat, alors que la fonction `ksmooth1` doit rouler pendant plusieurs secondes pour effectuer une estimation en 17 points, à partir de 10000 observations.

Le coeur du calcul de la fonction `density` est effectué par du code en langage C. C'est pour cette raison qu'elle est à ce point plus rapide que la fonction `ksmooth1`. Nous allons y revenir au truc 6.

Truc 2 : Faire seulement ce qui est nécessaire

L'idée derrière ce truc est simplement de ne pas alourdir un code d'évaluations inutiles.

Par exemple, si nous voulons calculer la somme des valeurs dans chacune des colonnes d'une matrice, la fonction `colSums` est plus rapide que la fonction `apply`.

```
mat <- matrix(x_long, nrow = 100, ncol = 1000)
microbenchmark(
  colSums(mat),
  apply(mat, 2, sum))
```

```
## Unit: microseconds
##                expr    min     lq     mean median      uq     max neval
##        colSums(mat)   62.5   83.5  105.433  103.9  120.65   371.3   100
##  apply(mat, 2, sum) 1421.3 1877.9 3118.662 2807.4 3320.45 12552.8   100
```

Ce résultat s'explique par le fait que la fonction `colSums` est spécialisée dans la tâche que nous cherchions à effectuer. Son code est simplifié, par rapport au code de `apply` qui peut appliquer n'importe quelle fonction sur n'importe quelle dimension d'un array. Nous pourrions aussi dire que la fonction `colSums` est une fonction optimisée.

C'est ce truc, de faire seulement ce qui est nécessaire, qui pousse certains programmeurs R à ne pas utiliser la fonction `return` pour retourner la sortie de leurs fonctions. L'appel à la fonction `return` amène une évaluation de plus à effectuer, sans être nécessaire. Pour faire un test, ajoutons un appel à la fonction `return` à la fin de la fonction `ksmooth2`.

```
ksmooth2return <- function(x, xpts, h)
{
  n <- length(x)
  D <- outer(x, xpts, "-")
  K <- dnorm(D / h)
  dens <- colSums(K) / (h * n)
  return(dens)
}
```

Maintenant, comparons les temps d'exécution des fonctions `ksmooth2` et `ksmooth2return`.

```r
microbenchmark(dens2 = ksmooth2(x = x, xpts = xpts, h = 1),
               dens2return = ksmooth2return(x = x, xpts = xpts, h = 1),
               unit = "us", times = 1000)
```

```
## Unit: microseconds
##         expr     min       lq     mean   median       uq     max neval
##        dens2 12825.1 13775.40 16171.31 14493.15 16946.40 82690.3  1000
##  dens2return 12904.5 13792.85 15969.58 14563.70 17056.65 84042.9  1000
```

La fonction `ksmooth2return` est effectivement légèrement plus lente que la fonction `ksmooth2`. Mais la différence entre les temps d'exécution médians des deux fonctions est à peine de l'ordre de quelques microsecondes. Alors d'autres programmeurs R préfèrent utiliser `return`, même s'il ralentit très légèrement les fonctions, dans le but d'avoir un code le plus clair possible.

Truc 3 : Exploiter les calculs matriciels et vectoriels

Nous avons déjà vu qu'en effectuant des calculs matriciels et vectoriels en R, comme dans la fonction `ksmooth2`, nous arrivons à faire un calcul beaucoup plus rapidement qu'avec une boucle, comme dans la fonction `ksmooth1`. Le calcul matriciel ou vectoriel en R est optimisé pour être très rapide. Il faut cependant faire attention à l'utilisation de la mémoire.

Nous aurions aussi pu coder la fonction `ksmooth` ainsi.

```r
#' Estimation de densité par noyau gaussien
#'
#' Version 3 : utilisation d'une boucle et d'un calcul vectoriel
#'
#' mêmes arguments et même sortie que ksmooth1
#'
ksmooth3 <- function(x, xpts, h)
{
  n <- length(x)
  dens <- double(length(xpts))
  for(i in 1:length(xpts)) {
    dens[i] <- sum(dnorm((xpts[i] - x)/h)) / (n * h)
  }
  dens
}
```

Cette version remplace la deuxième boucle par un calcul vectoriel.

Nous aurions même pu procéder comme suit.

```r
#' Estimation de densité par noyau gaussien
#'
#' Version 4 : utilisation d'une fonction de la famille des
#'             apply et d'un calcul vectoriel
#'
#' mêmes arguments et même sortie que ksmooth1
#'
ksmooth4 <- function(x, xpts, h)
{
  n <- length(x)
  sapply(
    X = xpts,
    FUN = function(xpts_i) {
      sum(dnorm((xpts_i - x) / h)) / (n * h)
```

```
    }
  )
}
```

Cette version remplace la seule boucle restante par l'utilisation d'une fonction de la famille des `apply`.

