HAL open science

Travailler avec R

Eric Marcon

To cite this version:

Eric Marcon. Travailler avec R. Doctorat. France. 2020. hal-03022820v1

HAL Id: hal-03022820

https://hal.science/hal-03022820v1

Submitted on 24 Nov 2020 (v1), last revised 14 Dec 2021 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Travailler avec R

Eric Marcon

Version : 2020-11-24

Ce document est réalisé de façon dynamique et reproductible grâce à :

- LaTeX, dans sa distribution Miktex (http://miktex.org/) et la classe memoir (http://www.ctan.org/pkg/memoir).
- R (http://www.r-project.org/) et RStudio (http://www.rstudio.com/)
- bookdown (http://bookdown.org/)

Son code source est sur GitHub : https://github.com/EricMarcon/travailleR/.

Le texte mis à jour en continu peut être lu sur https://ericmarcon.github.io/travailleR/.

Les versions d'étape sont déposées sur HAL : https://hal.archives-ouvertes.fr/.

UMR Écologie des forêts de Guyane
http://www.ecofog.gf

Les opinions émises par les auteurs sont personnelles et n'engagent ni l'UMR EcoFoG ni ses tutelles.

Photographie en couverture : Hadrien Lalagüe

Table des matières

Table des matières **iii**

Présentation **ix**
Objectifs ix
Conventions ix

1 Logiciels **1**
1.1 R 1
Installation 1
Rtools 1
Mise à jour 2
Librairies 2
1.2 RStudio 3
Installation 4
Dossier de travail 4
Solution retenue 4
1.3 Packages 5
Installation depuis CRAN 5
Installation depuis GitHub 6
Installation depuis Bioconductor 6
Solution retenue 7
1.4 Git 8
1.5 GitHub 8
1.6 MiKTeX 9
Installation 9
Mises à jour 9
1.7 Go 9

2 Utiliser R **11**
2.1 Les langages de R 11
Base 11
S3 12
S4 14
RC 15
S6 16
Tidyverse 16
2.2 Environnements 19
Organisation 19

Recherche 19
Espaces de nom des packages 21
2.3 Mesure du temps d'exécution 22
system.time 22
microbenchmark 23
Profilage 25
2.4 Boucles 25
Fonctions vectorielles 25
lapply 26
Boucles for 27
replicate 28
Vectorize 29
Statistiques marginales 29
2.5 Code C++ 29
2.6 Paralléliser R 30
mclapply (fork) 31
parLapply (socket) 34
foreach 34
Fonctionnement 34
Parallélisation 35
2.7 Etude de cas 36
Création des données 36
Spatstat 36
apply 36
boucle for 39
boucle foreach 39
RCpp 40
RcppParallel 41
Conclusions sur l'optimisation de la vitesse du code 43

3 Git et GitHub 45
3.1 Principes 45
Contrôle de source 45
git et GitHub 45
3.2 Créer un nouveau dépôt 46
A partir d'un projet existant 46
Prendre en compte des fichiers 47
Valider des modifications 47
Créer un dépôt vide sur GitHub 49
Lier git et GitHub 50
Pousser les premières modifications 51
Cloner un dépôt de GitHub 51
3.3 Usage courant 52
Tirer, modifier, valider, pousser 52
Régler les conflits 53
Voir les différences 53
Revenir en arrière 53
Voir l'historique 54
3.4 Branches 54

Créer une nouvelle branche 55
Changer de branche 55
Fusionner avec `merge` 55
Fusionner avec une requête de tirage 56
3.5 Usage avancé 57
Commandes de git 57
Taille d'un dépôt 57
Supprimer un dossier 58
3.6 Pages GitHub 60
Activation 60
Badges 60

4 Rédiger 63
4.1 Bloc-note Markdown (R Notebook) 63
4.2 Modèles R Markdown 65
4.3 Memo, article : bookdown 66
Modèle Memo EcoFoG 67
Créer 67
Ecrire 67
Tricoter 67
Mettre en ligne 68
Autres modèles 68
4.4 Présentation EcoFoG : bookdown 69
4.5 Ouvrage EcoFoG : bookdown 69
Créer 69
Ecrire 70
Tricoter 70
Finitions 70
Intégration continue 71
Google Analytics 71
4.6 Site web R Markdown 72
Modèle 72
Améliorations 72
Contôle de source 73
4.7 Site web personnel : blogdown 74
Installation des outils 74
Créer 75
Paramétrer 77
Ecrire 79
Page d'accueil 79
Menu de la page d'accueil 82
Billets 83
Publications 84
Communications 86
Autres éléments 87
Finitions 88
Intégration continue 88
Mises à jour 88
4.8 Exportation de figures 88

Formats vectoriels et raster 89
Fonctions 90
Package ragg 90

5 Package 93
5.1 Premier package 93
Création 94
Première fonction 94
Fichiers 94
Création 95
Contrôle de source 98
package.R 98
5.2 Organisation du package 99
Fichier DESCRIPTION 99
Fichier NEWS.md 100
5.3 Vignette et pkgdown 101
5.4 Code spécifique aux packages 102
Importation de fonctions 102
Méthodes S3 103
Classes 103
Méthodes 104
Attribution d'un objet à une classe 105
En pratique 106
Création d'une méthode générique 106
Création d'une classe 108
Documentation 110
Code C++ 112
Package bien rangé 114
5.5 Bibliographie 114
Préparation 114
Citations 114
5.6 Données 115
5.7 Tests unitaires 116
5.8 Fichier .gitignore 117
5.9 Intégration continue 118
5.10 CRAN 119
Test du package 119
Soumission 119
Maintenance 119

6 Intégration continue 121
6.1 Outils 121
Travis 121
Codecov 121
GitHub Pages 122
6.2 Intégration continue d'un projet de document . . . 122
Obtention d'un jeton d'accès personnel 122
Activation du dépôt sur Travis 122
Activation du dépôt sur CodeCov 123

Scripter les actions de Travis 123
Projet d'ouvrage 123
Site web blogdown 126
Packages R 128
Vérification de `.travis.yml` 130
Problèmes transitoires 130
Ajouter des badges 131

7 Shiny 133
7.1 Première application 133
7.2 Application plus élaborée 134
Méthode de travail 134
Exemple 134
7.3 Hébergement 138

8 Enseigner avec R 141
8.1 learnr 141
Premier tutoriel 141
Diffusion 142
8.2 GitHub Classrooms 142
Inscription 142
Organisations 142
Nouvelle salle de classe 143
Préparer un modèle de dépôt 143
Assigner une tâche 143
Contrôler le travail des étudiants 144

9 Conclusion 145

Bibliographie 147

Présentation

Objectifs

Ce document est le support du cours *Travailler avec R*.

Il propose une organisation du travail autour de R et RStudio pour, au-delà des statistiques, rédiger des documents efficacement avec R Markdown, aux formats variés (mémos, articles scientifiques, mémoires d'étudiants, livres, diaporamas), créer son site web et des applications R en ligne (Shiny), produire des packages et utiliser R pour l'enseignement. Il complète *Reproducible Research with R and R Studio*[1] par une approche plus concrète, avec des solutions prêtes à l'emploi.

[1] C. Gandrud (2015). *Reproducible Research with R and R Studio*. 2e éd. Chapman et Hall/CRC.

L'optimisation de l'utilisation des nombreux outils disponibles est traitée en détail : **rmarkdown**, **bookdown** et **blogdown** pour la rédaction, **roxygen2**, **testthat** et **pkgdown** pour les packages, le contrôle de source avec git et GitHub, l'intégration continue avec Travis et Codecov. Des exemples sont présentés à chaque étape, et le code nécessaire est fourni.

Le chapitre 1 est consacré à l'installation des outils nécessaires : R, git et LaTeX. Le chapitre 2 renvoie vers des documents externes consacrés à l'utilisation de R (langage, utilisation de base), qui n'est pas reprise ici. Le chapitre 3 présente le contrôle de source avec git et GitHub.

Le chapitre 4 montre comment rédiger des documents simples (mémos) ou complexes (ouvrages) avec R Markdown, intégrant les données, le code pour les traiter et le texte pour les présenter. Le chapitre 5 présente une méthode pas à pas pour créer efficacement un package. Le chapitre 6 introduit l'utilisation de l'intégration continue pour produire automatiquement des documents, vérifier le code des packages et produire leurs vignettes. Le chapitre 7 présente Shiny, l'outil de mise en ligne d'applications R. Enfin, le chapitre 8 introduit les outils destinés à l'enseignement de et avec R.

Conventions

Les noms des packages sont en gras dans le texte, exemple : **ggplot2**.

L'identifiant utilisé sur GitHub est noté *GitHubID*.

Le signe `|>` dans le code des exemples indique que la suite du code devrait se trouver sur la même ligne, mais est coupée pour le formatage de ce document. Son usage est limité aux fichiers de configuration YAML, surtout utilisés dans le chapitre 6. Dans tous les autres cas, le code peut être utilisé directement.

Chapitre 1

Logiciels

L'outil central est évidemment R, mais son fonctionnement est aujourd'hui difficilement envisageable sans son environnement de développement RStudio. Pour le contrôle de source, git et GitHub sont de fait les standards. L'ensemble doit être complété par une distribution LaTeX pour la production de documents au format PDF. Enfin, d'autres logiciels d'usage plus ponctuel peuvent être nécessaires, comme Go.

Leur installation et leur organisation cohérente sont présentées dans ce chapitre.

1.1 R

Installation

R est inclus dans les distributions de Linux : le paquet est nommé `r-base`. Il ne contient pas des outils de développement souvent nécessaires, donc il est préférable d'installer aussi le paquet `r-base-dev`. La version de R est souvent un peu ancienne. Pour disposer de la dernière version, il faut utiliser un mirroir de CRAN comme source des paquets : voir la documentation complète pour Ubuntu[1].

Sous Windows ou Mac, installer R après l'avoir téléchargé depuis CRAN[2].

[1] https://doc.ubuntu-fr.org/r

[2] https://cran.r-project.org/

Rtools

Sur Mac, l'installation de R est suffisante à partir de la version 4.0.0.

Sous Windows, l'installation doit être complétée par les "Rtools", qui contiennent les outils de développement dont ceux nécessaires à la compilation des packages contenant du code C++.

Le chemin des Rtools doit être déclaré à R, en exécutant dans la console de RStudio la commande suivante (adaptée à la version

4.0 des Rtools) :

```
# Rtools : déclaration du chemin,
# nécessite de redémarrer RStudio
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"',
           con = "~/.Renviron")
```

Les Rtools doivent être complétés par quelques utilitaires manquants, à installer quand le besoin apparaît (en général, un avertissement de R indiquant que le logiciel n'est pas installé).

La vérification des packages renvoie un avertissement si *qpdf*[3] n'est pas installé. Télécharger le fichier zip et coller tout le contenu du dossier `bin` dans le dossier `bin` de `Rtools` (`C:\Rtools40\usr\bin` pour la version 4.0).

[3] https://sourceforge.net/projects/qpdf/

Un autre avertissement est renvoyé en absence de *Ghostscript*[4] à télécharger et installer. Copier ensuite le contenu du dossier `bin` dans le dossier `bin` de `Rtools`.

[4] https://www.ghostscript.com/

Mise à jour

Il est conseillé d'utiliser la dernière version mineure de R : 4.0.x jusqu'à la sortie de la version 4.1. Il est obligatoire d'utiliser la toute dernière version pour préparer un package soumis à CRAN.

Des changements importants ont lieu entre les versions majeures (la version 4 ne permet pas d'utiliser un package compilé pour la version 3) mais aussi parfois entre versions mineures (un fichier de données binaires `.rda` enregistré sous la version 3.3 ne peut pas être lu par la version 3.6). Il est donc utile de mettre R à jour régulièrement.

L'installation d'une nouvelle version ne désinstalle par automatiquement les versions anciennes, ce qui permet d'en utiliser plusieurs en cas de besoin (par exemple, si un package ancien et indispensable n'est plus disponible). En usage courant, il est préférable de désinstaller manuellement les anciennes versions après l'installation d'une nouvelle.

Librairies

Les packages de R se trouvent dans deux dossiers :

- la bibliothèque système (*System Library*) contient les packages fournis avec R : **base**, **utils**, **graphics** par exemple. Elle se trouve dans un sous-répertoire du programme d'installation (`C:\Program Files\R\R-4.0.0\library` pour R version 4.0.0 sous Windows 10).
- La bibliothèque utilisateur (*User Library*) contient ceux installés par l'utilisateur. Elle se trouve dans le dossier personnel de l'utilisateur, dans un sous-dossier `R\win-library\4.0\`).

Si le dossier personnel de l'utilisateur est sauvegardé (par exemple, s'il est répliqué dans le cloud par OneDrive sous Windows), il n'est pas optimal d'y placer les packages. Pour que les packages soient installés automatiquement dans la bibliothèque système, il suffit que l'utilisateur y ait le droit d'écrire. Sous Windows, donner le droit "Modifier" au groupe des utilisateurs de l'ordinateur sur le dossier de la biliothèque, en plus du droit de lecture par défaut 1.1).

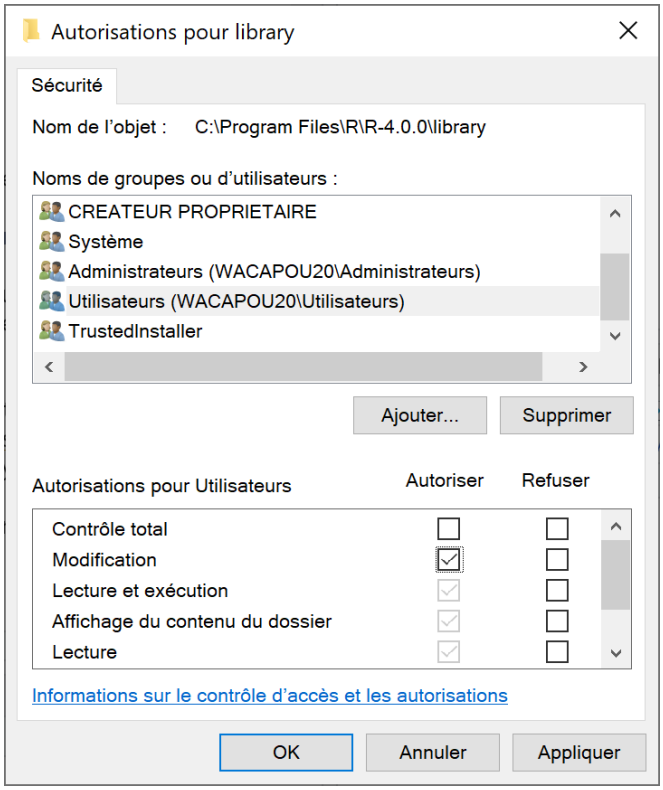


Fig. 1.1 : Activation du droit de modifier la bibliothèque système sous Windows.

Si la bibliothèque utilisateur est retenue, il faut penser à vider le dossier correspondant à l'ancienne version de R en cas changement de version mineure.

L'emplacement des librairies est donné par la fonction `.libPaths()` :

```
.libPaths()
```

```
## [1] "/home/travis/R/Library"        "/usr/local/lib/R/site-library"
## [3] "/opt/R/4.0.2/lib/R/library"
```

1.2 RStudio

RStudio est une interface graphique pour R et bien plus : il est conçu pour simplifier la gestion des projets, faciliter la rédaction et la production de documents et intégrer le contrôle de source par exemple.

Installation

Installer la dernière version de *RStudio Desktop* à partir du site de RStudio[5].

[5] https://rstudio.com/products/rstudio/download/

Une commande est disponible dans le menu *Help* de RStudio pour vérifier l'existence d'une version plus récente, à installer.

Dossier de travail

Le dossier de travail par défaut est le dossier personnel de l'utilisateur, appelé ~ par RStudio :

```
Sys.getenv("R_USER")
```

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

- `Mes Documents` sous Windows ;
- `Home` sous Mac ou Linux.

Il faut systématiquement travailler dans des sous-dossier de ~, par exemple : `~\Formation`.

Pour le bon fonctionnement des *RTools*, le nom complet du répertoire de travail ne doit pas contenir d'espace (utiliser les tirets bas _) ni de caractère spécial. Le dossier de travail en cours est obtenu par la commande `getwd()`.

```
getwd()
```

```
## [1] "/home/travis/build/EricMarcon/travailleR"
```

L'utilisation du contrôle de source (voir chapitre 3) crée de nombreux fichiers de travail. Les projets sous contôle de source ne devraient pas se trouver dans un dossier déjà sauvegardé par un autre moyen, comme un lecteur OneDrive sous Windows, sous peine d'une utilisation excessive des ressources : chaque validation de modifications engendre la sauvegarde des fichiers modifiés, mais aussi des fichiers de contrôle qui peuvent être de grande taille.

Solution retenue

L'organisation de l'environnement travail est une affaire personnelle, qui dépend des préférences de chacun. L'organisation proposée ici n'est qu'une possibilité, à adapter à ses propres choix, mais en respectant les contraintes mentionnées.

Sous Windows, une organisation optimale est la suivante :

- Dans son dossier personnel (`Mes Documents`, ~ pour R), un dossier `R` est utilisé pour les projets simples, sans contrôle de source. La sauvegarde de ce dossier est gérée par ailleurs.

- Un dossier hors du dossier personnel est utilisé pour les projets sous contrôle de source. L'utilisateur doit y avoir le droit d'écrire. Dans l'organisation de Windows, le dossier correspondant à ces critères est `%LOCALAPPDATA%`, typiquement `C:\Users\NomUtilisateur\AppData\Local`. Le dossier sera donc `%LOCALAPPDATA%\ProjetsR` à créer : exécuter `md %LOCALAPPDATA%\ProjetsR` dans une invite de commande. Epingler ce dossier à l'accès rapide de l'explorateur de fichiers (figure 1.2) : coller `%LOCALAPPDATA%\ProjetsR` dans la barre d'adresse de l'explorateur de fichiers, valider, puis faire un clic droit sur "Accès Rapide" et épingler le dossier.

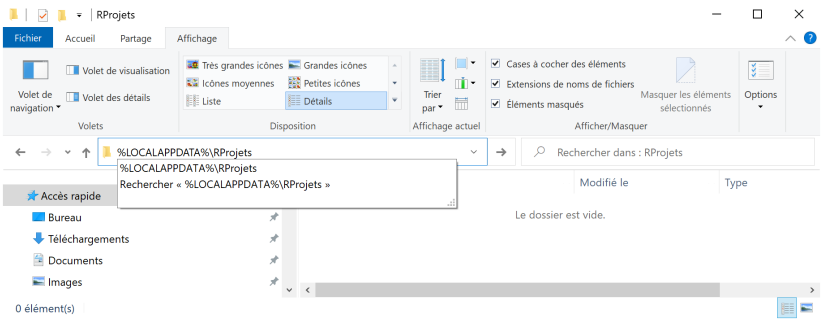


Fig. 1.2 : Dossier pour les projets sous contrôle de source, sous Windows.

1.3 Packages

Installation depuis CRAN

L'installation classique des packages fait appel à CRAN. Un bouton *Install* se trouve dans la fenêtre *Packages* de RStudio.

Les packages sont déposés sur CRAN par leurs auteur sous forme de code source, compressé dans une fichier `.tar.gz`. Ils sont disponibles pour le téléchargement dès leur validation. Ils doivent ensuite être mis au format binaire pour Windows (dans un fichier `.zip`), ce qui prend un peu de temps.

A la demande de l'installation d'un package sous Windows, CRAN propose la version source plutôt que la version binaire si elle est plus récente 1.3).

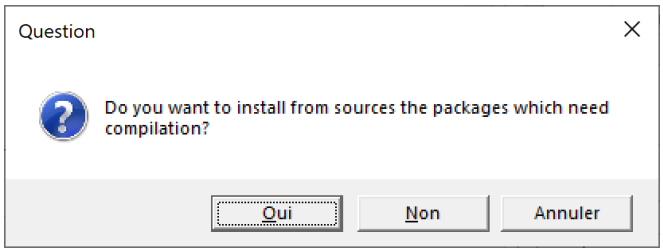


Fig. 1.3 : Activation du droit de modifier la bibliothèque système sous Windows.

La liste des packages concernés est affichée dans la console, par exemple :

```
  There are binary versions available but the source
  versions are later:
             binary   source needs_compilation
boot         1.3-24   1.3-25             FALSE
class        7.3-16   7.3-17              TRUE
```

Certains packages nécessitent une compilation (colonne `needs_compilation`), en général parce qu'ils contiennent du code C++. Ils ne pourront être installés que par les *Rtools*.

L'installation des packages en version source est beaucoup plus longue qu'en version binaire. Sauf si une version précise d'un package est nécessaire, il est donc préférable de refuser l'installation des versions source.

Les packages peuvent être mis à jour un peu plus tard, après leur compilation par CRAN.

Le bouton *Update* dans la fenête *Packages* de RStudio permet de mettre à jour tous les packages installés.

Installation depuis GitHub

Certains packages ne sont pas disponibles sur CRAN mais seulement sur GitHub parce qu'ils sont encore en développement ou parce qu'ils ne sont pas destinés à un large usage par la communauté des utilisateurs de R. Il peut aussi être utile d'installer une version de développement d'un package publié sur CRAN pour un usage ponctuel comme le test de nouvelles fonctionnalités.

L'installation est gérée par le package **remotes**. L'argument `build_vignettes` est nécessaire pour créer les vignettes du package.

```r
remotes::install_github("EcoFoG/EcoFoG", build_vignettes = TRUE)
```

Le nom du package est entré sous la forme "GitHubID/NomduPackage". L'installation est faite à partir du code source et nécessite donc les *Rtools* si une compilation est nécessaire. `install_github()` vérifie que la version sur GitHub est plus récente que l'éventuelle version installée sur le poste de travail et ne fait rien si elles sont identiques.

Installation depuis Bioconductor

Bioconductor est une plateforme complémentaire de CRAN qui héberge des packages spécialisés dans la génomique. L'installaton des packages de Bioconductor nécessite le package **BiocManager** pour sa fonction `install()`. Le premier

argument de la fonction est un vecteur de caractères contenant les noms des packages à installer, par exemple :

```
BiocManager::install(c("GenomicFeatures", "AnnotationDbi"))
```

La fonction `install()` appelée sans arguments met à jour les packages.

Solution retenue

A chaque mise à jour mineure de R, tous les packages doivent être réinstallés. La façon la plus efficace de le faire est de créer un script `Packages.R` à placer dans `~\R`. Il contient une fonction qui vérifie si chaque package est déjà installé pour ne pas le refaire inutilement.

```
# Installation des packages de R ####

# Installer les packages si nécessaire ####
InstallPackages <- function(Packages) {
  sapply(Packages, function(Package)
    if (!Package %in% installed.packages()[, 1])
      {install.packages(Package)})
}


# Outils de développement ####
InstallPackages(c(
  # Outils de développement. Importe remotes, etc.
  "devtools",
  # Exécution de Check par RStudio
  "rcmdcheck",
  # Fomatage du code R (utilisé par knitr)
  "formatR",
  # Documentation des packages dans /docs sur GitHub
  "pkgdown",
  # Biliographie avec roxygen
  "Rdpack",
  # Mesure des performances
  "rbenchmark",
  # Documentation automatique des packages
  "roxygen2",
  # Tests des packages
  "testthat"
  ))

# Markdown ####
InstallPackages(c(
  # Tricot
  "knitr",
  # Documents markdown complexes
  "bookdown",
  # Sites web
  "blogdown",
  # Modèle de présentation
  "xaringan",
  # Modèles d'articles
  "rticles"
  ))


# Tidyverse ####
InstallPackages("tidyverse")
```

```r
# Mes packages ####

# EcoFoG
remotes::install_github("EcoFoG/EcoFoG",
                        build_vignettes = TRUE)
```

La dernière partie du script est à compléter avec les packages utilisés régulièrement.

Ce script est à exécuter à chaque mise à jour de R, après avoir éventuellement activé le droit d'écriture dans la librairie système (voir section 1.1).

1.4 Git

git est le logiciel de contrôle de source utilisé ici. Son utilisation est détaillée dans le chapitre 3.

[6] https://git-scm.com/

Pour Windows et Mac, l'installation a lieu à partir du site[6].

git est intégré dans les distributions Linux. Pour Ubuntu, le package apt est `git-all`.

git est installé sans interface graphique, fournie par RStudio.

Après l'installation de git, il est possible que le terminal de RStudio ne fonctionne plus correctement et renvoie un message d'erreur contenant les éléments suivants :

```
*** fatal error - cygheap base mismatch detected
This problem is probably due to using incompatible
versions of the cygwin DLL.
```

Le message d'erreur est imprécis : la librairie qui ne doit exister qu'en un seul exemplaire n'est pas `cygwin1.dll` mais `msys-2.0.dll`. Rechercher ce fichier dans le dossier d'installation de git et le supprimer : il se trouve déjà dans le dossier de Rtools.

1.5 GitHub

GitHub est la plateforme accessible par un site web qui permet de partager le contenu des dépôts *git*. Pour l'utiliser, il suffit d'ouvrir un compte.

Le nom du compte GitHub est noté ici *GitHubID*. Chaque compte GitHub permet d'héberger des dépôts (un dépôt contient les fichiers d'un projet) à l'adresse https://github.com/GitHubID/NomDuDepot [7]. Chaque dépôt peut disposer d'un site web à l'adresse https://GitHubID.github.io/NomDuDepot/ [8]. Enfin, un site web global est prévu pour chaque utilisateur à l'adresse https://GitHubID.github.io/ [9].

[7] Exemple : https://github.com/EricMarcon/travailleR

[8] Exemple : https://EricMarcon.github.io/travailleR/

[9] Exemple : https://EricMarcon.github.io/

1.6 MiKTeX

Pour produire des documents au format PDF, une distribution Latex est nécessaire. La solution légère consiste à installer le package **tinytex** : sa fonction `install_tinytex()` installe une distribution Latex optimisée pour R Markdown.

Une distribution complète est préférée ici parce qu'elle permet l'utilisation de Latex au-delà de RStudio. MiKTeX[10] est une très bonne solution pour Windows et Mac.

[10] https://miktex.org/download

Installation

Télécharger le fichier d'installation et l'exécuter. Plusieurs choix sont à faire pendant l'installation :

- Installer le programme pour tous les utilisateurs (avec des droits d'administrateur) ;
- Le format par défaut du papier : choisir A4 ;
- Le mode d'installation des packages manquants : choisir "Always Install" pour qu'ils soient téléchargés automatiquement en cas de besoin.

Pour Linux, suivre les instructions sur le site de MiKTeX.

Mises à jour

MiKTeX est installé avec les packages LaTeX les plus utilisés. Si un document nécessite un package manquant, il est chargé automatiquement. Les mises à jour de packages doivent être faites périodiquement avec la console MiKTeX, accessible dans le menu Démarrer.

Quand elle est lancée sans élévation des privilèges, la console propose de passer en mode administrateur. Cliquer sur "Switch to Administrator mode".

Dans les paramètres (*Settings*), vérifier que les packages s'installent toujours automatiquement et que le format du papier est bien A4.

Dans le menu des mises à jour (*Updates*), cliquer sur "Check for updates" puis "Update now".

Si l'installation automatique est défaillante, il est possible d'installer manuellement un package dans le menu "Packages".

1.7 Go

[11] https://golang.org/

Go[11] n'est utilisé que par le générateur de sites web Hugo (voir section 4.7).

Télécharger le fichier d'installation et l'exécuter. A la fin de l'installation, exécuter la commande `go version` dans un terminal pour vérifier son bon fonctionnement.

Les mises à jour se font en installant la nouvelle version par dessus la précédente.

Chapitre 2

Utiliser R

La documentation consacrée à l'apprentissage de R est florissante. Les ouvrages suivants sont une sélection arbitraire mais utile pour progresser :

- L'Introduction à R et au tidyverse[1] est un excellent cours de prise en main.
- Advanced R[2] est la référence pour maîtriser les subtilités du langage et bien comprendre le fonctionnement de R.
- R for Data Science[3] présente une méthode de travail complète, cohérente avec le tidyverse.
- Enfin, Efficient R programming[4] traite de l'optimisation du code.

[1] J. Barnier (2020). *Introduction à R et au tidyverse.*

[2] H. Wickham (2014). *Advanced R.* Chapman et Hall/CRC.

[3] H. Wickham et G. Grolemund (2016). *R for Data Science.* O'Reilly Media.

[4] C. Gillespie et R. Lovelace (2016). *Efficient R Programming.* O'Reilly Media.

Quelques aspects avancés du codage sont vus ici, concernant l'optimisation des performances. La première étape consiste à disposer des outils permettant de mesurer le temps d'exécution du code.

2.1 Les langages de R

R comprend plusieurs langages de programmation. Le plus courant est S3, mais ce n'est pas le seul[5].

[5] https://adv-r.had.co.nz/OO-essentials.html

Base

Le coeur de R est constitué des fonctions primitives et structures de données de base comme la fonction `sum` et les données de type `matrix` :

```
pryr::otype(sum)
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
## [1] "base"
```

```r
typeof(sum)
```

```
## [1] "builtin"
```

```r
pryr::otype(matrix(1))
```

```
## [1] "base"
```

```r
typeof(matrix(1))
```

```
## [1] "double"
```

Le package **pryr** permet d'afficher le langage dans lequel des objets sont définis. La fonction `typeof()` affiche le type de stockage interne des objets :

- la fonction `sum()` appartient au langage de base de R et est une fonction primitive (*builtin*) ;
- les éléments de la matrice numérique contenant un seul 1 des réels à double précision, et la matrice elle-même est définie dans le langage de base.

Les fonctions primitives sont codées en C et sont très rapides. Elles sont toujours disponibles, quels que soient les packages chargés. Leur usage est donc à privilégier.

S3

S3 est le langage le plus utilisé, souvent le seul connu par les utilisateurs de R.

C'est un langage orienté objet dans lequel les classes, c'est-è-dire le type des objets, sont déclaratives.

```r
MonPrenom <- "Eric"
class(MonPrenom) <- "Prenom"
```

La variable `MonPrenom` est ici de classe "Prenom" par une simple déclaration.

