Formation R avancée :
Outils de développement et de performance

Robin Genuer et Boris Hejblum

2018-04-10

Contents

Présentation de la formation

Le but principal de cette formation est de vous donner des outils pour faciliter le développement de code (performant) avec `R`. L'aspect "performance" arrivera dans un deuxième temps, et les premiers outils présentés sont également très utiles dans des situations ne faisant pas intervenir de temps de calculs importants.

Nous allons centrer la présentation de ces outils de développement autour de la notion de package. Vous connaissez déjà cette notion, car vous avez déjà installer des packages depuis le CRAN par exemple. Vous savez également que c'est le moyen le plus standard dans `R` pour mettre à disposition du code. Nous allons vous montrer que le package est également un excellent outil pour développer le code.

Nous allons adopter le plan suivant :

1. Construire un package
2. Tracer les changements, partager son code, développement collaboratif et automatiser les tests dans un package
3. Mesurer le temps de calcul
4. *Profiler* le code
5. Utiliser *Rcpp* pour optimiser ce qui doit l'être
6. Paralléliser facilement le code

Vous pouvez télécharger le contenu de cette formation ici (ouvrir alors le fichier `index.html`)

Afin de suivre cette formation, il est nécessaire de disposer des logiciels suivants :

- la dernière version de *R* (v3.4.3 - https://cloud.r-project.org/)
- la dernière version de *RStudio* (v1.1.423 - https://www.rstudio.com/products/rstudio/download/#download)
- un *compilateur `C++`* (tel que `gcc` ou `clang` - natif sous les système UNIX, pour les utilisateurs Windows nous recommandons l'installation de Rtools, pour les utilisateurs Mac il peut être nécessaire d'installer les outils de développement Apple comme suggéré ici)
- les packages `R` suivants : *devtools*, *doParallel*, *itertools*, *microbenchmark*, *profvis*, *Rcpp*, *RcppArmadillo*, *roxygen2*, *testthat*, *mvtnorm*
- le client *GitHub Desktop*
- le logiciel *git*

Chapter 1

Construire un package `R`

Nous présentons ici comment construire un package efficacement à l'aide d'outils graphiques présents dans Rstudio et du package `devtools`.

Le site de référence sur ce sujet est le site R packages d'Hadley Wickham.

1.1 Initialiser un package

Une manière simple, et intégrée à Rstudio, pour **initialiser un package** est :

1. de créer un nouveau projet (menu déroulant en haut à droite dans Rstudio)
2. choisir "New Directory"
3. choisir "R package using devtools" (s'il n'est pas disponible, choisir "R package", différence étant qu'avec "R package", il faudra supprimé des fichiers créés automatiquement mais non utiles)
4. donner un nom au package, par exemple `mypkgr`

On récupère la structure **minimale pour un package R**, à savoir :

- un fichier `DESCRIPTION` dont les parties `Title`, `Version`, `Authors@R` et `Description` sont à éditer (d'autres parties pourront être éditer voire même ajouter de manière automatique, voir plus loin)
- un fichier `NAMESPACE` qui sera éditer automatiquement plus loin
- un dossier `R` dans lequel on va ajouter des fichiers de scripts R

Rstudio ajoute également trois **fichiers facultatifs** :

- `.gitignore`, relatif à `git`, outils de contrôle de version que nous verrons en détails dans la partie sur GitHub
- `mypkgr.Rproj` qui est un fichier spécifique de Rstudio, et permet de définir les caractéristiques et préférences du projet que nous venons de créer
- `.Rbuildignore` qui permet d'ignorer certains fichiers au moment où on construira le package un peu plus loin (par exemple, le fichier `mypkgr.Rproj` ne doit pas être inclus dans le package)

1.2 Ajouter une fonction : exemple fil rouge

Nous vous proposons de coder la fonction suivante, que nous reprendrons tout au long de la formation :

Nous souhaitons calculer la valeur de la densité d'une loi normale multivariée sur $\mathbb{R}^p$ en n points. Notre fonction doit pouvoir s'appliquer pour n'importe quelle loi normale multivariée (vecteur de moyennes dans $\mathbb{R}^p$ et matrice de variance-covariance d'ordre de p quelconques), et on souhaite pouvoir calculer toutes les valeurs de la densité évaluées sur les n points en un seul appel de la fonction.

Vous devez donc créer une fonction `mvnpdf()` dans un fichier nommé `mvnpdf.R` dans le dossier `R` du package, qui :

- prend en arguments :
  - `x` une matrice, à n colonnes (les observations) et p lignes
  - `mean` un vecteur de moyennes
  - `varcovM` une matrice de variance-covariance
  - `Log` un paramètre logique valant `TRUE` par défaut
- renvoie une liste contenant la matrice `x` ainsi qu'un vecteur des images des points de `x` par la fonction de densité de la variable aléatoire de loi normale multivariée considérée.

A vous de jouer !

ATTENTION ! Si vous cliquez trop vite sur le lien ci-dessous, cela invalidera votre participation à la formation !

Voici une proposition de fonction que vous pouvez télécharger ici.

Pour des conseils lors de la rédaction de code, voir la page R code du site d'Hadley.

1.3 Documenter une fonction

Il est important de bien documenter votre code. Tout projet a au moins 2 développeurs :

- vous
- vous dans 6 mois

Par égard à votre futur moi, soyez sympas et prenez le temps de documenter votre code !

Nous vous conseillons vivement d'utiliser le package `roxygen2` pour documenter vos packages. L'avantage principale étant d'avoir l'aide d'une fonction dans le même fichier que le code définissant cette fonction.

A vous de jouer !

1. Commencer par insérer le squelette de l'aide grâce à "Insert Roxygen Skeleton" situé dans le menu "Code" ou le sous-menu *Baguette magique*
2. Compléter la documentation en renseignant :

- le titre de la fonction (première ligne)
- la description de ce que fait la fonction (deuxième paragraphe)
- si vous renseignez un troisième paragraphe, cette partie ira dans la section "Details" de la page d'aide
- la signification des paramètres
- la sortie, après la balise `@return`

3. Générer la documentation à l'aide de "Document" dans le menu "More" de l'onglet "Build" (ou Ctrl+Shift+D ou `devtools::document()`). L'effet de cette commande est multiple :

- un dossier `man` a été créé et à l'intérieur, un fichier `mvnpdf.Rd` a été créé et contient les informations de l'aide de la fonction
- le fichier `NAMESPACE` a été modifié

En cas de bug ou par curiosité ET une fois que vous avez terminé vous pouvez consulter cette proposition.

Pour plus de détails sur la documentation de package et les balises `roxygen2`, voir la page Object documentation du site d'Hadley.

Finissons par évoquer une fonction du package `devtools` qui initialise une page d'aide pour le package dans son ensemble :

```
devtools::use_package_doc()
```

La page d'aide générée sera alors accessible, une fois le package installé, via :

```
?mypkgr
```

1.4 Tester le package de manière *intéractive*

Pour tester le package, vous devez le charger dans R à l'aide de : dans l'onglet "Build", le menu "More" puis "Load All" (ou Ctrl+Shift+L ou `devtools::load_all()`).

Vous pouvez alors utiliser votre package directement dans R : consulter l'aide de la fonction avec `?mvnpdf` et par exemple exécuter les commandes renseignées dans la section exemple de cette page d'aide.

```
?mvndpf
```

Ainsi, lors du développement, vous pouvez :

- Ajouter/Modifier le code R
- Re-charger le package Ctrl+Shift+L
- Essayer dans la console
- Et ainsi de suite…

1.5 Tester le package de manière *automatique*

Pour initialiser la fonctionnalité de tests automatiques dans le package, utiliser :

```
devtools::use_testthat()
```

Cette commande induit la création d'un dossier `tests` qui comprend un fichier `testthat.R` - à ne pas modifier - et un dossier `testthat` dans lequel on va insérer nos tests. Cet outils s'appuie sur la théorie des *tests unitaires*.

Voici par exemple, le contenu d'un fichier qu'on appellera `test-univariate.R` à mettre dans le dossier `testthat` :

```
context("Univariate gaussian test")

test_that("correct result for univariate gaussian", {
  expect_equal(mvnpdf(x=matrix(1.96), Log=FALSE)$y, dnorm(1.96))
  expect_equal(mvnpdf(x=matrix(c(1.96, -0.5), ncol = 2), Log=FALSE)$y,
```

```
                dnorm(c(1.96, -0.5)))
})
```

Et un deuxième, appelé `test-bivariate.R` :

```
context("Bivariate gaussian test")

test_that("correct results for bivariate gaussian", {
  expect_equal(mvnpdf(x=matrix(rep(1.96,2), nrow=2, ncol=1), Log=FALSE)$y,
               mvtnorm::dmvnorm(rep(1.96, 2)))
})
```

Pour exécuter ces tests, on peut utiliser dans l'onglet "Build", le menu "More", "Test package" (ou `devtools::test()` ou Ctrl+Shift+T).

L'avantage de ces tests automatiques est qu'ils vont s'exécuter à chaque fois qu'on effectuera un *check* du package.

Une bonne pratique est d'ajouter un test unitaire à chaque fois qu'un bug est identifier et résolu, afin de pouvoir immédiatement identifier et prévenir qu'une erreur identique ne se reproduise dans le futur.

1.6 Faire un *check* du package

Faire un *check* signifie vérifier que tout est correct dans le package. Il est **nécessaire** de "passer" le *check* pour pouvoir déposer le package sur le CRAN.

Pour exécuter celui-ci, utiliser "Check" dans l'onglet "Build" (`devtools::check()` ou Ctrl+Shift+E).

Lors du *check*, les tests que nous avons mis au point précédemment sont exécutées. C'est justement l'avantage d'avoir fait ces tests, nous n'avons plus besoin de s'en préoccuper, mais juste de réagir en cas d'erreurs renvoyées.

1.7 Installer le package

Pour le moment, le package n'existe que dans l'environnement associé au projet Rstudio qu'on a créé. Pour pouvoir l'utiliser dans R de manière générale, il faut l'installer (comme un package du CRAN par exemple).

Pour faire ça, utiliser "Install and Restart" dans l'onglet "Build" (`devtools::install()` ou Ctrl+Shift+B).

Et enfin, vous pouvez configurer le comportement de Rstudio pour qu'au moment de l'installation, il documente en même temps le package : aller dans l'onglet "Build", le menu "More" puis "Configure Build Tools". Cliquer ensuite sur "Configure" puis cocher la case en bas "Build and Reload".

1.8 Annexe 1.1 : ajouter d'une méthode S3

Dans la plupart des packages on est amenés à implémenter des méthodes S3, très souvent pour qu'à partir d'un objet résultat `res`, on puisse exécuter `print(res)`, `summary(res)`, `plot(res)`…

Voici un exemple de méthode `plot()` qu'on peut ajouter dans notre package :

```
#' Plot of the mvnpdf function
#'
#' @param x an object of class \code{mvnpdf} resulting from a call of
#' \code{mvnpdf()} function.
```

```
#' @param ... graphical parameters passed to \code{plot()} function.
#'
#' @return Nothing is returned, only a plot is given.
#' @export
#'
#' @examples
#' pdfvalues <- mvnpdf(x=matrix(seq(-3, 3, by = 0.1), nrow = 1), Log=FALSE)
#' plot(pdfvalues)
plot.mvnpdf <- function(x, ...) {
  plot(x$x, x$y, type = "l", ...)
}
```

Attention ! Pour que cette méthode fasse bien ce qu'on veut quand on l'applique au résultat de notre fonction `mvnpdf()`, il faut déclarer que ce résultat est de classe `mvnpdf`.