Ces deux nouvelles versions effectuent bien le même calcul que `ksmooth1`.

```
dens1 <- ksmooth1(x = x, xpts = xpts, h = 1)
dens3 <- ksmooth3(x = x, xpts = xpts, h = 1)
dens4 <- ksmooth4(x = x, xpts = xpts, h = 1)

all.equal(dens1, dens3)
```

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

```
all.equal(dens1, dens4)
```

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

Comparons maintenant les temps d'exécution des quatre versions de `ksmooth` écrites jusqu'à maintenant.

```
microbenchmark(unit = "ms",
               doubleBoucle_v1 = ksmooth1(x = x, xpts = xpts, h = 1),
  calculVectorielSeulement_v2 = ksmooth2(x = x, xpts = xpts, h = 1),
   boucleEtCalculVectoriel_v3 = ksmooth3(x = x, xpts = xpts, h = 1),
    applyEtCalculVectoriel_v4 = ksmooth4(x = x, xpts = xpts, h = 1))
```

```
## Unit: milliseconds
##                          expr      min        lq      mean   median        uq      max neval
##               doubleBoucle_v1 148.2374 158.79005 181.68885 161.6578 182.47020 427.0478   100
##  calculVectorielSeulement_v2  12.7607  12.93375  13.91572  13.1724  14.07775  29.4604   100
##   boucleEtCalculVectoriel_v3  12.3112  12.44710  13.82284  12.6600  13.60485  42.9883   100
##    applyEtCalculVectoriel_v4  12.3948  12.55975  13.95377  12.8877  13.81885  29.0852   100
```

Nous constatons que les deux dernières versions sont encore plus rapides que la version 2 !

Ce qu'il faut retenir de ces exemples est ceci :

- **Le code le plus rapide n'est pas toujours celui que nous croyons**. Il est parfois difficile de prédire quel bout de code sera le plus rapide. Ici, nous aurions pu croire que le calcul totalement vectoriel (`ksmooth2`) serait plus rapide qu'une boucle jumelée à un calcul vectoriel (`ksmooth3`). Pourtant, `ksmooth3` est légèrement plus rapide que `ksmooth2`. Il est donc toujours recommandé, lors de l'optimisation du temps d'exécution d'une fonction, d'essayer les différentes programmations possibles et de mesurer leurs temps d'exécution.
- **Les fonctions de la famille des `apply` ne sont pas nécessairement plus rapides qu'une boucle**. Ces fonctions cachent littéralement des boucles et ne représentent pas une sorte de calcul vectoriel. Leur utilisation est recommandée principalement parce qu'elles mènent à du code plus court et plus clair selon plusieurs programmeurs R, pas à du code nécessairement plus rapide.

Truc 4 : Éviter les allocations de mémoire inutiles

Allouer de l'espace dans la mémoire d'un ordinateur est une opération coûteuse en temps. À chaque fois qu'un nouvel objet est créé, une allocation en mémoire est effectuée.

Deux opérations plutôt anodines sont à éviter dans une boucle R, car elles provoquent une allocation en mémoire et ralentissent donc beaucoup la boucle. Il s'agit de :

1. l'utilisation d'un objet de dimension croissante,
2. l'assignation de valeur(s) à un ou des éléments d'un data frame.

Objets de dimension croissante :

Un objet de dimension croissante est, par exemple, une matrice à laquelle nous ajoutons, à chaque itération d'une boucle, une ligne avec `rbind` ou une colonne avec `cbind`, comme le fait l'instruction suivante :

```
matrice <- rbind(matrice, nouvelleLigne)
```

Avec un vecteur, une instruction similaire ferait plutôt appel à la fonction `c` ou `append` comme suit :

```
vecteur <- c(vecteur, nouvelElement)
```

Le problème avec ces assignations est qu'elles modifient la dimension d'un objet. L'objet ne requière donc plus la même quantité d'espace mémoire. Il ne serait pas une bonne idée de simplement utiliser les cases mémoires adjacentes pour agrandir l'objet, car ces cases mémoires sont potentiellement utilisées pour stocker d'autres objets R ou n'importe quelle valeur nécessaire à un processus en cours d'exécution sur l'ordinateur. L'ordinateur doit plutôt complètement déplacer l'objet dans de nouvelles cases mémoire qu'il sait être inutilisées afin de ne pas entrer en conflit avec quoi que ce soit. Ainsi, avec les commandes précédentes, nous avons peut-être l'impression de modifier le contenu de certaines cases mémoire alors qu'en réalité nous provoquons une nouvelle allocation de mémoire.