Contrairement au fonctionnement d'un langage orienté objet classique[6], les méthodes S3 sont liées aux fonctions, pas aux objets.

[6] https://www.troispointzero.fr/le-blog/introduction-a-la-programmation-orientee-objet-poo/

```r
# Affichage par défaut
MonPrenom
```

```
## [1] "Eric"
## attr(,"class")
## [1] "Prenom"
```

```
print.Prenom <- function(x) cat("Le prénom est", x)
# Affichage modifié
MonPrenom
```

```
## Le prénom est Eric
```

Dans cet exemple, la méthode `print()` appliquée à la classe "Prenom" est modifiée. Dans un langage orienté objet classique, la méthode serait définie dans la classe `Prenom`. Dans R, les méthodes sont définies à partir de méthodes génériques.

`print` est une méthode générique ("un générique") déclaré dans **base**.

```
pryr::otype(print)
```

```
## [1] "base"
```

Son code se résume à une déclaration `UseMethod("print")` :

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x3a54740>
## <environment: namespace:base>
```

Il existe beaucoup de méthodes S3 pour `print` :

```
head(methods("print"))
```

```
## [1] "print.acf"         "print.AES"         "print.all_vars"
## [4] "print.anova"       "print.ansi_string" "print.ansi_style"
```

Chacune s'applique à une classe. `print.default` est utilisée en dernier ressort et s'appuie sur le type de l'objet, pas sur sa classe S3.

```
typeof(MonPrenom)
```

```
## [1] "character"
```

```
pryr::otype(MonPrenom)
```

```
## [1] "S3"
```

Un objet peut appartenir à plusieurs classes, ce qui permet une forme d'héritage des méthodes. Dans un langage orienté objet classique, l'héritage permet de définir des classes plus précises ("PrenomFrancais") qui héritent de classes plus générales ("Prenom") et bénéficient de cette façon de leurs méthodes sans avoir à les redéfinir. Dans R, l'héritage consiste simplement à déclarer un vecteur de classes de plus en plus larges pour un objet :

```r
# Définition des classes par un vecteur
class(MonPrenom) <- c("PrenomFrancais", "Prenom")
# Ecriture alternative, avec inherits()
inherits(MonPrenom, what = "PrenomFrancais")
```

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

```r
inherits(MonPrenom, what = "Prenom")
```

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

Le générique cherche une méthode pour chaque classe, dans l'ordre de leur déclaration.

```r
print.PrenomFrancais <- function(x) cat("Prénom français:",
    x)
MonPrenom
```

```
## Prénom français: Eric
```

En résumé, S3 est le langage courant de R. Presque tous les packages sont écrits en S3. Les génériques sont partout mais passent inaperçus, par exemple dans des packages :

```r
library("entropart")
.S3methods(class = "SpeciesDistribution")
```

```
## [1] autoplot plot
## see '?methods' for accessing help and source code
```

La fonction `.S3methods()` permet d'afficher toutes les méthodes disponibles pour une classe, par opposition à `methods()` qui affiche toutes les classes pour lesquelles la méthode passée en argument est définie.

De nombreuses fonctions primitives de R sont des méthodes génériques. Utiliser l'aide `help(InternalMethods)` pour les découvrir.

S4

S4 est une évolution de S3 qui structure les classes pour se rapprocher d'un langage orienté objet classique :

- les classes doivent être définies explicitement, pas simplement déclarées ;
- les attributs (c'est-à-dire les variables décrivant les objets), appelés *slots*, sont déclarés explicitement ;
- le constructeur, c'est-à-dire la méthode qui crée un nouvelle instance d'une classe (c'est-à-dire une variable contenant un objet de la classe), est explicite.

En reprenant l'exemple précédent, la syntaxe S4 est la suivante :

```
# Définition de la classe Personne, avec ses slots
setClass("Personne",
         slots = list(Nom = "character", Prenom = "character"))
# Construction d'une instance
Moi <- new("Personne", Nom = "Marcon", Prenom = "Eric")
# Langage
pryr::otype(Moi)
```

```
## [1] "S4"
```

Les méthodes appartiennent toujours aux fonctions. Elles sont déclarées par la fonction `setMethod()` :

```
setMethod("print", signature = "Personne", function(x, ...) {
    cat("La personne est:", x@Prenom, x@Nom)
})
print(Moi)
```

```
## La personne est: Eric Marcon
```

Les attributs sont appelés par la syntaxe `variable@slot`.

En résumé, S4 est plus rigoureux que S3. Quelques packages sur CRAN : **Matrix**, **sp**, **odbc**… et beaucoup sur Bioconductor sont écrits en S4 mais le langage est maintenant clairement délaissé au profit de S3, notamment à cause du succès du **tidyverse**.

RC

RC a été introduit dans R 2.12 (2010) avec le package **methods**.

Les méthodes appartiennent aux classes, comme en C++ : elles sont déclarées dans la classe et appelées à partir des objets.

```
library("methods")
# Déclaration de la classe
PersonneRC <- setRefClass("PersonneRC",
    fields = list(Nom = "character", Prenom = "character"),
    methods = list(print = function() cat(Prenom, Nom)))
# Constructeur
MoiRC <- new("PersonneRC", Nom = "Marcon", Prenom ="Eric")
# Langage
pryr::otype(MoiRC)
```

```
## [1] "RC"
```

```
# Appel de la méthode print
MoiRC$print()
```

```
## Eric Marcon
```

RC est un langage confidentiel, bien que ce soit le premier "vrai" langage orienté objet de R.

S6

S6[7] perfectionne RC mais n'est pas inclus dans R : il nécessite d'installer son package.

[7] https://r6.r-lib.org/

Les attributs et les méthodes peuvent être publics ou privés. Une méthode `initialize()` est utilisée comme constructeur.

```
library(R6)
PersonneR6 <- R6Class("PersonneR6", public = list(Nom = "character",
    Prenom = "character", initialize = function(Nom = NA,
        Prenom = NA) {
        self$Nom <- Nom
        self$Prenom <- Prenom
    }, print = function() cat(self$Prenom, self$Nom)))
MoiR6 <- PersonneR6$new(Nom = "Marcon", Prenom = "Eric")
MoiR6$print()
```

```
## Eric Marcon
```

S6 permet de programmer rigoureusement en objet mais est très peu utilisé. Les performances de S6 sont bien supérieures à celles de RC mais sont inférieures à celles de S3[8].

[8] https://r6.r-lib.org/articles/Performance.html

La non-inclusion de R6 à R est montrée par **pryr** :

```
pryr::otype(MoiR6)
```

```
## [1] "S3"
```

Tidyverse

Le tidyverse est un ensemble de packages cohérents qui ont fait évoluer la façon de programmer R. L'ensemble des packages indispensables peut être chargé par le package **tidyverse** qui n'a pas d'autre utilité :

```
library("tidyverse")
```

Il ne s'agit pas d'un nouveau langage à proprement parler mais plutôt d'une extension de S3, avec de profondes modifications techniques, notamment l'évaluation non conventionnelle des expressions[9], qu'il n'est pas essentiel de maîtriser en détail.

[9] https://dplyr.tidyverse.org/articles/programming.html

Ses principes sont inscrits dans un manifeste[10]. Son apport le plus visible pour l'utilisateur sont l'enchaînement des commandes dans un flux (pipeline de code).

[10] https://cran.r-project.org/web/packages/tidyverse/vignettes/manifesto.html

En programmation standard, l'enchaînement des fonctions s'écrit par emboîtements successifs, ce qui en rend la lecture difficile, surtout quand des arguments sont nécessaires :

```
# Logarithme de base 2 de la moyenne de 100 tirages
# aléatoires dans une loi uniforme
log(mean(runif(100)), base = 2)
```

```
## [1] -1.127903
```

Dans le tidyverse, les fonctions s'enchaînent, ce qui correspond souvent mieux à la réflexion du programmeur sur le traitement des données :

```r
# 100 tirages aléatoires dans une loi uniforme
runif(100) %>%
  # Moyenne
  mean %>%
  # Logarithme
  log(base=2)
```

```
## [1] -0.9772102
```

Le tuyau `%>%` est un opérateur qui appelle la fonction suivante en lui passant comme premier argument le résultat de la fonction précédente. Les arguments supplémentaires sont passés normalement : pour la lisibilité du code, il est indispensable de les nommer. La plupart des fonctions de R sont utilisables sans difficultés dans le tidyverse bien qu'elles n'aient pas été prévues pour cela : il suffit que leur premier argument soit les données à traiter.

Le pipeline ne permet de passer qu'une seule valeur à la fonction suivante, ce qui interdit les fonctions multidimensionnelles, de type `f(x,y)`. La structure de données préférée est le *tibble*, qui est un dataframe amélioré : sa méthode `print()` est plus lisible, et il corrige quelques comportements non-intuitifs des dataframes, comme la conversion automatique en vecteurs des dataframes à une seule colonne. Les colonnes du dataframe ou du tibble permettent de passer autant de données que nécessaire.

Enfin, la visualisation des données est prise en charge par **ggplot2** qui s'appuie sur une grammaire des graphiques[11] solide sur le plan théorique. Schématiquement, un graphique est construit selon le modèle suivant :

[11] H. Wickham (2010). « A Layered Grammar of Graphics ». In : *Journal of Computational and Graphical Statistics* 19.1, p. 3-28.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

- les données sont obligatoirement un dataframe ;
- la géométrie est le type de graphique choisi (points, lignes, histogrammes…) ;
- l'esthétique (fonction `aes()`) désigne ce qui est représenté : c'est la correspondance entre les colonnes du dataframe et les éléments nécessaires à la géométrie ;

- la statistique est le traitement appliqué aux données avant de les transmettre à la géométrie (souvent “identity”, c’est-à-dire aucune transformation mais “boxplot” pour une boîte à moustache). Les données peuvent être transformées par une fonction d’échelle, comme `scale_y_log10()` ;
- la position est l’emplacement des objets sur le graphique (souvent “identity” ; “stack” pour un histogramme empilé, “jitter” pour déplacer légèrement les points superposés dans un `geom_point`) ;
- les coordonnées définissent l’affichage du graphique (`coord_fixed()` pour ne pas déformer une carte par exemple) ;
- enfin, les facettes offrent la possibilité d’afficher plusieurs aspects des mêmes données en produidant un graphique par modalité d’une variable.

L’ensemble formé par le pipeline et **ggplot2** permet des traitements complexes dans un code lisible. La figure 2.1 montre le résultat du code suivant :

```
# Données sur les diamants fournies par ggplot2
diamonds %>%
  # Ne conserver que les diamants de plus d'un demi-carat
  filter(carat > 0.5) %>%
  # Graphique : prix en fonction du poids
  ggplot(aes(x = carat, y = price)) +
    # Nuage de points
    geom_point() +
    # Echelle logarithmique
    scale_x_log10() +
    scale_y_log10() +
    # Régression linéaire
    geom_smooth(method = "lm")
```

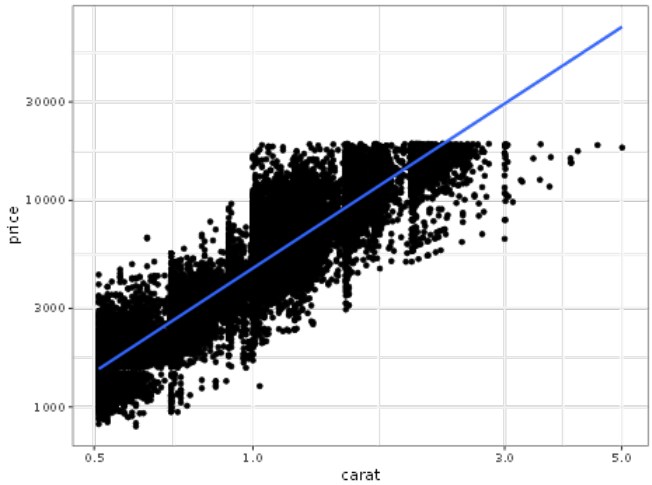


Fig. 2.1 : Prix des diamants en fonction de leur poids. Démonstration du code de **ggplot2** combiné au traitement de données du tidyverse.

Le tidyverse est documenté en détail dans Wickham et Grolemund[12] et **ggplot2** dans Wickham.[13]

[12] Wickham et Grolemund (2016), *R for Data Science*, cf. note 3, p. 11.

[13] H. Wickham (2017). *ggplot2 : Elegant Graphics for Data Analysis*. 2nd. Springer. DOI : 10.1007/978-3-319-24277-4.

2.2 Environnements

Les objets de R, données et fonctions, sont nommés. Comme R est modulaire, avec la possibilité de lui ajouter un nombre indéterminé de packages, il est très probable que des conflits de nom apparaissent. Pour les régler, R dispose d'un système rigoureux de précédence des noms : le code s'exécute dans un environnement défini, héritant d'environnements parents.

Organisation

R démarre dans un environnement vide. Chaque package chargé crée un environnement fils pour former une pile des environnements, dont chaque nouvel élément est appelé "fils" du précédent, qui est son "parent".

La console se trouve dans l'environnement global, fils du dernier package chargé.

```r
search()
```

```
##  [1] ".GlobalEnv"         "package:R6"         "package:entropart"
##  [4] "package:forcats"    "package:stringr"    "package:dplyr"
##  [7] "package:purrr"      "package:readr"      "package:tidyr"
## [10] "package:tibble"     "package:ggplot2"    "package:tidyverse"
## [13] "package:kableExtra" "package:stats"      "package:graphics"
## [16] "package:grDevices"  "package:utils"      "package:datasets"
## [19] "package:methods"    "Autoloads"          "package:base"
```

Le code d'une fonction appelée de la console s'exécute dans un environnement fils de l'environnement global :

```r
# Environnement actuel
environment()
```

```
## <environment: R_GlobalEnv>
```

```r
# La fonction f affiche son environnement
f <- function() environment()
# Affichage de l'environnement de la fonction
f()
```

```
## <environment: 0x92da450>
```

```r
# Environnement parent de celui de la fonction
parent.env(f())
```

```
## <environment: R_GlobalEnv>
```

Recherche

La recherche des objets commence dans l'environnement local. S'il n'est pas trouvé, il est cherché dans l'environnement parent, puis dans le parent du parent, jusqu'à l'épuisement des

environnements qui génère une erreur indiquant que l'objet n'a pas été trouvé.

Exemple :

```
# Variable q définie dans l'environnement global
q <- "GlobalEnv"
# Fonction définissant q dans son environnement
qLocalFonction <- function() {
    q <- "Fonction"
    return(q)
}
# La variable locale est retournée
qLocalFonction()
```

```
## [1] "Fonction"
```

```
# Fonction (mal écrite) utilisant une variable qu'elle
# ne définit pas
qGlobalEnv <- function() {
    return(q)
}
# La variable de l'environnement global est retournée
qGlobalEnv()
```

```
## [1] "GlobalEnv"
```

```
# Suppression de cette variable
rm(q)
# La fonction base::q est retournée
qGlobalEnv()
```

```
## function (save = "default", status = 0, runLast = TRUE)
## .Internal(quit(save, status, runLast))
## <bytecode: 0xcf4fc98>
## <environment: namespace:base>
```

La variable `q` est définie dans l'environnement global. La fonction `qLocalFonction` définit sa propre variable `q`. L'appel de la fonction retourne la valeur locale de la fonction parce qu'elle se trouve dans l'environnement de la fonction.

La fonction `qGlobalEnv` retourne la variable `q` qu'elle ne définit pas localement. Elle la recherche donc dans son environnement parent et trouve la variable définie dans l'environnement global. En supprimant la variable de l'environnement global par `rm(q)`, la fonction `qGlobalEnv()` parcourt la pile des environnements jusqu'à trouver un objet nommé `q` dans le package **base**, qui est la fonction permettant de quitter R. Elle aurait pu trouver un autre objet si un package contenant un objet `q`avait été chargé.

Pour éviter ce comportement erratique, une fonction ne doit *jamais* appeler un objet non défini dans son propre environnement.

Espaces de nom des packages

Il est temps de définir précisément ce que les packages rendent visible. Les packages contiennent des objets (fonctions et données) qu'ils *exportent* ou non. Ils sont habituellement appelés par la fonction `library()` qui effectue deux opérations :

- elle *charge* le package en mémoire, ce qui permet d'accéder à tous ses objets avec la syntaxe `package::objet` pour les objets exportés et `package:::objet` pour ceux qui ne le sont pas ;
- elle *attache* ensuite le package, ce qui place son environnement en haut de la pile.

Il est possible de détacher un package avec la fonction `unloadNamespace()` pour le retirer de la pile des environnements. Exemple :

```
# entropart chargé et attaché
library("entropart")
# Est-il attaché ?
isNamespaceLoaded("entropart")
```

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

```
# Pile des environnements
search()
```

```
##  [1] ".GlobalEnv"         "package:R6"         "package:entropart"
##  [4] "package:forcats"    "package:stringr"    "package:dplyr"
##  [7] "package:purrr"      "package:readr"      "package:tidyr"
## [10] "package:tibble"     "package:ggplot2"    "package:tidyverse"
## [13] "package:kableExtra" "package:stats"      "package:graphics"
## [16] "package:grDevices"  "package:utils"      "package:datasets"
## [19] "package:methods"    "Autoloads"          "package:base"
```

```
# Diversity(), une fonction exportée par entropart est
# trouvée
Diversity(1, CheckArguments = FALSE)
```

```
## None
##    1
```

```
# Détacher et décharger entropart
unloadNamespace("entropart")
# Est-il attaché ?
isNamespaceLoaded("entropart")
```

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

```
# Pile des environnements, sans entropart
search()
```

```
##  [1] ".GlobalEnv"        "package:R6"        "package:forcats"
##  [4] "package:stringr"   "package:dplyr"     "package:purrr"
##  [7] "package:readr"     "package:tidyr"     "package:tibble"
## [10] "package:ggplot2"   "package:tidyverse" "package:kableExtra"
## [13] "package:stats"     "package:graphics"  "package:grDevices"
## [16] "package:utils"     "package:datasets"  "package:methods"
## [19] "Autoloads"         "package:base"
```

```r
# Diversity() est introuvable
tryCatch(Diversity(1), error = function(e) print(e))
```

```
## <simpleError in Diversity(1): could not find function "Diversity">
```

```r
# mais peut être appelée avec son nom complet.  L'appel
# exécute implicitement
entropart::Diversity(1, CheckArguments = FALSE)
```

```
## None
##    1
```

L'appel de `entropart::Diversity()` charge le package (c'est-à-dire, exécute implicitement `loadNamespace("entropart")`) mais ne l'attache pas.

En pratique, il faut limiter le nombre de package attachés pour limiter le risque d'appeler une fonction non désirée homonyme de la fonction recherchée. Dans les cas critiques, il faut utiliser le nom complet de la fonction : `package::fonction()`.

Un problème fréquent concerne la `filter()` de **dplyr** homonyme de celle de **stats**. Le package **stats** est habituellement chargé avant **dplyr**, un package du tidyverse. `stats::filter()` doit donc être appelée explicitement.

Cependant, le package **dplyr** ou **tidyverse** (qui attache tous les packages du tidyverse) peut être chargé systématiquement en créant un fichier `.RProfile` à la racine du projet contenant la commande :

```r
library("tidyverse")
```

Dans ce cas, **dplyr** est chargé *avant* **stats** et c'est sa fonction qui est inaccessible.

2.3 Mesure du temps d'exécution

Le temps d'exécution d'un code long peut être mesuré très simplement par la commande `system.time`. Pour des temps d'exécution très courts, il est nécessaire de répéter la mesure : c'est l'objet du package **microbenchmark**.

system.time

La fonction retourne le temps d'exécution du code.

```
# Ecart absolu moyen de 1000 valeurs dans une loi
# uniforme, répété 100 fois
system.time(for (i in 1:100) mad(runif(1000)))
```

```
##    user  system elapsed
##   0.016   0.000   0.016
```

microbenchmark

Le package **microbenchmark* est le plus avancé.

L'objectif est de comparer la vitesse du calcul du carré d'un vecteur (ou d'un nombre) en le multipliant par lui-même ($x \times x$) ou en l'élevant à la puissance 2 (x^2).

```
# Fonctions à tester
f1 <- function(x) x * x
f2 <- function(x) x^2
f3 <- function(x) x^2.1
f4 <- function(x) x^3
# Initialisation
X <- rnorm(10000)
# Test
library("microbenchmark")
(mb <- microbenchmark(f1(X), f2(X), f3(X), f4(X)))
```

```
## Unit: microseconds
##   expr     min      lq      mean   median       uq      max neval
##  f1(X)   8.021   9.737  22.87879  10.4850  11.5425 1189.655   100
##  f2(X)  10.426  11.754  31.73730  12.3980  13.0665 1922.062   100
##  f3(X) 460.194 463.276 486.83238 466.4315 478.4350 1703.005   100
##  f4(X) 773.460 778.590 799.25864 782.3720 790.1645 2123.848   100
```

Le tableau retourné contient les temps minimum, médian, moyen, max et les premiers et troisièmes quartiles, ainsi que le nombre de répétitions. La valeur médiane est à comparer. Le nombre de répétition est par défaut de 100, à moduler (argument `times`) en fonction de la complexité du calcul.

Le résultat du test, un objet de type `microbenchmark`, est un tableau brut des temps d'exécution. L'analyse statistique est faite par les méthodes `print` et `summary`. Pour choisir les colonnes à afficher, utiliser la syntaxe suivante :

```
summary(mb)[, c("expr", "median")]
```

```
##    expr   median
## 1 f1(X)  10.4850
## 2 f2(X)  12.3980
## 3 f3(X) 466.4315
## 4 f4(X) 782.3720
```

Pour faire des calculs sur ces résultats, il faut les stocker dans une variable. Pour empêcher l'affichage dans la console, la solution la plus simple est d'utiliser la fonction `capture.output` en affectant son résultat à une variable.

```
dummy <- capture.output(mbs <- summary(mb))
```

Le test précédent est affiché à nouveau.

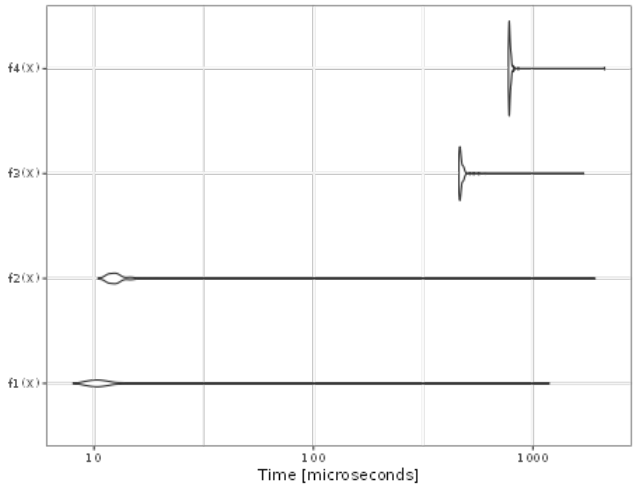
```
summary(mb)[, c("expr", "median")]
```

```
##    expr   median
## 1 f1(X)  10.4850
## 2 f2(X)  12.3980
## 3 f3(X) 466.4315
## 4 f4(X) 782.3720
```

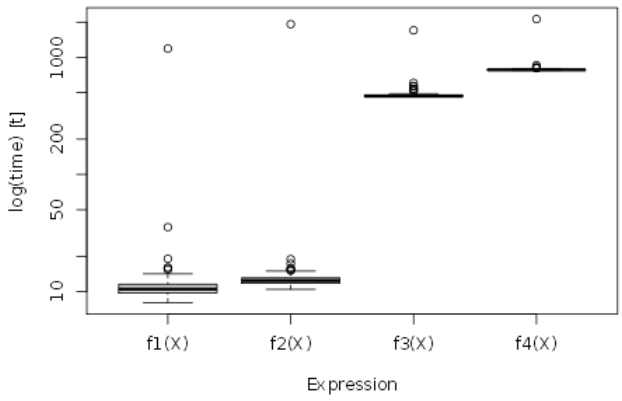
Le temps de calcul est à peu près identique entre $x \times x$ et x^2. Le calcul de puissance est nettement plus long, surtout si la puissance n'est pas entière, parce qu'il nécessite un calcul de logarithme. Le calcul de la puissance 2 est donc optimisé par R pour éviter l'usage du log.

Deux représentations graphiques sont disponibles : les violons représentent la densité de probabilité du temps d'exécution ; les boîtes à moustache sont classiques.

```
library("ggplot2")
autoplot(mb)
```




```
boxplot(mb)
```



Profilage

profvis est l'outil de profilage de RStudio.

Il permet de suivre le temps d'exécution de chaque ligne de code et la mémoire utilisée. L'objectif est de détecter les portions de code lentes, à améliorer.

```
library(profvis)
p <- profvis({
    # Calculs de cosinus
    cos(runif(10^7))
    # 1/2 seconde de pause
    pause(1/2)
})
htmlwidgets::saveWidget(p, "docs/profile.html")
```

Le résultat est un fichier HTML contenant le rapport de profilage[14]. On peut observer que le temps de tirage des nombres aléatoires est similaire à celui du calcul des cosinus.

Lire la documentation complète[15] sur le site de RStudio.

[14] https://EricMarcon.github.io/travailleR/profile.html

[15] https://rstudio.github.io/profvis/

2.4 Boucles

Le cas le plus fréquent de code long à exécuter est celui des boucles : le même code est répété un grand nombre de fois.

Fonctions vectorielles

La plupart des fonctions de R sont vectorielles : les boucles sont traitées de façon interne, extrêmement rapide. Il faut donc raisonner en termes de vecteurs plutôt que de scalaires.

```
# Tirage de deux vecteurs de trois nombres aléatoires
# entre 0 et 1
x1 <- runif(3)
x2 <- runif(3)
# Racine carrée des trois nombres de x1
sqrt(x1)
```

```
## [1] 0.9647806 0.7323823 0.4684709
```

```
# Sommes respective des trois nombres de x1 et x2
x1 + x2
```

```
## [1] 1.6809587 0.9307191 0.5185944
```

Il faut aussi écrire des fonctions vectorielles sur leur premier argument. La fonction `lnq` du package **entropart** retourne le logarithme déformé d'ordre q d'un nombre x.

```
# Code de la fonction
entropart::lnq
```

```
## function (x, q)
## {
##     if (q == 1) {
##         return(log(x))
##     }
##     else {
##         Log <- (x^(1 - q) - 1)/(1 - q)
##         Log[x < 0] <- NA
##         return(Log)
##     }
## }
## <bytecode: 0xde44300>
## <environment: namespace:entropart>
```

Pour qu'une fonction soit vectorielle, chaque ligne de son code doit permetre que le premier argument soit traité comme un vecteur. Ici : `log(x)` et `x^` sont une fonction et un opérateur vectoriels et la condition `[x < 0]` retourne aussi un vecteur.

lapply

Les codes qui ne peuvent pas être écrits comme une fonction vectorielle nécessitent des boucles.

`lapply()` applique une fonction à chaque élément d'une liste. Elle est déclinée sous plusieurs versions :

- `lapply()` renvoie une liste (économise le temps de leur réorganisation dans un tableau) ;
- `sapply()` renvoie un dataframe en rassemblant les listes par `simplify2array()` ;
- `vapply()` est presque identique mais demande que le type de données du résultat soit fourni.

```
# Tirage de 1000 valeurs dans une loi uniforme
x1 <- runif(1000)
# La racine carrée de peut être calculée pour le vecteur
# ou chaque valeur
identical(sqrt(x1), sapply(x1, FUN = sqrt))
```

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

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

```
##                                   expr   median
## 1                             sqrt(x1)   3.3210
## 2                lapply(x1, FUN = sqrt) 255.8515
## 3                sapply(x1, FUN = sqrt) 309.3555
## 4 vapply(x1, FUN = sqrt, FUN.VALUE = 0) 262.0395
```

`lapply()` est beaucoup plus lent qu'une fonction vectorielle. `sapply()` nécessite plus de temps pour `simplify2array()`, qui doit détecter comment rassembler les résultats. Enfin, `vapply()` économise le temps de détermination du type de données du résultat et permet d'accélérer le calcul avec peu d'efforts.

Boucles for

Les boucles sont gérées par la fonction `for`. Elles ont la réputation d'être lentes dans R parce que le code à l'intérieur de la boucle doit être interprété à chaque exécution. Ce n'est plus le cas depuis la version 3.5 de R : les boucles sont compilées systématiquement avant leur exécution. Le comportement du compilateur "juste à temps" est défini par la fonction `enableJIT`. Le niveau par défaut est 3 : les fonctions sont toutes compilées, et les boucles dans le code aussi.

Pour évaluer le gain de performance, le code suivant supprime toute compilation automatique, et compare la même boucle compilée ou non.

```
library("compiler")
# Pas de compilation automatique
enableJIT(level = 0)
```

```
## [1] 3
```

```
# Boucle pour calculer la racine carrée d'un vecteur
Boucle <- function(x) {
    # Initialisation du vecteur de résultat, indispensable
    Racine <- vector("numeric", length = length(x))
    # Boucle
    for (i in 1:length(x)) Racine[i] <- sqrt(x[i])
    return(Racine)
}
# Version compilée
Boucle2 <- cmpfun(Boucle)
# Comparaison
mb <- microbenchmark(Boucle(x1), Boucle2(x1))
(mbs <- summary(mb)[, c("expr", "median")])
```

```
##          expr   median
## 1  Boucle(x1) 676.9250
## 2 Boucle2(x1)  67.0825
```

```
# Compilation automatique par défaut depuis la version
# 3.5
enableJIT(level = 3)
```

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

Le gain est considérable : de 1 à 10.

Les boucles for sont maintenant nettement plus rapides que `lapply`.

