Pratique R

Calcul parallèle avec R

Vincent Miele
Violaine Louvet

edp sciences

Calcul parallèle avec R

Vincent Miele et Violaine Louvet

Calcul parallèle avec R

edp sciences

ISBN : 978-2-7598-2060-3

Imprimé en France

Collection Pratique R

dirigée par Pierre-André Cornillon et Eric Matzner-Løber

Département MASS
Université Rennes-2-Haute-Bretagne, France

Comité éditorial

Déjà paru dans la même collection :

Séries temporelles avec R

Yves Aragon, 2016
ISBN : 978-2-7598-1779-5 – EDP Sciences

Psychologie statistique avec R

Yvonnick Noël, 2015
ISBN : 978-2-7598-1736-8 – EDP Sciences

Réseaux bayésiens avec R

Jean-Baptiste Denis, Marco Scutati, 2014
ISBN : 978-2-7598-1198-4 – EDP Sciences

Analyse factorielle multiple avec R

Jérôme Pagès, 2013
ISBN : 978-2-7598-0963-9 – EDP Sciences

Régression avec R

Pierre-André Cornillon, Eric Matzner-Løber, 2011
ISBN : 978-2-8178-0184-1 – Springer

Méthodes de Monte-Carlo avec R

Christian P. Robert, George Casella, 2011
ISBN : 978-2-8178-0181-0 – Springer

REMERCIEMENTS

Mes premiers remerciements vont à Bernard Prum, grande figure de la Statistique française décédée il y a peu, qui m'a ouvert les portes du CNRS et celles du calcul parallèle. Fin 2004, bureau de Bernard : « Bernard, je voudrais m'inscrire à une formation sur le calcul parallèle à Grenoble, es tu d'accord ? ». « Du *parallèle* à Grenoble, je te connais, c'est une formation sur le ski parallèle que tu veux suivre ! ». S'en suivaient de grands éclats de rire. Je dédicace donc ce livre à mon très cher ami Bernard.
Je souhaite remercier le groupe Calcul dans son ensemble, *i.e.* les hommes et les femmes qui considèrent que l'entraide fait partie du travail. Bien sûr, j'adresse une mention spéciale à Violaine Louvet et Thierry Dumont, fondateurs du groupe, avec qui j'ai fait un bon bout de chemin à Lyon, au plus près des nouvelles technologies et techniques du calcul, et parfois même les pieds dans l'eau fraîche des Calanques de Marseille !
Mais il n'y a pas de calcul parallèle sans jolis problèmes scientifiques. Je souhaite en particulier remercier Laurent Duret et Simon Penel pour m'avoir fait confiance au moment où nous nous sommes lancés dans le grand bain des algorithmes parallèles.
Je remercie tout particulièrement mon acolyte Franck Picard qui, fin 2011, m'a dit ceci : « Vincent, tu es spécialiste du calcul parallèle, j'ai un package R qui demande des performances, ça te dirait de creuser les aspects du calcul parallèle dans R ? ».
J'adresse par ailleurs mes plus chaleureux remerciements aux membres du pôle informatique du LBBE qui ont toujours su mettre à ma disposition les dernières technologies et la puissance de calcul du CC LBBE/PRABI.
Je remercie également Stéphane Dray, Laurent Jacob, Martyn Plummer et Aurélie Siberchicot, mes collègues lyonnais de l'écosystème R, pour toute la considération qu'ils ont pu apporter aux différentes initiatives que j'ai eues autour de R.

Vincent Miele

Ce livre est l'aboutissement de nombreuses discussions et interactions avec des personnes issues d'horizons différents, unies dans un même besoin et une même envie de partage d'expériences et d'échanges. Je voudrais remercier collectivement ou nommément ces collègues pour la richesse de nos collaborations.
En premier lieu, mes pensées vont à Jacques Laminie qui m'a mis le pied à l'étrier du calcul intensif. Je tiens à remercier tout particulièrement Thierry Dumont pour la longue route que nous avons suivie ensemble, entre combustion et plasmas, entre méthodes numériques et architectures, entre Python et C++, entre Bedlewo et Grenade.
Je souhaite également remercier l'ensemble des personnes qui s'investissent depuis des années dans l'aventure du Groupe Calcul, mettant à la disposition des autres leurs expériences, leurs compétences et leur dynamisme.
Merci également aux chercheurs qui sont à l'origine de collaborations autour de projets scientifiques passionnants : Marc Massot, Stéphane Descombes, Emmanuel

Grenier, Francis Filbet en particulier.
Pour finir, je remercie tout particulièrement toutes les personnes des différents laboratoires que j'ai fréquentés : le laboratoire de Mathématiques d'Orsay, puis l'Institut Camille Jordan à Lyon, pour m'avoir permis un cheminement professionnel particulièrement riche.
Enfin, merci à Vincent pour m'avoir embarquée dans cette aventure !