Tester cette fonction, en exécutant l'exemple.
N'oubliez pas de re-charger le package (Ctrl+Shift+L), et de re-documenter le package (Ctrl+Shift+D).

Consulter le contenu du dossier `man` et les modifications qui ont été apportées au fichier `NAMESPACE`.

Voici une proposition de solution : le fichier contient le code complet de la fonction `mvnpdf()` et de la méthode `plot()` associée.

1.9 Annexe 1.2 : soumettre son package au CRAN

`devtools::check_cran()` puis `devtools::submit_cran()`

Chapter 2

Contrôle de version avec `git` et GitHub : hitorique de changement,

développement collaboratif et intégration continue.

Nous nous intéressons ici aux solutions proposées par RStudio et GitHub pour l'hébergement et le contrôle de version de projets.

2.1 Principe du contrôle de version

Le principe du contrôle de version est d'enregistrer les changements successifs apportés à des fichiers (notamment des fichiers R).

RStudio propose 2 solutions intégrées pour le contrôle de version :

- `git`
- `svn`

2.1.1 `git`

`git` est un logiciel de contrôle de version (c'est-à-dire un outils qui va enregistrer l'histoire des changements successifs de votre code et permettre de partager ces changements avec d'autres personnes). `git` est un logiciel en ligne de commande, et sa prise en main n'est pas nécessairement intuitive.

`git` fonctionne de la façon suivante : sur un serveur dans le 'cloud', une version **à jour** du code est disponible. À tout moment il est possible d'accéder à cette version du code en ligne. Chaque contributeur peut télécharger cette dernière version **à jour** (dans une action que l'on dénomme *pull*), avant de l'éditer localement. Une fois ses changements effectués, le contributeur peut alors mettre à jour la version en ligne du code afin que ses changements soient disponibles pour tout le monde (dans une action que l'on dénomme *push*)

NB : `git` a été pensé pour des fichiers légers (comme par exemple des fichiers texte) et est loin d'être optimisé pour des fichiers trop lourds et/où compressés.

2.1.2 `subversion`

`subversion` est l'autre solution disponible dans RStudio. Elle fonctionne de manière similaire à git, mais avec des fonctionnalités un peu plus réduites que nous détaillons pas ici (la différence majeure est que tout

les contributeurs travaillent simultanément sur la même version du code).

2.2 Utiliser `git` localement depuis RStudio

A vous de jouer !

1. Commencez par activer `git` depuis l'onglet "Git/SVN" de "Project Options" situé dans le menu "Tools" et suivre les instructions.
2. À partir de l'onglet "Git" maintenant apparu à côté de l'onglet "Build", enregistrer l'état actuel de votre package en réalisant votre premier "commit"
3. à partir de l'onglet "Git" maintenant apparu à côté de l'onglet "Build", enregistrer l'état actuel de votre package en réalisant votre premier "commit" :

- 3a. sélectionner les fichiers à suivre (ne pas sélectionner le fichier .Rpoj)
- 3b. écrire un message informatif (pour vos collaborateurs - ce qui inclut votre futur vous)
- 3c. cliquer sur "Commit"

4. ajouter une ligne "*.Rproj" au fichier ".gitignore" et effectuez un nouveau commit
5. visualiser les changements et leur historique à l'aide des outils de visualisation "Diff" et "History" accessible depuis l'onglet "Git"

2.2.1 Bonnes pratiques du *commit*

Idéalement, chaque commit ne devrait régler qu'un seul problème. Il devrait le régler dans son intégralité (être **complet**) et ne contenir des changements relatifs qu'uniquement à ce problème (être **minimal**). Il est alors important d'écrire des message de commit **informatifs** (pensez à vos collaborateur, qui incluent votre **futur vous**). Il faut également être concis, et décrire les raisons des changements plutôt que les changements eux-mêmes (visibles dans le *Diff*). Il est parfois difficile de respecter ces directives à la lettre, et celles-ci ne sont qu'un guide et ne doivent pas vous empêcher d'effectuer des *commits* réguliers.

Par ailleurs, la tentation d'avoir un historique de changements "propre" et bien ordonné est naturelle, mais se révèle une source de problèmes inutiles. Elle entre en contradiction avec l'objectif de traçabilité du contrôle de version. Le développement de code étant généralement un processus intellectuel complexe et non linéaire, il est normal que l'enregistrement des changements reflète ce cheminement. En pratique, votre futur-vous sera le premier utilisateur de votre historique de changements et la priorité est donc de vous faciliter la tache dans le futur lors de la résolution de bug où l'extension de fonctionnalités.

2.3 GitHub

GitHub est un site internet proposant une solution d'hébergement de code en ligne, et s'appuyant sur `git`. Il existe de nombreux sites web et services (gitlab, bitbucket, …) permettant d'héberger du code et s'appuyant sur `git`. GitHub est très populaire dans la communauté des développeurs `R`, et est relativement facile à utiliser, même pour un utilisateur novice.

Les avantages d'utiliser GitHub :

- une interface graphique simple pour suivre l'historique des changements de votre code
- la dernière version de développement de votre code est disponible en ligne et vous pouvez la référencer (on peut même référencer un numéro de commit précis pour geler une version spécifique du code)

- les utilisateurs disposent d'un canal clair et transparent pour signaler les bugs/difficultés
- cela facilite grandement le développement collaboratif

2.3.1 Mettre son package `R` sur GitHub

A vous de jouer !

1. rendez vous sur le site https://github.com/ et créez vous un compte GitHub (si vous hésitez, une convention courante est d'utiliser *prénomnom* comme nom d'utilisateur)
2. ouvrez le client "GitHub desktop" sur votre machine et connectez vous à votre compte GitHub.
3. ajouter un nouveau projet local en cliquant sur l'icone "+" en haut à gauche de la fenêtre du client, puis en choisissant "Add" et en rentrant le chemin du dossier où se trouve le code de votre package.
4. une fois le repertoire créer en local, publiez le sur GitHub en cliquant sur "Publish" en haut à droite de la fenêtre du client. Vérifiez sur le site de GitHub que votre code à bien été *uploadé* avec les 2 commits précédents.
5. Ajouter un fichier "README.Rmd" à votre package afin de disposer d'une belle page d'accueil sur GitHub :

- 5a. dans RStudio, executez la commande `devtools::use_readme_rmd()`
- 5b. à l'aide de l'outilds "Diff" de l'onglet "Git" de RStudio, étudier les changements opérer par la commande précédente
- 5c. éditez le fichier "README.Rmd" créé, puis créer le fichier README.md correspondant en executant `knitr` (cliquer su la pelotte de laine "Knit" en haut à gauche dans Rstudio), avant d'effectuer un 3^{e} commit contenant ces changements
- 5d. à ce stade, si vous visitez la page de votre répertoire sur GitHub, votre 3^{e} commit n'apparait pour l'instant pas. Il faut synchroniser le répertoire GitHub en ligne avec votre dossier local. Pour cela, vous avez 2 solutions : soit utiliser le bouton "Sync" en haut à droit de la fenêtre du client GitHub desktop ; soit directement depuis RStudio en cliquant sur "Push" depuis l'onglet "Git". Maintenant, les changement du 3^{e} commit sont visibles en ligne dur GitHub.

2.4 Collaboration pour la production du code

`git` et GitHub sont particulièrement efficaces lorsque plusieurs personnes collabore pour développer un code. En effet, chacun peut effectuer des *pull* et *push* successifs pour apporter des changements au code, de manière simultanée et en étant sûr de toujours travailler sur la dernière version. Nous allons voir différents concepts utiles dans le cas d'un tel travail collaboratif.

A vous de jouer !

1. En formant des groupes de 2, vous allez chacun ajouter votre binome comme "collaborator" à votre repertoire GitHub à partir de l'onglet "Settings" (sur GitHub).
2. Quelques instants plus tard le collaborateur ainsi ajouté reçoit un email l'invitant à accepter l'ajout. Cliquer sur le lien et accepter.
3. Dans le client "GitHub desktop", ajouter le répertoire de votre binôme en cliquant sur l'icone "+" en haut à gauche et en selectionnant "Clone", ce qui fait apparaitre la liste des

repertoires associés à votre compte GitHub non liés à un dossier local. Sélectionner le projet de votre binome.

2.4.1 *Branches*

Une des fonctionnalités assez utile de git est les *branches*. Cela permet d'opérer des changements importants dans le code sans perturber le fonctionnement actuel. C'est notamment utile pour explorer une piste de développement dont on ne sait pas si elle sera concluante au final.

D'ailleurs, vous utilisez déjà les branches depuis le depuis de cette partie. En effet, la branche par défaut est appelé "master".

Grâce à ce système de branches, on obtient un arbre des différents *commits* au cours du temps (où les nœuds correspondent à la séparations des branches).

2.4.2 *Merge*

Un *pull* se décompose en 2 actions de la part de `git` :

1. tout d'abord un *fetch*, qui correspond au téléchargement du code en ligne
2. suivi d'un *merge* qui fusionne la version locale avec les changements.

Après avoir conduit un développement expérimental dans une branche, on peut vouloir *merger* ces changements dans la branche "master" par exemple, après que l'expérience se soit révélée concluante."

Si un les changements concernent des parties distinctes du code, alors le *merge* peut s'effectuer sans problème. En revanche si les 2 versions à *merger* comportent des changements depuis leur dernier "commit* commun qui concerne les mêmes lignes de codes, alors on va rencontrer un (ou des) conflit(s), qu'il va falloir résoudre.

2.4.3 Les conflits

Prenons l'exemple suivant : le développeur D_1 et le développeur D_2 on tous les 2 *pullé* la version v0.1 du code à l'instant t sur leur machine respective. Ils travaillent chacun indépendamment pour apporter des changements au code. Au moment de *pusher* ses changements, le développeur D_2 reçoit un message d'erreur :

"Sync Error.
Please resolve all conflicted files, commit, then try syncing again."

Chaque fichier étant source de conflit a alors été automatiquement édité comme suit :

```
<<<<<<< HEAD
code dans votre version local
=======
code en ligne
>>>>>>> remote
```

Pour résoudre le conflit, il faut alors éditer chaque fichier un à un en choisissant s'il faut conserver la version locale ou bien celle en ligne, avant de pouvoir *commiter* à nouveau et enfin de *pusher* vos changements avec succès.

A vous de jouer !

1. Modifiez le fichier `README.Rmd` de votre binome, puis *commitez* votre changement et *pushez* le.

2. une fois que votre binôme a modifié votre `README.Rmd`, modifiez à votre tour le fichier à la même ligne, **SANS** *puller* les changements de votre binôme au préalable ! *Commitez* et essayez de *pusher* ces changements.
3. Résolvez le conflit.