Exemple :

Simulons une expérience aléatoire pour compter le nombre de fois qu'un dé doit être tiré afin d'atteindre une somme des valeurs obtenues supérieure ou égale à 50000. Nous voulons conserver en mémoire toutes les valeurs obtenues.

La première fonction que nous allons créer pour simuler cette expérience va utiliser un objet de dimension croissante pour garder une trace des résultats.

```
sommeDe1 <- function(sommeVisee = 50000){
  somme <- 0
  resultats <- integer(length = 0)  # ou resultats <- NULL
  while(somme < sommeVisee) {
    tirage <- sample(1:6, size = 1)
    somme <- somme + tirage
    resultats <- c(resultats, tirage)
  }
  resultats
}
```

La deuxième fonction que nous allons créer pour simuler cette expérience va plutôt utiliser un grand objet de taille fixe pour stocker les résultats.

```
sommeDe2 <- function(sommeVisee = 50000){
  somme <- 0
  resultats <- integer(sommeVisee)
  i <- 0
  while(somme < sommeVisee) {
    tirage <- sample(1:6, size = 1)
    somme <- somme + tirage
    i <- i + 1
    resultats[i] <- tirage
  }
  resultats[1:i]
}
```

Nous ne savons pas d'avance combien de lancés du dé devront être effectués pour atteindre une somme de `sommeVisee`, mais nous savons que ce sera au maximum `sommeVisee` lancés, puisque le plus petit résultat du lancé d'un dé est 1. Ainsi, nous créons d'abord un très grand vecteur, de longueur `sommeVisee`, et nous allons modifier les éléments de ce vecteur à chaque itération de la boucle (une itération = un lancé de dé). Nous modifions d'abord le premier élément puis le deuxième et ainsi de suite, grâce à l'indicateur de position `i` que nous incrémentons de 1 à chaque itération. À la fin, nous retournons seulement les éléments du vecteur de résultat qui ont été modifiés.

Comparons les temps d'exécution des deux fonctions.

```
microbenchmark(sommeDe1(), sommeDe2(), unit = "ms")
```

```
## Unit: milliseconds
##        expr      min       lq      mean   median       uq      max neval
##  sommeDe1() 234.8832 246.4969 279.46190 254.2296 269.2481 628.8068   100
##  sommeDe2()  56.2464  59.4830  65.37152  61.3204  64.2775 138.6909   100
```

Nous constatons donc qu'en termes de temps de calcul, il est préférable de créer un très grand objet, de le remplir, puis de mettre de côté les éléments inutilisés que de faire croître la taille d'un objet. C'est de la préallocation de mémoire. Par contre, encore là, il y a une limite à la grandeur de l'objet qui peut être créé.

Modification d'éléments dans un data frame

Lorsque nous modifions les éléments d'un objet R dans une boucle, comme nous avons fait dans la fonction `sommeDe2` par l'instruction suivante :

```
resultats[i] <- tirage
```

la modification s'effectue sans réallocation de mémoire à chaque itération si :

- l'objet en question est un objet atomique (vecteur, matrice ou array) ou une liste ;
- la valeur assignée est du même type que les éléments de l'objet initialisé (dans le cas d'un objet atomique).

Pour nous en convaincre, faisons quelques tests en utilisant la fonction `tracemem` qui affiche un message à chaque fois qu'un objet est copié en mémoire.

Voici une boucle qui modifie les éléments d'une matrice.

```
matrice <- matrix(integer(20), 4, 5)
tracemem(matrice)
```

```
[1] "<0000000019C9B820>"
```

```
for (i in 1:5){
  matrice[, i] <- 1:4
}
```

```
tracemem[0x0000000019c9b820 -> 0x0000000019f53170]:
```

```
untracemem(matrice)
```

L'objet `matrice` est copié une seule fois, au début de la boucle, mais pas à chaque itération.