Violaine Louvet

Les auteurs souhaitent également remercier Pierre-André Cornillon et Eric Matzner-Løber pour leur avoir donné l'opportunité de publier cet ouvrage. Ils remercient également les différents relecteurs anonymes ou pas (Erwan Le Pennec, Rémy Drouilhet et Martial Krawier) pour les commentaires constructifs qui ont permis d'améliorer cet ouvrage.

PREFACE

C'est avec un grand plaisir que je vous présente l'ouvrage « Calcul parallèle avec R » par Vincent Miele et Violaine Louvet, qui traite d'un sujet central pour le calcul haute performance dans R. En tant qu'utilisateur de R depuis 1996, et membre de la *R Core Team* depuis 2002, j'ai été témoin de l'augmentation incroyable de la popularité de R, qui constitue aujourd'hui un outil essentiel pour le traitement des données dans de nombreux domaines scientifiques. L'utilisation de plus en plus fréquente de R par des organisations commerciales à l'ère des « big data » s'est traduite par la formation du *R-Consortium* (`http://R-consortium.org`), un groupe d'entreprises du secteur technologique qui se sont unies pour apporter leur soutien à la communauté des utilisateurs de R. Mais surtout, le langage R possède aujourd'hui une communauté mature d'utilisateurs et de développeurs qui ont créé et partagé des milliers de packages via le *Comprehensive R Archive Network* (`CRAN`, `https://cran.r-project.org`).

L'histoire du succès de R a commencé il y a 30 ans. Une grande partie de la conception du logiciel R dérive du langage `S` développé au sein des laboratoires AT&T Bell dans les années 1980. Si on téléportait un utilisateur contemporain de R au début des années 1990, il n'aurait aucune difficulté à travailler avec la version `S3`, même s'il aurait probablement du mal à se passer de `CRAN`. Certaines caractéristiques fondamentales de R remontent à une époque où le paysage informatique était très différent. C'est pourquoi les limites inhérentes à cette conception ancienne finissent par devenir évidentes à un utilisateur cherchant une performance maximale.

R a toujours offert la possibilité d'améliorer sa performance en convertissant le haut niveau d'interprétation du code R en un langage compilé écrit en `C, C++` ou `Fortran`. Cette possibilité a été encore améliorée par les développeurs du package **Rcpp**, lequel offre une intégration harmonieuse entre R et `C++`. **Rcpp** est devenu la partie la plus importante de l'infrastructure de R, en dehors de sa distribution de base ; **Rcpp** est aujourd'hui utilisée par plus de 1200 packages `CRAN`.

Il faut pourtant constater que l'utilisation de codes compilés n'est plus suffisante pour obtenir des performances maximales. Comme Violaine et Vincent l'ont expliqué très clairement dans ce livre, les fabricants ont cessé de chercher à créer des processeurs plus rapides, il y a environ 10 ans, lorsqu'ils ont plutôt choisi d'augmenter le nombre de cœurs computationnels à l'intérieur des processeurs. L'exploitation de ces cœurs multiples exige pour le programmeur l'adoption de techniques de programmation parallèle. En outre, le traitement d'une très grande quantité de données ne peut plus se faire sur un ordinateur de bureau classique, car il exige d'avoir recours à un cluster. Autrefois réservé aux spécialistes de calcul haute performance, les clusters sont de plus en plus répandus grâce au services de *cloud computing*. La répartition des données et l'utilisation efficace de tous les nœuds d'un cluster nécessitent également une programmation parallèle, bien que dans une perspective quelque peu différente.