2.4.4 *Fork*

L'action *fork* permet de créer une copie qui vous appartient à partir d'un code disponible. Ainsi le code original ne sera pas impacté par vos changements. Cela revient à créer une branche, et la séparer de l'arbre pour pouvoir en assumer la propriété.

Cette action est principalement utile dans le cadre des *pull requests*.

2.4.5 *Pull request*

Il s'agit du moyen le plus facile de proposer des changements dans un code dont vous n'êtes pas collaborateur. GitHub propose une interface graphique facilitant leur traitement.

A vous de jouer !

1. Modifiez le `README.Rmd` de votre voisin qui n'est pas votre binôme après avoir *forké* son package.
2. Proposez votre changement sous la forme d'une *pull request*.
3. Acceptez la *pull request* sur le site de GitHub et faire le *merge*.

2.4.6 *Issues*

Pour n'importe quel répertoire GitHub, vous pouvez poster un commentaire sous forme d'*issue* afin d'alerter les développeurs sur un éventuel bug, ou une question sur l'utilisation du package, ou encore demander une fonctionnalité supplémentaire…

L'idéal est de proposer vous-même une *pull request* qui résout votre *issue* lorsque vous le pouvez (i.e. en avez les capacités et le temps).

A vous de jouer !

1. Utilisez `devtools::use_github_links()` afin d'ajouter les 2 lignes suivantes au fichier `DESCRIPTION` de votre package
   `URL: http://github.com/*prenom.nom*/mypkg`
   `BugReports: http://github.com/*prenom.nom*/mypkg/issues`
   grâce à la fonction `devtools::use_github_links()`
2. Visualisez les nouveau changements, puis *commitez* les.
3. Créez une *issue* sur le projet de votre binome

2.5 Intégration continue

À chaque changement, à chaque *commit* donc, il y a la possibilité d'introduire 1 (ou plusieurs) bugs qui vont empêcher le package de passer le *CRAN check*. Si l'on accumule trop de ces bugs, au moment de soumettre la nouvelle version, il peut y avoir beaucoup de corrections à apporter. C'est d'autant plus frustrant si le package passait le *CRAN check* auparavant…

Les services d'intégration continue permettent de *checker* votre package **automatiquement** après chaque *commit* ! En cas d'échec, vous recevez un mail qui vous en informe. Un certain nombre de ces services proposent une offre limitée gratuite pour les projets open-source.

Une autre raison d'utiliser l'intégration continue est qu'elle permet de tester votre package sur des infrastructures différentes de la votre (e.g. Windows, Ubuntu, Mac OS) et pour différentes versions de `R` (*current*, *devel*...)

2.5.1 Travis CI

Travis est un service d'intégration continue (*Continuous Integration*), qui permet de *checker* votre package à chaque *commit* sous Ubuntu. La commande `devtools::use_travis()` initialise le fichier de configuration `.travis.yml` nécessaire.

A vous de jouer !

1. Rendez vous sur le site https://travis-ci.org/ et créez vous un compte associé à votre GitHub en cliquant sur le bouton "SignIn with GitHub" en haut à droite.
2. Activez votre repertoire `mypkg` sur Travis
3. executez la commande `devtools::use_travis()` et *commitez* les changements et regardez ce qu'il se passe sur votre page Travis
4. ajouter un joli badge à votre README.md grâce au code suivant (obtenu dans la console R) :
   `[![Travis-CI Build Status](https://travis-ci.org/*prenomnom*/mypkgr.svg?branch=master)](https:/`
   et *commitez les changements*

Travis permet également de tester votre package sous Mac OS (même si ce service est parfois moins stable).

5. Ajoutez les lignes suivantes dans le fichier de configuration `.travis.yml` :

```
r:
 - release
 - devel
os:
 - linux
 - osx
```

N'hésitez pas à aller visiter les pages de packages connus sur GitHub pour observer comment ils configurent leur fichier `.travis.yml`.

2.5.2 Appveyor

Appveyor est l'analogue de Travis CI mais pour Windows.

A vous de jouer !

1. Rendez vous sur le site https://ci.appveyor.com/ et créez vous un compte associé à votre GitHub en cliquant sur le bouton "GitHub" pour le *Login*
2. Ajoutez votre repertoire `mypkg` sur Appveyor en cliquant sur "+ New project" en haut à gauche, puis en selectionnant GitHub et mypkg dans le menu.
3. Executez la commande `devtools::use_appveyor()` et *commitez* les changements et regardez ce qu'il se passe sur votre page Appveyor

4. ajouter un joli badge à votre README.md grâce au code suivant (obtenu dans la console R) :
`[![AppVeyor Build Status](https://ci.appveyor.com/api/projects/status/github/*prenomnom*/mypkgr`
et *commitez les changements*

2.5.3 R-hub

Le *R consortium* met à disposition le R-hub builder, et a pour ambition de bientôt proposer un service d'intégration continue spécialement dédié aux packages `R`.

2.6 Annexe 2.1 : couverture du code

Le package `covr` propose une solution pour mesurer la couverture des tests unitaires associés à un package. La couverture de test détermine la proportion du code source qui est effectivement utilisée lors de l'exécution des tests unitaires. La mesure de la couverture du code renforce la fiabilité d'un code et donne confiance à ses utilisateurs potentiels.

A vous de jouer !

1. Executez la commande `devtools::use_appveyor()`, ajouter un joli badge à votre README.md grâce au code suivant (obtenu dans la console R) :
`[![Coverage Status](https://img.shields.io/codecov/c/github/*prenomnom*/mypkgr/master.svg)](htt`
Et ajouter au fichier `.travis.yml`:

```
after_success:
  - Rscript -e 'covr::codecov()'
```

2. *Commitez* ces changements.

Pour plus d'information n'hésitez pas à consulter la vignette de `covr`.

Chapter 3

Mesurer et comparer des temps d'exécution

La première étape avant d'optimiser un code est de pouvoir mesurer son temps d'exécution, afin de pouvoir comparer les temps d'exécution entre différente implémentations.

3.1 Mesurer des temps d'exécution avec `system.time()`

Pour mesure le temps d'exécution d'une commande R, on peut utiliser la fonction `system.time()` comme ceci :

```
system.time(mvnpdf(x=matrix(rep(1.96, 2), nrow=2, ncol=1), Log=FALSE))
```

```
##    user  system elapsed
##   0.001   0.001   0.004
```

Le problème qui apparaît sur cet exemple est que l'exécution est tellement rapide que `system.time()` affiche `0` (ou une valeur très proche). De plus, on voit qu'il y a une certaine variabilité quand on relance plusieurs fois la commande.

Ainsi si on souhaite comparer notre code avec la fonction `mvtnorm::dmvnorm()`, on ne peut pas utiliser `system.time()` :

```
system.time(mvtnorm::dmvnorm(rep(1.96, 2)))
```

```
##    user  system elapsed
##   0.005   0.001   0.009
```

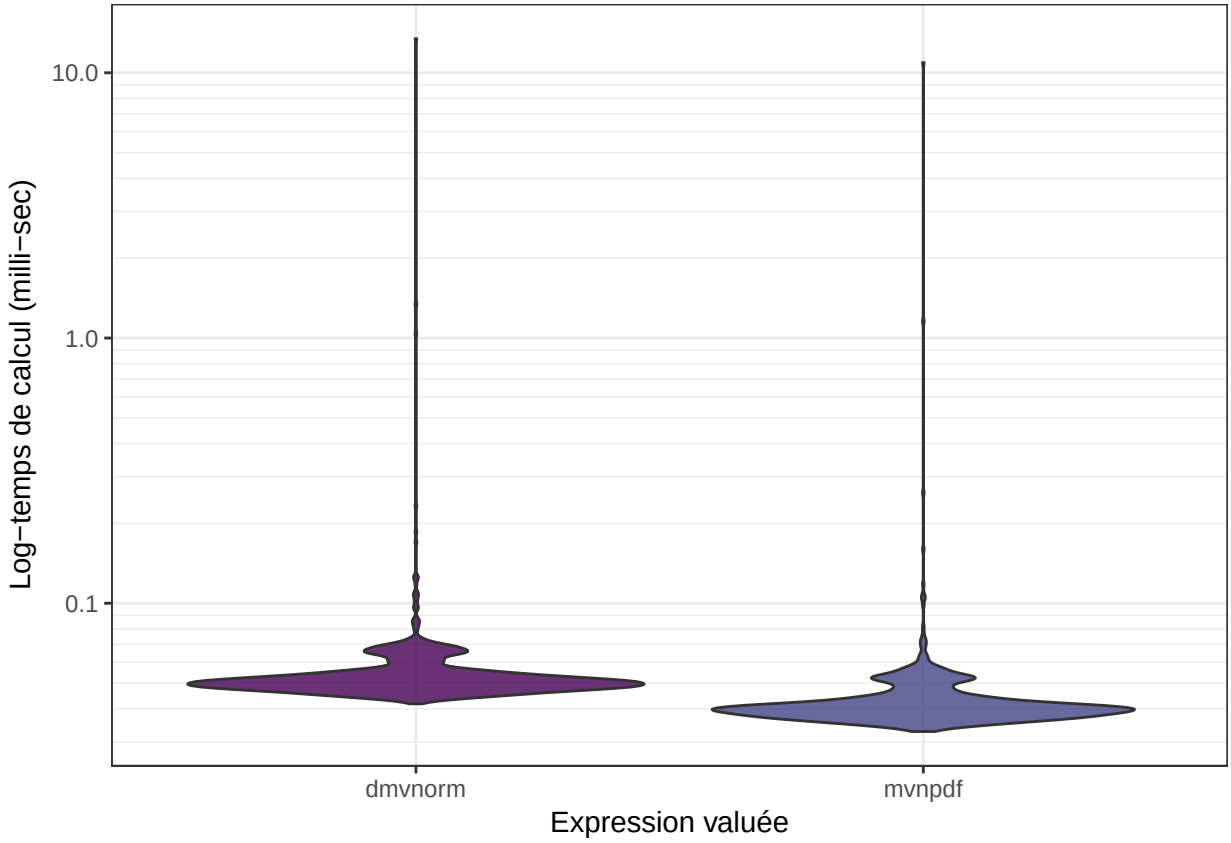
On pourrait se dire qu'il faut augmenter la complexité de notre calcul, mais il y a mieux : utiliser le package `microbenchmark` !