Nous observons le même comportement avec une liste.

```
liste <- vector(mode = "list", length = 5)
tracemem(liste)
```

```
[1] "<000000001A2DC048>"
```

```
for (i in 1:5){
  liste[[i]] <- 1:4
}
```

```
tracemem[0x000000001a2dc048 -> 0x0000000019d88738]:
```

```
untracemem(liste)
```

Cependant, R se comporte différemment lors de la modification d'un élément dans un data frame

```
df <- as.data.frame(matrix(integer(20), 4, 5))
tracemem(df)
```

```
[1] "<0000000019B7D730>"
```

```
for (i in 1:5){
  df[, i] <- 1:4
}
```

```
tracemem[0x0000000019b7d730 -> 0x0000000018e92818]:
tracemem[0x0000000018e92818 -> 0x0000000018e92950]: [<-.data.frame [<-
tracemem[0x0000000018e92950 -> 0x0000000018e929b8]: [<-.data.frame [<-
tracemem[0x0000000018e929b8 -> 0x0000000018e92a20]:
tracemem[0x0000000018e92a20 -> 0x0000000018e92b58]: [<-.data.frame [<-
tracemem[0x0000000018e92b58 -> 0x0000000018e92bc0]: [<-.data.frame [<-
tracemem[0x0000000018e92bc0 -> 0x0000000018e92c28]:
tracemem[0x0000000018e92c28 -> 0x000000001a2ab810]: [<-.data.frame [<-
tracemem[0x000000001a2ab810 -> 0x000000001a2ab878]: [<-.data.frame [<-
tracemem[0x000000001a2ab878 -> 0x000000001a2ab8e0]:
tracemem[0x000000001a2ab8e0 -> 0x000000001a2aba18]: [<-.data.frame [<-
tracemem[0x000000001a2aba18 -> 0x000000001a2aba80]: [<-.data.frame [<-
tracemem[0x000000001a2aba80 -> 0x000000001a2abae8]:
tracemem[0x000000001a2abae8 -> 0x000000001a2abc20]: [<-.data.frame [<-
tracemem[0x000000001a2abc20 -> 0x000000001a2abc88]: [<-.data.frame [<-
```

```
untracemem(df)
```

À chaque itération de cette boucle, l'objet `df` est recopié 3 fois. Ces allocations de mémoire répétées prennent du temps.

Reprenons l'exemple de la simulation d'une expérience aléatoire pour compter le nombre de fois qu'un dé doit être tiré afin d'atteindre une somme des valeurs obtenues supérieure ou égale à 50000. Voici deux autres fonctions réalisant cette expérience, qui diffèrent seulement par le type de l'objet utilisé pour stocker les résultats. La fonction `sommeDe3` utilise une matrice et la fonction `sommeDe4` un data frame. Quelle fonction est la plus rapide ?

```
# En utilisant une matrice pour stocker les résultats
sommeDe3 <- function(sommeVisee = 50000){
  resultats <- matrix(NA, ncol = 2, nrow = sommeVisee)
  # Colonne 1 : numéro de l'itération
  resultats[, 1] <- 1:sommeVisee
  # Boucle
  somme <- 0
  i <- 0
  while(somme < sommeVisee) {
    tirage <- sample(1:6, size = 1)
    somme <- somme + tirage
    i <- i + 1
```

```
    # Colonne 2 : résultat obtenu au lancé du dé
    resultats[i, 2] <- tirage
  }
  resultats[1:i, ]
}

# En utilisant un data frame pour stocker les résultats
sommeDe4 <- function(sommeVisee = 50000){
  resultats <- as.data.frame(matrix(NA, ncol = 2, nrow = sommeVisee))
  # Colonne 1 : numéro de l'itération
  resultats[, 1] <- 1:sommeVisee
  # Boucle
  somme <- 0
  i <- 0
  while(somme < sommeVisee) {
    tirage <- sample(1:6, size = 1)
    somme <- somme + tirage
    i <- i + 1
    # Colonne 2 : résultat obtenu au lancé du dé
    resultats[i, 2] <- tirage
  }
  resultats[1:i, ]
}

microbenchmark(sommeDe3(), sommeDe4(), unit = "ms")
```

```
## Unit: milliseconds
##        expr      min        lq      mean    median        uq       max neval
##  sommeDe3()  57.8001  61.06065  72.91964  65.98515  76.87975  196.0028   100
##  sommeDe4() 766.0980 801.61845 858.22237 840.76730 871.33720 1714.1097   100
```

Ici, l'utilisation du data frame est environ 12 fois plus lente que l'utilisation d'une matrice !

Bref, autant que possible, il vaut mieux **éviter d'utiliser une data frame pour stocker des résultats générés dans une boucle**.

La lenteur des opérations de manipulation de data frame est bien connue en R. Des alternatives plus rapides existent, notamment les objets de classe `data.table` offerts par le package `data.table`. Ce package avait été mentionné dans les notes sur les structures de données en R.