La programmation parallèle est bien plus « *close to the metal* » que la programmation séquentielle. Elle exige que le développeur soit familier avec l'architecture

du système qu'il utilise et qu'il en connaisse les limites que ce système impose à l'efficacité des programmes informatiques. Voilà pourquoi ce livre est si important. Il traite non seulement des outils de programmation parallèle R et C++ interfacé avec R en utilisant **Rcpp**, mais aussi des concepts fondamentaux de l'architecture de votre ordinateur qu'il vous faut connaître pour mener à bien la programmation. Enfin, il vous apprend à « penser parallèle ».

le 19 avril 2016,
Martyn Plummer

Table des matières

1 A la recherche de performances **1**
- 1.1 L'organisation de projet . . . 1
  - 1.1.1 Ne jamais optimiser prématurément . . . 1
  - 1.1.2 Le processus de développement . . . 2
  - 1.1.3 Bonnes pratiques de développement . . . 3
- 1.2 Les différentes approches en R pour gagner en performances . . . 9
  - 1.2.1 Apprendre à mesurer les performances temps et mémoire . 9
  - 1.2.2 Améliorer/optimiser le code R . . . 15
  - 1.2.3 Implémenter les points chauds de calcul avec des langages compilés . . . 21
  - 1.2.4 Utiliser plusieurs unités de calcul . . . 24

2 Fondamentaux du calcul parallèle **25**
- 2.1 Évolution des ordinateurs et nécessité du calcul parallèle . . . 25
- 2.2 Les architectures parallèles . . . 27
  - 2.2.1 Éléments de vocabulaire autour du processeur . . . 28
  - 2.2.2 Éléments de vocabulaire autour de la mémoire . . . 29
  - 2.2.3 Typologie des infrastructures de calcul . . . 35
- 2.3 Le calcul parallèle . . . 37
  - 2.3.1 Éléments de vocabulaire . . . 37
  - 2.3.2 Penser parallèle . . . 39
- 2.4 Limites aux performances . . . 41
  - 2.4.1 Loi d'Amdahl . . . 41
  - 2.4.2 Loi de Gustafson . . . 42
  - 2.4.3 Et dans la vraie vie . . . 43

3 Calcul parallèle avec R sur machine multi-cœurs **45**
- 3.1 Principe général . . . 45
- 3.2 Le package `parallel` et son utilisation . . . 46
  - 3.2.1 L'approche `snow` . . . 48
  - 3.2.2 L'approche `multicore` . . . 55
  - 3.2.3 `foreach + doParallel` . . . 60

3.2.4 Génération de nombres pseudo-aléatoires 62
3.3 L'équilibrage de charge 63
3.3.1 Durée des tâches connue et ordonnancement statique 65
3.3.2 Durée des tâches inconnue et ordonnancement dynamique . 66
3.3.3 Granularité et équilibrage de charge 70

4 C++ parallèle interfacé avec R **73**
4.1 Principe général 73
4.2 openMP 78
4.2.1 Les bases d'openMP 78
4.2.2 R et `openMP` 82
4.3 Les threads de `C++11` 84
4.3.1 Les bases des threads de `C++11` 84
4.3.2 R et les threads de `C++11` 85
4.4 RcppParallel 87

5 Calculs et données distribués avec R sur un cluster **89**
5.1 Cluster orienté HPC 89
5.1.1 Les bases de `MPI` 90
5.1.2 R et `MPI` 92
5.2 Cluster orienté *big data* 100

A Notions complémentaires **103**
A.1 Problématique du calcul flottant 103
A.2 La complexité 104
A.3 Typologie des langages 106
A.4 Les accélérateurs 107
A.4.1 GP-GPU, General-Purpose computation on Graphic Processing Unit 107
A.4.2 Carte many-cœurs 109
A.5 Exemples d'utilisation de **Rcpp** avec la fonction `cppfunction`du package **inline** 109

Bibliographie **114**

Index **115**

Chapitre 1

A la recherche de performances

La recherche de performance est une démarche qui s'inscrit dans un cadre global. Il est généralement contre-productif de s'attaquer à l'optimisation d'un code sans mettre en place au préalable des éléments d'organisation, souvent de bon sens (gestion de projet, suivi de version), qui permettent d'améliorer considérablement l'efficacité du travail réalisé. Nous proposons dans ce chapitre des pistes à explorer pour une approche à la fois organisationnelle et technique de l'optimisation de code R : optimiser sa façon de programmer et optimiser ses programmes.