```
# Test
mb <- microbenchmark(vapply(x1, FUN = sqrt, 0), Boucle(x1))
summary(mb)[, c("expr", "median")]
```

```
##                        expr   median
## 1 vapply(x1, FUN = sqrt, 0) 254.0765
## 2                Boucle(x1)  66.1980
```

Attention, le test de performance peut être trompeur :

```
# Préparation du vecteur de résultat
Racine <- vector("numeric", length = length(x1))
# Test
mb <- microbenchmark(vapply(x1, FUN = sqrt, 0),
                     for(i in 1:length(x1))
                       Racine[i] <- sqrt(x1[i]))
summary(mb)[, c("expr", "median")]
```

```
##                                            expr   median
## 1                       vapply(x1, FUN = sqrt, 0)  267.189
## 2 for (i in 1:length(x1)) Racine[i] <- sqrt(x1[i]) 2330.077
```

Dans ce code, la boucle for n'est pas compilée donc elle est beaucoup plus lente que dans le cadre normal de son utilisation (dans une fonction ou au niveau supérieur du code).

Les boucles longues permettent un suivi de leur progression par une barre de texte, ce qui est un autre avantage. La fonction suivante exécute des pauses d'un dixième de seconde pendant le temps passé en paramètre (en secondes).

```
BoucleSuivie <- function(duree = 1) {
    # Barre de progression
    pgb <- txtProgressBar(min = 0, max = duree * 10)
    # Boucle
    for (i in 1:(duree * 10)) {
        # Pause d'un dixième de seconde
        Sys.sleep(1/10)
        # Suivi de la progression
        setTxtProgressBar(pgb, i)
    }
}
BoucleSuivie()
```

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

replicate

`replicate()` répète une instruction.

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

```
## [1] 0.64386543 0.03692524 0.99038972
```

est équivalent à `runif(3)`, avec des performances similaires à celles de `vapply` : de 50 à 100 fois plus lent qu'une fonction vectorielle.

```r
mb <- microbenchmark(replicate(1000, runif(1)), runif(1000))
summary(mb)[, c("expr", "median")]
```

```
##                        expr    median
## 1 replicate(1000, runif(1)) 2207.2215
## 2               runif(1000)   28.0435
```

Vectorize

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

Statistiques marginales

`apply` applique une fonction aux lignes ou colonnes d'un objet en deux dimensions.

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

```r
# Somme des colonnes numériques du jeu de données diamonds de ggplot()
# Boucle identique à l'action de apply(, 2, )
BoucleSomme <- function(Table) {
  Somme <- vector("numeric", length = ncol(Table))
  for (i in 1:ncol(Table)) Somme[i] <- sum(Table[, i])
  return(Somme)
}
mb <- microbenchmark(BoucleSomme(diamonds[-(2:4)]),
                     apply(diamonds[-(2:4)], 2, sum),
                     colSums(diamonds[-(2:4)]))
summary(mb)[, c("expr", "median")]
```

```
##                              expr   median
## 1   BoucleSomme(diamonds[-(2:4)]) 2.017111
## 2 apply(diamonds[-(2:4)], 2, sum) 3.953311
## 3       colSums(diamonds[-(2:4)]) 1.394338
```

`apply` clarifie le code mais est plus lent que la boucle, qui est à peine plus lente que `colSums`.

2.5 Code C++

L'intégration de code C++ dans R est largement simplifiée par le package **Rcpp** mais reste difficile à déboguer et donc à réserver à du code très simple (pour éviter toute erreur) et répété un grand nombre de fois (pour mériter l'effort). La préparation des données et leur vérification doivent être exécutées sous R, de même que le traitement et la présentation des résultats.

L'utilisation habituelle est l'inclusion de code C++ dans un package, mais l'utilisation hors package est possible :

- Le code C++ peut être inclus dans un document C++ (fichier avec l'extension `.cpp`) : il est compilé par la commande `sourceCpp()` qui crée les fonctions R permettant d'appeler le code C++.
- Dans un document RMarkdown, des bouts de code Rcpp peuvent être créés pour y insérer le code C++ : ils sont compilés et interfacés pour R au moment du tricotage.

L'exemple suivant montre comment créer une fonction C++ pour calculer le double d'un vecteur numérique.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

Une fonction R du même nom que la fonction C++ est maintenant disponible.

```
timesTwo(1:5)
```

```
## [1]  2  4  6  8 10
```

Les performances sont deux ordres de grandeur plus rapides que le code R (voir l'étude de cas, section 2.7).

2.6 Paralléliser R

Lorsque des calculs longs peuvent être découpés en tâches indépendantes, l'exécution simultanée (*parallèle*) de ces tâches permet de réduire le temps de calcul total à celui de la tâche la plus longue, auquel s'ajoute le coût de la mise en place de la parallélisation (création des tâches, récupération des résultats…).

Lire l'excellente introduction de Josh Errickson[16] détaille les enjeux et les contraintes de la parallélisation.

[16] http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html

Deux mécanismes sont disponibles pour l'exécution de code en parallèle :

- *fork* : le processus en cours d'exécution est dupliqué sur plusieurs coeurs du processeur de l'ordinateur de calcul. C'est la méthode la plus simple mais elle ne fonctionne pas sous Windows (limite du système d'exploitation).
- *socket* : un cluster est constitué, soit physiquement (un ensemble d'ordinateurs exécutant R est nécessaire) soit logiquement (une instance de R sur chaque coeur de l'ordinateur utilisé). Les membres du cluster communiquent par le réseau (le réseau interne de l'ordinateur utilisé pour un cluster logique).

Différents packages de R permettent de mettre en oeuvre ces mécanismes.

mclapply (fork)

La fonction `mclapply` du package **parallel** a la même syntaxe que `lapply` mais parallélise l'exécution des boucles. Sous Windows, elle n'a aucun effet puisque le système ne permet pas les *fork* : elle appelle simplement `lapply`. Cependant, un contournement existe pour émuler `mclapply` sous Windows en appelant `parLapply`, qui utilise un cluster.

```
##
## mclapply.hack.R
##
## Nathan VanHoudnos
## nathanvan AT northwestern FULL STOP edu
## July 14, 2014
##
## A script to implement a hackish version of
## parallel:mclapply() on Windows machines.
## On Linux or Mac, the script has no effect
## beyond loading the parallel library.

require(parallel)
```

```
## Loading required package: parallel
```

```
## Define the hack
mclapply.hack <- function(...) {
  ## Create a cluster
  size.of.list <- length(list(...)[[1]])
  cl <- makeCluster( min(size.of.list, detectCores()) )

  ## Find out the names of the loaded packages
  loaded.package.names <- c(
    ## Base packages
    sessionInfo()$basePkgs,
    ## Additional packages
    names( sessionInfo()$otherPkgs ))

  tryCatch( {

    ## Copy over all of the objects within scope to
    ## all clusters.
    this.env <- environment()
    while( identical( this.env, globalenv() ) == FALSE ) {
      clusterExport(cl,
                    ls(all.names=TRUE, env=this.env),
                    envir=this.env)
      this.env <- parent.env(environment())
    }
    clusterExport(cl,
                  ls(all.names=TRUE, env=globalenv()),
                  envir=globalenv())

    ## Load the libraries on all the clusters
    ## N.B. length(cl) returns the number of clusters
    parLapply( cl, 1:length(cl), function(xx){
      lapply(loaded.package.names, function(yy) {
        require(yy , character.only=TRUE)})
    })

    ## Run the lapply in parallel
```

```
    return( parLapply( cl, ...) )
  }, finally = {
    ## Stop the cluster
    stopCluster(cl)
  })
}

## Warn the user if they are using Windows
if( Sys.info()[['sysname']] == 'Windows' ){
  message(paste(
    "\n",
    "   *** Microsoft Windows detected ***\n",
    "   \n",
    "   For technical reasons, the MS Windows version of mclapply()\n",
    "   is implemented as a serial function instead of a parallel\n",
    "   function.",
    "   \n\n",
    "   As a quick hack, we replace this serial version of mclapply()\n",
    "   with a wrapper to parLapply() for this R session. Please see\n\n",
    "     http://www.stat.cmu.edu/~nmv/2014/07/14/
    implementing-mclapply-on-windows \n\n",
    "   for details.\n\n"))
}

## If the OS is Windows, set mclapply to the
## the hackish version. Otherwise, leave the
## definition alone.
mclapply <- switch( Sys.info()[['sysname']],
                    Windows = {mclapply.hack},
                    Linux   = {mclapply},
                    Darwin  = {mclapply})

## end mclapply.hack.R
```

Le code suivant teste la parallélisation d'une fonction qui renvoie son argument inchangé après une pause d'un quart de seconde. Ce document est tricoté avec 2 coeurs, qui sont tous utilisés.

```
f <- function(x, time = 0.25) {
    Sys.sleep(time)
    return(x)
}
nbCoeurs <- detectCores()
# Série : temps théorique = nbCoeurs/4 secondes
(tserie <- system.time(lapply(1:nbCoeurs, f)))
```

```
##    user  system elapsed
##   0.000   0.002   0.502
```

```
# Parallèle : temps théorique = 1/4 secondes
(tparallele <- system.time(mclapply(1:nbCoeurs, f)))
```

```
##    user  system elapsed
##   0.001   0.013   0.260
```

La mise en place de la parallélisation a un coût de l'ordre du temps de calcul au-delà du quart de seconde attendu : environ 0.01 secondes ici. Le temps d'exécution est bien plus long en parallèle sous Windows parce que la mise en place du cluster prend bien plus de temps que la parallélisation n'en fait gagner.

La parallélisation est intéressante pour des tâches plus longues, comme une pause d'un seconde.

```
# Série
system.time(lapply(1:nbCoeurs, f, time = 1))
```

```
##    user  system elapsed
##   0.001   0.000   2.002
```

```
# Parallèle
system.time(mclapply(1:nbCoeurs, f, time = 1))
```

```
##    user  system elapsed
##   0.003   0.004   1.009
```

Le temps additionnel nécessaire pour l'exécution parallèle du nouveau code est relativement plus faible : les coûts deviennent inférieurs à l'économie quand le temps de chaque tâche s'allonge.

Si le nombre de tâches parallèles dépasse le nombre de coeurs, les performances s'effondrent parce que la tâche supplémentaire doit être exécutée après les premières.

```
system.time(mclapply(1:nbCoeurs, f, time = 1))
```

```
##    user  system elapsed
##   0.000   0.008   1.010
```

```
system.time(mclapply(1:(nbCoeurs + 1), f, time = 1))
```

```
##    user  system elapsed
##   0.000   0.013   2.008
```

Le temps reste ensuite stable jusqu'au double du nombre de coeurs. La figure 2.2 montre l'évolution du temps de calcul en fonction du nombre de tâches.

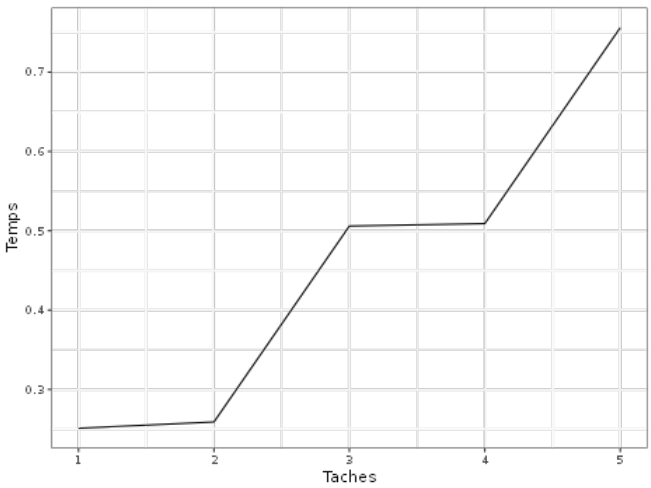


Fig. 2.2 : Temps d'exécution en parallèle de tâches nécessitant un quart de secondes (chaque tâche est une pause d'un quart de seconde). Le nombre de tâches varie de 1 à deux fois le nombre de coeurs (égal à 2) plus une.

Si le nombre de tâches est petit, il faut donc l'adapter au nombre de coeurs sous peine de perte de performance.

parLapply (socket)

`parLapply` nécessite de créer un cluster, exporter les variables utiles sur chaque noeud, charger les packages nécessaires sur chaque noeud, exécuter le code et enfin arrêter le cluster. Le code de chaque étape se trouve dans la fonction `mclapply.hack` ci-dessus.

Pour un usage courant, `mclapply` est plus rapide, sauf sous Windows, et plus simple (y compris sous Windows grâce au contournement ci-dessus.)

foreach

Fonctionnement

Le package **foreach** permet un usage avancé de la parallélisation. Lire ses vignettes.

```
# Manuel
vignette("foreach", "foreach")
# Boucles imbriquées
vignette("nested", "foreach")
```

Indépendamment de la parallélisation, **foreach** redéfinit les boucles *for*.

```
for (i in 1:3) {
    f(i)
}
# devient
library("foreach")
```

```
##
## Attaching package: 'foreach'


## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
foreach(i = 1:3) %do% {
    f(i)
}
```

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

La fonction `foreach` retourne une liste contenant les résultats de chaque boucle. Les éléments de la liste peuvent être combinés par une fonction quelconque, comme `c`.

```
foreach(i = 1:3, .combine = "c") %do% {
    f(i)
}
```

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

La fonction `foreach` est capable d'utiliser des itérateurs, c'est-à-dire des fonctions qui ne passent à la boucle que les données dont elle a besoin sans charger les autres en mémoire. Ici, l'itérateur `icount` passe les valeurs 1, 2 et 3 individuellement, sans charger le vecteur 1 :3 en mémoire.

```
library("iterators")
foreach(i = icount(3), .combine = "c") %do% {
    f(i)
}
```

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

Elle est donc très utile quand chaque objet de la boucle utilise une grande quantité de mémoire.

Parallélisation

Remplacer l'opérateur `%do%` par `%dopar%` parallélise les boucles, à condition qu'un adaptateur, c'est-à-dire un package intermédiaire entre `foreach` et un package chargé de l'implémentation de la parallélisation, soit chargé. **doParallel** est un adaptateur pour utiliser le package **parallel** livré avec R.

```
library(doParallel)
registerDoParallel(cores = nbCoeurs)
# Série
system.time(foreach(i = icount(nbCoeurs), .combine = "c") %do%
    {
        f(i)
    })
```

```
##    user  system elapsed
##   0.003   0.000   0.504
```

```
# Parallèle
system.time(foreach(i = icount(nbCoeurs), .combine = "c") %dopar%
    {
        f(i)
    })
```

```
##    user  system elapsed
##   0.014   0.003   0.262
```

Le coût fixe de la parallélisation est faible.

2.7 Etude de cas

Cette étude de cas permet de tester les différentes techniques vues plus haut pour résoudre un problème concret. L'objectif est de calculer la distance moyenne entre deux points d'un semis aléatoire de 1000 points dans une fenêtre carrée de côté 1.

Création des données

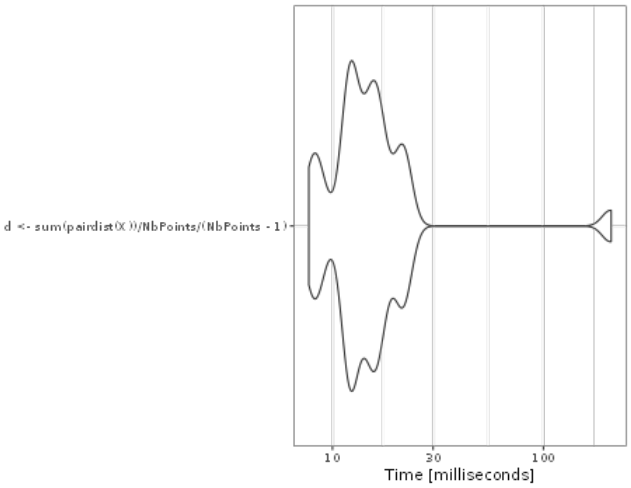
Le semis de points est créé avec le package **spatstat**.

```
NbPoints <- 1000
library("spatstat")
X <- runifpoint(NbPoints)
```

Spatstat

La fonction `pairdist()` de **spatstat** retourne la matrice des distances entre les points. La distance moyenne est calculée en divisant la somme par le nombre de paires de points distincts.

```
mb <- microbenchmark(d <- sum(pairdist(X))/NbPoints/(NbPoints -
    1))
# suppressmessages pour éliminer les messages superflus
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5103392
```

La fonction est rapide parce qu'elle est codée en langage C dans le package **spatstat** pour le coeur de ses calculs.

apply

La distance peut être calculée par deux `sapply()` imbriqués.

```
fsapply1 <- function() {
    distances <- sapply(1:NbPoints, function(i) sapply(1:NbPoints,
        function(j) sqrt((X$x[i] - X$x[j])^2 + (X$y[i] -
            X$y[j])^2)))
    return(sum(distances)/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply1())
```

```
##    user  system elapsed
##   4.746   0.012   4.758
```

```
d
```

```
## [1] 0.5103392
```

Un peu de temps peut être gagnée en remplaçant sapply par vapply : le format des résultats n'a pas à être déterminé par la fonction. Le gain est négligeable sur un long calcul comme celui-ci mais important pour des calculs courts.

```
fsapply2 <- function() {
    distances <- vapply(1:NbPoints, function(i) vapply(1:NbPoints,
        function(j) sqrt((X$x[i] - X$x[j])^2 + (X$y[i] -
            X$y[j])^2), 0), 1:1000 + 0)
    return(sum(distances)/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply2())
```

```
##    user  system elapsed
##   4.328   0.000   4.328
```

```
d
```

```
## [1] 0.5103392
```

Le format de sortie n'est pas toujours évident à écrire :

- il doit respecter la taille des données : un vecteur de taille 1000 pour la boucle externe, un scalaire pour la boucle interne.
- il doit respecter leur type : `0` pour un entier, `0.0` pour un réel. Dans la boucle externe, l'ajout de `0.0` au vecteur d'entiers le transforme en vecteur de réels.

Une amélioration plus significative consiste à ne calculer les racines carrées qu'à la fin de la boucle, pour profiter de la vectorisation de la fonction.

```
fsapply3 <- function() {
    distances <- vapply(1:NbPoints, function(i) vapply(1:NbPoints,
        function(j) (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2,
        0), 1:1000 + 0)
    return(sum(sqrt(distances))/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply3())
```

```
##    user  system elapsed
##   4.326   0.008   4.334
```

```
d
```

```
## [1] 0.5103392
```

Les calculs sont effectués deux fois (distance entre les points i et j, mais aussi entre les points j et i) : un test sur les indices permet de diviser presque le temps par 2 (pas tout à fait parce que les boucles sans calcul, qui retournent 0, prennent du temps).

```
fsapply4 <- function() {
    distances <- vapply(1:NbPoints, function(i) {
        vapply(1:NbPoints, function(j) {
            if (j > i) {
                (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2
            } else {
                0
            }
        }, 0)
    }, 1:1000 + 0)
    return(sum(sqrt(distances))/NbPoints/(NbPoints - 1) *
        2)
}
system.time(d <- fsapply4())
```

```
##    user  system elapsed
##   2.534   0.008   2.542
```

```
d
```

```
## [1] 0.5103392
```

En parallèle, le temps de calcul n'est pas amélioré parce que les tâches individuelles sont trop courtes.

```
fsapply5 <- function() {
    distances <- mclapply(1:NbPoints, function(i) {
        vapply(1:NbPoints, function(j) {
            if (j > i) {
                (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2
            } else {
                0
            }
        }, 0)
    })
    return(sum(sqrt(simplify2array(distances)))/NbPoints/(NbPoints -
        1) * 2)
}
system.time(d <- fsapply5())
```

```
##    user  system elapsed
##   4.924   0.240   2.601
```

```
d
```

```
## [1] 0.5103392
```

boucle for

Une boucle for est plus rapide et consomme moins de mémoire parce qu'elle ne stocke pas la matrice de distances.

```
distance <- 0
ffor <- function() {
    for (i in 1:(NbPoints - 1)) {
        for (j in (i + 1):NbPoints) {
            distance <- distance + sqrt((X$x[i] - X$x[j])^2 +
                (X$y[i] - X$y[j])^2)
        }
    }
    return(distance/NbPoints/(NbPoints - 1) * 2)
}
system.time(d <- ffor())
```

```
##    user  system elapsed
##   1.671   0.000   1.670
```

```
d
```

```
## [1] 0.5103392
```

boucle foreach

Deux boucles foreach imbriquées sont nécessaires ici : elles sont extrêmement lente en comparaison d'une boucle simple. Le test est lancé ici avec 10 fois moins de points, donc 100 fois moins de distances à calculer.

```
NbPointsReduit <- 100
Y <- runifpoint(NbPointsReduit)
fforeach1 <- function(Y) {
    distances <- foreach(i = 1:NbPointsReduit, .combine = "cbind") %:%
        foreach(j = 1:NbPointsReduit, .combine = "c") %do%
        {
            if (j > i) {
                (Y$x[i] - Y$x[j])^2 + (Y$y[i] - Y$y[j])^2
            } else {
                0
            }
        }
    return(sum(sqrt(distances))/NbPointsReduit/(NbPointsReduit -
        1) * 2)
}
system.time(d <- fforeach1(Y))
```

```
##    user  system elapsed
##   1.824   0.000   1.824
```

```
d
```

```
## [1] 0.5267848
```

Les boucles foreach imbriquées sont à réserver à des tâches très longues (plusieurs secondes au moins) pour amortir les coûts fixes de leur mise en place. La parallélisation ralentit encore le calcul ici parce que les tâches élémentaires sont courtes.

```
registerDoParallel(cores = detectCores())
fforeach3 <- function(Y) {
  distances <-
    foreach(i=icount(NbPointsReduit),
            .combine='+') %dopar% {
      distance <- 0
      for (j in 1:Y$n) {
        distance <- distance +
          sqrt((Y$x[i]-Y$x[j])^2 + (Y$y[i]-Y$y[j])^2)
      }
      distance
    }
  return(distances/NbPointsReduit/(NbPointsReduit-1))
}
system.time(d <- fforeach3(Y))
```

```
##    user  system elapsed
##   0.125   0.037   0.099
```

```
d
```

```
## [1] 0.5267848
```

foreach dispose d'adaptateurs optimisés permettant d'utiliser des clusters physiques par exemple. Son intérêt est limité avec le package **parallel**.

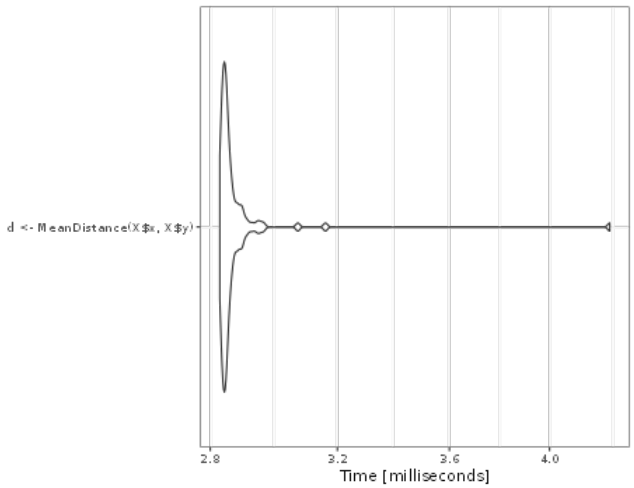
RCpp

La fonction C++ permettant de calculer les distances est la suivante.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double MeanDistance(SEXP Rx, SEXP Ry) {
  // x, y coordinates of points
  NumericVector x(Rx);
  NumericVector y(Ry);
  double distance=0;
  double dx, dy;
  for (int i=0; i < (x.length()-1); i++) {
    for (int j=i+1; j < x.length(); j++) {
    // Calculate distance
        dx = x[i]-x[j];
        dy = y[i]-y[j];
        distance += sqrt(dx*dx + dy*dy);
    }
  }
  return distance/(double)(x.length()/2*(x.length()-1));
}
```

Elle est appelée dans R très simplement. Le temps d'exécution est très court.

```
mb <- microbenchmark(d <- MeanDistance(X$x, X$y))
# suppressmessages pour éliminer les messages superflus
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5103392
```

RcppParallel

RcppParallel permet d'interfacer du code C++ parallélisé, au prix d'une syntaxe plus complexe qu'avec **RCpp**. Une documentation est disponible[17].

[17] http://rcppcore.github.io/RcppParallel/

La fonction C++ exportée dans R ne réalise pas les calculs mais organise seulement l'exécution en parallèle d'une autre fonction, non exportée, de type `Worker`.

Deux fonctions (C++) de parallélisation sont disponibles pour deux types de tâches :

- `parallelReduce` pour l'accumulation d'une valeur, utilisée ici pour additionner les distances.
- `parallelFor` pour remplir une matrice de résultats.

La syntaxe du `Worker` est un peu laborieuse mais assez simple à adapter : les constructeurs initialisent les variables C à partir des valeurs transmises par R et déclarent la parallélisation.

```
// [[Rcpp::depends(RcppParallel)]]
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp;
using namespace RcppParallel;

// Fonction de travail, non exportée
struct TotalDistanceWrkr : public Worker
{
  // source vectors
  const RVector<double> Rx;
  const RVector<double> Ry;

  // accumulated value
  double distance;
```

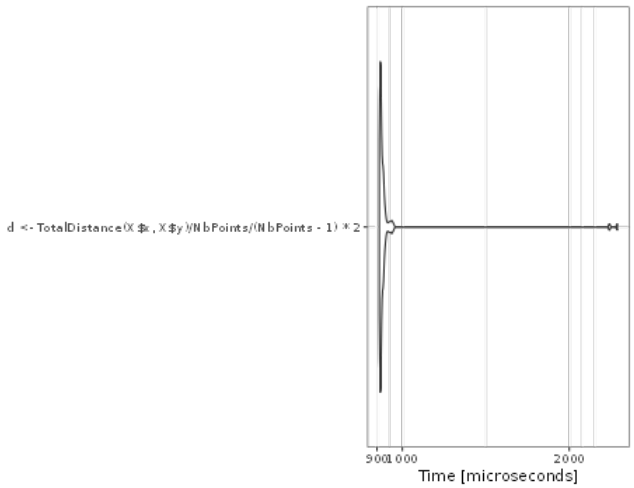
```
  // constructors
  TotalDistanceWrkr(const NumericVector x, const NumericVector y) :
    Rx(x), Ry(y), distance(0) {}
  TotalDistanceWrkr(const TotalDistanceWrkr& TotalDistanceWrkr, Split) :
    Rx(TotalDistanceWrkr.Rx), Ry(TotalDistanceWrkr.Ry),  distance(0) {}

  // count neighbors
  void operator()(std::size_t begin, std::size_t end) {
    double dx, dy;
    unsigned int Npoints = Rx.length();

    for (unsigned int i = begin; i < end; i++) {
      for (unsigned int j=i+1; j < Npoints; j++) {
          // Calculate squared distance
          dx = Rx[i]-Rx[j];
          dy = Ry[i]-Ry[j];
          distance += sqrt(dx*dx + dy*dy);
      }
    }
  }

  // join my value with that of another Sum
  void join(const TotalDistanceWrkr& rhs) {
    distance += rhs.distance;
  }
};


// Fonction exportée
// [[Rcpp::export]]
double TotalDistance(NumericVector x, NumericVector y) {

  // Declare TotalDistanceWrkr instance
  TotalDistanceWrkr totalDistanceWrkr(x, y);

  // call parallel_reduce to start the work
  parallelReduce(0, x.length(), totalDistanceWrkr);

  // return the result
  return totalDistanceWrkr.distance;
}
```

L'usage dans R est identique à celui des fonctions C++ interfacées par **RCpp**.

```
(mb <- microbenchmark(d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints -
    1) * 2))
```

```
## Unit: microseconds
##                                                      expr    min      lq
##  d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints - 1) * 2 908.53 912.097
##      mean   median       uq      max neval
##  948.1121 914.2095 922.9775 2442.329   100
```

```
# suppressmessages pour éliminer les messages superflus
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5103392
```

Le temps de mise en place des tâches parallèles est bien plus long que le temps de calcul en série.

En multipliant le nombre de points par 50, le temps de calcul en série doit être multiplié par 2500 environ.

```
NbPoints <- 50000
X <- runifpoint(NbPoints)
system.time(d <- MeanDistance(X$x, X$y))
```

```
##    user  system elapsed
##   7.066   0.000   7.066
```

En parallèle, le temps augmente peu.

```
system.time(d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints -
    1) * 2)
```

```
##    user  system elapsed
##   4.552   0.000   2.287
```

Conclusions sur l'optimisation de la vitesse du code

L'optimisation du temps de calcul sous R peut être compliquée si elle passe par la parallélisation et l'écriture de code C++. L'effort doit donc être concentré sur les calculs réellement long alors que la lisibilité du code doit rester la priorité pour le code courant.

L'utilisation de boucles for n'est plus pénalisante depuis la version 3.5 de R. L'écriture de code vectoriel, utilisant `sapply()` se justifie toujours pour sa lisibilité.

Le code C est assez facile à intégrer grâce à **RCpp** et sa parallélisation n'est pas très coûteuse avec **RCppParallel**.

Le choix de paralléliser le code doit être évalué selon le temps d'exécution de chaque tâche parallélisable. S'il dépasse quelques secondes, la parallélisation se justifie. `mclapply()` remplace `lapply()` sans aucun effort, mais nécessite un hack (fourni ici) sous Windows. `foreach()` ne remplace pas `for()` aussi simplement et ne se justifie que pour des tâches très lourdes en termes de mémoire et de temps de calcul, en particulier sur des clusters de calcul.

Chapitre 3

Git et GitHub

Le contrôle de source consiste à enregistrer l'ensemble des modifications apportées sur les fichiers suivis. Les avantages sont nombreux : traçabilité et sécurité du projet, possibilité de collaborer efficacement, de revenir en arrière, de tenter de nouveaux développements sans mettre en péril la version stable…

3.1 Principes

Contrôle de source

L'outil standard est aujourd'hui *git*.

Les commandes de git peuvent être exécutées dans le terminal de RStudio.