Truc 5 : Faire du calcul en parallèle

Une importante technique pour réaliser des calculs informatiques plus rapidement est le calcul en parallèle. Il s'agit d'un type de calcul qui, dans sa version la plus simple,

- brise un long calcul en petits blocs de calcul indépendants ;
- réalise ces blocs de calcul sur plusieurs unités de calcul, simultanément (donc en parallèle) ;
- rassemble à la fin tous les résultats.

Les unités de calcul utilisées peuvent être localisées sur CPU (pour *Central Processing Unit*) ou sur GPU (pour *Graphical Processing Unit*). Nous pouvons exploiter différentes unités sur une seule machine ou encore sur plusieurs noeuds de calcul dans une grappe de serveurs.

Je vais réaliser un exemple dans lequel je vais exploiter tous les coeurs du CPU de mon ordinateur.

Il existe un très grand nombre de packages R pour réaliser du calcul en parallèle (https://cran.r-project.org/web/views/HighPerformanceComputing.html). Un de ces packages vient avec l'installation de R. Il s'agit du

package `parallel`. Ce package est donc déjà installé sur votre ordinateur si R y est installé. Cependant, le package n'est pas chargé par défaut lors de l'ouverture d'une nouvelle session R. Chargeons-le.

```
library(parallel)
```

Tout d'abord, voyons combien de coeurs compte mon ordinateur.

```
detectCores()
```

```
## [1] 4
```

Il compte 4 coeurs logiques.

Maintenant, si nous travaillons sous Windows, il faut d'abord établir une connexion entre R est les différents coeurs avec la fonction `makeCluster`.

```
coeurs <- detectCores()
grappe <- makeCluster(coeurs - 1)
grappe
```

```
## socket cluster with 3 nodes on host 'localhost'
```

Notons que la première fois que j'ai soumis cette commande, Windows m'a demandé une autorisation.

Remarquons aussi que je n'ai utilisé que 3 des 4 coeurs disponibles sur mon ordinateur dans le but de laisser un coeur libre pour les autres processus actifs.

Ensuite, je vais comparer la fonction `ksmooth4`, qui utilise `sapply`, à une autre version de `ksmooth` qui utilise la version parallèle du `sapply`, offerte par le package `parallel`, nommée `parSapply`.

```
#' Estimation de densité par noyau gaussien
#'
#' Version 5 : utilisation de parSapply et d'un calcul vectoriel
#'
#' mêmes arguments et même sortie que ksmooth1
#'
ksmooth5 <- function(grappe, x, xpts, h)
{
  n <- length(x)
  parSapply(
    cl = grappe,
    X = xpts,
    FUN = function(xpts_i) {
      sum(dnorm((xpts_i - x) / h)) / (n * h)
    }
  )
}
```

Ces fonctions effectuent bien le même calcul.

```
dens4 <- ksmooth4(x = x, xpts = xpts, h = 1)
dens5 <- ksmooth5(grappe = grappe, x = x, xpts = xpts, h = 1)

all.equal(dens4, dens5)
```

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

Laquelle est la plus rapide ?

```
microbenchmark(unit = "ms",
     sapply_v4 = ksmooth4(x = x, xpts = xpts, h = 1),
  parSapply_v5 = ksmooth5(grappe = grappe, x = x, xpts = xpts, h = 1))
```

```
## Unit: milliseconds
##          expr     min      lq     mean   median       uq     max neval
##     sapply_v4 16.6670 17.6102 18.77782 18.08640 18.95755 39.7285   100
##  parSapply_v5  8.3712  9.1249 10.60558  9.49235 10.27545 31.3514   100
```

Le calcul en parallèle a permis de réduire un peu le temps d'exécution. Même si 3 coeurs ont été exploités, le calcul n'est pas 3 fois plus rapide, car :

- mon ordinateur possède en fait 2 coeurs physiques, chacun séparé en 2 coeurs logiques (pour un total de 4 coeurs logiques), et des coeurs logiques ne sont pas aussi rapides que des coeurs physiques ;
- toutes les communications entre les coeurs et R requièrent aussi un peu de temps.