1.1 L'organisation de projet

1.1.1 Ne jamais optimiser prématurément

On nomme *optimisation de code* l'ensemble des efforts réalisés pour augmenter ses performances, c'est-à-dire accélérer la vitesse du code et/ou diminuer son empreinte mémoire (*i.e.* l'espace mémoire qu'il requiert). Mais à partir de quelle phase de développement doit-on commencer à se soucier des performances de calcul ? Herb Sutter nous dit : « ne jamais optimiser prématurément » (Sutter & Alexandrescu, 2004). Et de rajouter, « l'exactitude des résultats, la simplicité et la clarté du code passent en premier ». Il est en effet primordial de respecter un rythme et une méthode de développement qui garantissent avant tout que les résultats sont et resteront exacts tout au long de la vie du code R. Exact au sens où on estime les résultats obtenus conformes aux attentes. L'*exactitude* des calculs sur ordinateur n'est pas toujours simple à définir (voir Annexe A.1, p.103, sur la problématique du calcul flottant). Pour préserver la « justesse » des résultats, nous proposons de suivre une démarche de développement itératif (voir §1.1.2, p.2). Les performances viendront progressivement, après 1/ validation de l'exacti-

tude des résultats et 2/ analyse fine de l'efficacité de l'implémentation (voir §1.2.1, p.9). Bien sûr, le programmeur expérimenté pourra rapidement faire des choix de structures de données, d'algorithmes, de façon de coder qui favorisent de bonnes performances *a priori* (*i.e* sans les mesurer) tout en veillant à garantir en premier lieu l'exactitude. Gardons en tête cette formule :

« On n'est jamais assez surpris
par les bonnes performances d'un code faux. » (les auteurs)

1.1.2 Le processus de développement

S'organiser pour chercher de la performance, c'est d'abord s'organiser pour mener à bien un projet informatique. Le développeur en R pourra s'inspirer librement des réflexions abouties et mises en œuvre dans le milieu du génie logiciel. En particulier, les approches *agile* (terme choisi par opposition à la lourdeur avérée des projets informatiques du siècle dernier) méritent d'être étudiées. En 2001, dix-sept personnalités du génie logiciel signent l'*Agile Manifesto* (Beck *et al.*, 2001) qui repense la gestion de projets de développement en informatique comme à la fois un processus de développement logiciel, un état d'esprit et un ensemble de bonnes pratiques. Plusieurs méthodes dites agiles verront le jour, en particulier l'*Extreme programming* : celle-ci n'a d'extrême que le nom, Kent Beck définissant d'ailleurs la méthode comme « une tentative de réconcilier l'humain avec la productivité »(Beck & Andres, 2004). Nous proposons ici une sélection libre de principes énoncés dans ces méthodes :

— le principe fondamental est le *développement itératif* qui consiste à adopter des cycles de développement (conception, implementation, tests) très courts. Pour la conception, la recherche de la simplicité, de la solution la plus simple, est suggérée. Le code évolue par petites unités fonctionnelles, il est donc recommandé de mettre en place un code modulaire (fonctions, classes ou données les plus indépendantes possibles) ;

— les modifications sont intégrées très fréquemment (*intégration continue*, éventuellement plusieurs fois par jour) et il est indispensable de réaliser un suivi des modifications grâce à un *gestionnaire de version* (voir §1.1.3, p.7) ;

— cependant, toutes ces modifications doivent être testées : l'unité de mesure de la progression du projet n'est pas le nombre de lignes de codes mais bien le nombre de lignes qui fonctionnent et donnent un résultat juste. Ainsi, les tests sont la clé de voûte du développement itératif et doivent être implémentés et conçus avant la fonctionnalité qu'ils sont censés tester : on parle de développement piloté par les tests (voir §1.1.3, p.5) ;

— les méthodes agiles recommandant la ré-écriture ou la re-conception (*refactoring*) fréquentes de parties du logiciel non optimales : le changement fait partie de la vie du code ;