3.2 Comparer des temps d'exécution avec `microbenchmark()`

Comme son nom l'indique, ce package permet justement de comparer des temps d'exécution même quand ceux-ci sont très faibles. De plus, la fonction `microbenchmark()` va répéter un certain nombre de fois l'exécution des commandes et donc va stabiliser le résultat.

```
library(microbenchmark)
mb <- microbenchmark(mvtnorm::dmvnorm(rep(1.96, 2)),
                     mvnpdf(x=matrix(rep(1.96,2)), Log=FALSE),
                     times=1000L)
mb
```

```
## Unit: microseconds
##                                          expr    min     lq     mean
##                 mvtnorm::dmvnorm(rep(1.96, 2)) 41.777 48.060 69.37492
##  mvnpdf(x = matrix(rep(1.96, 2)), Log = FALSE) 32.846 37.787 54.58270
##  median      uq      max neval cld
##   50.71 54.8220 13446.74  1000   a
##   40.16 43.7665 10930.70  1000   a
```

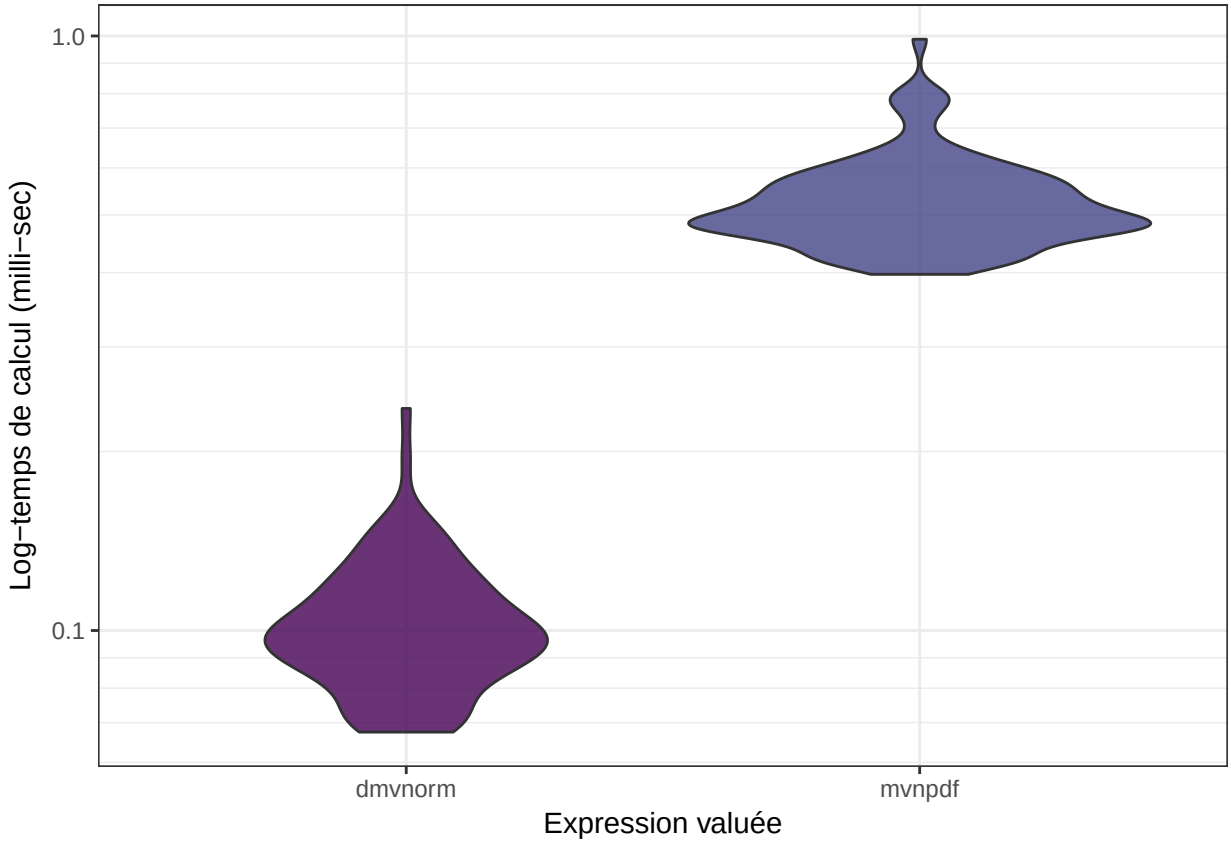


Les deux fonctions `mvnpdf()` et `dmnvorm()` étant capables de prendre en entrée une matrice, on peut également comparer leurs comportements dans ce cas :

```
n <- 100
mb <- microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                     mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     times=100L)
mb
```

```
## Unit: microseconds
##                                                    expr     min
##         mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))  67.482
##  mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 397.244
##        lq     mean   median      uq     max neval cld
```

```
##   85.8100 103.0665  97.4795 116.399 236.320   100  a
##  473.0355 527.4503 503.3420 564.096 987.397   100   b
```



Il s'est passé un quelque chose… Et on va diagnostiquer ce problème dans la suite.

Chapter 4

Profiler son code

On parle de *profiling* en anglais. Il s'agit de déterminer ce qui prend du temps dans un code. Le but étant une fois trouvé le bloc de code qui prend le plus de temps dans l'exécution d'optimiser uniquement cette brique.

Pour obtenir un *profiling* du code ci-dessous, sélectionner les lignes de code d'intérêt et aller dans le menu "Profile" puis "Profile Selected Lines" (ou Ctrl+Alt+Shift P).

```
n <- 10e4
pdfval <- mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE)
```

OK, we get it ! Concaténer un vecteur au fur et à mesure dans une boucle n'est vraiment pas une bonne idée.

4.1 Comparaison avec une version plus habile de `mnvpdf()`

Considérons une nouvelle version de `mvnpdf()`, appelée `mvnpdfsmart()`. Télécharger le fichier puis l'inclure dans le package.

Profiler la commande suivante :

```
n <- 10e4
pdfval <- mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE)
```

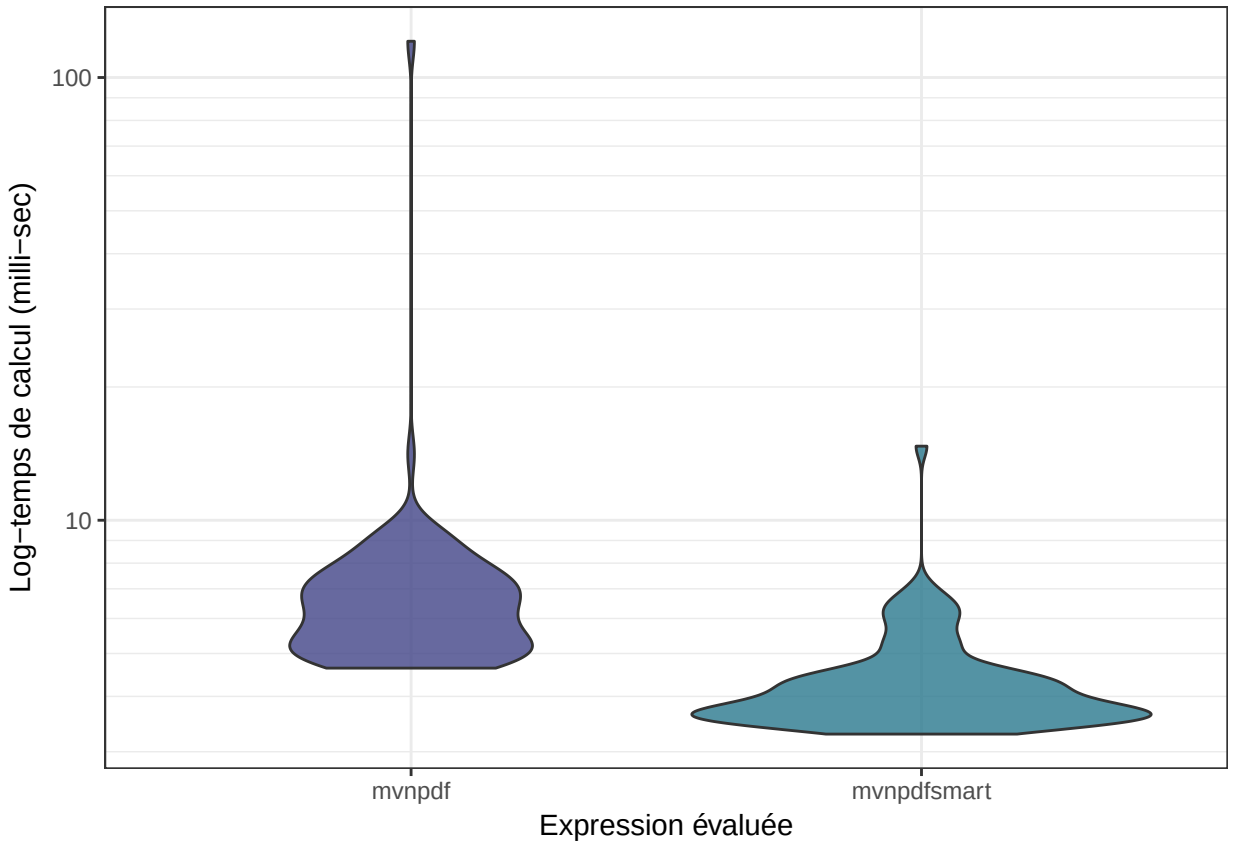
On a effectivement résolu le problème et on apprend maintenant de manière plus fine ce qui prend du temps dans notre fonction.

Pour confirmer que `mvnpdfsmart()` est effectivement bien plus rapide que `mvnpdf()` on peut re-faire une comparaison avec `microbenchmark()` :

```
n <- 1000
mb <- microbenchmark(mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     times=100L)
mb
```

```
## Unit: milliseconds
##                                                          expr      min
##       mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 4.634469
##  mvnpdfsmart(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 3.288446
##        lq     mean   median       uq       max neval cld
```

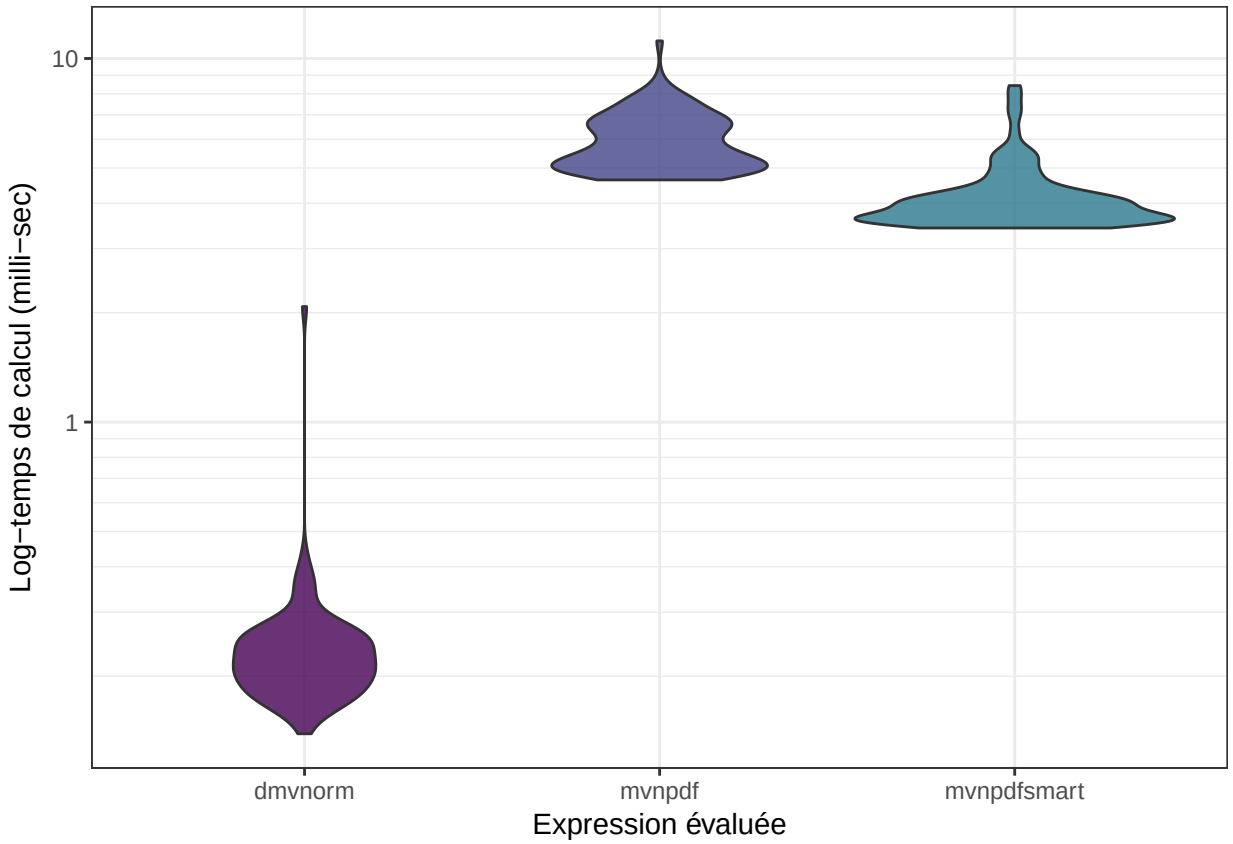
```
##  5.206888 7.599990 6.266242 7.454142 120.75959   100   b
##  3.621145 4.336717 3.904676 4.512889  14.70005   100  a
```