Une fois le calcul terminé, il est recommandé de fermer les connexions entre R et les coeurs de calcul avec la fonction `stopCluster`.

```
stopCluster(grappe)
```

Nous aurions pu aller chercher une amélioration plus importante du temps de calcul en utilisant plus d'unités de calcul. Le département de mathématiques et de statistique possède une grappe de calcul pouvant être utilisée par les étudiants du département pour faire du calcul en parallèle. Calcul Québec gère aussi des supercalculateurs pour le calcul en parallèle utilisable gratuitement par tout chercheur (et ses étudiants) admissible aux subventions provenant des conseils de recherche canadiens, à la condition d'avoir obtenu des accès aux ressources : http://www.calculquebec.ca/fr/acces-aux-ressources. Finalement, plusieurs plateformes de *cloud computing* permettent d'utiliser des serveurs de calculs à faible coût (par exemple Amazon Web Services, Microsoft Azure, Google Cloud Platform).

Lancer des calculs en parallèle sur une grappe de calcul ne s'effectue pas tout à fait comme le lancement de calculs en parallèle sur une seule machine. La communication avec la grappe s'effectue typiquement via des protocoles SSH et les programmes R se lancent en mode batch grâce à la commande `Rscript`. Ce sujet ne sera pas couvert ici, car il est plutôt complexe et la mise en oeuvre de calculs en parallèle dépend des ressources à notre disposition. Pour plus d'informations, je vous réfère à un document que j'ai écrit sur le sujet, qui est disponible ici : https://stt4230.rbind.io/autre_materiel/calcul_parallele_r/.

Truc 6 : Reprogrammer en C ou C++ les bouts de code les plus lents

Note : La matière présentée dans cette section ne sera pas évaluée.

Un dernier truc pour rendre du code R plus rapide est de reprogrammer ses bouts les plus lents en C ou C++. Le langage R étant un langage interprété, il n'est pas aussi rapide que du C ou du C++, qui sont des langages plus près du langage machine.

Nous n'utilisons pas ce truc pour réaliser des analyses de données plus rapidement, mais plutôt pour créer une fonction qui réalise rapidement un certain calcul.

Il existe quelques outils pour intégrer du code C ou C++ en R. Un outil très populaire pour intégrer du code C++ en R est le package `Rcpp` (http://www.rcpp.org/). Le R de base offre pour sa part les fonctions `.C`, `.Call` et `.External` pour ce faire (voir le manuel *Writing R Extensions*, chapitre 5). Je vais me contenter ici d'illustrer l'utilisation de la fonction `.C`, qui est la méthode la plus simple, mais la moins puissante.

La fonction `ksmooth1` peut être reprogrammée en C comme suit (Peng et de Leeuw, 2002) :

```
#include <R.h>
#include <Rmath.h>

void kernel_smooth(double *x, int *n, double *xpts,
       int *nxpts, double *h, double *result)
{
 int i, j;
 double d, ksum;

 for(i=0; i < *nxpts; i++)
 {
  ksum = 0;
  for(j=0; j < *n; j++)
  {
   d = xpts[i] - x[j];
   ksum += dnorm(d / *h, 0, 1, 0);
  }
  result[i] = ksum / ((*n) * (*h));
 }
}
```

Du code C destiné à être appelé en R avec la fonction `.C` se doit de respecter les propriétés suivantes (Peng et de Leeuw, 2002) :

- Les fonctions C appelées en R doivent être de type « void ». Elles doivent retourner les résultats des calculs par leurs arguments.
- Les arguments passés aux fonctions C sont des pointeurs à un nombre ou à un tableau. Il faut donc correctement déréférencer les pointeurs dans le code C afin d'obtenir la valeur d'un élément dont l'adresse est contenue dans le pointeur. Un pointeur est déréférencé en ajoutant `*` devant celui-ci.
- Il est préférable d'inclure dans tout fichier contenant du code C à être appelé en R le fichier d'en-tête R.h en ajoutant au début du fichier de code C la ligne :

  ```
  #include <R.h>
  ```

  De plus, il est possible d'utiliser en C certaines fonctions mathématiques R en incluant le fichier d'en-tête Rmath.h dans le fichier de code C par la ligne :

  ```
  #include <Rmath.h>
  ```

  Les fonctions mathématiques R utilisables en C sont énumérées dans le manuel de R *Writing R Extensions*, chapitre 6.
- Le fichier contenant le code C doit porter l'extension *.c*.

Une fois le code C écrit, il reste trois étapes à compléter pour intégrer du code C en R avec la fonction `.C`.

1. Compiler le code C afin de créer un objet partagé si nous travaillons sur Linux ou une « bibliothèque de liens dynamiques » (an anglais *DLL*) si nous travaillons sur Windows ou Mac OS X ;
2. Charger en R l'objet partagé ou le DLL créé à l'étape précédente avec la fonction `dyn.load` ;
3. Appeler en R les fonctions créées dans le code C avec la fonction d'interface `.C`.