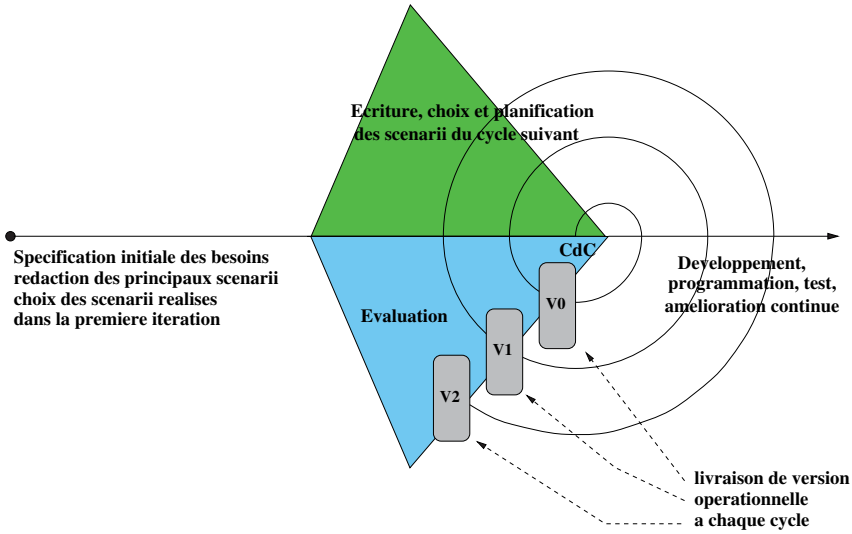

Fig. 1.1 – Illustration du développement itératif, CdC=cahier des charges.

— l'adoption de *conventions de nommage* (règles de choix du noms des éléments du programme) dans le code source est fondamentale (voir §1.1.3, p.4) ;

— enfin, l'humain est un facteur fondamental dans la réussite d'un projet informatique. *Primo*, le rythme de développement doit être *durable*, le stress étant considéré comme contre-productif par les promoteurs des méthodes agiles. *Secondo*, les membres de l'équipe (depuis les développeurs jusqu'au responsable du projet) doivent se sentir co-responsables de la vie du logiciel quel que soit leur position dans la chaîne de décision : on parle d'*appropriation collective* du code. Par exemple, le temps de développement associé au *refactoring* doit être accepté par tous et ne doit pas être considéré comme du temps perdu, ni ne doit servir à justifier une chasse aux sorcières contre le développeur qui a implémenté une partie à ré-écrire.

1.1.3 Bonnes pratiques de développement

La mise en place de bonnes pratiques et l'utilisation d'outils *ad hoc* peuvent paraître au premier abord fastidieuses, inutiles et chronophages. Une bonne organisation du développement peut pourtant apporter des gains en temps, en énergie et en sérénité, que l'on soit développeur « isolé » (personne implémentant un code pour son propre usage) ou développeur dans un projet collaboratif.

Les conventions de nommage

Pour R, le développeur pourra s'inspirer des conventions proposées dans la communauté (voir tab. 1.1), mais devra surtout maintenir ces conventions sur toute la durée du projet et dans l'intégralité du code : la rédaction d'un petit document qui fixe les idées peut dès lors être une bonne idée !

Nom de la convention (anglais) `+ example`	Principe
alllowercase `adjustcolor`	tout en minuscule cohérent mais peu lisible
period.separated `plot.new`	minuscules et mots séparés par un point spécifique à R et très utilisé dans le `core`
underscore_separated `seq_along`	minuscules et mots séparés par un tiret-bas recommandé par GNU
lowerCamelCase `colMeans`	première lettre minuscule et mots séparés par une majuscule
UpperCamelCase `Matrix` class	idem mais première lettre majuscule classique pour les noms de classes à combiner avec lowerCamelCase
Bioconductor	`https://www.bioconductor.org/developers/how-to/coding-style/`

Tableau 1.1 – Quelques conventions de nommage : les cinq premières sont présentées dans Bååth (2012) tandis que la dernière convention fait référence au projet Bioconductor.

Le choix des noms de variables et fonctions doit permettre l'*auto-documentation* du code : il s'agit de donner des noms explicites, qui font sens et/ou référence à des éléments présentés dans un article. L'auto-documentation est « agile » car toute modification du code modifie de fait intrinsèquement la documentation (puisque le code est sa propre documentation). A noter que l'anglais est à privilégier, car R est basé sur la philosophie de l'*open source* et le code source doit être lisible par le plus grand nombre « worldwide ».