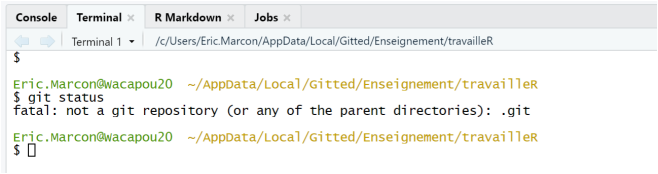


Fig. 3.1 : Capture d'écran du terminal de RStudio. La commande `git status` supposée décrire l'état du dépôt renvoie une erreur si le projet R n'est pas sous contrôle de source.

La commande `git status` (figure 3.1) retourne l'état du dépôt (*repository*), c'est-à-dire l'ensemble des données gérées par git pour suivre le projet en cours.

RStudio intègre une interface graphique pour git suffisante pour se passer de la ligne de commande dans le cadre d'une utilisation standard, présentée ici.

git et GitHub

git est le logiciel installé sur le poste de travail.

GitHub est une plateforme, accessible par le web[1], qui permet de partager le contenu des dépôts git (pour travailler à plusieurs) et de partager de la documentation sous la forme d'un site web (*GitHub Pages*).

[1] https://github.com/

Comme GitHub permet au minimum la sauvegarde des dépôts git, les deux sont toujours utilisés ensemble. GitHub n'est pas la seule plateforme utilisable mais la principale. Les alternatives sont Bitbucket[2] et GitLab[3] par exemple.

[2] https://bitbucket.org/

[3] https://about.gitlab.com/

3.2 Créer un nouveau dépôt

A partir d'un projet existant

Dans un projet R existant, activer le contrôle de source dans les options du projet (figure 3.2). La commande exécutée est `git init`. Redémarrer RStudio à la demande.

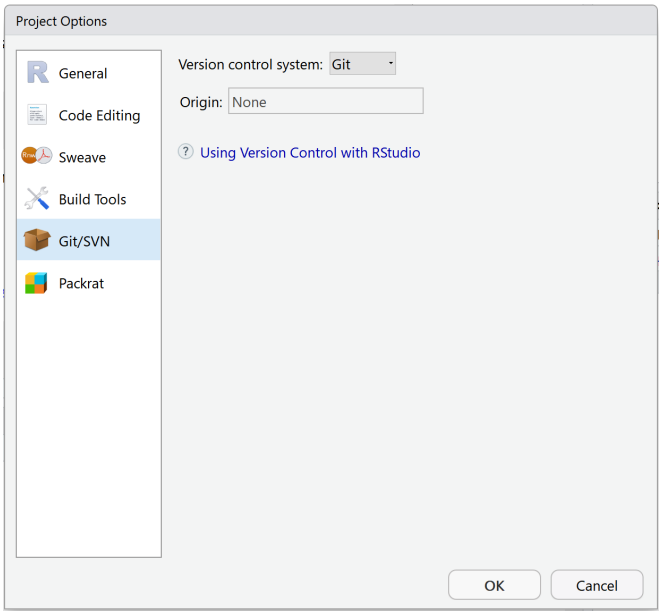


FIG. 3.2 : Activation du contrôle de source dans le menu "Tools > Project Options...".

Une nouvelle fenêtre *Git* apparaît dans le panneau supérieur droit. Elle contient la liste des fichiers du projet (figure 3.3).

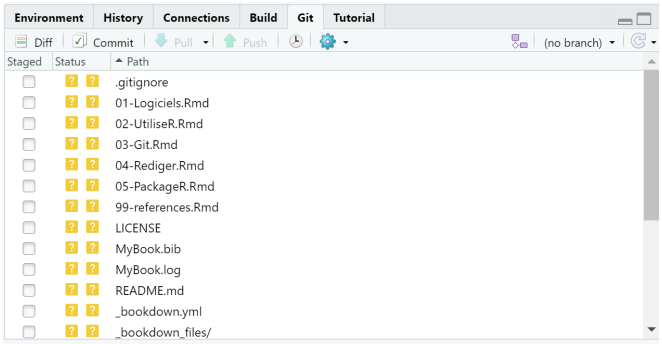


FIG. 3.3 : Fichiers du projet, pas encore pris en compte par git.

A ce stade, les fichiers ne sont pas pris en compte par git : leur statut est un double point d'interrogation jaune. Pour git, le répertoire de travail local est un *bac à sable* où toutes les modifications sont possibles sans conséquences.

Le fichier `.gitignore` contient la liste des fichiers qui n'ont jamais vocation à être pris en compte, qu'il est donc inutile d'afficher dans la liste : les fichiers intermédiaires produits automatiquement par exemple. La syntaxe des fichiers `.gitignore` est détaillée dans la documentation de git[4]. En règle générale, utiliser un fichier existant : les modèles de documents notamment incluent leur fichier `.gitignore`.

[4] https://git-scm.com/docs/gitignore

Prendre en compte des fichiers

Dans la fenêtre git, cocher la case *Staged* permet de prendre en compte (*Stage*) chaque fichier. La commande exécutée est `git add <NomDeFichier>`. Les fichiers pris en compte une première fois ont le statut "A" pour "Added".

Les fichiers pris en compte font partie de l'*index* de git.

Valider des modifications

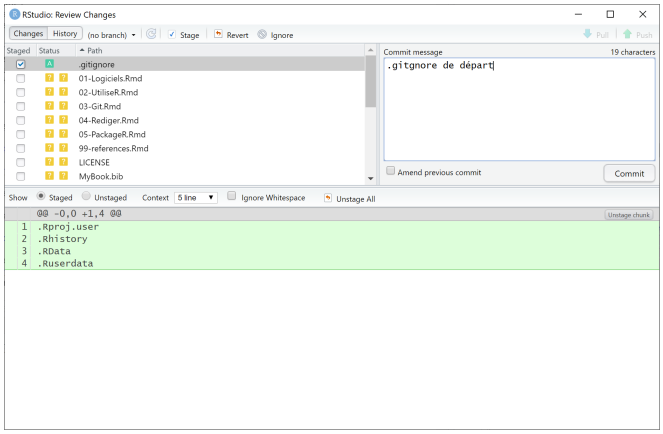


Fig. 3.4 : Fenêtre de validation des modifications prises en compte.

Les fichiers pris en compte peuvent être validés (*Commit*) en cliquant sur le bouton *Commit* dans la fenêtre *Git*. Une nouvelle fenêtre s'ouvre (figure 3.4), qui permet de visualiser toutes les modifications par fichier (ajouts en verts, suppressions en rouge). Le grain de modification traité par git est la ligne de texte, terminée par un retour à la ligne. Les fichiers binaires comme les images sont traités en bloc.

Chaque validation (*Commit*) est accompagnée d'un texte de description. La première ligne est la description courte. Une description détaillée peut être ajoutée après un saut de ligne. Pour la lisibilité de l'historique du projet, chaque *commit* correspond donc à une action, correspondant à la description courte : tous les fichiers modifiés ne sont pas forcément pris en compte et validés en une fois. La commande exécutée est `git commit -m "Message de validation"`.

Les validations sont liées à leur auteur, qui doit être identifié par git. En règle générale, git utilise les informations du système.

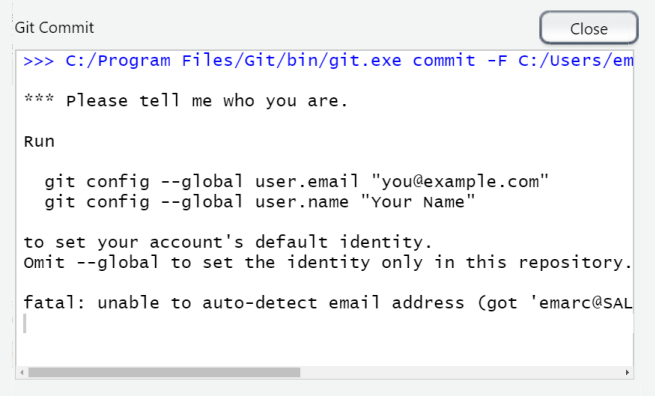


FIG. 3.5 : Fenêtre de demande d'indentification.

S'il n'y parvient pas, une fenêtre demande à l'utilisateur de s'identifier avant d'effectuer son premier *commit* (figure 3.5). Les commandes indiquées sont à exécuter dans le terminal de RStudio. Elles peuvent aussi être utilisées pour vérifier les valeurs connues par git :

```
git config user.name
git config user.email
```

Dès la première validation, la branche principale du dépôt, appelée "master", est créée. Une branche est une version du dépôt, avec son propre historique et donc ses propres fichiers. Les branches permettent : * de développer de nouvelles fonctionnalités dans un projet, sans perturber la branche principale qui peut contenir une version stable. Si le développement est retenu, sa branche pourra être fusionnée avec la branche *master* pour constituer une nouvelle version stable. * de contenir des fichiers totalement différents de ceux de la branche principale, pour d'autres objectifs. Sur GitHub, les pages web de présentation du projet peuvent être placés dans une branche appelée "gh-pages" qui ne sera jamais fusionnée.

Le dépôt git est complètement constitué. Dans le vocabulaire de git, il comprend trois *arbres* (figure 3.6) :

- le répertoire de travail, ou bac à sable, qui contient les fichiers non pris en compte : inconnus, modifiés, supprimés ou renommés (case *Staged* décochée) ;
- l'index, qui contient les fichiers pris en compte (case *Staged* cochée) ;
- la tête, qui contient les fichiers validés.

Le statut des fichiers est représenté par deux icônes dans la fenêtre *Git* de RStudio : deux points d'interrogation quand ils n'ont pas été pris en compte par git. Ensuite, l'icône de droite décrit la différence entre le le répertoire de travail et l'index. Celle de gauche décrit la différence entre l'index et la tête. Un fichier

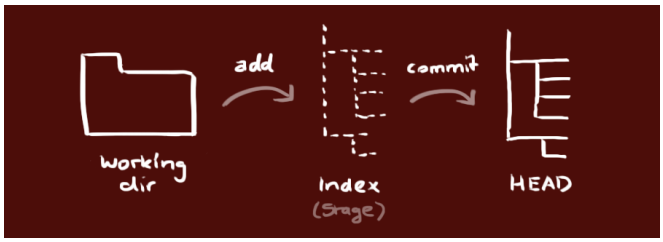


FIG. 3.6 : Les trois arbres de git. Source : https://rogerdudler.github.io/git-guide/index.fr.html

modifié aura donc l'icône `M` affichée à droite avant d'être pris en compte, puis à gauche après prise en compte. Il est possible, même s'il vaut mieux l'éviter, de modifier à nouveau un fichier pris en compte avant qu'il soit validé : alors, les deux icônes seront affichées.

Créer un dépôt vide sur GitHub

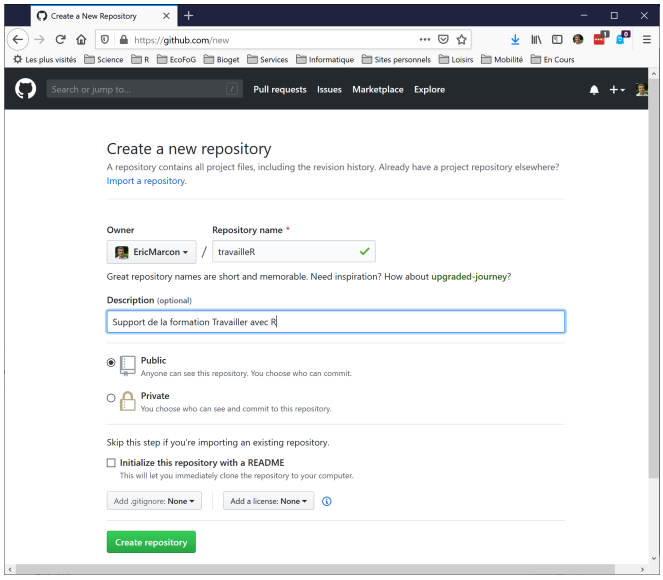


FIG. 3.7 : Création d'un dépôt sur GitHub.

Un dépôt vide sur GitHub doit être créé (figure 3.7) :

- Sur GitHub, cliquer sur le bouton vert *New repository*.
- Saisir le nom du dépôt, identique à celui du projet R local.
- Ajouter une description, qui apparaîtra uniquement sur la page GitHub du dépôt.
- Choisir le statut du dépôt :
  - Public : visible par tout le monde
  - Privé : visible seulement par les collaborateurs du projet, ce qui exclut de compléter par des pages web de présentation.
- Ne pas ajouter de `README`, `.gitIgnore` ou licence : le projet doit être vide.
- Cliquer sur "create Repository".
- Copier l'adresse du dépôt (https://github.com/...)

Lier git et GitHub

Dans RStudio, un premier *commit* doit au moins avoir eu lieu pour que la branche principale du projet, nommée “master”, existe. En haut à droite de la fenêtre *Git* (figure 3.3), il est affiché “(no branch)” avant cela. Ensuite, il est affiché “master”, le nom par défaut de la branche principale du projet. Le projet peut alors être lié au dépôt GitHub.

Cliquer sur le bouton violet à côté de “master” : une fenêtre apparaît (habituellement utilisée pour la création d’une nouvelle branche, voir section 3.4). Saisir le nom de la branche “master”, cliquer sur “Add Remotes” et compléter :

- Remote Name : `origin` ;
- Remote URL : coller l’adresse du dépôt GitHub ;
- Cliquer sur *Add*.

Cocher la case “Sync with Remote”.

Au message indiquant qu’une branche *master* existe déjà, cliquer sur “Overwrite”.

Le nom *origin* est une convention de git. Il peut être modifié mais l’organisation du projet sera plus lisible en respectant la convention.

A la première connexion de RStudio à GitHub, une fenêtre d’authentification permet de saisir ses identifiants GitHub (figure 3.8).

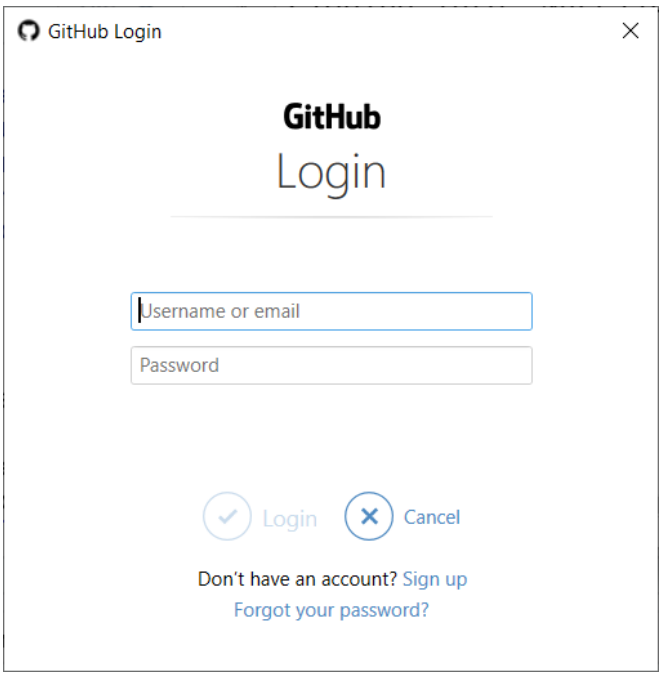


Fig. 3.8 : Création d’un dépôt sur GitHub.

GitHub délivre un jeton d’accès personnel (PAT) à git (voir section 6.2 pour plus de détails) pour éviter de répéter la saisie des

informations d'authentification à chaque opération. Un message électronique est envoyé au titulaire du compte GitHub pour l'en avertir.

Pousser les premières modifications

La manipulation précédente a automatiquement poussé (*Push*) les modifications validées sur GitHub. Par la suite, il faudra cliquer sur le bouton *Push* de la fenêtre *Git* pour le faire.

Sur GitHub, les fichiers résultant des modifications enregistrées par git sont maintenant visibles.

Chaque *commit* réalisé localement est compté par git et un message "Your branch is ahead of 'origin/master' by *n* commits" affiché dans en haut de la fenêtre *Git* indique qu'il est temps de mettre à jour GitHub en poussant l'ensemble de ces *commits*. Cliquer sur le bouton *Push* pour le faire.

A ce stade, le projet doit disposer d'un fichier `README.md` qui présente son contenu sur GitHub. Son contenu minimal est un titre et quelques lignes de description :

```
# Nom du Projet

Description en quelques lignes.
```

Il est conseillé d'utiliser des badges [5], à placer juste après le titre, pour déclarer l'état de maturité du projet, par exemple :

[5] https://github.com/orangemug/stability-badges

```
![stability-wip](https://img.shields.io/badge/|>
stability-work_in_progress-lightgrey.svg)
```

Cloner un dépôt de GitHub

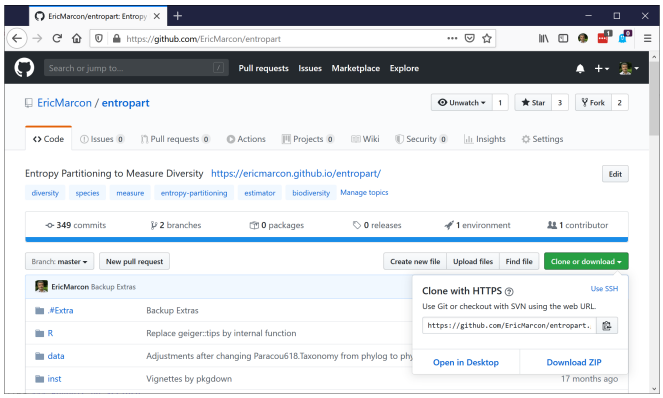


Fig. 3.9 : Création d'un dépôt sur GitHub.

Tout dépôt sur GitHub peut être installé (on dit *cloné*) sur le poste de travail en copiant son adresse qui apparaît en cliquant sur le bouton vert (figure 3.9).

Dans RStudio, créer un nouveau projet et, dans l'assistant, choisir "Version Control", "Git" et coller l'adresse dans le champ "Repository URL". Le nom répertoire à créer pour le projet est déduit automatiquement de l'adresse. Choisir le répertoire dans lequel celui du projet va être créé et cliquer sur "Create Project". Le projet créé est lié au dépôt distant sur GitHub.

Pour travailler à plusieurs sur le même projet, le propriétaire du projet doit donner l'accès au projet à des collaborateurs (figure 3.10), c'est-à-dire d'autres utilisateurs GitHub dans les réglages du dépôt (*Settings*).

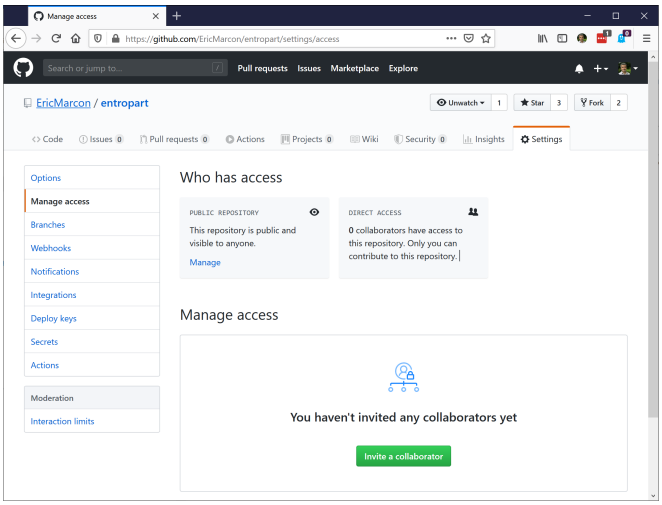


FIG. 3.10 : Création d'un dépôt sur GitHub.

Les collaborateurs sont invités par un message envoyé par *GitHub*.

3.3 Usage courant

Tirer, modifier, valider, pousser

Toute séance de travail sur un projet commence en tirant (Bouton *Pull*) de la fenêtre *Git* pour intégrer au dépôt local les mises à jour effectuées sur GitHub par d'autres collaborateurs.

Les modifications apportées aux fichiers du projet sont ensuite prises en compte (cocher les cases *Staged*) et validées (*Commit*) avec un message explicatif. Une bonne pratique consiste à valider les modifications à chaque fois qu'une tâche élémentaire, qui peut être décrite dans le message explicatif, est terminée plutôt que d'effectuer des *commits* regroupant de nombreux changements avec une description vague.

Dès que possible, pousser (*Push*) les mises à jour pour qu'elles soient visibles par les collaborateurs.

Régler les conflits

Il n'est pas possible de pousser les modifications validées si un collaborateur a modifié le dépôt distant sur GitHub. Il faut alors les tirer pour les intégrer au dépôt local avant de pousser les modifications fusionnées.

Un conflit a lieu si un *Pull* importe dans le fichier local une modification qui ne peut pas être fusionnée automatiquement parce qu'une modification contradictoire a eu lieu localement. Git considère chaque ligne comme un élément indivisible : la modification de la même ligne sur le dépôt distant et le dépôt local génère donc un conflit.

Git insère dans le fichier contenant un conflit les deux versions avec une présentation particulière :

```
<<<<<<<<< HEAD # Version importée du conflit
Lignes en conflit, version importée
========= # limite entre les deux versions
Lignes en conflit, version locale
>>>>>>>>> # Fin du conflit
```

Les lignes de formatage contenant les `<<<<`, les `====` et les `>>>>` doivent être supprimés et une seule version des lignes problématiques conservée, qui peut être différente des deux versions originales. La résolution du conflit doit être prise en compte et validée.

Pour limiter les conflits dans un document contenant du texte (typiquement, un document R Markdown), une bonne pratique consiste à traiter chaque phrase comme une ligne, terminée par un retour à la ligne qui ne sera pas visible dans le document mis en forme : un saut de ligne est nécessaire pour séparer les paragraphes.

Voir les différences

Dans la fenêtre *Git* de RStudio, le menu contextuel (affiché par un clic droit) "Diff" peut être utilisé pour afficher les modifications apportées à chaque fichier (figure 3.11).

Revenir en arrière

Le menu contextuel "Revert" permet d'annuler toutes les modifications apportées à un fichier (affichées par *Diff*) et de rétablir son contenu validé la dernière fois (son état dans la tête).

Il n'est pas simple de revenir en arrière au-delà de la dernière validation parce que les modifications ont pu être prises en compte par des collaborateurs : leur suppression rendrait le projet incohérent.

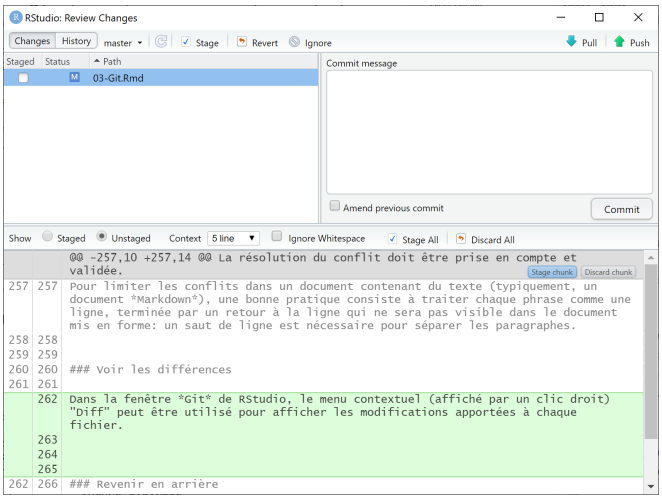


FIG. 3.11 : Différences entre le répertoire de travail et la tête.

Voir l'historique

Le bouton en forme d'horloge de la fenêtre *Git* de RStudio affiche l'historique du projet (figure 3.12).

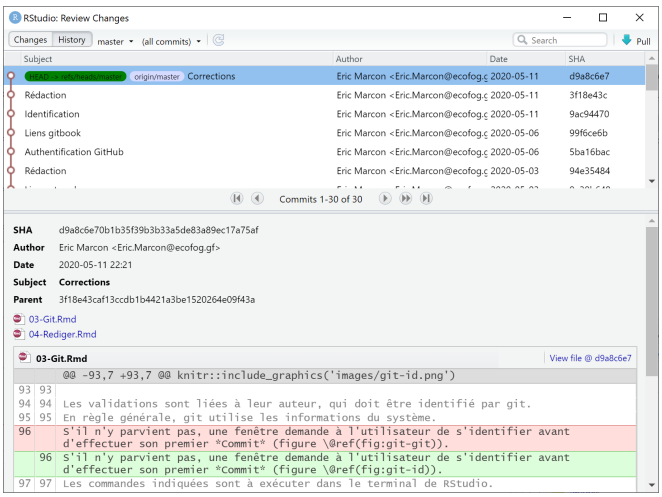


FIG. 3.12 : Historique des validations dans le dépôt.

En haut se trouve la tête, puis toutes les validations (*commits*) qui l'ont constituée. Pour chaque validation, les différences de chaque fichier peuvent être affichées en cliquant sur le nom du fichier dans la partie basse de la fenêtre.

3.4 Branches

Les branches d'un projet sont des versions différentes mais simultanées. Un usage typique est le développement d'une nouvelle fonctionnalité. Si son écriture prend du temps, le projet est perturbé par le chantier en cours : le code peut ne plus fonctionner. Si le développement s'avère impossible ou inutile, il faut pouvoir l'abandonner sans dommage. Pour l'isoler pendant

sa réalisation et se permettre de le valider ou de l'abandonner à la fin, il faut le placer dans une branche.

La branche principale du projet s'appelle "master" ou "main" à partir de novembre 2020[6]). Elle doit toujours être dans un état stable : c'est elle qui est clonée à partir de GitHub par d'autres utilisateurs éventuels.

[6] https://github.com/github/renaming

Le changement de convention pour le nom de la branche "master" fait qu'à partir de novembre 2020, les projets créés sur GitHub clonés dans RStudio ont pour branche principale "main" alors que les projets créés sur RStudio puis liés à GitHub conservent le nom "master".

Créer une nouvelle branche

Cliquer sur le bouton violet *New Branch* dans la fenêtre *git* de RStudio. Saisir son nom et cliquer sur *Create*.

La nouvelle branche est maintenant active.

Les commandes git peuvent aussi être exécutées dans le terminal (pour créer la branche et l'activer) :

```
git branch new_branch
git checkout new_branch
```

Changer de branche

Sélectionner la branche à activer dans la liste des branches locales de la la fenêtre *git*.

Les *commits* s'appliquent à la branche active. Chaque branche se comporte comme une version différente du projet.

Attention : pour éviter la confusion, sauvegarder les modifications, prendre en compte et valider les changements avant de changer de branche.

Fusionner avec `merge`

L'interface graphique de RStudio ne prévoit pas les fusions, il faut donc utiliser le terminal : tout d'abord, se placer dans la branche cible (possible avec l'interface graphique) :

```
git checkout master
```

Ensuite, fusionner :

```
git merge new_branch
```

Dans la majorité des situations, la fusion sera automatique ("Fast Forward"). Il est possible que des conflits apparaissent :

utiliser la commande `git status` pour afficher la liste des fichiers concernés, les ouvrir, régler le confit et effectuer un *commit*.

La branche fusionnée n'est pas supprimée : elles peut être utilisée à nouveau pour d'autres développements ou supprimée manuellement avec la commande suivante :

```
git branch -d new_branch
```

Fusionner avec une requête de tirage

L'autre façon de fusionner est plus formelle mais aussi plus générale : elle permet de fusionner une branche dans un dépôt d'un autre utilisateur pour y contribuer, ou de faire valider sa branche par un autre membre de l'équipe dans un projet collaboratif.

Pour contribuer au projet d'un autre utilisateur de GitHub[7], il faut commencer par en créer un *fork*, c'est-à-dire une copie sous la forme d'un dépôt lié à l'original. Il sera possible de tirer les modifications de l'original pour rester à jour[8] (par opposition à une simple copie instantanée possible en téléchargeant un Zip du projet) et, à la fin du développement, de fusionner le *fork* au dépôt original (par opposition à un clone qui ne permettrait pas de contribuer par la suite).

[7] https://git-scm.com/book/fr/v2/GitHub-Contribution-%C3%A0-un-projet

[8] https://ardalis.com/syncing-a-fork-of-a-github-repository-with-upstream/

Ensuite, il faut créer une branche de développement comme précédemment, la modifier et finalement demander au propriétaire du dépôt de la fusionner. Ce processus est décrit en détail dans la documentation de git .

Dans le cadre plus simple d'une branche de son propre projet comme dans le cas d'un *fork*, la branche de développement est prête à être fusionnée. Elle doit avoir être poussée sur GitHub. Sur la page GitHub du projet, un bouton *Create Pull Request* permet de demander la fusion. Un message décrivant les modifications proposées avec leur argumentaire doit être ajouté.

Le propriétaire du projet (les membres de l'équipe dans le cadre d'un projet collaboratif, ou soi-même si l'équipe se réduit à une personne) est averti de la requête de tirage. Sur la page du projet original, il est possible de voir le message, la liste des modifications (chronologie des *commits* ou comparaison des fichiers), d'engager un discussion avec l'auteur de la requête… Si la requête n'est pas retenue, elle peut être fermée. Si elle est validée, le bouton *Merge Pull Request* permet de fusionner la branche de développement avec la branche "master" (ou une autre) du projet source.

Les requêtes de tirage sont le seul moyen de contribuer à un dépôt sur lequel on ne dispose pas de droits d'écriture. C'est aussi le moyen de fusionner une branche de développement dans sont propre projet en en gardant une trace explicite (dans la rubrique *Pull requests* de la page GitHub du projet). Dans le cadre

d'un projet collaboratif, les propositions d'un membre (auteur de la requête) peuvent être validées par un autre (qui accepte la fusion).

3.5 Usage avancé

Commandes de git

Au-delà de l'usage courant permis par l'interface graphique de RStudio, des manipulations avancées des projets sont permises en utilisant git en ligne de commande. Quelques exemples utiles sont présentés ici.

Un petit guide des commandes est proposé par Roger Dudler[9]. Il résume les commandes essentielles, donc intégrées à l'interface graphique de RStudio. Des liens vers des références plus complètes sont donnés en bas de la page.