Retournons donc à l'exemple. Supposons que le code C ci-dessus se trouve dans le fichier `C:/coursR/ksmoothC.c`. Dans le *terminal* sous Linux ou Mac OS X et dans une *fenêtre invite de commandes* sous Windows, il faut se positionner dans le répertoire contenant le fichier et lancer la commande suivante :

```
R CMD SHLIB ksmoothC.c
```

Notez qu'en RStudio, nous pouvons facilement ouvrir un terminal ou une fenêtre invite de commandes par le menu *Tools* > *Shell...*.

Cette commande fonctionnera seulement si un compilateur C/C++ est installé sur l'ordinateur. Les outils nécessaires au développement de packages R en fournissent un. Si la commande a fonctionné, l'objet partagé ou le DLL sera créé. Sur Windows, il s'agit d'un fichier portant l'extension *.dll*.

Maintenant, chargeons cet objet en R avec la fonction `dyn.load`, comme dans cet exemple réalisé sur Windows :

```
dyn.load("C:/coursR/ksmoothC.dll")
```

Il ne reste plus qu'à écrire la « fonction R enveloppe », qui appelle la fonction écrite en C, comme dans cet exemple :

```
#' Estimation de densité par noyau gaussien
#'
#' Version _C : code C + appel à la fonction .C
#'
#' mêmes arguments et même sortie que ksmooth1
#'
ksmooth_C <- function(x, xpts, h) {
  n <- length(x)
  nxpts <- length(xpts)
  dens <- .C("kernel_smooth", as.double(x), as.integer(n),
             as.double(xpts), as.integer(nxpts), as.double(h),
             result = double(length(xpts)))
  dens$result
}
```

Dans l'appel à la fonction `.C`, le nom de la fonction doit obligatoirement être entre guillemets. Il est préférable de s'assurer que chaque argument passé à la fonction C est du bon type en appliquant aux arguments une fonction telle `as.integer`, `as.double`, `as.character` ou `as.logical`.

Est-ce que la fonction `ksmooth_C` effectue bien le même calcul que `ksmooth1` ?

```
dens1 <- ksmooth1(x = x, xpts = xpts, h = 1)
densC <- ksmooth_C(x = x, xpts = xpts, h = 1)

all.equal(dens1, densC)
```

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

Oui.

Maintenant, voyons si cette nouvelle version de `ksmooth` est plus rapide que certaines des autres fonctions que nous avons développées. Comparons aussi `ksmooth_C` à la fonction `density`

```
microbenchmark(unit = "ms",
  ksmooth1 = ksmooth1(x = x, xpts = xpts, h = 1),
  ksmooth2 = ksmooth2(x = x, xpts = xpts, h = 1),
  ksmooth3 = ksmooth3(x = x, xpts = xpts, h = 1),
  ksmooth_C = ksmooth_C(x = x, xpts = xpts, h = 1),
  density = density(x = x, bw = 1, kernel = "gaussian", from = -4, to = 4, n = 17))
```

```
## Unit: milliseconds
##       expr        min         lq        mean      median          uq        max neval
##   ksmooth1 160.018944 164.591159 170.1964157 167.9260415 170.7282935 220.236163   100
##   ksmooth2  13.402644  13.713477  15.6816264  14.8917480  16.5846660  61.744760   100
##   ksmooth3  12.471381  12.584219  14.0803544  12.7721640  14.8104705  58.695675   100
##  ksmooth_C  10.264700  10.315124  10.6443776  10.3643150  10.5529655  13.869510   100
##    density   0.513059   0.635065   0.6897806   0.6770265   0.7061175   1.384377   100
```

La fonction `ksmooth_C` bat `ksmooth1` (double boucle), `ksmooth2` (calcul vectoriel seul avec `outer`) et `ksmooth3` (boucle et calcul vectoriel), quoiqu'elle n'est pas beaucoup plus rapide que cette dernière. Cependant, `density` demeure beaucoup plus rapide que tout ce que nous avons programmé.

Mais pourquoi `density` est-il tellement plus rapide alors qu'il fait appel à du code C, tout comme `ksmooth_C` ? Premièrement, parce que ce code C est interfacé en R par la fonction `.Call` plutôt que `.C`. L'interface `.Call` est plus compliquée d'utilisation que `.C`, mais plus efficace. La fonction `density` est aussi plus rapide parce que son code C a lui aussi été optimisé.