Dans le code suivant, aucune convention ni cohérence n'est adoptée et, de fait, le code est peu lisible :

```
i <- 1.45 # nommage pas explicite
          # et i mal choisi car entier en général

Y <- 5    # alternance de majuscules-minuscules
```

```
          # sans règle
z <- matrix(0,2,2)

func63 <- function(a, b) return sqrt(a^2+b^2)
          # nom de fonction pas explicite
```

A contrario, les bonnes pratiques de nommage sont mise en application dans le code ci-dessous :

```
impact.factor <- 1.45  # nommage explicite, en minuscule
nb.journal <- 5        # et choix du . pour séparer les mots

M <- matrix(0,2,2)     # choix des majuscules pour les matrices

euclidian.distance <- function(x, y) sqrt(sum((x - y) ^ 2))
                       # nom de fonction explicite
```

Avant tout : les tests

Le but des tests est de garantir que les modifications apportées à un code (depuis les premières lignes de code) sont vérifiées et n'altèrent pas l'exactitude des résultats. Il est donc plus que nécessaire de mettre en œuvre une organisation qui fera mentir l'adage suivant :

> « *Le problème, ce n'est pas qu'on n'a pas fait de tests, mais c'est qu'on ne les a pas gardés... et qu'on ne peut plus les refaire tourner automatiquement.* »
> (Wickham, 2014).

Il existe un éventail de tests qui suivent la terminologie suivante :

- *tests unitaires* : la règle numéro un, proposée dans les méthodes agiles, est d'écrire le test d'une fonctionnalité/d'un module... avant justement d'implémenter celle/celui-ci. Ce sont les tests unitaires. On notera par exemple que le projet R Bioconductor incite les développeurs à implémenter des tests unitaires et propose un ensemble de bonnes pratiques à leur destination (Bioconductor, 2016) ;
- *tests d'intégration* : il s'agit ici de vérifier l'assemblage des composants élémentaires pour réaliser un composant de plus haut niveau. Le concept d'intégration continue consiste à systématiser et automatiser ces tests dès lors que les composants élémentaires sont modifiés (eux-même testés avec les tests unitaires) ;
- *test de non-régression* : les tests de (non-)régression permettent de s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non

modifiées, lorsqu'on ajoute une fonctionnalité ou qu'on réalise un changement dans une autre partie du programme. Ces tests sont fastidieux car ils nécessitent d'être le plus exhaustif possible.

Au niveau de R, une approche minimaliste peut consister à créer un répertoire (`devtests` par exemple), y installer des données et des scripts de tests remplis de `stopifnot()` comme ci-après et lancer régulièrement ces scripts avec `Rscript` :

```
log.likelihood <- compute.loglikelihood("data_test1.txt")
stopifnot(abs(log.likelihood - (-772.0688)) < 1e-8)
```

Ceci peut être adapté à la mise en œuvre de tests de régression (car peu nombreux) mais il s'agit cependant de « bricolage » et rien n'est automatique ici.
Pour gagner en robustesse, le développeur se tournera vers les packages **RUnit** et **testthat** qui sont les deux alternatives actuelles les plus communes pour aider à la réalisation de tests automatiques, en particulier les tests unitaires (associés à des modules, des classes, des fonctionnalités indépendantes). Nous évoquerons brièvement ici l'utilisation de **testthat** (Wickham, 2011). Avec **testthat**, voici la feuille de route, à implémenter dans le cadre de la structure d'un package R que nous appellerons **mypkg** :

1. Ajouter `testthat` dans le champ `Suggests` de `DESCRIPTION`.
2. Créer un répertoire `tests/testthat`
3. Dans ce répertoire, créer un ensemble de tests aux noms explicites de la forme `test-modulexxx.R` ou `test-fonctionnalitéxxx.R`. Ces tests contiennent des appels aux fonctions de vérifications `expect_equal, expect_identical, expect_match, expect_output,` etc.

   ```
   library(mypkg)
   context("Unit test for the model log-likelihood")
   test_that("Produces the correct log likelihood.", {
       expect_equal(object=compute.loglikelihood("data_test1.txt"),
       expected=-772.0688, tolerance = 1e-8)
   })
   ```

4. Créer un fichier `tests/testthat.R` qui va rassembler les tests lancés par `R CMD check` et qui contient les lignes suivantes :

   ```
   library(testthat)
   test_check("mypkg")
   ```

Pour terminer, on aura bien noté ici que, dans les deux cas `stopifnot()` ou `expect_equal`, l'égalité entre valeurs réelles n'est pas requise, les calculs étant réalisés en arithmétique flottantes (voir Annexe A.1). C'est pour cela que l'on compare ici les valeurs réelles avec un seuil (`tolerance`) à 1e-8.