[9] https://rogerdudler.github.io/git-guide/index.fr.html

Taille d'un dépôt

Pour connaître l'espace disque occupé par un dépôt, utiliser la commande `git count-objects -vH`[10].

[10] https://git-scm.com/docs/git-count-objects

Les données pour ce document au stade de la rédaction sont présentées à titre d'exemple.

```
$ git count-objects -v
count: 200
size: 2.66 MiB
in-pack: 0
packs: 0
size-pack: 0
prune-packable: 0
garbage: 0
size-garbage: 0
```

La taille totale est sur la ligne *size*. Les packs sont une méthode utilisée par git pour réduire la taille du dépôt : des fichiers similaires sont stockés sous la forme d'une partie commune et de différences. La ligne *prune-packable* donne la taille d'objets stockés à la fois sous forme individuelle et dans des packs. Si leur taille est importante, exécuter `git prune-packed` pour la ramener à zéro.

La ligne *size-garbage* donne la taille des objets qui peuvent être supprimés. `git gc` les supprime, mais pas seulement : il optimise le stockage.

```
$ git gc
Enumerating objects: 194, done.
Counting objects: 100% (194/194), done.
```

```
Delta compression using up to 8 threads
Compressing objects: 100% (188/188), done.
Writing objects: 100% (194/194), done.
Total 194 (delta 83), reused 0 (delta 0)

$ git count-objects -vH
count: 1
size: 5.72 KiB
in-pack: 194
packs: 1
size-pack: 4.00 MiB
prune-packable: 0
garbage: 0
size-garbage: 0 bytes
```

Ici, la majorité des objets du dépôt a été placée dans un pack (mais sa taille est supérieure à celle des objets individuels).

Il est généralement inutile d'effectuer la collecte des déchets manuellement : git gère bien l'organisation de ses dépôts.

GitHub limite la taille des dépôts. En mai 2020, la limite est de 100 Go. La taille de tous les dépôts d'un utilisateur authentifié peut être affichée dans les réglages de son compte ("Personnal Settings", "Repositories")[11].

[11] https://github.com/settings/repositories

Supprimer un dossier

Toutes les modifications apportées à un dépôt sont stockées dans son historique. Il peut être utile d'en supprimer dans quelques cas particuliers :

- si un fichier contenant des informations confidentielles a été validé par mégarde. La validation de sa suppression ne le retire pas de l'historique, et les informations confidentielles restent visibles en consultant l'historique.
- si des fichiers volumineux ne sont plus nécessaires, par exemple des fichiers PDF produits par R Markdown (chapitre 4), binaires (donc inadaptés à git) et reproductibles à partir du code.

Typiquement, le dossier `docs` est utilisé pour stocker les documents produits à partir de code R Markdown. Les fichiers HTML et PDF doivent s'y trouver pour constituer les pages GitHub du projet. Chaque modification du dépôt génère une nouvelle version de ces fichiers dont le volume de l'historique devient rapidement considérable. Une solution efficace consiste à déléguer la création de ces fichiers à un système d'intégration continue (chapitre 6) et à retirer le dossier `docs` de la branche principale (*master*) du dépôt. Il faut alors supprimer tout son

historique pour récupérer la place qu'il occupe, qui peut être l'essentiel de la taille du dépôt.

Les commandes de suppression complète d'un dossier d'un dépôt son présentées ici[12]. Le dépôt doit être propre, c'est-à-dire sans modifications non validées, et les versions distantes et locales synchronisées.

[12] https://stackoverflow.com/questions/10067848/remove-folder-and-its-contents-from-git-githubs-history

Les trois commandes suivantes suppriment complètement le dossier `docs` de l'historique du dépôt git :

```
git filter-branch --tree-filter "rm -rf docs" |>
    --prune-empty HEAD
git for-each-ref --format="%(refname)" refs/original/ |>
    | xargs -n 1 git update-ref -d
```

Le dossier n'est pas supprimé du répertoire de travail. Il doit donc être ajouté au fichier `.gitignore` pour ne plus être suivi. La modification de `.gitignore` doit être validée. Ces opérations peuvent être réalisées avec l'interface de RStudio ou en ligne de commande :

```
echo docs/ >> .gitignore
git add .gitignore
git commit -m 'Removing docs folder from git history'
```

Le nettoyage du dépôt est nécessaire pour supprimer physiquement les données retirées :

```
git gc
```

Enfin, le dépôt doit être poussé. L'option `--force` implique le remplacement du contenu du dépôt distant par celui du dépôt local : toutes les modifications faites par des collaborateurs sont effacées, c'est pourquoi cette opération de nettoyage implique l'arrêt complet du projet pendant qu'elle a lieu.

```
git push origin master --force
```

Ce code peut être utilisé pour supprimer totalement n'importe quel fichier ou dossier d'un dépôt en remplaçant simplement `docs` dans la commande `git filter-branch` initiale. La réduction de la taille du dépôt peut être suivie en utilisant `git count-objects -vH` avant l'opération, avant `git gc` (la taille du dépôt reste stable mais a été déplacée vers *garbage*) et à la fin (la taille du dépôt est sensiblement réduite).

3.6 Pages GitHub

Tout projet sur GitHub doit avoir contenir un fichier `README.md` pour le présenter. Ce fichier est écrit au format Markdown.

Le fichier peut être placé dans le dossier `docs` pour fournir à fois la page d'accueil du dépôt et de son site web ou bien dupliqué par le script `GitHubPages.R` dans les projets de documents. Un dépôt contenant un mémo écrit en R Markdown (voir section 4.3) est utilisé comme exemple[13].

[13] https://github.com/EricMarcon/Krigeage

Son fichier `README.md` existe aux deux emplacements : il est écrit par le développeur à la racine du projet et dupliqué par `GitHubPages.R`.

Activation

Pour activer les pages GitHub, il faut ouvrir les propriétés du dépôt (*Settings*) et modifier la rubrique "GitHub Pages" (dans "Options"). Sélectionner la branche du projet et le dossier contenant les pages web, ici : `master` et `/docs`. En option, le choix d'un thème personnalise l'apparence des pages.

Le site web est accessible à une adresse[14] du domaine *github.io*.

[14] https://EricMarcon.github.io/Krigeage/

Le fichier `README.md` affiché en page d'accueil a un aspect très différent mais le même contenu que celui affiché avec le code sur la page du dépôt dans GitHub.

L'intérêt des pages GitHub est de permettre un accès simple aux documents formatés quand le dépôt contient une production écrite et ou à la documentation des packages R. Ces contenus seront présentés dans le chapitre suivant.

Un site web principal est proposé avec chaque compte GitHub, à l'adresse https://GitHubID.github.io[15]. Il sera utilisé pour héberger un site web personnel produit par **blogdown**.

[15] Exemple : https://EricMarcon.github.io/Krigeage/

Badges

Les badges sont de petites images, éventuellement mises à jour dynamiquement, qui renseignent rapidement sur le statut d'un projet. Ils doivent être placés immédiatement après le titre du fichier `README.md`.

Une bonne pratique consiste à indiquer l'avancement dans le cycle de vie du projet. Les badges correspondants sont listés sur le site du Tidyverse [16].

[16] https://www.tidyverse.org/lifecycle/

Leur code MarkDown est le suivant :

```
![stability-wip]
(https://img.shields.io/badge/lifecycle-maturing-blue.svg)
```

L'adresse du fichier image correspond à la position dans le cycle de vie : il suffit de copier l'adresse de l'image du badge désiré sur le site du Tidyverse pour l'obtenir.

Chapitre 4

Rédiger

R et RStudio permettent de rédiger efficacement des documents de tous formats, du simple bloc-note à la thèse, en passant par des diaporamas. Les outils pour le faire sont l'objet de ce chapitre, complété par la production de sites web (y compris un site personnel).

4.1 Bloc-note Markdown (R Notebook)

Dans un fichier `.R`, le code doit toujours être commenté pour faciliter sa lecture. Quand l'explication du code nécessite plusieurs lignes de commentaire par ligne ou bloc de code, il est temps d'inverser la logique et de placer le code dans un texte.

L'outil le plus simple est le bloc-note Markdown (Menu "File > New File > R Notebook"). Le modèle de document contient son mode d'emploi.

Le langage qui permet de formater le texte est Markdown[1], un langage de balisage simple à utiliser :

- Les paragraphes sont séparés par des sauts de ligne ;
- Le document est structuré par des titres : leur ligne commence par un nombre de `#` correspondant à leur niveau ;
- Les formats de caractères sont limités à l'essentiel : italique ou gras (texte entouré par une ou deux `*`) ;
- D'autres codes simples permettent tous les formatages utiles.

Ce langage est le pivot du logiciel pandoc[2], dédié à la conversion de documents de formats différents.

Le package **rmarkdown**[3] fait le lien entre R et Markdown, en s'appuyant sur l'interface de RStudio qui n'est pas indispensable mais simplifie énormément son utilisation. Le dialecte de Markdown utilisé par le package est appelé *R Markdown*. Sa syntaxe est résumée dans une antisèche[4]. Sa documentation

[1] https://fr.wikipedia.org/wiki/Markdown

[2] https://fr.wikipedia.org/wiki/Pandoc

[3] Y. Xie (2015). *Dynamic Documents with R and knitr*. 2nd. Boca Raton, Florida : Chapman et Hall/CRC.

[4] https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf

complète est en ligne.[5]

Les équations sont écrites au format LaTeX[6].

L'organisation la plus simple d'un document *R Markdown* est visible dans le modèle de bloc-note. Il commence par un en-tête au format YAML[7] :

```
---
title: "R Notebook"
output: html_notebook
---
```

La première entrée est le titre, la seconde le format de sortie : plus précisément le nom de la fonction chargée de traiter le document.

Le document contient du texte formaté en Markkdown et des bouts de code (*code chunks*) entourés par trois accents graves (la syntaxe markdown d'un bloc de code) et une description du langage, ici `r`. Ces bouts de code sont traités par **knitr** qui transforme le résultat de l'exécution du code R en Markdown et l'intègre au texte du document.

Traiter un document R Markdown s'appelle le *tricoter* (*knit*). La chaîne de production est la suivante :

- **knitr** traite les bouts de code : calculs, production de figures ;
- **rmarkdown** intègre la production des bouts de code et texte pour produire un fichier Markdown standard ;
- pandoc (installé avec RStudio) convertit ce fichier au format HTML, LaTeX ou Word ;
- LaTeX produit un fichier PDF quand ce format est demandé.

RStudio permet de lancer le tricot par des boutons plutôt que par des commandes : dans la fenêtre source (celle du haut à gauche), un bouton *Knit* accompagne les documents R Markdown. Pour les bloc-notes R Markdown, il est remplacé par un bouton *Preview* avec les mêmes fonctions. Il peut être déroulé pour choisir le format de sortie : HTML, Word, PDF (en passant par LaTeX) et, pour les bloc-notes, une commande *Preview* qui affiche le document en HTML sans exécuter les bouts de code pour gagner du temps. Dès le premier tricot au format Word ou HTML, on remarquera que le bouton *Preview* disparaît.

Au final, l'utilisation de R Markdown combine plusieurs avantages :

- La simplicité de la rédaction : le texte brut est plus facile à lire et à formater qu'en LaTeX par exemple ;

[5] Y. Xie et al. (2018). *R Markdown : The Definitive Guide*. Boca Raton, Florida : Chapman et Hall/CRC.

[6] https://fr.wikibooks.org/wiki/LaTeX/%C3%89crire_des_math%C3%A9matiques

[7] https://fr.wikipedia.org/wiki/YAML

- L'automatisation de la production : le formatage et la mise en page sont entièrement automatiques ;
- La reproductibilité : chaque document peut être autosuffisant accompagné de ses données. Relancer le tricotage régénère entièrement le document, y compris les calculs nécessaires et la production des figures.

Elle a aussi quelques inconvénients :

- Le formatage dépend de modèles, et developper de nouveaux modèles n'est pas simple ;
- Les erreurs de tricot sont parfois difficiles à corriger, notamment quand elles interviennent à l'étape de la compilation LaTeX ;
- La reproductibilité consomme du temps de calcul. Pour limiter ce problème, un système de cache permet de ne pas réévaluer tous les bouts de code R à chaque modification du texte. La production de gros documents peut aussi être déléguée à un système d'intégration continue (chapitre 6).

4.2 Modèles R Markdown

Des modèles de document plus élaborés que le bloc-note sont fournis par des packages, dont **rmarkdown**. Ils sont accessibles par le menu "File > New File > R Markdown..." (figure 4.1)).

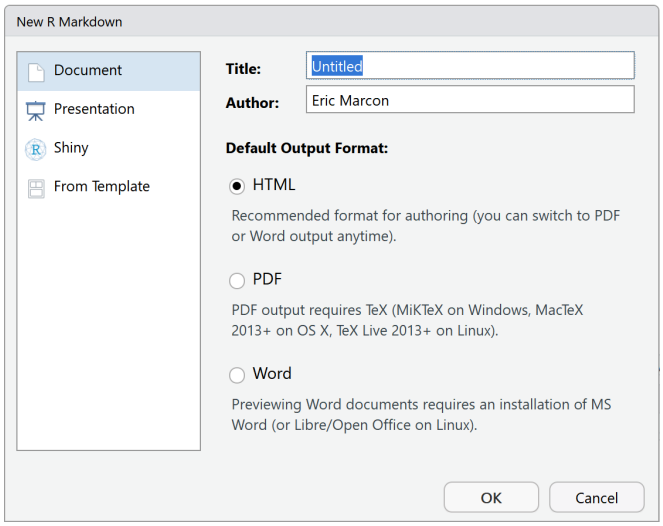


Fig. 4.1 : Nouveau document Markdown à partir d'un modèle.

Les modèles les plus simples sont *Document* et *Presentation*. Les informations à fournir sont le titre et le nom de l'auteur, et le format du document attendu (qui pourra être modifié plus tard). Ces modèles créent un seul fichier dont l'enregistrement ne sera obligatoire qu'au moment de tricoter.

La syntaxe est la même que celle du bloc-note. Dans l'entête, une entrée supplémentaire est utilisée pour la date, qui peut être calculée par R à chaque tricot :

```
date: "2020-11-24"
```

Le code R en ligne (par opposition aux bouts de code) peut être utilisé partout dans un document R Markdown. Il commence par un guillemet inversé suivi de `r`et se termine par un autre guillement inversé.

Les documents peuvent être tricotés au format HTML, PDF (via LaTeX) ou Word. L'entête du fichier R Markdown est réécrit quand le tricot est lancé par le bouton de RStudio qui place en premier le format de sortie utilisé et l'ajoute si nécessaire.

Les présentations peuvent être tricotées dans deux formats HTML, ioslide[8] ou Slidy[9], au format Beamer (PDF)[10] ou en Powerpoint[11].

Le niveau 2 de plan (`##`) marque le changement de diapositive.

Du code supplémentaire, présenté dans les documentations des formats HTML, permet d'utiliser des fonctionnalités spécifiques.

Ces modèles sont simples mais assez peu utiles : le bloc-note R est plus facile à utiliser que le modèle de document pour des documents très simples. Des modèles plus élaborés sont disponibles.

[8] https://bookdown.org/yihui/rmarkdown/ioslides-presentation.html

[9] https://bookdown.org/yihui/rmarkdown/slidy-presentation.html

[10] https://bookdown.org/yihui/rmarkdown/beamer-presentation.html

[11] https://bookdown.org/yihui/rmarkdown/powerpoint-presentation.html

4.3 Memo, article : bookdown

R Markdown ne permet pas de rédiger un article scientifique. La bibliographie ne pose pas de problème parce qu'elle est gérée par pandoc pour les documents HTML ou Word et sous-traitée à LaTeX pour les documents PDF. Les équations, figures et tableaux sont numérotés par LaTeX mais pas en HTML. Les références croisées (les renvois à un numéro de figure par exemple) ne sont pas supportés. Enfin, les légendes de figures ou tableaux ne supportent que du texte brut, sans aucun formatage.

bookdown comble ces manques. Le package a été conçu pour la rédaction d'ouvrages comportant plusieurs chapitres mais peut être utilisé pour des articles. Le package ne fournit pas directement de modèles.

Le package **EcoFoG** disponible sur GitHub fournit les modèles présentés ici. Il doit être installé :

```
remotes::install_github("EcoFoG/EcoFoG")
```

Modèle Memo EcoFoG

Le modèle Memo EcoFoG produit un document HTML simple avec une table des matières flottante (voir l'exemple[12]). Le format PDF est proche du modèle *article* de LaTeX (exemple[13]) .

Le modèle contient sa propre documentation.

[12] https://EricMarcon.github.io/Krigeage/Krigeage.html

[13] https://EricMarcon.github.io/Krigeage/Krigeage.pdf

Créer

Utiliser le menu "File > New File > R Markdown…" puis sélectionner "From template" (figure 4.1). La liste des modèles disponible et le package qui les propose est alors affichée.

Sélectionner le modèle Memo EcoFoG du package **EcoFoG**, choisir le nom du projet ("Name :", qui sera le nom du dossier dans lequel il sera créé, et son dossier parent ("Location :"). Dans l'organisation proposée en section 1.2, le dossier parent est `%LOCALAPPDATA%\ProjetsR`. Le nom du projet ne doit contenir aucun caractère spécial (accent, espace…) pour assurer sa portabilité sur tous les systèmes d'exploitation (Windows, Linux, MacOS).

Les modèles élaborés créent un dossier avec de nombreux fichiers (bibliographie, styles, modèle LaTex…), contrairement aux modèles simples qui créent seulement un fichier.

Quand un dossier est créé, par exemple par le modèle Memo EcoFoG, il faut en faire un projet RStudio : dans le menu des projets (en haut à droite de la fenêtre de RStudio), utiliser le menu "New Project…" puis "Existing Directory" et sélectionner le dossier qui vient d'être créé.

Ecrire

Les instructions pour utiliser le modèle sont contenues dans le texte fourni par défaut.

Tricoter

Le document peut être tricoté en plusieurs formats :

- *html_document2* est le format HTML pour lequel le modèle a été conçu : un bloc-note avec une table des matières flottante ;
- *gitbook* est un format HTML alternatif, utilisé normalement pour les ouvrages ;
- *pdf_book* produit un document PDF suivant le modèle LaTeX *article*, couramment utilisé directement en LaTeX ;
- *word_document2* crée un ficher Word.

Mettre en ligne

Le script `GithubPages.R` fourni avec le modèle place tous les documents tricotés (HTML et PDF) dans le dossier `docs`, avec une copie du fichier `README.md`. De cette façon, il est possible d'activer les pages GitHub du projet (sur le dossier `docs` de la branche `master`). Le fichier `README.md` sera la page d'accueil du site web produit.

Le script retourne de nombreux avertissements, sans conséquence, quand un fichier doit être écrasé par exemple.

Pour le projet présenté en exemple, le fichier `README.md` est le suivant :

```
# Krigeage avec R

Techniques pour interpoler les valeurs d'une variable
continue.

[Voir le support](https://EricMarcon.github.io/
Krigeage/Krigeage.html) ou le [télécharger]
(https://EricMarcon.github.io/Krigeage/Krigeage.pdf)
```

La première ligne contient le titre, la seconde une description. Les liens pointent vers le fichier HMTL et le fichier PDF produits, tous les deux dans le dossier `docs`. Les liens sont absolus, avec l'adresse complète des fichiers, et non relatifs, parce que le fichier `README.md` est le même à la racine du projet (affiché par GitHub en dessous du code) et dans le dossier `docs` (utilisé par les pages GitHuB comme page d'accueil du site).

En pratique, on tricote au format HTML pendant toute la phase de rédaction du mémo, parce que la production est très rapide. Quand le document est stabilisé, il faut le tricoter au format HTML et au format PDF. Enfin, l'exécution du script `GithubPages.R` (l'ouvrir et cliquer sur le bouton *Source*) place tous les fichiers produits dans `docs`. Il reste à pousser le dépôt sur GitHub et activer les pages GitHub.

La production des documents a lieu sur le poste de travail, elle n'est pas sous-traitée à un service en ligne (intégration continue, voir plus bas) contrairement à celle des ouvrages, qui nécessitent plus de ressources.

Autres modèles

Le modèle *Article* d'**EcoFoG** est destiné à la production d'articles PDF pour l'autoarchivage (typiquement, le dépôt sur HAL) bien formatés, au format A4 en double colonne[14].

[14] Exemple : https://EricMarcon.github.io/Rochebrune2018/Entropie.pdf

Le format HTML est le même que celui du modèle de mémo.

Le package **rticles** a pour ambition de fournir des modèles pour toutes les revues scientifiques qui acceptent une soumission d'articles en LaTeX. Il propose donc des modèles Markdown qui produisent des fichiers PDF conformes aux exigences des revues et la possibilité de récupérer le fichier `.tex` intermédiaire (pandoc produit un fichier `.tex` transmis au compilateur LaTeX). Le package ne permet pas de tricot HTML parce qu'il utilise la syntaxe LaTeX dans le document R Markdown au lieu d'utiliser **bookdown** pour gérer les références bibliographique et les références croisées. Il n'est pas possible d'échanger directement du contenu R Markdown standard avec des documents écrits pour **rticles**, ce qui limite beaucoup l'intérêt du package.

4.4 Présentation EcoFoG : bookdown

Le modèle Présentation EcoFoG permet de créer des présentations au format HTML et PDF (beamer) simultanément, comme le montre l'exemple[15].

[15] https://EricMarcon.github.io/Chao1/, choisir Lecture (HTML) ou Téléchargement (PDF).

La démarche est identique à celle du Memo EcoFoG. Les niveaux de titre permettent de séparer les parties de la présentation (`#`) et les diapositives (`##`). Deux formats sont disponibles en HTML : ioslides[16] et Slidy[17] . Quelques spécificités dans le code permettent d'affiner la présentation des diapositives, pour un affichage sur deux colonnes par exemple : elles sont documentées dans le modèle .

[16] https://bookdown.org/yihui/rmarkdown/ioslides-presentation.html

[17] https://bookdown.org/yihui/rmarkdown/slidy-presentation.html

4.5 Ouvrage EcoFoG : bookdown

Le modèle d'ouvrage est destiné aux documents longs, qui présentent une différence importante avec les documents précédents : un document long est composé de plusieurs chapitres, chacun placé dans son fichier `.Rmd`.

Le format HTML est gitbook[18], le standard de la lecture en ligne de documents de ce type. Le format PDF est dérivé du modèle LaTeX *memoir*[19], optimisé aussi pour les documents longs .

[18] https://www.gitbook.com/

[19] https://www.ctan.org/pkg/memoir

Ce document a été écrit avec ce modèle.

Créer

La création d'un projet d'ouvrage est identique à celle présentée plus haut : le modèle est : Ouvrage EcoFoG. Le dossier créé doit être transformé en chapitre, comme pour un Memo EcoFoG.

Chaque chapitre de l'ouvrage est un fichier Rmd, dont le nom commence normalement par son numéro (ex. : `01-intro.Rmd`). Tous les fichiers Rmd présents dans le dossier du projet sont en

réalité traités comme des chapitres, triés par ordre de nom de fichier, dont ceux fournis par le modèle (démarrage et syntaxe) qui doivent être supprimés à l'exception de `99-references.Rmd` qui contient la bilibiographie, placée à la fin. Le fichier `index.Rmd` est particulier : il contient l'entête du document et le premier chapitre.

Ecrire

Ce premier chapitre est placé dans l'avant-propos de l'ouvrage imprimé : il ne doit pas être numéroté (d'où le code `{-}` à côté du titre) dans la version HTML. Il se termine obligatoirement par la commande LaTeX `\mainmatter` qui marque le début du corps de l'ouvrage.

Les niveaux de plan commencent sont `#` pour les chapitres (un seul par fichier), `##` pour les sections, etc.

Tricoter

La compilation au format PDF est faite par XeLaTeX, qui doit être installé.

Pendant la rédaction, il est fortement conseillé de ne créer que le fichier HTML, ce qui est beaucoup plus rapide qu'une compilation LaTeX. Chaque chapitre peut être visualisé très rapidement en cliquant sur le bouton *Knit* au-dessus de la fenêtre de source. Le livre entier est créé en cliquant sur le bouton *Build Book* de la fenêtre *Build* de RStudio. La liste déroulante du bouton permet de créer tous les documents ou de se limiter à un format.

Les fichiers produits sont placés directement dans le dossier `docs`, qui sera utilisé par les pages GitHub pour permettre la lecture en ligne et le téléchargement du PDF. La page d'accueil du site web est créée par bookdown à partir du fichier `index.Rmd` : le fichier `README.md` n'est pas dupliqué dans `docs` le script `GithubPages.R` ne doit pas être utilisé (il n'est pas fourni dans le modèle).

Finitions

La mise en page est assurée de façon totalement automatique par pandoc (en HTML) et LaTeX (en PDF).

Il est souvent utile d'aider LaTeX à résoudre quelques dépassements de marge dus à de trop grandes contraintes de mise en page : pour la lisibilité optimale, les colonnes sont étroites, mais le code (texte formaté entre deux apostrophes inversées) n'autorise pas la césure.

Si une ligne de texte dépasse dans la marge de droite dans le document PDF, la solution consiste à ajouter manuellement

le code `\break` à l'emplacement désiré pour le retour à la ligne dans le document R Markdown. La commande n'a aucun effet sur le document HTML mais force la césure en LaTeX. Pour couper du texte formaté (entre astérisques pour l'italique ou plus fréquemment entre appostrophes inversées pour du code), il faut terminer le formatage avant `\break` et le recommencer après. Exemple, pour forcer le retour à la ligne avant `fichier.Rmd` :

```
Le fichier `/chemin/`\break`fichier.Rmd`
```

Les bouts de code R sont formatés automatiquement par **knitr** quand l'option `tidy=TRUE` leur est appliquée. Le comportement par défaut est indiqué dans les options de **knitr**, dans un bout de code au début de chaque document :

```r
# knitr options
knitr::opts_chunk$set(
  cache=TRUE, warning=FALSE, echo = TRUE,
  fig.env='SCfigure', fig.asp=.75,
  fig.align='center', out.width='80%',
  tidy=TRUE,
  tidy.opts=list(blank=FALSE, width.cutoff=55),
    size="scriptsize",
  knitr.graphics.auto_pdf = TRUE)
```

La largeur maximale d'une ligne de code formaté est ici de 55 caractères, optimal pour le modèle "Ouvrage EcoFoG". Il arrive que le formatage automatique ne fonctionne pas parce que **knitr** ne parvient pas à trouver une coupure de ligne respectant toutes les contraintes, ce qui provoque un dépassement de marge dans le code. Dans ce cas, formater manuellement le bout de code en lui ajoutant l'option `tidy=FALSE`.

Les blocs de code littéral, délimités par trois apostrophes inversées, doivent être formatés manuellement, en évitant toute ligne de plus de 55 caractères.

Intégration continue

La construction d'un ouvrage prend du temps, surtout s'il contient des calculs. Elle doit être lancée au format gitbook et au format PDF. En production, elle peut être confiée à Travis-CI (chapitre 6.2).

Google Analytics

Le suivi de l'audience de l'ouvrage peut être confié à Google Analytics. Pour cela, il faut créer un compte et ajouter une *propriété* Google Analytics, c'est-à-dire un site web, puis un flux de données, ici un flux web[20].

[20] https://support.google.com/analytics/answer/9304153?hl=fr&ref_topic=9303319

Google Analytics fournit un script de configuration nommé `gtag.js` à placer à la racine du dossier du projet. Enfin, déclarer

le script dans l'entête des pages web en ajoutant une instruction dans `_output.yml`, dans sa première section.

```
bookdown::gitbook:
  includes:
    in_header: gtag.js
```

4.6 Site web R Markdown

Un site web constitué de pages écrites avec R Markdown (sans les fonctionnalités de **bookdown**) et un menu peut être créé très simplement, avec un résultat de bonne facture[21].

[21] https://rstudio.github.io/learnr/ par exemple.

Modèle

Dans RStudio, dans le menu des projets en haut à droite, cliquer sur *New Project…* puis "New Directory" puis "Simple R Markdown website". Saisir le nom du projet, sélectionner le dossier dans lequel le projet sera créé en cliquant sur "Browse" et enfin cliquer sur "Create Project".

Le site par défaut contient deux pages : `index`, la page d'accueil, et `about`, la page "A propos". Le fichier `_site.yml` contient le nom du site et le contenu de sa barre de navigation : un titre et le fichier correspondant. D'autres pages seront ajoutées en créant de nouveaux fichiers `.Rmd` et en les ajoutant au fichier `_site.yml`.

Améliorations

Le modèle de site peut facilement être amélioré en complétant `_site.yml` :

- en ajoutant une icône GitHub dans la barre de navigation pour renvoyer vers le code source du site :
- en choisissant la méthode de tricot des pages, pour utiliser **bookdown** au lieu de **rmarkdown** ;
- en plaçant les fichiers du site dans le dossier `docs` et ainsi séparer le code et la production.