Synthèse

Optimisation de temps d'exécution

Outils pour analyser la performance d'un programme R :

- calcul global de temps d'exécution :
  - la fonction `system.time` (R de base),
  - la fonction `microbenchmark` (package du même nom) ;
- calcul plus détaillé de temps d'exécution :
  - par fonction appelée (*call stack*) : les fonctions `Rprof` et `summaryRprof` (R de base),
  - par ligne de code : la fonction `profvis` (package du même nom).

Conseil : comparaison de différentes options
$\longrightarrow$ Le code le plus rapide n'est pas toujours celui que nous croyons.

Stratégies d'optimisation du temps d'exécution

Trucs pour du code R plus rapide :

1. Utiliser des fonctions optimisées
2. Faire seulement ce qui est nécessaire
3. Exploiter les calculs matriciels et vectoriels
4. Éviter les allocations de mémoire inutiles
5. Faire du calcul en parallèle
6. Reprogrammer en C ou C++ les bouts de code les plus lents

Contrainte : **compromis temps d'exécution / mémoire utilisée**

Truc 1 : Utiliser des fonctions optimisées

Profiter du travail des autres

Le web regorge de fonctions R : la distribution de base de R, le CRAN, Bioconductor, etc.

Truc 2 : Faire seulement ce qui est nécessaire

Ne pas alourdir son code d'évaluations inutiles

Truc 3 : Exploiter les calculs matriciels et vectoriels

R est optimisé pour le calcul vectoriel

(fonctions de la famille des `apply` = boucles, pas calcul vectoriel)

Truc 4 : Éviter les allocations de mémoire inutiles

Allouer de l'espace dans la mémoire d'un ordinateur = opération coûteuse en temps

Opérations à éviter dans une boucle R :

1. utilisation d'un objet de dimension croissante,
2. assignation de valeur(s) dans un data frame.

Truc 5 : Faire du calcul en parallèle

Version la plus simple du calcul en parallèle :

- brise un long calcul en petits blocs de calcul indépendants ;
- réalise ces blocs de calcul sur plusieurs unités de calcul (coeurs d'un processeur) simultanément (donc en parallèle) ;
- rassemble à la fin tous les résultats.

Domaine très large : https://cran.r-project.org/web/views/HighPerformanceComputing.html

Domaine très technique : dépend des ressources à notre portée (CPU versus GPU, plusieurs coeurs sur notre ordinateur, grappe de serveurs de calcul, plateforme de cloud computing, etc.)

Point de départ en R : package `parallel`

Truc 6 : Reprogrammer en C ou C++ les bouts de code les plus lents

Meilleure solution pour développer une fonction très rapide, pas pour réaliser rapidement des analyses de données.

- fonction d'interface `.C`,
- fonction d'interface `.Call` ou `.External`,
- package `Rcpp` pour l'intégration de code C++.

Références

- **Outils d'analyse de la performance d'un programme R**
  - package `microbenchmark` : https://CRAN.R-project.org/package=microbenchmark
  - Alternatives au package `microbenchmark` : package `bench` https://github.com/r-lib/bench
  - package `profvis` : https://rstudio.github.io/profvis/

- **Optimisation de temps d'exécution**
  - Wickham, H. (2014). *Advanced R*. CRC Press.
    - Édition 1, Chapitre 17, URL http://adv-r.had.co.nz/Profiling.html
    - Édition 2, Chapitre 24, URL https://adv-r.hadley.nz/perf-improve.html#code-organisation
  - Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design*, No Starch Press. Chapitre 14.
  - Adler, J. (2012). *R in a Nutshell*, Second edition. O'Reilly. Chapitre 24.
  - http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html

- **Interfacer du code dans un autre langage (en particulier C ou C++)**
  - Peng, R. D., & de Leeuw, J. (2002). An Introduction to the .C Interface to R. UCLA : Academic Technology Services, Statistical Consulting Group. URL http://www.biostat.jhsph.edu/~rpeng/docs/interface.pdf

  - R Core Team (2018). *Writing R Extensions*. R Foundation for Statistical Computing. Chapitre 5. URL http://cran.r-project.org/doc/manuals/r-release/R-exts.html#System-and-foreign-language-interfaces
  - Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design*, No Starch Press. Chapitre 15.
  - Utilisation du package `Rcpp` :
    * http://www.rcpp.org/
    * Wickham, H. (2014). *Advanced R*. CRC Press. Chapitre 19. URL http://adv-r.had.co.nz/Rcpp.html

- **Calcul en parallèle**
  - https://stt4230.rbind.io/autre_materiel/calcul_parallele_r/
  - McCallum, E., & Weston, S. (2011). *Parallel R*. O'Reilly.
  - Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design, No Starch Press*. Chapitre 16.