La gestion de version

Le *gestionnaire de version* est l'outil le plus courant et sans doute le plus indispensable pour toute personne travaillant sur un document, que ce soit les sources d'un code, une publication... ou un livre ! Le principe est simple : il permet de conserver la trace chronologique de toutes les modifications qui ont été apportées aux fichiers. En corollaire, il est ainsi relativement aisé de gérer les différentes versions d'un code et de créer des *branches de développement* : celles-ci sont des bifurcations dans la vie du code qui permettent le test de nouvelles fonctionnalités sans perturber les évolutions de la version principale par exemple. Le gestionnaire de version permet aussi facilement de développer à plusieurs en proposant la gestion des conflits et la fusion des modifications lorsque plusieurs personnes ont travaillé sur le même fichier.
Si les avantages d'un outil de gestion de version sont évidents pour un travail collaboratif, quel intérêt présente-il pour quelqu'un qui développe un « petit » code tout seul ? Une réponse pragmatique à cette question consiste à en poser une autre :

« Quel développeur n'a jamais effectué des modifications dans son programme qu'il a immédiatement regrettées, et pour lequel il a du réaliser des « undo » sans pouvoir revenir parfaitement à la situation de départ ? »

Il existe essentiellement deux types de systèmes de gestion de version : les systèmes *centralisés* comme `subversion` (`https://subversion.apache.org/`), et les systèmes *décentralisés* comme `git` (`https://git-scm.com/`). Dans le premier cas, il n'existe qu'un seul dépôt de l'ensemble des fichiers et des versions qui fait référence, ce qui en simplifie la gestion. Dans le second, plusieurs dépôts cohabitent, permettant plus de souplesse dans le travail individuel. On notera que les systèmes décentralisés prennent de plus en plus le pas sur les systèmes centralisés, en particulier du fait de la possibilité de faire des enregistrements fréquents de versions (des *commit*) localement sans les propager à tout le système.

Supposons que l'on souhaite la chronologie des modifications apportées au fichier source `estimate.dynsbm.R`. La commande `svn log` de l'outil `subversion` répond à nos besoins : elle retourne les *logs* (des messages), *i.e.* les informations écrites par les développeurs successifs qui ont modifié le code, la date ainsi que le volume des modifications. Ci-dessous, `svn log` nous donne les logs des versions 53 à 56 :

```
svn log
------------------------------------------------------------------------
r56 | louvet | 2016-01-05 12:48:10 +0100 (mar. 05 janv. 2016) | 1 ligne
Adding two additionnal parameters in estimate.dynsbm.

------------------------------------------------------------------------
```

```
cppFunction('double add(double x, double y){
 return(x*y);
}
')
add(1.5, 6.3)
[1] 9.45
```

— des vecteurs (`NumericVector, IntegerVector, LogicalVector`) en entrée de fonction :

```
cppFunction('
double crossprodC(NumericVector x, NumericVector y){
 double crp = 0.;
 int n = x.size();
 for(int i=0; i<n; i++) crp += x[i]*y[i];
 return(crp);
}
')
crossprodC(c(1.5,6.3), c(0.3,4.3))
[1] 27.54
```

— une matrice (`NumericMatrix, IntegerMatrix`) en sortie de fonction :

```
cppFunction('
NumericMatrix subMatrix(LogicalVector x, NumericMatrix mat){
 int nr = mat.nrow(), nc = mat.ncol();
 if (x.size() != nr)  warning("Inconsistent dimensions");
 int nr2 = 0;
 for(int i=0; i<nr; i++) if(x[i]) nr2++;
 NumericMatrix mat2(nr2,nc);
 int i2 = 0;
 for(int i=0; i<nr; i++)
  if(x[i]){
   for(int j=0; j<nc; j++)
    mat2(i2,j) = mat(i,j);
   i2++;
  }
 return(mat2);
}
')
subMatrix(c(TRUE,FALSE,TRUE), rbind(c(1.,2),c(3.,4.),c(5.,6.)))
     [,1] [,2]
[1,]    1    2
[2,]    5    6
```

— une fonction de modification « *in-place* » (*i.e.* sans copie) d'un vecteur :

```
cppFunction('void squareInPlace(IntegerVector x) {
 int n = x.size();
 for(int i = 0; i < n; ++i) x[i] += x[i]*x[i];
}
')
x = c(1.5, 6.3)
squareInPlace(x)
x
[1]  3.75 45.99
```