Le fichier `_site.yml` complété est le suivant :

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "About"
      href: about.html
```

```
  right:
    - icon: fa-github
      href: https://github.com/rstudio/rmarkdown
output_dir: "docs"
output:
  bookdown::html_document2:
    theme: sandstone
    highlight: tango
    toc: true
    toc_float: yes
```

L'icône de GitHub fait partie de la collection Font Awesome dont toutes les icônes gratuites[22] sont utilisables avec la même syntaxe : "fa-nom".

[22] https://fontawesome.com/icons?d=gallery&m=free

Le lien correspondant à l'icône doit être celui du dépôt GitHub du site web.

La syntaxe de la section `output` est la même que celle des documents vus plus haut, comme le mémo EcoFoG. Elle s'applique à toutes les pages (dont l'entête YAML est réduite au minimum). Les thèmes disponibles sont ceux de rmarkdown[23].

[23] https://bookdown.org/yihui/rmarkdown/html-document.html#appearance-and-style

L'option `highlight` indique la façon dont le code R éventuellement affiché sera formaté. Enfin, la table des matières est flottante, ce qui signifie que sa position s'ajuste quand la fenêtre défile.

Contôle de source

Le projet doit être placé sous contrôle de source et poussé sur GitHub (chapitre 3). Le fichier `.gitignore` est le suivant :

```
# R
.Rbuildignore
.RData
.Rhistory
.Rprofile
.Rproj.user

# Web Site
/_site/
/*_cache/
/*_files/
```

Activer les pages GitHub (section 3.6) sur le dossier `docs` pour héberger le site. Ajouter un fichier vide nommé `.nojekyll` dans `docs` pour que les pages GitHub ne tentent pas de reformater le site. On peut uliser le terminal de RStudio pour exécuter :

```
touch docs/.nojekyll
```

4.7 Site web personnel : blogdown

Pour créer une page web personnelle, *Hugo* est un générateur de site statique capable de produire des pages HTML à partir de code Markdown. Les sites statiques ont l'avantage, en comparaison aux sites dynamiques gérés par un système de gestion de contenu (CMS, par exemple : Wordpress, Joomla, SPIP), d'être portables sur n'importe quel serveur web sans support de base de données ni de code à exécuter côté le serveur (tel que PHP) et d'être très rapides puisque les pages sont créées une seule fois et non à chaque consultation. Un site Hugo peut être hébergé par exemple sur la page personnelle de tout utilisateur de GitHub dont l'adresse est de la forme "NomUtilisateur.github.io".

Hugo propose de nombreux thèmes, qui sont des modèles de structure de sites, donc le thème **Academic**, destiné aux chercheurs. Dans RStudio, le package **blogdown** est prévu pour produire facilement des pages web avec Hugo. Ces pages peuvent contenir du code R : elles sont très proches d'un Memo EcoFoG dont le contenu peut être facilement copié et collé. Nous utiliserons donc cette solution, pour un site comme celui proposé en exemple[24].

[24] https://EricMarcon.github.io/

La structure du site web est simple :

- une page d'accueil, contenant divers composants paramétrables comme la biographie de l'auteur, une sélection de publications, de billets de blogs ou d'autres éléments et un formulaire de contact ;
- des pages détaillant les divers éléments (publications, billets, etc.) écrites en R Markdown.

Installation des outils

La première étape consiste à installer le package **blogdown** dans R.

```
install.packages("blogdown")
```

blogdown est capable d'installer Hugo sous Windows, macOS ou Linux.

```
blogdown::install_hugo()
```

La documentation complète de **blogdown** est disponible[25].

[25] https://bookdown.org/yihui/blogdown/

Les versions récentes de Hugo utilisent *Go* (le langage de programmation) pour installer leurs modules à la volée : ici le thème Academic est chargé depuis GitHub au moment de la création du site. Go doit donc être installé[26].

[26] https://golang.org/doc/install

Créer

La façon la plus simple consiste à créer un dépôt sur GitHub à partir du modèle. Sur la page du dépôt *starter-academic*[27], cliquer sur le bouton *Use this template*, s'authentifier éventuellement sur GitHub, puis saisir le nom du dépôt qui contiendra le projet, par exemple "MySite". Créer le dépôt. Copier l'adresse du dépôt en cliquant sur le bouton *Code* puis sur le bouton à droite de l'adresse (figure 4.2).

[27] https://github.com/wowchemy/starter-academic

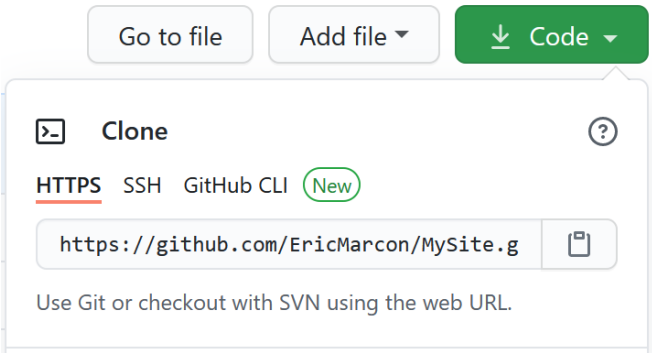


FIG. 4.2 : Copie de l'adresse d'un dépôt à cloner sur GitHub.

Dans RStudio, créer un nouveau projet à partir de GitHub : dans le menu des projets en haut à droite, cliquer sur *New Project…* puis *Version Control* puis *Git* puis coller l'adresse dans le champ "Repository URL" (figure 4.3). Sélectionner le dossier dans lequel le projet sera créé en cliquant sur "Browse" et enfin cliquer sur "Create Project".

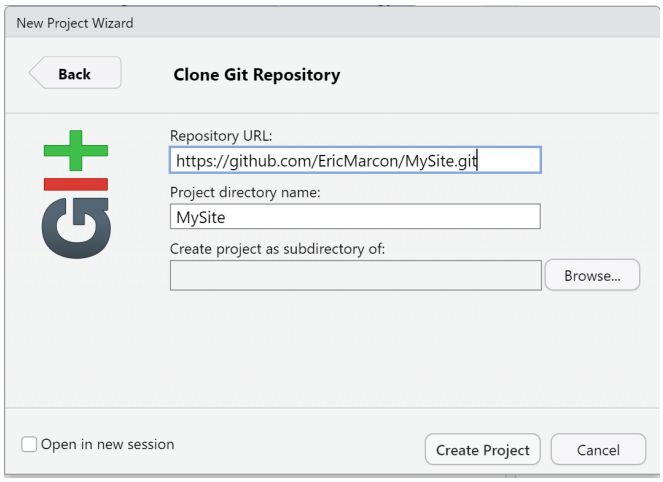


FIG. 4.3 : (ref :rediger-rediger-Projet-GitHub)

Le projet créé est une copie exacte du modèle, qui doit être personnalisée.

RStudio ajoute automatiquement à la fin du fichier `.gitignore` une ligne pour ignorer ses fichiers de travail (dossier `.Rproj.user`). Ajouter une ligne de commentaire pour le signaler. Le contenu de `.gitignore` doit être le suivant :

```
# R
.Rbuildignore
.RData
.Rhistory
.Rprofile
.Rproj.user

# Hugo
/resources/
/public/

# blogdown
/static/en/
/static/fr/
*.rmarkdown
_index.html
index.html
```

Un bug de **blogdown** nécessite de déplacer le fichier `config/_default/config.toml` à la racine du projet.

Prendre en compte ces modification dans git en faisant un commit.

Exécuter

```
blogdown:::serve_site()
```

pour construire le site web. Il apparaît dans la fenêtre *Viewer* de RStudio, dont le bouton d'agrandissement permet l'affichage dans le navigateur internet par défaut du système.

Pour modifier le contenu du site, il est préférable d'arrêter le serveur web par la commande :

```
blogdown:::stop_server()
```

Si le site est multilingue (Français et Anglais par exemple), son contenu (dossier `content`) doit être copié dans un dossier correspondant à chaque langue. Par exemple, le fichier `content/authors/admin/_index.md` qui contient les informations sur le propriétaire du site est remplacé par `content/en/authors/admin/_index.md` et `content/fr/authors/admin/_index.md` si le site supporte l'Anglais et le Français. En pratique, créer un dossier `en` et un dossier `fr` dans `content`. Copier tout le contenu de content (sauf les deux nouveaux dossiers) dans `en` puis déplacer ce même contenu dans `fr`.

Paramétrer

Les fichiers de configuration du site sont bien documentés et offrent de nombreuses options. Les principales sont passées en revue ici pour une création rapide d'un site fonctionnel.

Le fichier `config.toml` contient les paramètres généraux du site. Les lignes à mettre à jour sont celle du titre du site (le nom du propriétaire puisqu'il s'agit d'un site personnel) et son adresse publique. Pour le site exemple :

```
title = "Eric Marcon"
baseurl = "https://EricMarcon.github.io/"
```

Il contient aussi la ligne de sélection de la langue par défaut ("en" ou "fr" au choix) et celle qui permet de placer les fichiers produits par Hugo dans chaque dossier de langue ("true" obligatoirement pour un site multilingue) :

```
defaultContentLanguage = "fr"
defaultContentLanguageInSubdir = true
```

Le dossier `config/_default/` contient les autres fichiers de configuration.

`languages.toml` contient les paramètres linguistiques et les traductions de menus. Pour chaque langue, la version utilisée et le dossier de contenu sont précisés :

```
[en]
  languageCode = "en-us"
  contentDir = "content/en"
[fr]
  languageCode = "fr-fr"
  contentDir = "content/fr"
```

Pour les langues additionnelles, le titre du site, les paramètres d'affichage des dates et la traduction des menus sont ajoutés. Dans la section `[fr]` :

```
[en]
  languageCode = "en-us"
  contentDir = "content/en"
[fr]
  languageCode = "fr-fr"
  contentDir = "content/fr"
```

Ces lignes sont commentées dans le modèle et doivent dont être décommentées en retirant les `#` en têtes de lignes.

Les menus sont décrits plus bas.

`params.toml` décrit l'aspect du site. Les options sont regroupées par sujet, par exemple "Theme" pour l'apparence générale. Dans "Basic Info", la ligne

```
site_type = "Person"
```

sélectionne un site personnel. Il est possible d'utiliser Academic pour un site de projet scientifique ou un site d'unité, non documentés en détail ici. Les principales différences sont, pour un site collectif :

- la gestion des auteurs : dans le dossier `/contents/<langue>/authors`, un seul dossier `admin` est utilisé pour un site personnel alors qu'un dossier par personne est nécessaire pour un site collectif ;
- un composant décrit plus bas, qui permet de présenter les personnes, doit être activé.

La description du site dans la langue par défaut est saisie, à destination des moteurs de recherche :

```
description = "Eric Marcon's Homepage"
```

Elle doit être traduite dans le fichier `languages.toml`, dans chaque langue.

Dans "Site Features", nous sélectionnons la coloration du code R, l'activation du formatage des équations et l'avertissement légal pour l'utilisation des cookies.

```
highlight_languages = ["r"]
math = true
privacy_pack = true
```

La ligne `edit_page` doit être mise à jour : remplacer le dépôt par défaut "https://github.com/gcushen/hugo-academic" par celui du site.

"Contact details" contient les informations pour contacter le propriétaire du site. Elles doivent être saisies.

"Regional Settings" contient les paramètres d'affichage de date pour la langue par défaut (ceux des autres langues sont dans `languages.toml`). Ils n'ont normalement pas à être modifiés.

"Comments" permet d'activer les commentaires des visiteurs en bas de pages, avec Disqus ou Comment.io (un compte est nécessaire chez le fournisseur). "Marketing" permet d'activer le suivi de fréquentation du site en saisissant simplement son identifiant Google Analytics (à créer avec un compte Google). "Content Management System" contient la ligne `netlify_cms`

dont la valeur doit être `false` si le site n'est pas hébergé par Netlify. Enfin "Icon Pack Extensions" permet d'activer les icônes Academicons si nécessaire.

Ecrire

Utiliser la documentation en ligne[28] en complément des informations principales détaillées ici. L'exemple utilisé ici est le site personnel de l'auteur[29] .

[28] https://wowchemy.com/docs/page-builder/

[29] https://EricMarcon.github.io/

La méthode de travail consiste à progresser pas à pas en testant puis validant chaque étape :

- Effectuer les modifications ;
- Construire le site et vérifier le résultat : `blogdown:::serve_site()` ;
- Arrêter le site : `blogdown::stop_server()` ;
- Si le résultat n'est pas satisfaisant, recommencer ;
- Valider les modifications (*commit*).

Page d'accueil

La page d'accueil du site est constituée par une suite d'éléments (*widgets*) qui se trouvent dans `/contents/<langue>/home`. Chaque élément est décrit par un fichier markdown. Le premier est `index.md`. Il n'est normalement jamais modifié. Son contenu est le suivant :

```
+++
# Homepage
type = "widget_page"
headless = true  # Homepage is headless, other widget
pages are not.
+++
```

Le fichier ne contient qu'un entête au format TOML, encadré par une ligne de `+++`. Le type de composant (`type`) indique qu'il s'agit d'une page de composants, dans laquelle les autres composants du dossier trouveront leur place. `headless = true` signifie que la page n'a pas d'en-tête.

Le composant `demo.md` (figure 4.4) est un composant de type "blank", c'est-à-dire une page de texte libre : il sert ici à présenter le modèle Academic Kickstart et doit donc être désactivé. L'entête contient ses informations de formatage (titre, nombre de colonnes, couleurs…) et le contenu de la page est écrit en markdown. Les composants apparaissent par ordre croissant de poids (`weight` dans l'entête) : 15 marque le premier composant dans le modèle Academic. Le composant peut être désactivé en supprimant son fichier ou en modifiant sa propriété `active` dans l'entête :

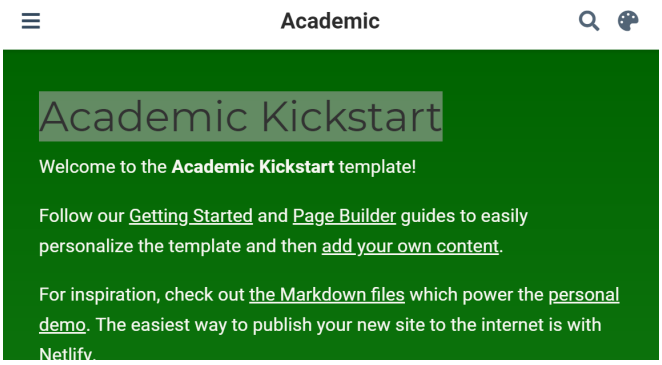


FIG. 4.4 : Composant `demo` dans Academic.

```
active = false  # Activate this widget? true/false
```

FIG. 4.5 : Composant `about` dans Academic.

Le composant suivant est `about.md` (figure 4.5). Il présente le propriétaire du site. Son titre doit être localisé. Dans le dossier `/content/fr/home`, sa valeur sera :

```
title = "Biographie"
```

L'auteur (`author`) doir correspondre à un dossier de `/contents/<langue>/authors`. `admin` convient parfaitement pour un site personnel. Academic permet de créer des sites d'équipes : dans cette configuration, un dossier par personne serait nécessaire. L'image affichée par le composant est le fichier `avatar.jpg` placé dans ce dossier. Limiter la taille du fichier pour la performance du site (moins d'un mégaoctet est une taille raisonnable), tout en assurant une taille minimale de quelques centaines de pixels de côté pour la qualité de l'affichage.

Le contenu du composant est lu dans le fichier `_index.md` du même dossier, qui contient toutes les informations sur l'auteur. Son organisation est assez claire : modifier son contenu à partir de l'exemple fourni. Si des icônes de type `ai` sont utilisées, activer le pack d'icône Academicons dans `config/_default/params.toml`.

Le composant talents (`skills`, figure 4.6) présente les compétences de l'auteur de façon graphique. Une collection

FIG. 4.6 : Composant `skills` dans Academic.

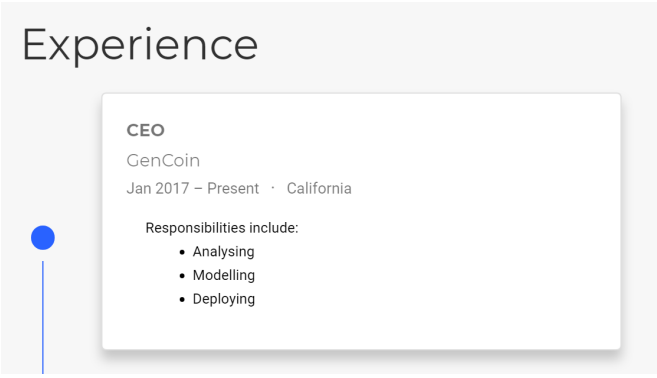


FIG. 4.7 : Composant `experience` dans Academic.

d'icônes est disponible, et des icônes nouvelles peuvent être ajoutées.

Le composant expérience (`experience`, figure 4.7) liste les expériences professionnelles. Toutes les informations sont saisies dans son entête.

Le composant `accomplishments` présente les formations professionnelles et permet d'accéder à leurs certificats.

Le composant `posts` va chercher son contenu dans le dossier `/contents/<langue>/post` qui contient les billets de blog (voir plus bas). Le fichier `posts.md` contient des options de mise en page dans son entête.

Le composant `projects` fonctionne de la même façon. La différence entre les deux composants est leur mise en forme : `posts` est du type "pages", qui affiche les éléments les plus récents, alors que `projects` est de type "portfolio", qui affiche les éléments sélectionnés qui contiennent la description `featured: true` dans leur propre entête. Il est possible de créer des composants de ces types librement, en spécifiant le dossier contenant les éléments dans "page-type". Exemple : créer un composant nommé `software.md` en renommant `projects.md`, modifier sa ligne `page_type = "software"` et créer un dossier `/contents/<langue>/software` pour y placer du contenu.

Les composants `publications` et `featured` sont de type "pages" et "portfolio" respectivement et prennent leur contenu dans le dossier `publication`.

Le composant `tags` présente un nuage de mots à partir des mots-clés déclarés dans tous les fichiers de contenu (billets de

blog, publications…) sous la forme suivante dans leur entête :

```
tags = ["Mot Clé 1", "Autre Mot Clé"]
```

Enfin, le composant `contact` permet d'afficher un formulaire de contact. Il utilise les informations du fichier `config/_default/params.toml` dans sa partie commençant par :

```
############################
## Contact details
##
```

Pour afficher une carte, entrer la latitude et la longitude de l'adresse dans la ligne `coordinates`. Pour afficher un formulaire de messagerie, choisir le service *formspree.io* (`email_form = 2` dans `contact.md`). Pour activer le service de messagerie, il faudra construire le site web, s'envoyer un premier message en utilisant le formulaire et suivre les instructions de Formspree.

Le composant `people` est utilisé dans les sites collectifs pour présenter les membres. Le composant `slider` permet d'afficher un carrousel (des éléments défilants) en haut de page. Pour comprendre son fonctionnement, le plus simple consiste à l'activer.

Menu de la page d'accueil

La page d'accueil comporte un menu qui permet de naviguer rapidement vers ses composants ou vers d'autres pages. Il est paramétré dans `config/_default/menus.toml`. Les éléments du menu ont un nom affiché, un lien (commençant par `#` pour pointer vers un composant ou un chemin relatif dans le site comme `publication/`), et un poids qui définit leur ordre d'affichage, de la même façon que celui des composants de la page d'accueil.

Un menu à deux éléments pour pointer vers l'accueil du site et les billets de blogs est donc le suivant :

```
[[main]]
  name = "Home"
  url = "#about"
  weight = 10

[[main]]
  name = "Posts"
  url = "#posts"
  weight = 20
```

Le menu doit être traduit dans chaque langue dans le fichier `config/_default/languages.toml` :

```
[fr]
  [[fr.menu.main]]
    name = "Accueil"
    url = "#about"
    weight = 10
  [[fr.menu.main]]
    name = "Articles"
    url = "#posts"
    weight = 20
```

Billets

Le site est alimenté par des billets de blog placés dans le dossier `/contents/<langue>/post`. Il doivent être traduits et placés dans le dossier `post` de chaque langue pour être disponibles dans la langue correspondante. L'exemple utilisé ici est un guide pour estimer correctement la densité d'une variable bornée[30].

Son code est sur GitHub[31].

Un billet est placé dans un dossier (`/content/fr/post/densite`) qui contient son code R Markdown et éventuellement des images, des données pour alimenter le code et d'autres éléments appelés par le code. Hugo supporte des fichiers markdown natifs. L'apport de **blogdown** relativement à un site Hugo natif est le support de R Markdown, donc la possibilité d'exécuter tout code R comme dans un bloc-note (dont le contenu peut être réutilisé sans modification).

Le fichier principal d'un billet est `index.Rmd`. **blogdown** crée un fichier `index.html` pendant la construction du site : il peut être ignoré (dans `.gitgnore`) et supprimé à tout moment. Si une image `featured.png` (optimale pour un schéma) ou `featured.jpg` (optimale pour un photo) est placée dans le dossier, elle sera utilisée comme vignette du billet.

`index.Rmd` comprend un entête au format yaml (entourée par des `---`) ou toml (entourée par des `+++`) qui décrit son affichage :

```
---
title: "Titre du billet"
subtitle: "Sous-titre"
summary: "Résumé"
authors: []
tags: ["Mot Clé 1", "Autre Mot Clé"]
categories: []
date: 2020-04-17
featured: false
draft: false
```

[30] https://EricMarcon.github.io/post/densite/

[31] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/post/densite

```
# Featured image
# To use, add an image named `featured.jpg/png` to
# your page's folder.
# Focal points: Smart, Center, TopLeft, Top, TopRight,
# Left, Right, BottomLeft, Bottom, BottomRight.
image:
  caption: ""
  focal_point: ""
  preview_only: false

bibliography: references.bib
---
```

Les auteurs sont utilisés dans les sites collectifs. Les tags permettent d'alimenter le composant nuage de mots s'il est activé dans la page d'accueil. Les catégories permettent de rechercher des pages au contenu similaire (recherche par mot-clé sur le site). L'option `featured: true` fait apparaître le billet dans les composants de type `featured` sur la page d'accueil. L'option `draft: true` cache le billet.

Les éléments suivants précisent l'affichage de la vignette : légende et position. L'option `preview_only: true` limite l'affichage aux miniatures (sur la page d'accueil), retirant donc l'image du billet lui-même.

Les éléments d'entête nécessaires au corps de texte R Markdown, comme le nom du fichier contenant les références bibliographiques, placé dans le même dossier, sont ajoutés.

Le corps du texte est celui d'un document R Markdown standard, avec du code R inclus. Un bout de code initial permet de fixer les options de R et charger les packages nécessaires.

En pratique, la façon la plus efficace de créer un nouveau billet est de copier le dossier complet d'un billet précédent, de le renmmer et de modifier son contenu. La commande `blogdown::new_post()` peut aussi être utilisée mais ne gère pas les langues multiples (et crée donc le billet dans le dossier `/contents/post` à moins de préciser l'argument `subdir`).

La reconstruction du site ne met par défaut pas à jour les pages basées sur un fichier `.Rmd`. Pour le faire, il faut forcer la commande `build_site()`.

```r
blogdown::build_site(build_rmd = TRUE)
blogdown::serve_site()
```

Publications

Les publications sont organisées comme les billets, mais placées dans le dossier `/contents/<langue>/publications`.

L'exemple utilisé est un article de revue[32] avec son code[33] .

[32] https://EricMarcon.github.io/publication/marcon-2003-a/

[33] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/publication/marcon-2003-a

Un fichier `cite.bib` contenant la référence au format BibTex est placé dans le dossier. Le nom du dossier est de préférence celui de l'identifiant de la publication. L'entête du fichier `index.md` (ici au format Markdown, mais `.Rmd` est possible si du code R est nécessaire) contient les mêmes informations que le fichier BibTex, mais au format approprié (yaml), et les éléments propres à Academic (`featured`) :

```
---
title: "Evaluating the geographic concentration of |>
industries using distance-based methods"
authors: ["Eric Marcon", "Florence Puech"]
publication_types: ["2"]
abstract: "We propose (...)"
publication: "*Journal of Economic Geography*"
doi: "10.1093/jeg/lbg016"

date: 2003-10-01
featured: false
---
```

Les types de publication sont :

- 0 = Uncategorized ;
- 1 = Conference paper ;
- 2 = Journal article ;
- 3 = Preprint / Working Paper ;
- 4 = Report ;
- 5 = Book ;
- 6 = Book section ;
- 7 = Thesis ;
- 8 = Patent.

Des boutons sont affichés en haut de la page de la publication en fonction des informations trouvées :

- PDF : si la ligne `url` est présente dans l'en-tête ;
- Citation : si le fichier `cite.bib` est orésent dans le dossier ;
- DOI : si la ligne `doi` est présente dans l'en-tête.

Le corps de la publication contient un lien (au format HTML) vers le site Dimension qui fournit des informations bibliométriques. Ce lien peut être réutilisé très simplement, en remplaçant simplement le DOI du document :

```
<span class="__dimensions_badge_embed__"
  data-doi="10.1093/jeg/lbg016"></span>
```

```
<script async src="https://badge.dimensions.ai/
  badge.js" charset="utf-8"></script>
```

Enfin, un fichier `/contents/<langue>/publications/_index.Rmd` permet de présenter la bibliographie complète. Il est accessible à partir du composant `publications` de la page d'accueil qui affiche un lien "Plus de Publications".

Le fichier exemple[34] avec son code[35] permet d'interroger Google Scholar pour obtenir le réseau de coauteurs, l'indice h et le nombre de citations annuelles de l'auteur . Il est réutilisable en modifiant simplement l'identifiant Google Scholar à la ligne 30 .

En faisant exécuter le code régulièrement, par exemple par Travis-CI (voir ci-dessous), les statistiques affichées sont maintenues à jour sans intervention humaine.

[34] https://EricMarcon.github.io/publication/

[35] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/publication/marcon-2003-a

Communications

Les communications sont organisées comme les publications, dans le dossier `/contents/<langue>/talk`.

L'exemple utilisé est une communication en Français, donc dans `/contents/fr/talk`[36] avec son code[37] .

Une image peut être utilisée plus facilement que pour une publication.

L'entête contient des lignes particulières adaptées aux communications :

[36] https://EricMarcon.github.io/talk/chao1/

[37] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/talk/chao1

```
---
title: "Construction de l'estimateur de biodiversité |>
Chao1"
event: "Semaine des mathématiques 2020"
event_url: https://eduscol.education.fr/cid59178/|>
semaine-des-mathematiques.html

location: Université de Guyane

summary: []
abstract: |
  Pour estimer le nombre d'espèces (richesse
  spécifique) d'une communauté à partir d'un
  échantillon, l'estimateur Chao1 est l'outil
  le plus utilisé.

  Sa construction est expliquée et son efficacité
  est testée sur des données simulées.