Et on peut également voir si on devient compétitif avec `dmvnorm()` :

```
n <- 1000
mb <- microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                     mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     times=100L)
mb
```

```
## Unit: microseconds
##                                                          expr      min
##              mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))  138.951
##       mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 4635.767
##  mvnpdfsmart(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 3418.769
##        lq      mean    median        uq       max neval cld
##   187.882  246.6448  219.5445  257.7615  2078.450   100 a
##  5078.005 5957.4809 5622.3955 6733.6165 11178.180   100   c
##  3610.798 4177.1537 3924.1510 4292.6860  8422.558   100  b
```



Il y a encore du travail...

4.2 Comparaison avec une version optimisée dans R

Boris est arrivée après plusieurs recherches et tests à une version optimisée avec les outils de R.

Inclure la fonction `mvnpdfoptim()` dans le package, puis profiler cette fonction :

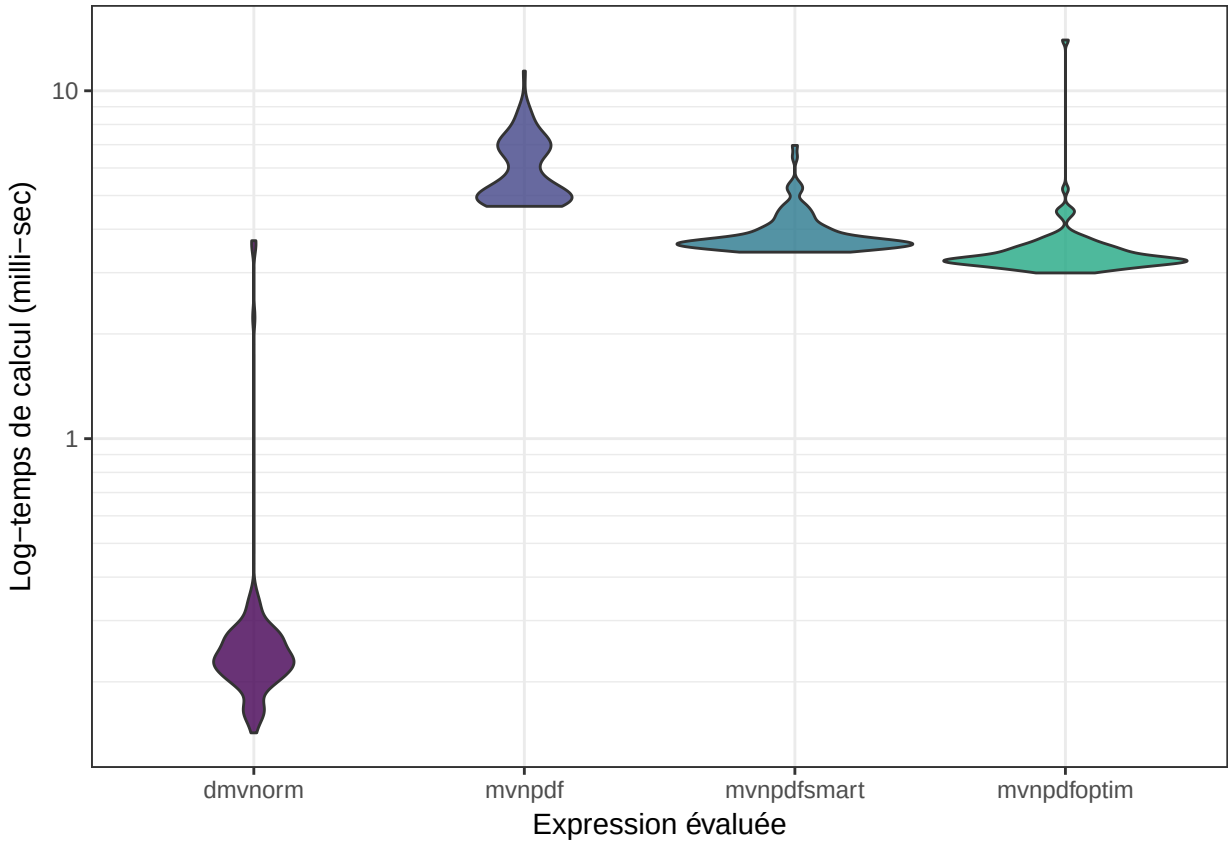
```
n <- 10e4
pdfval <- mvnpdfoptim(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE)
```

Et un petit `microbenchmark()` :

```
n <- 1000
mb <- microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                     mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfoptim(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     times=100L)
mb
```

```
## Unit: microseconds
##                                                        expr      min
##             mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))  142.695
##       mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 4653.811
##  mvnpdfsmart(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 3436.588
##  mvnpdfoptim(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 2994.890
##        lq      mean    median       uq       max neval  cld
```

```
##   208.464  323.8774  233.2215  266.453  3707.693   100 a
##  4865.005 5960.0235 5350.2790 6939.080 11395.828   100    d
##  3585.891 3915.2049 3695.5255 4020.365  6965.204   100   c
##  3204.261 3507.1549 3281.6170 3522.945 13993.819   100  b
```



Pour finir on peut profiler la fonction `dmvnorm()` :

```
n <- 10e5
pdfval <- mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))
```

Chapter 5

Rcpp ou comment intégrer facilement du code C++dans un package R

Rcpp (R-C-Plus-Plus) est un package qui facilite l'interface entre C++ et R. R est un langage interprété, ce qui facilite un certain nombre de choses (notamment nous donne accès à la console dans laquelle on peut évaluer du code à la volée). Néanmoins, cette facilité d'utilisation se compense entre autre par des temps de calcul supérieurs à ceux de langages de plus bas niveau, tels que C, Fortran et C++ (mais qui nécessitent eux une compilation).

On dirigera le lecteur curieux vers le livre en ligne *Rcpp for everyone* de Masaki E. Tsuda, qui constitue une ressource très complète pour comprendre l'utilisation de Rcpp en plus de l'introduction que l'on peut trouver dans le livre *Advanced R* d'Hadley Wickham.

5.1 Première fonction en Rcpp

A vous de jouer !

1. Afin de rendre votre package prêt pour l'utilisation avec Rcpp, commencez par executer la commande suivante :

```
devtools::use_rcpp()
```

2. Constatez les changements apportés
3. il faut également ajouter les 2 commentaires roxygen suivants dans la page d'aide du package dans son ensemble :

```
#' @useDynLib mypkgr
#' @importFrom Rcpp sourceCpp, .registration = TRUE
NULL
```

Nous allons maintenant créer une première fonction en Rcpp permettant d'inverser une matrice. Pour cela, nous allons nous appuyer sur la library C++ Armadillo. Il s'agit d'une *library* d'algèbre linéaire moderne et simple, hautement optimisée, et interfacée avec R via le package RcppArmadillo.

C++ n'est pas un langage très différent de R. Les principales différences qui nous concernent :

- C++est très efficaces pour le boucles *for* (y compris les boucles for emboîtées). Attention : il y a souvent un sens qui est plus rapide que l'autre (ceci est dû à la manière dont C++ attribue et parcours la mémoire).

- Chaque commande doit se terminer par un point virgule ';'

- C++est un langage typé : il faut déclarer le type de chaque variable avant de pouvoir l'utiliser.

A vous de jouer !

1. Créez un nouveau fichier C++ depuis RStudio (via le menu File > New File > C++ File), et enregistrez le dans le dossier src. Prenez le temps de le lire et essayez de comprendre chaque ligne.

2. Compilez et chargez votre package (via le bouton "Install and Restart") et essayez d'utiliser la fonction timesTwo() depuis la console.

3. Installez le package RcppArmadillo, et n'oubliez pas de faire les ajouts nécessaires dans DESCRIPTION (cf. Rcpp précédement - vous pouvez expérimentez avec la fonction RcppArmadillo::RcppArmadillo.package.skeleton() qui a le désavantage de créer beaucoup de fichiers inutiles)

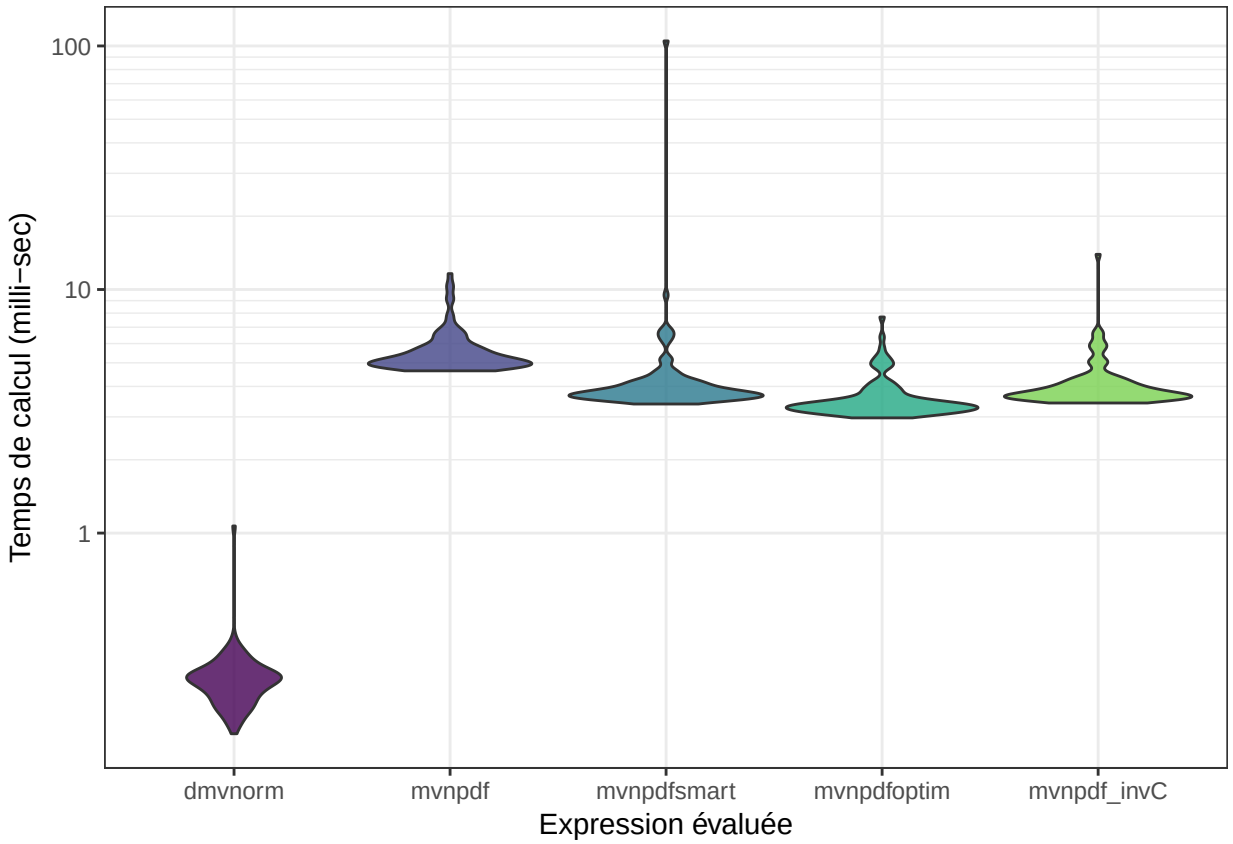
4. À l'aide de la documentation des packages Rcpp et RcppArmadillo de celle de la library Armadillo, tentez d'écrire une courte fonction invC en C++ calculant l'inverse d'une matrice.