# Talk start and end times.
#   End time can optionally be hidden by
```

```
# prefixing the line with `#`.
date: "2020-03-11T11:00:00Z"
date_end: "2020-03-11T12:00:00Z"
all_day: false

# Schedule page publish date (NOT talk date).
publishDate: "2020-04-14"

# Is this a featured talk? (true/false)
featured: false

image:
  caption: 'Produit scalaire des vecteurs $v_0$ |>
et $v_2$'
  focal_point: Smart

url_code: "https://github.com/EricMarcon/Chao1"
url_pdf: "https://EricMarcon.github.io/Chao1/|>
Chao1.pdf"
url_slides: "https://EricMarcon.github.io/Chao1/|>
Chao1.html"

# Enable math on this page?
math: true
---
```

Les liens (`url_code` par exemple) font apparaître des boutons qui permettent d'afficher respectivement le code source de la présentation, un fichier pdf et les diapositives en ligne.

Autres éléments

Il est possible d'ajouter librement des éléments supplémentaires sur le site :

- dans `/contents/<langue>/`, créer un dossier dont le nom est le type d'éléments (exemple : `recette`) ;
- ajouter des éléments dans ce dossier, chacun dans son propre dossier ;
- le fichier obligatoire est `index.md` ou `index.Rmd` avec un en-tête contenant possiblement tous les champs rencontrés dans les éléments `post`, `publication` et `talk` ;
- le fichier de vignette, `featured.png` ou `featured.jpg`, est facultatif ;
- tous les fichiers nécessaires au tricot (images, données) peuvent être ajoutés dans le même dossier ;
- dans `/contents/<langue>/home`, ajouter un composant de la page d'accueil en copiant-collant un élément existant de type "pages" (comme `publications`) ou "portfolio"

(comme `featured`) et le paramétrer pour qu'il pointe sur le bon dossier (dans l'exemple : `page-type=recette`) et ajuster son apparence (nombre d'éléments par exemple) et sa position (poids) ;
- ajouter éventuellement une entrée de menu pour pointer sur le composant, avec le même poids que le composant.

Les fichiers d'index peuvent porter l'extension `.Rmd` ou `.md`. Dans le premier cas, ils seront traités par **blogdown**, qui supporte l'intégration de code R. Dans l'autre cas, ils seront traités par Hugo, qui ne gère que le format markdown standard. Les fichiers `.md` nécessitent moins de ressource et sont donc préférés quand ils suffisent.

Finitions

L'icône du site, qui apparaît dans la barre d'adresse des navigateurs web, se trouve dans `assets/images`. Le fichier `icon.png` peut être remplacé.

Intégration continue

La construction du site web en production peut être confiée à Travis-CI (section 6.2), y compris sa mise à jour périodique si des pages du site traitent des données qui évoluent dans le temps.

Mises à jour

Le thème Academic est régulièrement mis à jour. La version utilisée est indiquée dans le fichier `go.mod`. Pour utiliser la dernière version officielle, exécuter dans la console R la commande suivante :

```
blogdown::hugo_cmd("mod get -u")
```

Les fichiers `go.mod` et `go.sum`, qui contient les codes de hachage des fichiers du module, sont mis à jour.

Chaque changement de version peut nécessiter des adaptations du contenu du site, référencées dans la documentation en ligne du thème[38].

[38] https://wowchemy.com/updates/

Mettre Hugo à jour en même temps :

```
blogdown::update_hugo()
```

4.8 Exportation de figures

Quand la production de documents avec R Markdown n'est pas possible, les figures issues de R doivent être exportées sous forme de fichiers pour être intégrés dans un autre processus

d'écriture. Il est préférable de créer des scripts pour créer les figures de façon reproductible et au format optimal.

Formats vectoriels et raster

Les figures doivent en général être produites dans un format vectoriel :

- SVG pour la publication d'affiches ou de posters ;
- EMF (Extended Meta-File) pour Word ou la suite Microsoft Office qui ne supporte pas d'autres formats ;
- EPS (Encapsulated PostScript) ou PDF (Portable Document Format) pour LaTeX.

Les figures raster (composées d'un ensemble de points, comme les photographies) sont rares dans R. La fonction `image()` utilisée pour afficher des cartes utilise par défaut des polygones plutôt que des points. La figure 4.8 montre le résultat du code suivant :

```
x <- 10 * (1:nrow(volcano))
y <- 10 * (1:ncol(volcano))
image(x, y, volcano, col = hcl.colors(100, "terrain"), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5), add = TRUE,
    col = "brown")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
```

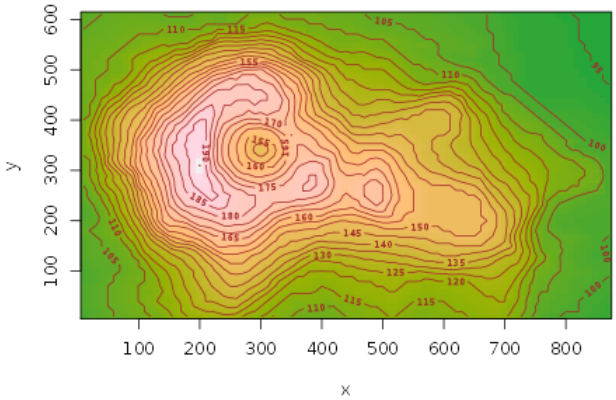


Fig. 4.8 : Courbes de niveau du volcan Maunga Whau, code fourni en exemple de l'aide de la foncion `image()`.

Elle est composée d'un ensemble de rectangles colorés : il s'agit bien d'une image vectorielle.

Si nécessaire, des images peuvent être produites aux formats BMP (bitmap, sans compression), JPEG (compressées avec perte de qualité), PNG (compressées sans perte de qualité, avec transparence possible) ou Tiff (compressées ou non).

Fonctions

La fonction `postscript()` produit un fichier EPS. Le code R doit appeler la fonction pour créer le fichier, produire la figure, puis fermer le fichier, par exemple :

```
# Ouverture du fichier
postscript("Fig1.eps", width = 6, height = 4, horizontal = FALSE)
# Création de la figure
plot(cars)
# Fermeture du fichier
dev.off()
```

```
## agg_png
##       2
```

La largeur et la hauteur (en pouces) d'un fichier vectoriel n'ont pas d'importance, mais leur rapport fixe l'aspect de la figure. La taille des textes est fixe : augmenter la taille de la figure revient donc à diminuer la taille relative des textes : procéder par essais successifs, en veillant à ce que les légendes restent lisibles à la taille finale de la figure.

L'argument `horizontal` fixe l'orientation de la figure de façon assez imprévisible : procéder par essais.

Les fonction `eps()`, `pdf()`, `bmp()`, `jpeg()`, `png()` et `tiff()` fonctionnent de la même manière. Se référer à l'aide des fonctions pour le choix des options (résolution, niveau de compression, etc.). La fonction `emf()` est fournie par le package **devEMF**.

Les polices de caractères ne sont pas incluses dans les fichiers EPS ou PDF. Si nécessaire, la fonction `embedFonts()` permet d'y remédier, à condition que GhostScript soit installé.

Package ragg

Le package **ragg**[39] améliore la qualité des fichiers PNG, JPEG et TIFF. Les fonctions optimisées sont `agg_png()`, `agg_jpeg()`et `agg_tiff()`. Leur usage est le même que celui des fonctions de **grDevices**.

[39] https://ragg.r-lib.org/

Les documents R Markdown produisent des images au format PNG pour leur version HTML. **ragg** améliore leur qualité : le package doit être installé et `dev = "ragg_png"` doit être ajoutée aux options de **knitr**. Pour ce document, les options déclarées dans `index.Rmd` sont les suivantes :

```
knitr::opts_chunk$set(
  cache=TRUE, warning=FALSE,
  echo = TRUE,
  fig.env='SCfigure', fig.asp=.75, fig.align='center', out.width='80%',
  dev = "ragg_png",
  tidy=TRUE, tidy.opts=list(blank=FALSE, width.cutoff=55), size="scriptsize",
  knitr.graphics.auto_pdf = TRUE)
```

Enfin, **ragg** peut être utilisé comme moteur de rendu graphique par défaut dans RStudio à partir de la versoin 1.4

(Menu “Tools > Global Options > General > Graphics > Backend”).

Chapitre 5

Package

Les packages de R permettent d'étendre les fonctionnalités du logiciel par du code fourni par la communauté des développeurs. Ils sont la clé du succès de R parce qu'ils permettent de diffuser rapidement de nouvelles méthodes issues de la recherche ou d'ajouter de nouveaux outils qui peuvent devenir des standards, comme le **tidyverse**.

Il est utile de produire un package quand on a écrit des nouvelles fonctions qui forment un ensemble cohérent. Un package à usage personnel ou limité à une équipe de travail est simple à mettre en place et le temps gagné en utilisant facilement la version à jour de chaque fonction amortit très rapidement le temps consacré à la fabrication du package. Ce type de package a vocation à être hébergé sur GitHub.

Des packages à usage plus large, qui fournissent par exemple le code correspondant à une méthode publiée, sont placés dans le dépôt CRAN, d'où ils pourront être installés par la commande standard `install.packages()`. CRAN effectue des vérifications poussées du code et n'accepte que les packages passant sans aucun avertissement sa batterie de tests. Ils doivent respecter la politique[1] du dépôt.

[1] https://cran.r-project.org/web/packages/policies.html

La documentation pour la création de packages est abondante. L'ouvrage de référence est celui de Wickham,[2] à consulter en tant que référence.

[2] H. Wickham (2015). *R Packages*. 1st. O'Reilly Media, Inc.

L'approche utilisée ici consiste à créer un premier package très rapidement pour comprendre que la démarche est assez simple. Il sera ensuite enrichi des éléments nécessaires à un package diffusé à d'autres utilisateurs que son concepteur : une documentation complète et des tests de bon fonctionnement notamment.

5.1 Premier package

Cette introduction reprend les recommandations du blog *Créer un package en quelques minutes*[3] de ThinkR.

[3] https://thinkr.fr/creer-package-r-quelques-minutes/

Création

Les packages ont une organisation stricte dans une structure de fichiers et de répertoires figée. Il est possible de créer cette structure manuellement mais des packages spécialisés peuvent s'en charger :

- **usethis** automatise la création des dossiers ;
- **roxygen2** permet d'automatiser la documentation obligatoire des packages ;
- **devtools** est la boîte à outils du développeur, permettant notamment de construire et tester les packages ;

Les trois sont à installer en premier lieu :

```
install.packages(c("usethis", "roxygen2", "devtools"))
```

Le package à créer sera un projet RStudio. Dans le menu des projets, sélectionner "New Project > New Directory > R package using devtools…", choisir le nom du projet et son dossier parent. Le package s'appellera **multiple**, dans le dossier `%LOCALAPPDATA%\ProjetsR` en suivant les recommandations de la section 1.2.

Le nom du package doit respecter les contraintes des noms de projets : pas de caractères spéciaux, pas d'espaces… Il doit aussi être évocateur de l'objet du package. Si le package doit être diffusé, toute sa documentation sera rédigée en Anglais, y compris son nom.

La structure minimale est crée :

- un fichier `DESCRIPTION` qui indique que le dossier contient un package et précise au minimum son nom ;
- un fichier `NAMESPACE` qui déclare comment le package intervient dans la gestion des noms des objets de R (son contenu sera mis à jour par **roxygen2**) ;
- un dossier `R` qui contient le code des fonctions offertes par le package (vide à ce stade).

Le package peut être testé tout de suite : dans la fenêtre *Build* de RStudio, cliquer sur "Install and Restart" construit le package et le charge dans R, après avoir redémarré le programme pour éviter tout conflit.

Dans la fenêtre *Packages*, **multiple** est maintenant visible. Il est chargé, mais ne contient rien.

Première fonction

Fichiers

Les fonctions sont placées dans un ou plusieurs fichier `.R` dans le dossier `R`. L'organisation de ces fichiers est libre. Pour

cet exemple, un fichier du nom de chaque fonction sera créé. Des fichiers regroupant les fonctions similaires ou un seul fichier contenant tout le code sont des choix possibles.

Le choix fait ici est le suivant :

- un fichier qui contiendra le code commun à tout le package : `package.R` ;
- un fichier commun à toutes les fonctions : `fonctions.R`.

Création

La première fonction, `double()`, est créée et enregistrée dans le fichier `fonctions.R` :

```
double <- function(number) {
    return(2 * number)
}
```

A ce stade, la fonction est interne au package et n'est pas accessible depuis l'environnement de travail. Pour s'en persuader, contruire le package (*Install and Restart*) et vérifier le bon fonctionnement de la fonction :

```
double(2)
```

Le résultat est un vecteur composé de deux 0 parce que la fonction appelée est un homonyme du package **base** (voir sa documentation en tapant `?double`) :

```
base::double(2)
```

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

Pour que la fonction de notre package soit visible, elle doit être *exportée* en la déclarant dans le fichier `NAMESPACE`. C'est le travail de **roxygen2** qui gère en même temps la documentation de chaque fonction. Pour l'activer, placer le curseur dans la fonction et appeler le menu "Code > Insert Roxygen Skeleton". Des commentaires sont ajoutés avant la fonction :

```
#' Title
#'
#' @param number
#'
#' @return
#' @export
#'
#' @examples
double <- function(number) {
    return(2 * number)
}
```

Les commentaires à destination de **roxygen2** commencent par `#'` :

- la première ligne contient le titre de la fonction, c'est-à-dire un descriptif très court : son nom en général ;
- la ligne suivante (séparée par un saut de ligne) peut contenir sa description (rubrique *Description* dans l'aide) ;
- la suivante (après un autre saut de ligne) peut contenir plus d'informations (rubrique *Details* dans l'aide) ;
- les arguments de la fonction sont décrits par les lignes `@param` ;
- `@return` décrit le résultat de la fonction ;
- `@export` déclare que la fonction est exportée : elle sera donc utilisable dans l'environnement de travail ;
- des exemples peuvent être ajoutés.

La documentation doit être complétée :

```
#' double
#'
#' Double value of numbers.
#'
#' Calculate the double values of numbers.
#'
#' @param number a numeric vector.
#'
#' @return A vector of the same length as `number` containing the
#'   transformed values.
#' @export
#'
#' @examples
#' double(2)
#' double(1:4)
double <- function(number) {
    return(2 * number)
}
```

Ne pas hésiter à s'inspirer de l'aide de fonctions existantes pour respecter les standards de R (ici : `?log`) :

- penser que les fonctions sont normalement vectorielles : `number` est par défaut un vecteur, pas un scalaire ;
- certains éléments commencent par une majuscule et se terminent par un point parce que ce sont des paragraphes dans le fichier d'aide ;
- le titre n'a pas de point final ;
- la description des paramètres ne commence pas par une majuscule.

La prise en compte des changements dans la documentation nécessitent d'appeler la fonction `roxygenize()`. Dans la fenêtre *Build*, le menu "More > Document" permet de le faire. Ensuite, construire le package (*Install and Restart*) et vérifier le résultat en exécutant la fonction et en affichant son aide :

```
double(2)
`?`(double)
```

Il est possible d'automatiser la mise à jour de la documentation à chaque construction du package par le menu "Build > Configure Build Tools..." : cliquer sur "Configure" et cocher la case "Automatically reoxygenize when running Install and Restart". C'est un choix efficace pour un petit package mais pénalisant quand le temps de mise à jour de la documentation s'allonge avec la complexité du package. La reconstruction du package est le plus souvent utilisée pour tester des modifications du code : sa rapidité est essentielle.

La documentation pour **roxygen2** supporte le format Markdown[4].

[4] https://roxygen2.r-lib.org/articles/markdown.html

A ce stade, le package est fonctionnel : il contient une fonction et un début de documentation. Il est temps de lancer une vérification de son code : dans la fenêtre *Build*, cliquer sur "Check" ou utiliser la commande `devtools::check()`. L'opération *réoxygène* le package (met à jour sa documentation), effectue un grand nombre de tests et renvoie la liste des erreurs, avertissements et notes détectées. L'objectif est toujours de n'avoir aucune alerte : elles doivent être traitées immédiatement. Par exemple, le retour suivant est un avertissement sur la non-conformité de la licence déclarée :

```
> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    `use_gpl3_license()`
  Standardizable: FALSE

0 errors v | 1 warning x | 0 notes v
Erreur : R CMD check found WARNINGs
```

Pour la corriger, mettre à jour, exécuter la commande de mise à jour de la licence, en commençant par votre nom :

```r
options(usethis.full_name = "Eric Marcon")
usethis::use_gpl3_license()
```

```
## v Setting active project to '/home/travis/build/EricMarcon/travailleR'
```

```
## v Leaving 'LICENSE.md' unchanged
```

```
## v Adding '^LICENSE\\.md$' to '.Rbuildignore'
```

La liste des licences valides est fournie par R[5].

[5] https://svn.r-project.org/R/trunk/share/licenses/license.db

Après correction, relancer les tests jusqu'à la disparition des alertes.

Contrôle de source

Il est temps de placer le code sous contrôle de source.

Activer le contrôle de source dans les options du projet (figure 3.2). Redémarrer RStudio à la demande.

Créer un dépôt sur GitHub et y pousser le dépôt local, comme expliqué dans le chapitre 3.

Créer le fichier `README.md` :

```
# multiple

An R package to compute mutiple of numbers.
```

Le développement du package est ponctué par de nombreux commits à chaque modification et une publication (push) à chaque étape, validée par une incrémentation du numéro de version.

package.R

Le fichier `package.R` est destiné à recevoir le code R et surtout les commentaires pour **roxygen2** qui concernent l'ensemble du package.

Le premier bloc de commentaire produira l'aide du package (`?multiple`).

```
#' multiple-package
#'
#' Multiples of numbers
#'
#' This package allows simple computation of multiples
#' of numbers, including fast algorithms for integers.
#'
#' @name multiple
#' @docType package
NULL
```

Son organisation est identique à celle des documentations de fonctions, avec deux déclarations particulières pour le nom du package et le type de documentation. Le code `NULL` après les commentaires indique à **roxygen2** qu'il n'y a pas de code R lié.

La documentation est mise à jour par la commande `roxygen2::roxygenise()`. Après reconstruction du package, vérifier que l'aide est est apparue : `?multiple`.

5.2 Organisation du package

Fichier DESCRIPTION

Le fichier doit être complété :

```
Package: multiple
Title: Calculate multiples of numbers
Version: 0.0.0.9000
Authors@R:
  person(given = "Eric",
           family = "Marcon",
           role = c("aut", "cre"),
           email = "e.marcon@free.fr",
           comment = c(ORCID = "0000-0002-5249-321X"))
Description: This package allows simple computation
  of multiples of numbers, including fast algorithms
  for integers.
License: GPL-3
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

Le nom du package est figé et ne doit pas être modifié.

Son titre doit décrire en une ligne à quoi il sert. Le titre est affiché dans la fenêtre *Packages* à côté des noms des packages.

La version doit respecter les conventions :

- Le premier nombre est la verson majeure, 0 tant que le package n'est pas stable puis 1. La version majeure ne change que si le package n'est plus compatible avec ses versions précédentes, ce qui oblige les utilisateurs à modifier leur code.
- Le deuxième est la version mineure, incrémentée quand des fonctionnalités nouvelles sont ajoutées.
- Le troisième est la version de correction : 0 à l'origine, incrémentée à chaque correction de code sans nouvelle fonctionnalité.
- Le quatrième est réservé au développement, et commence à 9000. Il est incrémenté à chaque version instable et disparaît quand une nouvelle version stable (*release*) est produite.

Exemple : une correction de bug sur la version 1.3.0 produit la version 1.3.1. Les versions de développement suivantes (instables, non destinées à l'usage en production) sont 1.3.1.9000 puis 1.3.1.9001, etc. Le numéro de version doit être mis à jour à chaque fois que le package est poussé sur GitHub. Quand le développement est stabilisé, la nouvelle version, destinée à être

utilisée en production, est 1.3.2 si elle n'apporte pas de nouvelle fonctionnalité ou 1.4.0 dans le cas contraire.

La description des auteurs est assez lourde mais simple à comprendre. Les identifiants Orcid des auteurs académiques peuvent être utilisés. Si le package a plusieurs auteurs, ils sont placés dans une fonction `c()` : `c(person(...), person())` pour deux auteurs. Dans ce cas, il faut préciser le rôle de chacun :

- "cre" pour le créateur du package
- "aut" pour un auteur parmi les autres
- "ctb" pour un contributeur, qui peut avoir signalé un bug ou fourni un peu de code.

La description du package en un paragraphe permet de donner plus d'informations.

La licence précise la façon dont le package peut être utilisé et modifié. GPL-3 est une bonne valeur par défaut, mais d'autres choix sont possibles[6].

[6] https://r-pkgs.org/description.html#description-license

L'option `LazyData` signifie que les données d'exemples fournies avec le package peuvent être utilisées sans les appeler au préalable par la fonction `data()` : c'est le standard actuel.

Enfin, les deux dernières lignes sont gérées par **roxygen2**.

Fichier NEWS.md

Le fichier `NEWS.md` contient l'historique du package. Les nouvelles versions sont ajoutées en haut du fichier.

Créer une première version du fichier :

```
# multiple 0.0.0.9000

## New features

* Initial version of the package
```

Les titres de premier niveau doivent contenir le nom du package et sa version. Les titres de niveau 2 sont libres, mais contiennent en général des rubriques comme "New features" et "Bug Fixes".

Pour ne pas multiplier les versions décrites, il est conseillé de modifier la version en cours et de compléter la documentation jusqu'au changement de version de correction (troisième nombre). Ensuite, l'entrée correspondant à cette version reste figée et une nouvelle entrée est ajoutée.

5.3 Vignette et pkgdown

Une vignette est indispensable pour documenter correctement le package :

```
usethis::use_vignette("multiple")
```

Le fichier `multiple.Rmd` est créé dans le dossier `vignettes`. Ajouter un sous-titre dans son entête : la description courte du package :

```
title: "multiple"
subtitle: "Multiples of numbers"
```

Le reste de l'entête permet à R de construire la vignette à partir de code R Markdown.

Le corps de la vignette contient par défaut du code R pour déclarer les options de présentation des bouts de code et le chargement du package. Une introduction à l'utilisation du package doit être écrite dans ce documents, en R Markdown.

Le package **pkgdown** permet de créer un site d'accompagnement du package[7], qui reprend le fichier `README.md` comme page d'accueil, la vignette dans une rubrique "Get Started", l'ensemble des fichiers d'aide avec leurs exemples exécutés (section "Reference"), le fichier `NEWS.md` pour un historique du package (section "Changelog") et des informations du fichier `DESCRIPTION`.

[7] Exemple : https://EricMarcon.github.io/entropart/

Créer le site avec **usethis**

```
usethis::use_pkgdown()
```

Construire ensuite le site. Cette commande sera exécutée à nouveau à chaque changement de version du package :

```
pkgdown::build_site()
```

Le site est placé dans le dossier `docs`. Ouvrir le fichier `index.htm` avec un navigateur web pour le visualiser. Dès que le projet sera poussé sur GitHub, activer les pages du dépôt pour que le site soit visible en ligne (voir section 3.6).

pkgdown place le site dans le dossier `docs`.

Ajouter l'adresse des pages GitHub dans une nouvelle ligne du fichier `DESCRIPTION` :

```
URL: https://GitHubID.github.io/multiple
```

L'ajouter aussi dans le fichier `_pkgdown.yml` qui a été créé vide, ainsi que l'option suivante :

```
url: https://GitHubID.github.io/multiple

development:
  mode: auto
```

pkgdown place le site dans le dossier `docs/dev` si le site d'une version stable (à trois nombres) du package existe dans `docs` et que la version en cours est une version de développement (à quatre nombres). De cette façon, les utilisateurs d'une version de production du package ont accès au site sans qu'il soit perturbé par les versions de développement.

Le site peut être enrichi de plusieurs façons :

- En ajoutant des articles au format R Markdown dans le dossier `vignettes/articles` ;
- En améliorant sa présentation (regroupement des fonctions par thèmes, ajout de badges, d'un sticker…) : se référer à la vignette de **pkgdown**.

5.4 Code spécifique aux packages

Importation de fonctions

Créons une nouvelle fonction dans `fonctions.R` qui ajoute un bruit aléatoire à la valeur double :

```r
fuzzydouble <- function(number, sd = 1) {
    return(2 * number + rnorm(length(number), 0, sd))
}
```

Le bruit est tiré dans une loi normale centrée d'écart-type `sd` et ajouté à la valeur calculée.

`rnorm()` est une fonction du package **stats**. Même si le package est systématiquement chargé par R, le package d'appartenance de la fonction doit obligatoirement être déclaré : les seules exceptions sont les fonctions du package **base**.

Le package **stats** doit d'abord être déclaré dans `DESCRIPTION` qui contient une instruction `Imports:`. Tous les packages utilisés par le code de **multiple** seront listés, séparés par des virgules.

```
Imports: stats
```

Cette "importation" signifie simplement que le package **stats** doit être chargé, mais pas nécessairement attaché (voir section 2.2), pour que **multiple** fonctionne.

Ensuite, la fonction `rnorm()` doit être trouvable dans l'environnement du package **multiple**. Il y a plusieurs façons de remplir cette obligation. D'abord, le commentaire suivant pourrait être fourni pour **roxygen2** :

```
#' @import stats
```

Tout l'espace de nom du package **stats** serait attaché et accessible au package **multiple**. Ce n'est pas une bonne pratique parce qu'elle multiplie les risques de conflits de noms (voir section 2.2). Notons que la notion d'importation utilisée ici est différente de celle de `DESCRIPTION`, bien qu'elles aient le même nom.

Il est préférable d'importer uniquement la fonction `rnorm()` en la déclarant dans la documentation de la fonction :

```
#' @importFrom stats rnorm
```

Ce n'est pas une pratique idéale non plus parce que l'origine de la fonction n'apparaîtrait pas clairement dans le code du package.

La bonne pratique est de ne rien importer (au sens de **roxygen2**) et de qualifier systématiquement les fonctions d'autres packages avec la syntaxe `package::fonction()`. C'est la solution retenue ici parce que la directive `@importFrom` importerait la fonction dans tout le package **multiple**, pas seulement dans la fonction `fuzzydouble()`, au risque de créer des effets de bord (modifier le comportement d'une autre fonction du package qui n'assumerait pas l'importation de `rnorm()`). Finalement, le code de la fonction est le suivant :

```
#' fuzzydouble
#'
#' Double value of numbers with an error
#'
#' Calculate the double values of numbers
#' and add a random error to the result.
#'
#' @param number a numeric vector.
#' @param sd the standard deviation of the Gaussian error added.
#'
#' @return A vector of the same length as `number`
#'  containing the transformed values.
#' @export
#'
#' @examples
#' fuzzydouble(2)
#' fuzzydouble(1:4)
fuzzydouble <- function(number, sd = 1) {
    return(2 * number + stats::rnorm(length(number), 0,
        sd))
}
```

Méthodes S3

Les méthodes S3 sont présentées en section 2.1.

Classes

Les objets appartiennent à des classes :

```r
# Classe d'un nombre
class(2)
```

```
## [1] "numeric"
```

```r
# Classe d'une fonction
class(sum)
```

```
## [1] "function"
```

En plus des classes de base, les développeurs peuvent en créer d'autres.

Méthodes

L'intérêt de créer de nouvelles classes est de leur adapter des méthodes existantes, le cas le plus courant étant `plot()`. Il s'agit d'une méthode générique, c'est-à-dire un modèle de fonction, sans code, à décliner selon la classe d'objet à traiter.

```r
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x25e68d8>
## <environment: namespace:base>
```

Il existe dans R de nombreuses déclinaisons de `plot` qui sont des fonctions dont le nom est de la forme `plot.class()`. **stats** fournit une fonction `plot.lm()` pour créer une figure à partir d'un modèle linéaire. De nombreux packages créent des classes adaptées à leurs objets et proposent une méthode `plot` pour chaque classe. Les fonctions peuvent être listées :

```r
# Quelques fonctions plot()
head(methods(plot))
```

```
## [1] "plot,ANY-method"   "plot,color-method" "plot.AccumCurve"
## [4] "plot.acf"          "plot.ACF"          "plot.addvar"
```

```r
# Nombre total
length(methods(plot))
```

```
## [1] 146
```

Inversement, les méthodes disponibles pour une classe peuvent être affichées :

```r
methods(class = "lm")
```

```
##  [1] add1           alias          anova          case.names     coerce
##  [6] confint        cooks.distance deviance       dfbeta         dfbetas
## [11] drop1          dummy.coef     effects        extractAIC     family
## [16] formula        fortify        hatvalues      influence      initialize
## [21] kappa          labels         logLik         model.frame    model.matrix
## [26] nobs           plot           predict        print          proj
## [31] qqnorm         qr             residuals      rstandard      rstudent
## [36] show           simulate       slotsFromS3    summary        variable.names
## [41] vcov
## see '?methods' for accessing help and source code
```

La méthode `print` est utilisée pour afficher tout objet (elle est implicite quand on saisit seulement le nom d'un objet) :

```r
my_lm <- lm(dist ~ speed, data = cars)
# Equivalent de '> my_lm'
print(my_lm)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

Le méthode `summary` affiche un résumé lisible de l'objet :

```r
summary(my_lm)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Les autres méthodes ont été créées spécifiquement pour les besoins du package **stats**.

Attribution d'un objet à une classe

Pour qu'un objet appartient à une classe, il suffit de le déclarer :

```r
x <- 1
class(x) <- "MyClass"
class(x)
```

```
## [1] "MyClass"
```

Une façon plus élégante de le faire est d'ajouter la nouvelle classe à l'ensemble des classes auquel l'objet appartient déjà :

```r
y <- 1
class(y) <- c("MyClass", class(y))
class(y)
```

```
## [1] "MyClass" "numeric"
```

Il n'y a aucune vérification de cohérence entre la structure réelle de l'objet et une structure de la classe qui serait déclarée ailleurs : le développeur doit s'assurer que les méthodes trouveront bien les bonnes données dans les objets qui déclarent lui appartenir. Dans le cas contraire, des erreurs se produisent :

```r
class(y) <- "lm"
tryCatch(print(y), error = function(e) print(e))
```

```
## <simpleError: $ operator is invalid for atomic vectors>
```

En pratique

Création d'une méthode générique

De nouvelles méthodes génériques peuvent être créées et déclinées selon les classes.

A titre d'exemple, créons une méthode générique `triple` qui calculera le triple des valeurs dans le package **multiple**, déclinée en deux fonctions distinctes : une pour les entiers et une pour les réels. Les calculs sur les nombres entiers sont bien plus rapides que ceux sur les réels, ce qui justifie l'effort d'écrire deux versions du code.

```r
# Méthode générique
triple <- function(x, ...) {
    UseMethod("triple")
}
```

La méthode générique ne contient pas de code au-delà de sa déclaration. Sa signature (c'est-à-dire l'ensemble de ses arguments) est importante parce que les fonctions dérivées de cette méthode devront obligatoirement avoir les mêmes arguments dans le même ordre et pourront seulement ajouter des arguments supplémentaires avant `...` (qui est obligatoire). Comme la nature du premier argument dépendra de la classe de chaque objet, l'usage est de l'appeler `x`.

La méthode est déclinée en deux fonctions :

```
triple.integer <- function(x, ...) {
    return(x * 3L)
}
triple.numeric <- function(x, ...) {
    return(x * 3)
}
```

Dans sa version entière, `x` est multiplié par `3L`, le suffixe `L` signifiant que 3 doit être compris comme un entier. Dans sa version réelle, 3 est noté `3.0` pour montrer clairement qu'il s'agit d'un réel.

Le choix de la fonction dépend de la classe de l'objet passé en argument.

```
# Argument entier
class(2L)
```

```
## [1] "integer"
```

```
# Résultat entier par la fonction triple.integer
class(triple(2L))
```

```
## [1] "integer"
```

```
# Argument réel
class(2)
```

```
## [1] "numeric"
```

```
# Résultat réel par la fonction triple.numeric
class(triple(2))
```

```
## [1] "numeric"
```

```
# Performance
microbenchmark::microbenchmark(triple.integer(2L), triple.numeric(2),
    triple(2L))
```

```
## Unit: nanoseconds
##                expr  min     lq     mean median     uq     max neval
##  triple.integer(2L)  292  315.5 17669.36  327.0  359.5 1730199   100
##   triple.numeric(2)  296  320.0 14578.07  340.0  368.0 1421755   100
##          triple(2L) 1011 1046.0  1233.15 1065.5 1157.0    7662   100
```

La mesure des performances par le package **microbenchmark** montre que la fonction `triple.integer()` est légèrement plus rapide que `triple.numeric` comme attendu mais que la méthode générique consomme beaucoup plus de temps que les calculs très simples ici. R teste en effet l'existence de fonctions correspondant à la classe de l'objet passé en argument aux méthodes génériques. Comme un objet peut appartenir à plusieurs classes, il recherche une fonction adaptée à la première classe, puis aux classes suivantes successivement. Cette recherche prend beaucoup de temps et justifie de réserver l'usage de méthodes génériques à

la lisibilité du code plutôt qu'à une recherche de performance : l'intérêt des méthodes génériques est de fournir à l'utilisateur du code une seule fonction pour un objectif donné (`plot` pour réaliser une figure) quelles que soient les données à traiter.

Création d'une classe

Dans un package, on créera des classes si les résultats des fonctions le justifient : structure de liste et identification de la classe à un objet ("lm" est la classe des modèles linéaires). Pour toute classe créée, les méthodes `print`, `summary` et `plot` (si une représentation graphique est possible) doivent être écrites.