5. Lorsque vous avez réussi à compiler votre fonction invC et qu'elle est accèssible depuis R créer une fonction mvnpdf_invC() à partir de l'implémentation de mvnpdfsmart en remplaçant uniquement les calculs d'inverse matriciel par un appel à invC.

6. Evaluer le gain en performance de cette nouvelle implémentation mvnpdf_invC

```
n <- 1000
mb <- microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                     mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfoptim(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdf_invC(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     times=100L)
mb
```

```
## Unit: microseconds
##                                                          expr      min
##              mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))  150.025
##       mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 4635.483
##  mvnpdfsmart(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 3388.023
##  mvnpdfoptim(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 2971.749
##  mvnpdf_invC(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE) 3420.709
##        lq      mean   median       uq        max neval cld
##   212.932  253.8442  250.043  271.625   1069.619   100 a
##  4876.709 5612.1058 5167.281 5784.028  11620.816   100   c
##  3605.399 5096.6506 3796.157 4169.661 104875.664   100  bc
##  3193.479 3608.7400 3352.521 3663.050   7712.733   100  b
##  3601.569 4140.9298 3770.406 4187.278  13944.776   100  bc
```




```
profvis::profvis(mvnpdfoptim(x=matrix(1.96,
    nrow = 2, ncol = 1000), Log=FALSE))
profvis::profvis(mvnpdfoptim(x=matrix(1.96,
    nrow = 100, ncol = 1000), Log=FALSE))
```

5.2 Optimisation grâce à C++

En règle générale, on ne gagne pas beaucoup en temps de calcul en remplaçant une fonction R optimisée par une fonction en C++. En effet, la plupart des fonctions de base de R s'appuie en réalité déjà sur des routines C ou Fortran bien optimisée. Le gain se limite alors simplement à la suppression des vérifications des arguments et de la gestion des différents types.

A vous de jouer !

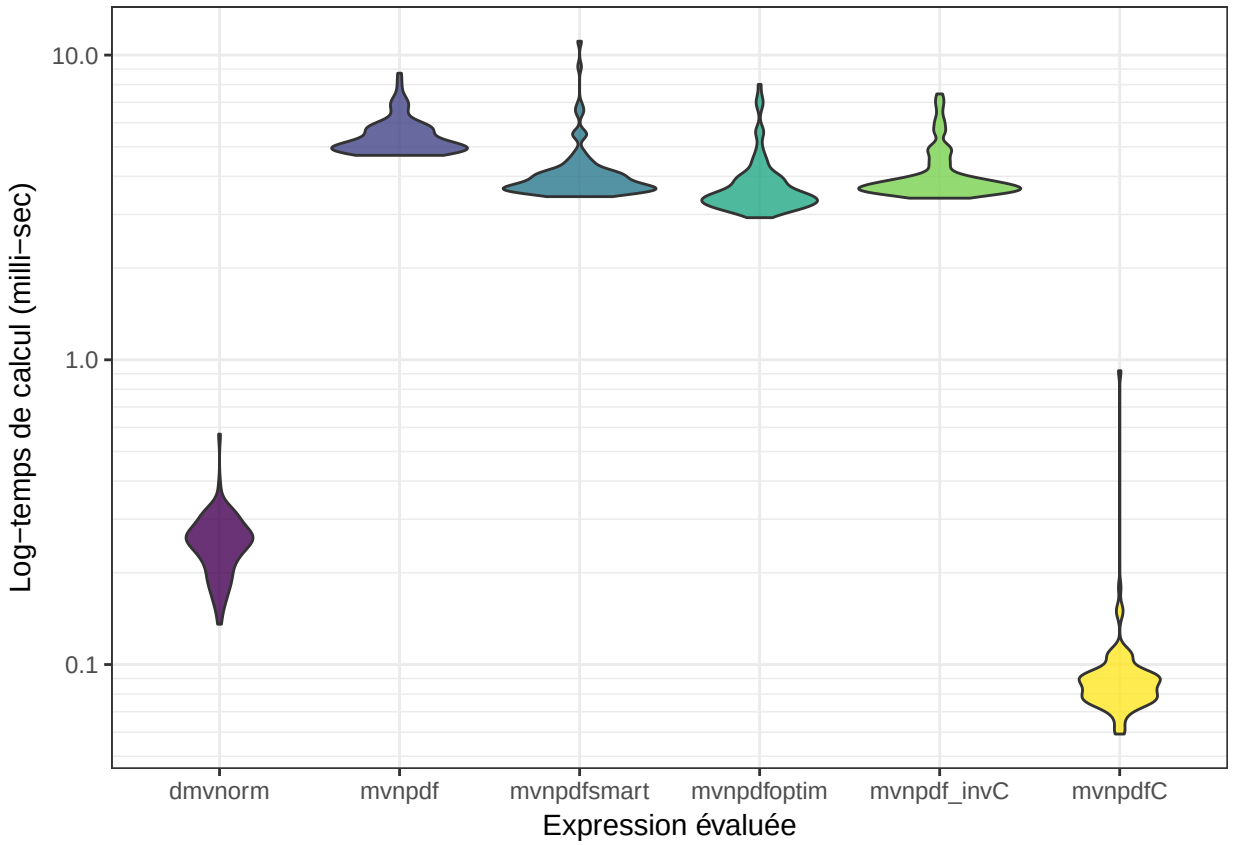
1. À partir de `mvnpdfsmart`, proposez une implémentation completement en C++ du calcul de densité de la loi Normale multivariée `mvnpdfC()`.
2. Evaluer le gain en performance de cette nouvelle implémentation `mvnpdf_invC`

Vous pouvez télécharger notre proposition de `mvnpdfC.cpp` ici.

```
n <- 1000
mb <- microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                     mvnpdf(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfsmart(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfoptim(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdf_invC(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                     mvnpdfC(x=matrix(1.96, nrow = 2, ncol = n), mean = rep(0, 2), varcovM = diag(2), L
```

```
                      times=100L)
mb
```

```
## Unit: microseconds
##                                                                                 expr
##                                        mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))
##                                     mvnpdf(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##                                mvnpdfsmart(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##                                mvnpdfoptim(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##                                mvnpdf_invC(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##  mvnpdfC(x = matrix(1.96, nrow = 2, ncol = n), mean = rep(0, 2),    varcovM = diag(2), Log = FALSE)
##       min       lq       mean   median        uq       max neval  cld
##   135.609  215.264  250.56381  251.547  280.2630   570.377   100 a
##  4687.518 4890.295 5428.58398 5142.587 5756.0445  8718.142   100    d
##  3431.621 3619.650 4128.27395 3804.160 4123.1495 11103.888   100   c
##  2926.824 3262.829 3751.33631 3440.896 3902.2660  8011.456   100  b
##  3391.585 3587.412 4098.44866 3749.930 4146.6030  7455.592   100   c
##    59.149   76.903   96.44677   85.956   93.0575   920.902   100 a
```



À noter que vous pouvez utiliser des fonctions Rcpp en dehors de l'architecture d'un package grâce à la fonction `Rcpp::sourceCpp()`. Mais comme nous avons qu'il est préférable de gérer tous ces code sous la forme de package, il est peu probable que vous en ayez besoin !

5.3 Annexe 6.1 : l'optimisation prématurée n'est pas une bonne idée

Chambers, *Software for Data Analysis: Programming with R*, Springer, 2008 :

> *Including additional C code is a serious step, with* ***some added dangers*** *and often a* ***substantial amount of programming and debugging*** *required.* ***You should have a good reason.***

Chapter 6

Parallélisation du code R

6.1 Introduction à l'execution parallèle sous R

En dehors de l'optimisation du code et des algorithmes, une autre façon d'obtenir un code performant est de tirer profit des architectures parallèles des ordinateurs modernes. Il s'agit alors de **paralléliser** son code afin de faire des opérations simultanées sur des parties distinctes d'un même problèmes, en utilisant différent cœurs de calcul. On ne réduit pas le temps de calcul total nécessaire, mais l'ensemble des opérations s'exécute plus rapidement.

Il existe un nombre non négligeable d'algorithmes qui sont d'un "parallélisme embarrassant", c'est-à-dire dont les calculs peuvent se décomposer en plusieurs sous-calculs indépendants. En statistique, il est ainsi souvent facile et direct de paralléliser selon les différentes observations ou selon les différentes dimensions. Typiquement, il s'agit d'opérations que l'on peut écrire sous la forme de boucle dont les opérations sont indépendantes d'une itération de la boucle à l'autre.

Les opérations nécessaires pour l'établissement d'un code parallèle sont les suivantes :

1. Démarrer m processus "travailleurs" (i.e. cœurs de calcul) et les initialiser
2. Envoyer les fonctions et données nécessaires pour chaque tache aux travailleurs
3. Séparer les taches en m opérations d'envergure similaire et les envoyer aux travailleurs
4. Attendre que tous les travailleurs aient terminer leurs calculs et obtenir leurs résultats
5. Rassembler les résultats des différents travailleurs
6. Arrêter les processus travailleurs

Selon les plateformes, plusieurs protocoles de communications sont disponibles entre les cœurs. Sous les systèmes UNIX, le protocole *Fork* est le plus utilisé, mais il n'est pas disponible sous Windows où on utilise préférentiellement le protocole *PSOCK*. Enfin, pour les architecture de calcul distribuée où les cœurs ne se trouvent pas nécessairement sur le même processeur physique, on utilise généralement le protocole *MPI*. L'avantage des packages `parallel` et `doParallel` est que la même syntaxe permettra d'exécuter du code en parallèle quelque soit le protocole de communication retenu.

Il existe un nombre important de packages et d'initiatives permettant de faire du calcul en R. Depuis `R 2.14.0`, le package `parallel` est inclus directement dans `R` et permet de démarrer et d'arrêter un "cluster" de plusieurs processus travailleur (étape 1). En plus du package `parallel`, on va donc utiliser le package `doParallel` qui permet de gérer les processus travailleurs et la communication (étapes 1) et l'articulation avec le package `foreach`qui permet lui de gérer le dialogue avec les travailleurs (envois, réception et rassemblement des résultats - étapes 2, 3, 4 et 5).

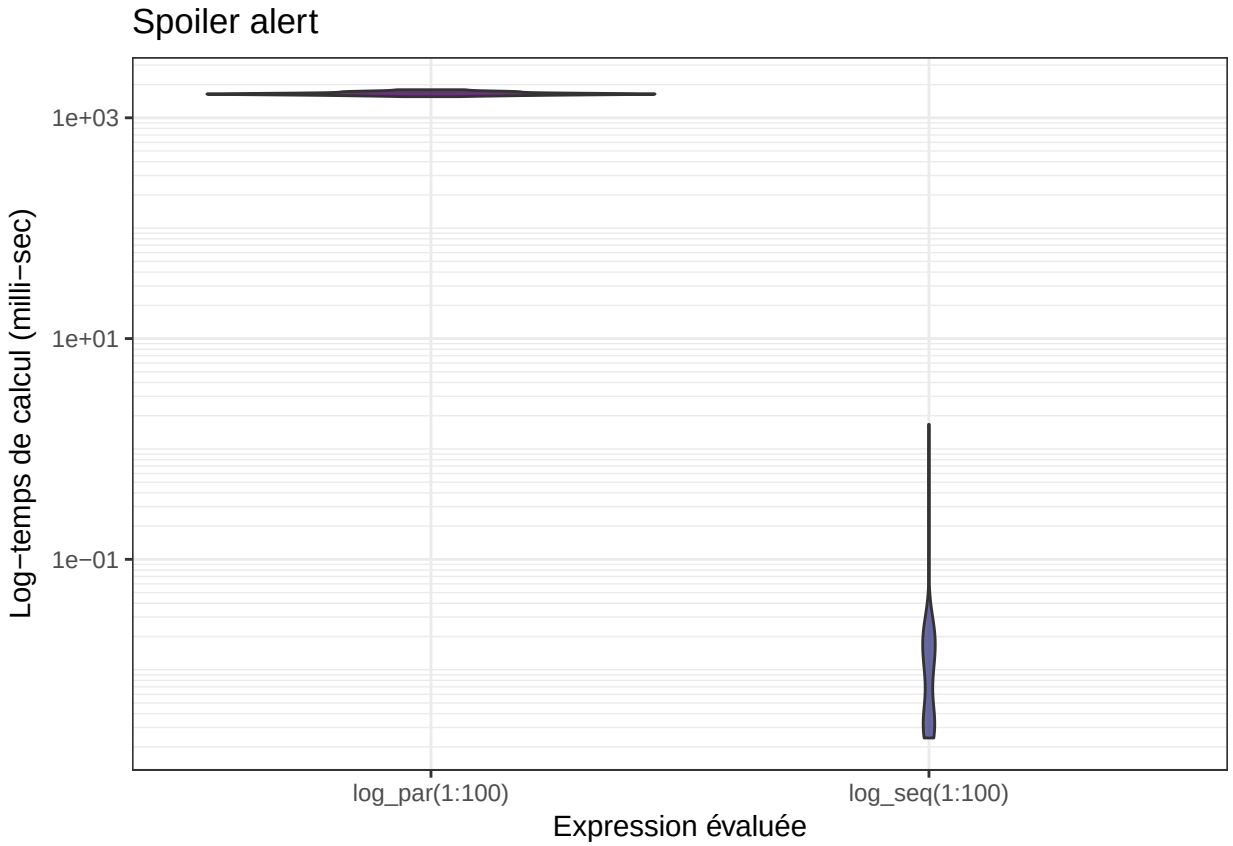
6.2 Première fonction parallèle en R

À vous de jouer !

On va commencer par écrire une fonction simple qui calcule le logarithme n nombres:

1. Déterminez combien de coeurs sont disponibles sur votre marchine grâce à la fonction `parallel::detectCores()`.
2. À l'aide de la fonction `parallel::makeCluster()`, créez un cluster de coeur (en prenant garde à **laisser un coeur disponible** pour traiter les autres processus) et déclarer ce cluser via la fonction `doParallel::registerDoParallel()`.
3. À l'aide de l'opérateur `%dopar%` du package `foreach`, calculez le log des n nombres en parallèle et concaténer les résultats dans un vecteur.
4. Fermez enfin les connections de votre cluster via la fonction `parallel::stopCluster(cl)`.
5. Comparez le temps d'éxecution avec celui d'une fonction séquentielle sur les 100 premiers entiers, grâce à la commande :
   `microbenchmark(log_par(1:100), log_seq(1:100), times=10)`

```
library(microbenchmark)
library(parallel)
library(foreach)
library(doParallel)

log_par <- function(x){

  Ncpus <- parallel::detectCores() - 1

  cl <- parallel::makeCluster(Ncpus)

  doParallel::registerDoParallel(cl)

  res <- foreach(i=1:length(x), .combine='c') %dopar% {
    log(x[i])
  }

  parallel::stopCluster(cl)

  return(res)
}

log_seq <- function(x){
  # res <- numeric(length(x))
  #
  # for(i in 1:length(x)){
  #   res[i] <- log(x[i])
  # }
  #
  # return(res)
  return(log(x))
}

mb <- microbenchmark(log_par(1:100), log_seq(1:100), times=50)
```



La version parallèle tourne beaucoup plus lentement… Car en fait, si les tâches individuelles sont trop rapides, `R` va passer plus de temps à communiquer avec les cœurs, qu'à faire les calculs effectifs.

Il faut qu'une itération de la boucle soit relativement longue pour que le calcul parallèle apporte un gain en temps de calcul !

En augmentant n, on observe une réduction de la différence entre les 2 implémentations (le temps de calcul en parallèle augmente très lentement comparé à l'augmentation de celui de la fonction séquentielle).

NB : les itérateurs d'`itertools` sont très performants mais ne peuvent servir que lorsque le code à l'intérieur du foreach est vectorisé (il est toujours possible de vectoriser le code à l'intérieur, par exemple avec une fonction de type `apply`). Ils minimisent le nombre de communication entre les coeurs.

6.3 Parallélisation efficace

On va maintenant se pencher sur un autre cas d'utilisation. Imaginons que l'on ait un grand tableau de données de taille comportant 10 observations pour 100 000 variables (e.g. des mesures de génomique), et que l'on veuille calculer la médiane pour chacune de ces variables.

```
x <- matrix(rnorm(1e6), nrow=10)
dim(x)
```

```
## [1]     10 100000
```

Pour un utilisateur averti de `R`, une telle opération se programme facilement à l'aide de la fonction `apply` :

```
colmedian_apply <- function(x){
  return(apply(x, 2, median))
```

```
}
system.time(colmedian_apply(x))
```

```
##    user  system elapsed
##   2.959   0.021   2.986
```

En réalité, une boucle `for` n'est pas plus lente à condition d'être bien programmée :

```
colmedian_for <- function(x){
  ans <- rep(0, ncol(x))
  for (i in 1:ncol(x)) {
    ans[i] <- median(x[,i])
  }
}
system.time(colmedian_for(x))
```

```
##    user  system elapsed
##   2.941   0.018   2.971
```

```
microbenchmark(colmedian_apply(x), colmedian_for(x), times=20)
```

```
## Unit: seconds
##                expr      min       lq     mean   median       uq      max
##  colmedian_apply(x) 2.808868 3.025870 3.110023 3.129606 3.210489 3.394848
##    colmedian_for(x) 2.731714 2.787841 2.905075 2.843305 3.012438 3.202375
##  neval cld
##     20   b
##     20  a
```

À vous de jouer !

Essayons d'améliorer encore ce temps de calcul en parallélisant :

1 . Parallélisez le calcul de la médiane de chacune des 100 000 variables. Observe-t-on un gain en temps de calcul ?

2. Proposez une implémentation alternative grâce à la fonction `itertools::isplitIndices()` qui permet de séparer vos données (les n nombres) en autant de groupes que vous avez de coeurs. Comparez à nouveau les temps de calcul.

```
colmedian_par <- function(x){

  Ncpus <- parallel::detectCores() - 1
  cl <- parallel::makeCluster(Ncpus)
  doParallel::registerDoParallel(cl)

  res <- foreach::foreach(i=1:ncol(x), .combine='c')%dopar%{
          return(median(x[,i]))
  }

  parallel::stopCluster(cl)
  return(res)
}
system.time(colmedian_par(x))
```

```
##    user  system elapsed
##  26.502   2.628  31.125
```

```
library(itertools)
colmedian_parIter <- function(x){

  Ncpus <- parallel::detectCores() - 1
  cl <- parallel::makeCluster(Ncpus)
  doParallel::registerDoParallel(cl)

  iter <- itertools::isplitIndices(n=ncol(x), chunks = Ncpus)
  res <- foreach::foreach(i=iter, .combine='c')%dopar%{
          return(apply(x[, i], 2, median))
  }

  parallel::stopCluster(cl)
  return(res)
}
system.time(colmedian_parIter(x))
```

```
##    user  system elapsed
##   0.102   0.038   2.682
```

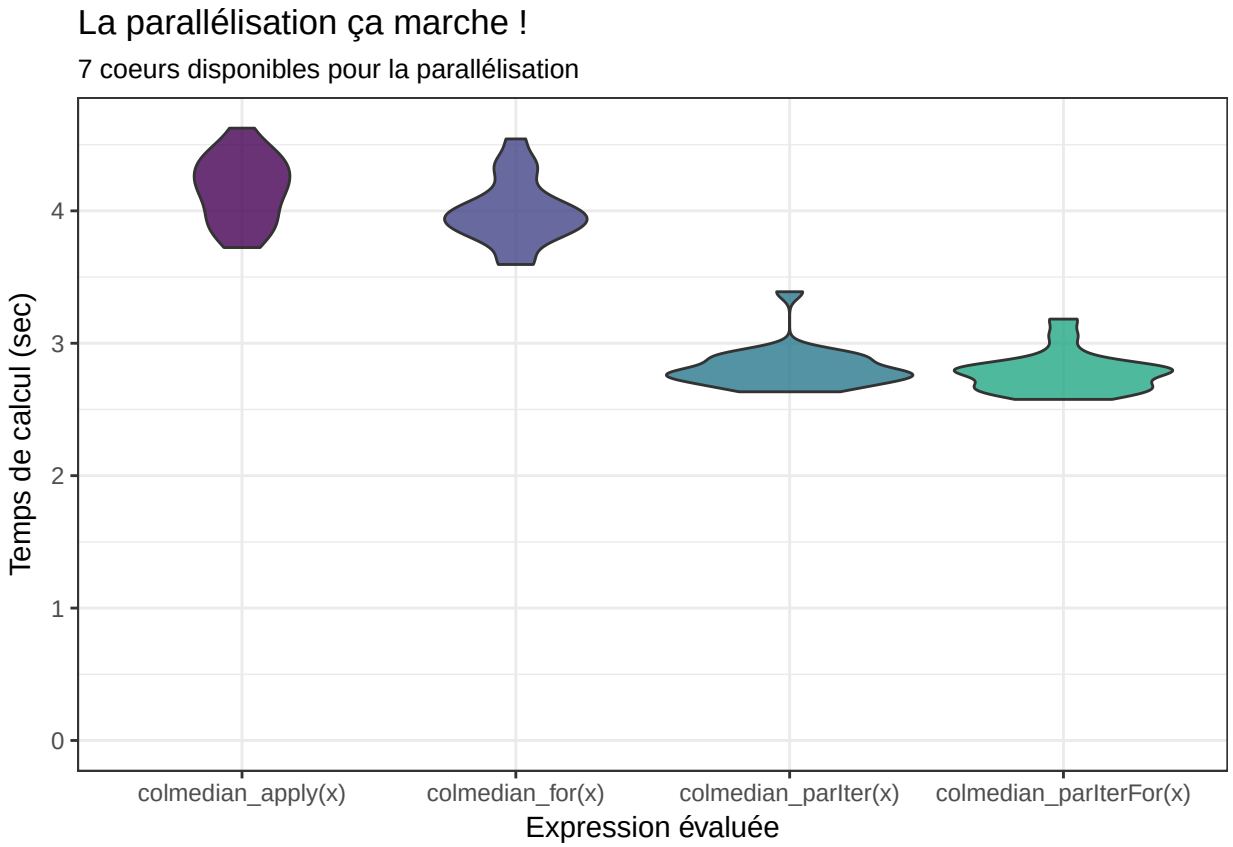
```
colmedian_parIterFor <- function(x){

  Ncpus <- parallel::detectCores() - 1
  cl <- parallel::makeCluster(Ncpus)
  doParallel::registerDoParallel(cl)

  iter <- itertools::isplitIndices(n=ncol(x), chunks = Ncpus)
  res <- foreach(i=iter, .combine='c') %dopar% {
    xtemp <- x[,i]
    ans <- rep(0, ncol(xtemp))
    for (j in 1:ncol(xtemp)) {
      ans[j] <- median(xtemp[,j])
    }
    return(ans)
  }

  parallel::stopCluster(cl)
  return(res)
}
system.time(colmedian_parIterFor(x))
```

```
##    user  system elapsed
##   0.098   0.038   2.652
```

```
mb <- microbenchmark(colmedian_apply(x),
                     colmedian_for(x),
                     colmedian_parIter(x),
                     colmedian_parIterFor(x), times=20)
mb
```

```
## Unit: seconds
##                      expr      min       lq     mean   median       uq
##       colmedian_apply(x) 3.723003 3.938663 4.163266 4.173559 4.350612
##         colmedian_for(x) 3.595036 3.846750 3.996339 3.952059 4.102757
##     colmedian_parIter(x) 2.633559 2.723737 2.816580 2.775805 2.883743
```

```
##  colmedian_parIterFor(x) 2.576067 2.666277 2.776994 2.780711 2.818846
##       max neval cld
##  4.624729    20   b
##  4.543317    20   b
##  3.389220    20  a
##  3.182542    20  a
```



Le package `itertools` permet de séparer facilement des données ou des taches (étape 3) tout en minimisant les communiquations avec les différents travailleurs. Il s'appuie sur une implémentation des itérateurs en R. Son utilisation nécessite néanmoins de vectoriser le code à l'intérieur du `foreach`. Expérimentez avec le petit code ci-dessous :

```
myiter <- itertools::isplitIndices(n=30, chunks = 3)

# Une première fois
iterators::nextElem(myiter)
```

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

```
# Une deuxième fois... Oh ?!
iterators::nextElem(myiter)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
# Encore !
iterators::nextElem(myiter)
```

```
##  [1] 21 22 23 24 25 26 27 28 29 30
```

```
# Encore ?
iterators::nextElem(myiter)
```

```
## Error: StopIteration
```

6.4 Parallélisation dans notre exemple fil rouge

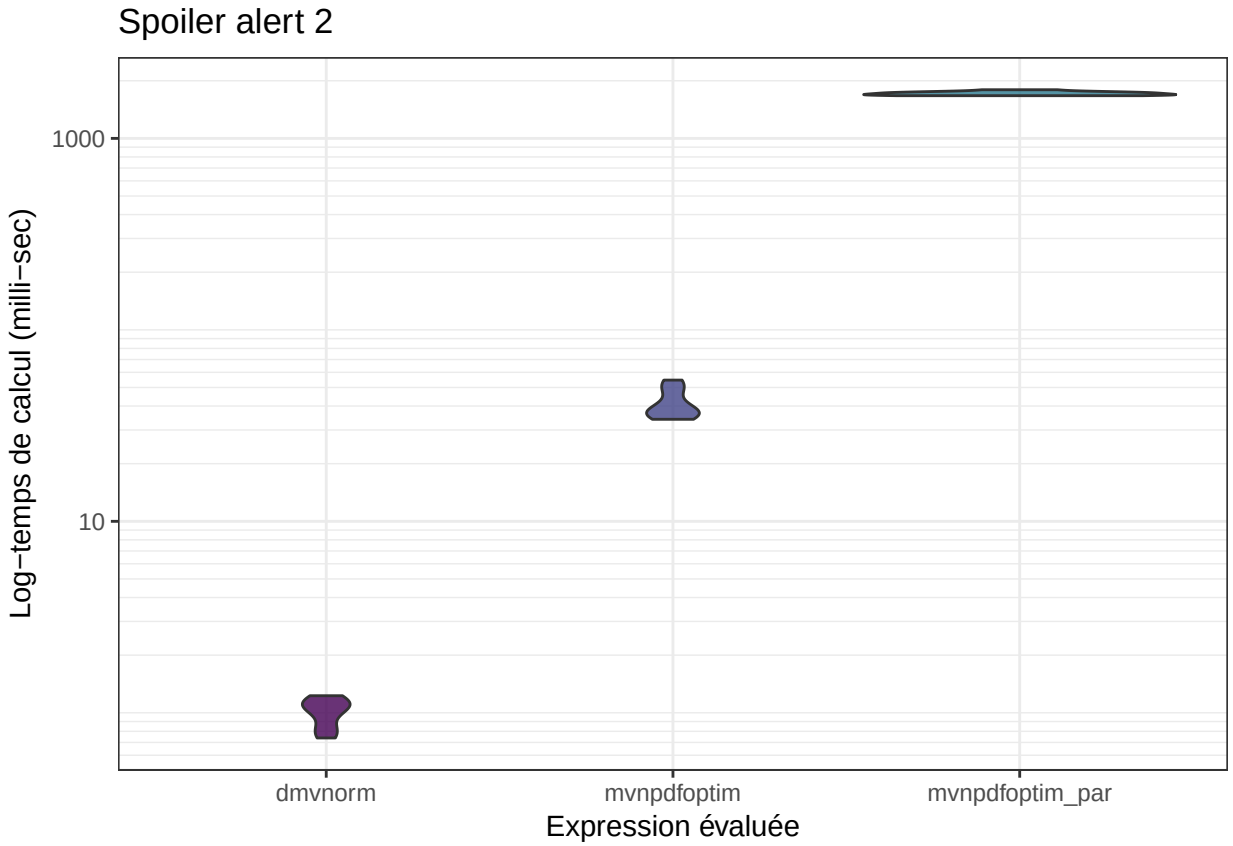
À vous de jouer !

1 . À partir de la fonction `mvnpdfoptim()` et/ou `mvnpdfsmart()`, proposez une implémentation parallélisant les calculs sur les observations (colonnes de x)

2. Comparez les temps de calcul sur 10 000 observations

```
n <- 10000
mb <- microbenchmark::microbenchmark(mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2)),
                                     mvnpdfoptim(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                                     mvnpdfoptim_par(x=matrix(1.96, nrow = 2, ncol = n), Log=FALSE),
                                     times=10L)
mb
```

```
## Unit: microseconds
##                                                      expr
##                  mvtnorm::dmvnorm(matrix(1.96, nrow = n, ncol = 2))
##      mvnpdfoptim(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##  mvnpdfoptim_par(x = matrix(1.96, nrow = 2, ncol = n), Log = FALSE)
##          min          lq        mean      median          uq         max
##      738.597     838.664    1012.381    1080.053    1120.811    1227.742
##    34100.296   36861.244   41196.843   37189.185   47822.766   54650.883
##  1671004.487 1679309.531 1715844.930 1709887.151 1738234.701 1795110.177
##  neval cld
##     10 a
##     10  b
##     10   c
```



6.5 Conclusion

La parallélisation permet de gagner du temps, mais il faut d'abord bien optimiser son code. Quand on parallélise un code, le gain sur la durée d'exécution dépend avant tout du ratio entre le temps de communication et le temps de calcul effectif pour chaque tache.

Chapter 7

Miscélanées

7.1 Debugging avec `browser()`

7.2 `attach`

7.3 gestion mémoire

7.4 copies et variables locales/globales dans les fonctions

7.5 naming

7.6 `gpplot2`

Chapter 8

Take Home message

1. **FAITES DES PACKAGES**
2. utilisez `git`, au moins pour vous en local
3. **si besoin** (i.e. après optimisation du code `R` lui même), n'ayez pas peur de vous tourner vers `Rcpp` et/ou la parallélisation de votre code

Références

- Les livres en ligne d'Hadley Wickham sont vraiment excellents et contiennent beaucoup de compléments par rapport à tous ce qu'on a traité dans cette formation.
  - le site sur la construction de package *R packages*.
  - le site *Advanced R* pour tout ce qui est optimisation, Rcpp, calcul parallèle.
  - le site *R for Data Science* est également très complet et comprend des chapitres sur la gestion des structures de données dans R, mais aussi la modélisation et des éléments sur les graphiques et Rmarkdown.
- le livre en ligne Rcpp for everyone de Masaki E. Tsuda est également très bien fait

Chapter 9

Les thèmes à aborder

- bonnes pratiques (naming, etc, cf Hadley Wickam & Jenny Brian, importance of consistency)
- **Robin** construire un package
- **Robin** le documenter avec *roxygen2*
- **Robin** tester son code lors du CHECK pour éviter l'introduction de bugs avec *testthat*
- **Boris** versioning collaboratif avec GitHub
- **Boris** automatisé le check automatique avec Travis CI et Appveyor
- Gains en performance 1 **Robin** Mesurer le temps de compilation : *microbenchmark* et profiling (*profvis*) 2 **Boris** parallélisation : *parallel*, *doParallel*, *iterators* 3 **Boris** C++ via *Rcpp*
- miscélanées sur R (attach, big.matrix, list indéxée recursivement, gestion mémoire, copies et variables locales/globales dans les fonctions)
- resources à suivre (livres en ligne de Hadley Wickam, r-hub https://builder.r-hub.io/, etc)
- reproducibilité et Rmarkdown