Ecrivons une fonction `multiple()` dont le résultat sera un objet d'une nouvelle classe, "multiple", qui sera une liste mémorisant les valeurs à multiplier, le multiplicateur et le résultat.

```
multiple <- function(number, times = 1) {
    # Calculate the multiples
    y <- number * times
    # Save in a list
    result <- list(x = number, y = y, times = times)
    # Set the class
    class(result) <- c("multiple", class(result))
    return(result)
}
# Classe du résultat
my_multiple <- multiple(1:3, 2)
class(my_multiple)
```

```
## [1] "multiple" "list"
```

L'appel à la fonction `multiple()` renvoie un objet de classe "multiple", qui est aussi de classe "list". En absence de fonction `print.multiple()`, R cherche la fonction `print.list()` qui n'existe pas et se rabat sur la fonction `print.default()` :

```
my_multiple
```

```
## $x
## [1] 1 2 3
##
## $y
## [1] 2 4 6
##
## $times
## [1] 2
##
## attr(,"class")
## [1] "multiple" "list"
```

La fonction `print.multiple` doit donc être écrite pour un affichage lisible, limité au résultat :

```
print.multiple <- function(x, ...) {
    print.default(x$y)
}
# Nouvel affichage
my_multiple
```

```
## [1] 2 4 6
```

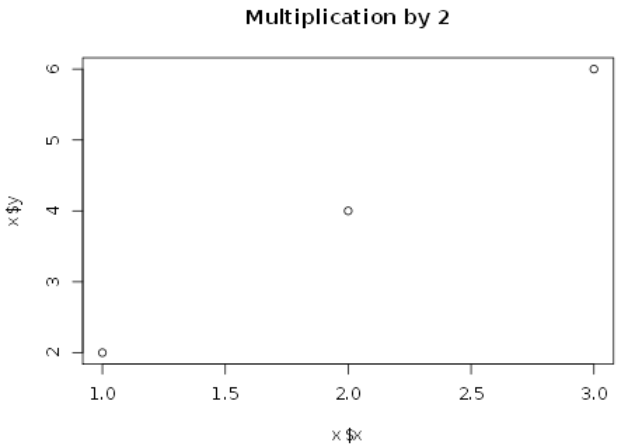
Les détails peuvent être présentés dans la fonction `summary` :

```
summary.multiple <- function(object, ...) {
    print.default(object$x)
    cat("multiplied by", object$times, "is:\n")
    print.default(object$y)
}
# Nouvel affichage
summary(my_multiple)
```

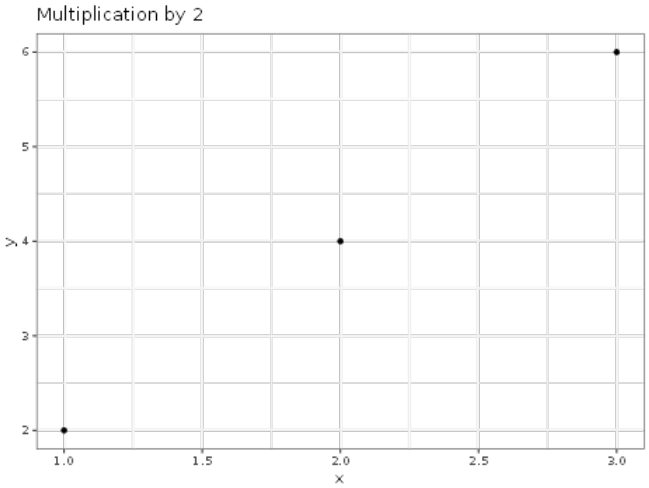
```
## [1] 1 2 3
## multiplied by 2 is:
## [1] 2 4 6
```

Enfin, une fonction `plot` et une fonction `autoplot` complètent l'ensemble :

```
plot.multiple <- function(x, y, ...) {
  plot.default(y=x$y, x=x$x, type = "p",
    main = paste("Multiplication by", x$times), ...)
}

autoplot.multiple <- function(object, ...) {
  data.frame(x = object$x, y = object$y) %>%
    ggplot2::ggplot() +
    ggplot2::geom_point(ggplot2::aes_(x = ~x, y = ~y)) +
    ggplot2::labs(title = paste("Multiplication by",
                                object$times))
}

plot(my_multiple)
```




```
autoplot(my_multiple)
```



Pour des raisons techniques liées à l'évaluation non conventionnelle dans le tidyverse, les fonctions `aes()` doivent être remplacées par `aes_()` dans les packages et ajouter un `~` devant les noms des variables. Dans le cas contraire, la vérification du package renvoie une note indiquant que les variables `x` et `y`, utilisées par les arguments de `aes()` n'ont pas été déclarées et n'existent peut-être pas dans l'environnement local (voir section 2.2).

Documentation

Les méthodes génériques et les fonctions qui les déclinent doivent être documentées comme n'importe quelle autre fonction.

La gestion de l'espace des noms est un peu plus complexe :

- Les méthodes génériques doivent être exportées :

```
#' @export
```

- Les fonctions dérivées de méthodes génériques ne doivent pas l'être mais être déclarées comme méthode, avec le nom de la méthode générique et la classe traitée. Etrangement, **roxygen2** demande qu'une directive d'exportation soit ajoutée mais ne l'applique pas (comme il se doit) dans le fichier `NAMESPACE` qui est utilisé par R :

```
#' @method plot multiple
#' @export
```

- Les fonctions dérivées de méthodes génériques venant d'un autre package nécessitent d'importer la méthode générique si elle n'est pas fournie par **base** (`print` est fourni par **base** et n'est donc pas concerné) :

```
#' @importFrom graphics plot
#' @importFrom ggplot2 autoplot
```

Dans `DESCRIPTION`, le package d'origine de chaque générique doit être listé dans la directive `Depends:`, pas dans `Imports:` :

```
Depends: R (>= 2.10), ggplot2, graphics
```

Enfin, l'importation de fonctions du tidyverse nécessite aussi quelques précautions :

- le package**tidyverse** est réservé à l'usage interactif de R : il n'est pas question de l'importer dans `DESCRIPTION` parce que ses dépendances peuvent changer et aboutir à des résultats imprévisibles ;
- Le package **magrittr** fournit les tuyaux, principalement `%>%`. Il doit être importé dans `DESCRIPTION`.

```
Imports: magrittr, stats
```

- Comme il n'est pas possible de les préfixer `%>%` par le nom du package, il faut importer la fonction en utilisant les délimiteurs prévus pour les fonctions dont le nom contient des caractères spéciaux :

```
#' @importFrom magrittr `%>%`
```

Finalement, le code complet est le suivant :

```
#' Multiplication of a numeric vector
#'
#' @param number a numeric vector
#' @param times a number to multiply
#'
#' @return an object of class `multiple`
#' @export
#'
#' @examples
#' multiple(1:2,3)
multiple <- function(number, times = 1) {
  # Calculate the multiples
  y <- number * times
  # Save in a list
  result <- list(x = number, y = y, times = times)
  # Set the class
  class(result) <- c("multiple", class(result))
  return(result)
}

#' Print objects of class multiple
#'
#' @param x an object of class `multiple`.
#' @param ... further arguments passed to the generic method.
#'
#' @export
#'
#' @examples
```

```r
#' print(multiple(2,3))
print.multiple <- function(x, ...) {
  print.default(x$y)
}

#' Summarize objects of class multiple
#'
#' @param object an object of class `multiple`.
#' @param ... further arguments passed to the generic method.
#'
#' @export
#'
#' @examples
#' summary(multiple(2,3))
summary.multiple <- function(object, ...) {
  print.default(object$x)
  cat("multiplied by", object$times, "is:\n")
  print.default(object$y)
}


#' Plot objects of class multiple
#'
#' @param x a vector of numbers
#' @param y a vector of multiplied numbers
#' @param ... further arguments passed to the generic method.
#'
#' @importFrom graphics plot
#' @export
#'
#' @examples
#' plot(multiple(2,3))
plot.multiple <- function(x, y, ...) {
  plot.default(y=x$y, x=x$x, type = "p",
               main = paste("Multiplication by", x$times), ...)
}


#' autoplot
#'
#' ggplot of the `multiple` objects.
#'
#' @param object an object of class `multiple`.
#' @param ... ignored.
#'
#' @return a `ggplot` object
#' @importFrom ggplot2 autoplot
#' @importFrom magrittr `%>%`
#' @export
#'
#' @examples
#' autoplot(multiple(2,3))
autoplot.multiple <- function(object, ...) {
  data.frame(x = object$x, y = object$y) %>%
    ggplot2::ggplot() +
    ggplot2::geom_point(ggplot2::aes_(x = ~x, y = ~y)) +
    ggplot2::labs(title = paste("Multiplication by",
                                object$times))
}
```

Code C++

L'utilisation de code C++ a été vue en section 2.5. Pour intégrer ces fonctions dans un packages, il faut respecter les règles suivantes :

- les fichiers `.cpp` contenant le code sont placés dans le dossier `/src` du projet ;
- le code est commenté pour **roxygen2** de la même façon que les fonctions R, mais avec le marqueur de commentaire du langage C :

```
#include <Rcpp.h>
using namespace Rcpp;

//' timesTwo
//'
//' Calculates the double of a value.
//'
//' @param x A numeric vector.
//' @export
// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

- dans `DESCRIPTION`, importer les packages. **Rcpp**, et **RcppParallel** si du code parallélisé est utilisé (supprimer ses références sinon), doivent être déclarés dans `LinkingTo` :

```
Imports: Rcpp, RcppParallel
LinkingTo: Rcpp, RcppParallel
```

- les commentaires pour **roxygen2** doivent être ajoutés à `package.R` (“multiple” est le nom du package) :

```
#' @importFrom Rcpp sourceCpp
#' @importFrom RcppParallel RcppParallelLibs
#' @useDynLib multiple, .registration = TRUE
```

- les fichiers de travail de C++ sont exclus du contrôle de source dans `.gitignore` :

```
# C binaries
src/*.o
src/*.so
src/*.dll
```

Ces modifications sont en partie effectuées automatiquement, pour **Rcpp** seulement, par **usethis**, mais l’insertion manuelle du code est plus rapide et fiable : ne pas utiliser cette commande.

```
# usethis::use_rcpp()
```

La construction du package entraînera la compilation du code : les Rtools sont donc indispensables.

Package bien rangé

Tout package moderne doit être compatible avec le tidyverse, ce qui nécessite peu d'efforts :

- pour permettre l'utilisation de pipelines, l'argument principal des fonctions doit être le premier ;
- les fonctions qui transforment des données doivent accepter un dataframe ou un tibble comme premier argument et retourner un objet du même format ;
- les méthodes `plot()` doivent être doublées de méthodes `autoplot()` avec les mêmes arguments qui produisent le même graphique avec **ggplot2**.

5.5 Bibliographie

La documentation d'un package fait appel à des références bibliographiques. Elles peuvent être gérées automatiquement avec **Rdpack** et **roxygen2**. Les références utilisées dans les fichiers R Markdown (vignette, site produit par **pkgdown**) ne sont pas concernées.

Préparation

Les références bibliographiques doivent être placées dans un fichier bibtex `REFERENCES.bib` placé dans le dossier `inst`. Ce dossier contient des fichiers qui seront placés à la racine du dossier du package quand il sera installé.

Ajouter la ligne suivante à `DESCRIPTION` :

```
RdMacros: Rdpack
```

Ajouter aussi le package `Rdpack` à la liste des packages importés :

```
Imports: magrittr, stats, Rcpp, Rdpack
```

Enfin, importer la fonction `reprompt()` de **Rdpack** en ajoutant les lignes suivantes à la documentation pour **roxygen2** dans `package.R` :

```
#' @importFrom Rdpack reprompt
```

Citations

Les références sont citées en utilisant la commande `\insertCite{key}{package}` dans la documentation destinée à **roxygen2**. `package` est le nom du package dans lequel le fichier `REFERENCES.bib` doit être cherché : ce sera normalement

le package en cours, mais les références d'autres packages sont accessibles, à la seule condition qu'ils utilisent **Rdpack**.

`key` est l'identifiant de la référence dans le fichier. Exemple[8] : documentation du package **SpatDiv** hébergé sur GitHub, fichier `.R` du package :

[8] Package **SpatDiv** sur GitHub : https://github.com/EricMarcon/SpatDiv/blob/master/R/package.R

```
#' SpatDiv
#'
#' Spatially Explicit Measures of Diversity
#'
#' This package extends the **entropart** package
#' \insertCite{Marcon2014c}{SpatDiv}.
#' It provides spatially explicit measures of
#' diversity such as the mixing index.
```

La référence citée se trouve dans `inst/REFERENCES.bib` :

```
@Article{Marcon2014c,
  author  = {Marcon, Eric and Herault, Bruno},
  title   = {entropart, an R Package to Partition
             Diversity},
  journal = {Journal of Statistical Software},
  year    = {2015},
  volume  = {67},
  number  = {8},
  pages   = {1--26},
}
```

Les citations sont entre parenthèses. Pour placer le nom de l'auteur hors de la parenthèse, ajouter la déclaration `;textual` :

```
\insertCite{Marcon2014c;textual}{SpatDiv}
```

Pour citer plusieurs références (forcément du même package), les séparer par des virgules.

A la fin de la documentation d'un objet utilisant des citations, ajouter systématiquement une liste des références :

```
#' @references
#' \insertAllCited{}
```

5.6 Données

Des données peuvent être intégrées à un package, notamment pour la clarté des exemples.

La méthode la plus simple consiste à utiliser **use_this**. Créer des variables contenant les données à sauvegarder puis les sauvegarder :

```
seq1_10 <- 1:10
seq1_100 <- 1:100
usethis::use_data(seq1_10, seq1_100)
```

Un fichier `.rda` est créé dans le dossier `data` pour chaque variable créée. Avec l'option `LazyData` activée dans `DESCRIPTION`, les variables seront disponibles dès le chargement du package, mais ne seront effectivement chargées en mémoire qu'après leur première utilisation.

Chaque variable doit être documentée dans le fichier `package.R` :

```
#' seq1_10
#'
#' A sequence of numbers from 1 to 10
#'
#' @format A numeric vector.
#' @source Values computed by the R software,
#'   \url{https://www.r-project.org/}
"seq1_10"
```

Le nom de la variable est donné entre guillemets après le bloc de commentaires (à la place du code R d'une fonction). `@format` décrit le format des données et `@source` permet d'indiquer leur source.

5.7 Tests unitaires

Dans l'idéal, tout le code inclus dans un package devrait être testé de multiples façons :

- contre les erreurs de syntaxe : les procédures de vérification de R s'en chargent assez bien ;
- pour vérifier la conformité des résultats de calculs aux valeurs attendues ;
- contre la survenue d'erreurs si les utilisateurs n'utilisent pas le code comme le développeur l'a prévu (arguments incorrects passés aux fonctions, données inadéquates…).

Les tests unitaires sont utilisés dans les deux derniers objectifs. Ils s'appuient sur **testthat** à intégrer au package :

```
usethis::use_testthat()
```

```
##
## Attaching package: 'testthat'

## The following object is masked from 'package:dplyr':
##
##     matches

## The following object is masked from 'package:purrr':
##
##     is_null

## The following object is masked from 'package:tidyr':
##
##     matches
```

Les tests doivent être ajoutés sous la forme de fichiers `.R` dont le nom commence obligatoirement par `test` dans le dossier `tests/testthat`.

Chaque test (donc le contenu de chaque fichier) commence par son contexte, c'est-à-dire ce un ensemble de tests. Exemple, dans un fichier `test_double.R` :

```
context("function double")
```

Les tests sont contenus dans des fichiers qui les regroupent par thème, par exemple `test_double.R`. Le nom de chaque test est passé comme argument de la fonction `test_that()` :

```
test_that("Double values are correct", {
    skip_on_cran()
    x <- 1:2
    # 2 x 2 should be 4
    expect_equal(double(x), c(2, 4))
    # The result should be a number (type = 'double')
    expect_type(double(x), "double")
    # Error management
    expect_error(double("a"))
})
```

```
## Test passed
```

Toutes les fonctions commençant par `expect` permettent de comparer leur premier argument à un résultat : dans l'exemple ci-dessus, le résultat de `double(1:2)` doit être `2 4` et le type de ce vecteur doit être réel à double précision. Le dernier test vérifie qu'une chaîne de caractère passée comme argument génère une erreur, ce qui n'est pas optimal : si le package traitait l'erreur, le message retourné pourrait être testé.

La commande `skip_on_cran()`, à utiliser systématiquement, évite de lancer les tests sur CRAN quand le package y sera déposé : CRAN dispose de ressources limitées et restreint strictement le temps de vérification des packages sur sa plateforme. Les tests devront donc être réalisés sur GitHub, grâce à l'intégration continue, voir section 5.9.

Les tests peuvent être lancés par le menu "More > Test package" de la fenêtre *Build* ou par la commande `devtools::test()`.

Il est conseillé d'écrire les tests dès qu'une fonction du package est stabilisée.

5.8 Fichier .gitignore

Le fichier `.gitignore` obtenu à ce stade est incomplet. Il peut être remplacé par celui-ci :

```
# History files
```

```
.Rhistory
.Rapp.history
# Session Data files
.RData
# Example code in package build process
*-Ex.R
# Output files from R CMD build
/*.tar.gz
# Output files from R CMD check
/*.Rcheck/
# RStudio files
.Rproj.user/
.Rprofile
# knitr and R markdown default cache directories
*_cache/
/cache/
# Temporary files created by R markdown
*.utf8.md
*.knit.md
# C binaries
src/*.o
src/*.so
src/*.dll
/src-i386/
/src-x64/
# uncomment if pkgdown is run by Travis
# docs/
```

La dernière ligne concerne le dossier `docs/`, qui reçoit le site web produit par **pkgdown**. Elle est commentée tant que la production du site est réalisée localement, mais décommentée si elle est confiée à Travis-CI (voir section suivante).

5.9 Intégration continue

La vérification (*Check*) du package doit être effectuée à chaque étape du développement, ce qui consomme un temps considérable. Elle peut être automatisée très simplement avec le service Travis-CI, déclenché à chaque modification du dépôt sur GitHub. L'analyse de la couverture du code par les tests (quelles parties du codes sont testées ou non) sera ajoutée.

Travis est également capable de reconstruire la documentation du package avec **pkgdown**, autre opération consommatrice de ressources, après la réussite des tests.

La section 6.2 détaille le moyen de le faire.

5.10 CRAN

Les packages dont l'audience dépasse l'entourage de l'auteur peuvent être déposés sur CRAN. Les règles à respecter sur CRAN sont nombreuses[9]. Elles sont vérifiées par la commande de vérification `R CMD check` avec l'option `-- as.cran`. La vérification ne doit renvoyer aucune erreur, aucun avertissement, ni aucune note avant de soumettre le package.

[9] https://cran.r-project.org/web/packages/policies.html

Test du package

La vérification du package par Travis dans le cadre de l'intégration continue n'est pas suffisante. Le package doit être testé sur la version de développement de R. Le site *R-hub builder*[10] permet de le faire simplement.

[10] https://builder.r-hub.io/

Le package, dont la version ne doit pas être de développement (limitée à trois nombres, voir section 5.2), doit être construit au format source : dans la fenêtre *Build* de RStudio, cliquer sur "More > Build Source Package". Sur le site *R-hub builder*, cliquer sur *Advanced*, sélectionner le fichier source du package et la plateforme de test : *Debian Linux, R-devel, GCC*.

Soumission

Quand le package est au point, la soumission à CRAN se fait par le site web dédié[11].

[11] https://xmpalantir.wu.ac.at/cransubmit/

En cas de rejet, traiter les demandes et soumettre à nouveau en incrémentant le numéro de version.

Maintenance

Des demandes de corrections sont envoyées par CRAN de temps à autre, notamment lors des changements de version de R. L'adresse de messagerie du responsable du package (*maintainer*) doit rester valide et les demandes doivent être traitées rapidement. Dans le cas contraire, le package est archivé.

Les nouvelles versions du package sont soumises de la même façon que la première.

Chapitre 6

Intégration continue

L'intégration continue consiste à confier à un service externe la tâche de vérifier un package, produire des documents Markdown pour les pages web d'un dépôt GitHub ou tricoter entièrement un site web à partir du code.

Toutes ces tâches peuvent être accomplies localement sur le poste de travail mais prennent du temps et risquent de ne pas être répétées à chaque mise à jour. Dans le cadre de l'intégration continue, elles le sont systématiquement, de façon transparente pour l'utilisateur. En cas d'échec, un message d'alerte est envoyé.

La mise en place de l'intégration continue se justifie pour des projets lourds, avec des mises à jour régulières. plutôt que pour des projets contenant un simple document Markdown rarement modifié.

6.1 Outils

Travis

L'outil utilisé le plus fréquemment pour des projets R déposés sur GitHub est *Travis CI*[1].

[1] https://travis-ci.org/

Il est nécessaire d'ouvrir un compte sur le site. Il est possible de s'authentifier avec son compte GitHub pour limiter le nombre de mots de passe.

Codecov

Pour évaluer le taux de couverture du code des packages R, c'est-à-dire la proportion du code testé d'une façon ou d'une autre (exemples, tests unitaires, vignette), le service *Codecov*[2] s'intègre parfaitement à Travis.

[2] https://codecov.io/

Il faut là encore ouvrir un compte, de préférence en s'authentifiant par GitHub.

GitHub Pages

Les pages web de GitHub peuvent être hébergées dans le répertoire `docs` de la branche master du projet : c'est la solution retenue quand elle sont produites sur le poste de travail.

Si elles sont produites par Travis, elle le seront obligatoirement dans une branche dédiée appelée `gh-pages`.

6.2 Intégration continue d'un projet de document

L'objectif est de faire tricoter par Travis un projet Markdown, comme ce document. Cette pratique est appropriée pour les projets d'ouvrages, qui nécessitent beaucoup de ressources pour leur construction. Dans ce type de projet, le code est tricoté par knitr pour produire plusieurs documents, typiquement aux formats HTML et PDF, accessibles sur les pages GitHub. Quand les documents sont produits localement, ils sont placés dans le dossier `docs` et poussés sur GitHub.

Pour que Travis s'en charge, quelques réglages sont nécessaires.

Obtention d'un jeton d'accès personnel

Pour écrire sur GitHub, le service d'intégration continue devra s'authentifier au moyen d'une clé privée, appelée *Personal Access Token* (PAT).

Les jetons sont créés sur GitHub, dans les paramètres de son compte d'utilisateur, dans *Developer Settings, Personal Access Tokens"[3].

[3] https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line

Générer un nouveau jeton, le décrire en tant que "Travis" et lui donner l'autorisation "repo", c'est-à-dire modifier *tous* les dépôts (il n'est pas possible de limiter l'accès à un dépôt particulier). Le jeton est une chaîne de caractère qui ne pourra pas être relue plus tard : elle doit être sauvegardée comme un mot de passe.

Activation du dépôt sur Travis

Sur le site de Travis, afficher les paramètres de son compte d'utilisateur. La liste des dépôts GitHub est présentée. Pour en activer un, cliquer sur l'interrupteur gris à côté de son nom (figure 6.1).

Cliquer ensuite sur "Settings" pour enregistrer le jeton d'accès. Dans la zone *Environment variables*, compléter :

- Name : GITHUB_TOKEN
- Value : <coller le jeton>

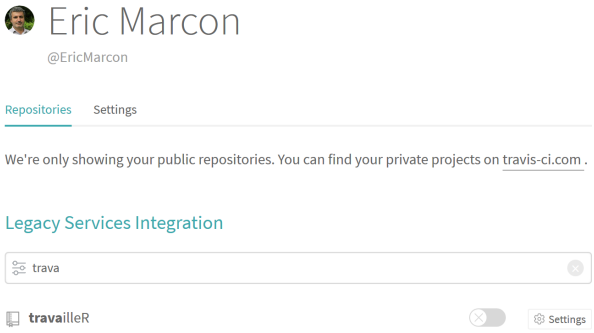


FIG. 6.1 : Activation d'un dépôt dans Travis CI.

Cliquer sur "Add". Le jeton ne pourra plus être lu sur Travis, pas plus que sur GitHub. Il sera utilisé comme une variable, à partir de son nom, pour n'apparaître dans aucun fichier.

Le nom *GITHUB_TOKEN* est une convention : on pourrait le modifier, mais on ne le fait pas.

Activation du dépôt sur CodeCov

L'opération est similaire sur le site Codecov : l'analyse de la couverture du code des packages est utile pour détecter les portions de code non testées. En revanche, l'analyse de la couverture des projets de document n'a pas d'intérêt.

Pour activer un dépôt, il faut d'authentifier sur le site de CodeCov avec son compte GitHub. La liste des dépôts est affichée et peut être actualisée. Si les dépôts à traiter sont hébergés par une organisation, par exemple les dépôts d'une salle de classe GitHub, il faut actualiser la liste des organisations en suivant les instructions (un lien permet de modifier rapidement les options de GithHub pour autoriser la lecture d'une organisation par Codecov) et à nouveau mettre à jour la liste des dépôts. Enfin, quand le dépôt recherché est visible, il faut l'activer. Il est inutile d'utiliser le système de jetons de Codecov pour les projets traités par Travis.

Scripter les actions de Travis

Projet d'ouvrage

Le script de contrôle de Travis doit être nommé `.travis.yml`. Il est au format YAML. Un script simple pour un projet `bookdown` ressemble à ceci (le signe `|>` indique que la suite du code devrait se trouver sur la même ligne, mais est coupée pour le formatage de ce document) :

```
language: r

script:
```

```
- Rscript -e "bookdown::render_book('index.Rmd', |>
    'bookdown::pdf_book', quiet = TRUE)"
- Rscript -e "bookdown::render_book('index.Rmd', |>
    'bookdown::gitbook', quiet = TRUE)"

deploy:
  provider: pages
  skip_cleanup: true
  github_token: $GITHUB_TOKEN
  keep_history: true
  local_dir: docs
  on:
    branch: master
```

Ce script minimal indique que le *langage* est R.

La partie *script* exécute des commandes sous R : `render_book` pour construire un livre, aux formats PDF et HTML. Le fichier PDF doit exister pour que le lien vers son téléchargement soit affiché dans la barre de commandes du format HTML, qui doit donc être créé en dernier.

La partie *deploy* gère le déploiement du résultat, écrit dans le dossier `docs` de la branche *master* par `render_book`. La commande `skip_cleanup` permet d'éviter que le contenu de `docs` soit effacé entre les phases `script` et `deploy`. La version 2 de l'outil de déploiement de Travis CI supprimera cette commande.

Le fournisseur `pages` tire la branche `gh-pages` depuis GitHub, valide les modifications et pousse le résultat. Le jeton GitHub est passé au script par Travis par la variable enregistrée précédemment, ce qui lui donne l'autorisation d'écrire sur GitHub.

Enfin, l'option `keep_history` permet de conserver l'historique de la branche `gh-pages`. Elle peut être mise à `false` pour l'effacer (commande `git push --force`) puisque le contenu n'est que le résultat d'une production automatique, l'historique important étant celui du code.

Ce script peut être amélioré en ajoutant un cache :

- pour éviter à Travis de recompiler les packages R à chaque exécution, et économiser plusieurs minutes ;
- pour éviter à knitR d'évaluer les calculs de tous les bouts de codes non modifiés, qui peuvent nécessiter de longs calculs.

Le code correspondant de la partie `cache`, à placer avant `script` est le suivant :

```
cache:
```

```
  packages: true
  directories: _bookdown_files
```

Enfin, Travis interrompt son exécution en absence de retour pendant 10 minutes. Si des bouts de code exécutent des calculs longs, il faut augmenter cette limite avec la commande `travis_wait` qui reçoit comme premier argument le nombre de minutes permises, ici une heure :

```
script:
  - travis_wait 60 Rscript -e |>
      "bookdown::render_book('index.Rmd', |>
      'bookdown::pdf_book', quiet = TRUE)"
  - travis_wait 60 Rscript -e |>
      "bookdown::render_book('index.Rmd', |>
      'bookdown::gitbook', quiet = TRUE)"
```

Les packages R à installer depuis CRAN peuvent être déclarés avant le script par une entrée `r_packages` et une entrée `r_github_packages` pour les packages sur GitHub. Ce n'est pas une bonne pratique pour les packages du projet : la bonne méthode consiste à utiliser un fichier `DESCRIPTION`, comme si le projet était un package R. Les directives `Depends` et `Imports` de ce fichier `DESCRIPTION` sont prises en compte par Travis et installés.

Certains packages nécessitent pour leur installation sous Linux (à partir de leur code source) celle de packages *apt*. Par exemple, le package R *sf* nécessite l'ajout des entrées suivantes :

```
addons:
  apt:
    packages:
      - libudunits2-dev
      - libgdal-dev
      - libgeos-dev
      - libproj-dev
```

Pour savoir quels packages *apt* installer, il est nécessaire de se référer à la documentation du package R et prendre en compte les instructions pour l'installation sous Ubuntu.

A la fin de l'exécution de son script, Travis peut envoyer systématiquement un message de compte-rendu :

```
notifications:
  email:
    on_success: always
```

Le comportement par défaut est d'envoyer un message au propriétaire du compte GitHub en cas d'échec ou seulement au premier succès d'un script qui échouait.

Finalement, les deux fichiers suivants sont ceux utilisés pour l'intégration continue de ce livre.

- Fichier `.travis.yml` :

  language : r

  cache : packages : true directories : _bookdown_files

  script : - travis_wait 60 Rscript -e |> "bookdown ::render_book('index.Rm
  |> 'bookdown ::gitbook', quiet = TRUE)" - travis_wait 60 Rscript -e |> "bookdown ::render_book('index.Rmd', |> 'bookdown ::pdf_book', quiet = TRUE)"

  deploy : provider : pages skip_cleanup : true github_token : $GITHUB_TOKEN keep_history : true local_dir : docs on : branch : master

- Fichier `DESCRIPTION` :

  Package : travailleR Title : Travailler avec R Version : 1.0.0 Authors@R : c(person("Eric", "Marcon", , "e.marcon@free.fr", |> c("aut", "cre"))) URL : https://github.com/EricMarcon/travailleR Depends : R (>= 3.1.0) Imports : bookdown, kableExtra, tidyverse, magrittr, gridExtra

Site web blogdown

Le fichier `.travis.yml` est très similaire :

```
language: r
latex: false

branches:
  only:
    - master

cache:
  packages: yes
  directories:
    - $HOME/bin

script:
  - Rscript -e 'blogdown::install_hugo(force=TRUE)'
  - Rscript -e 'blogdown::build_site(local = TRUE, |>
      build_rmd = TRUE)'
  - touch public/.nojekyll

deploy:
```

```
  provider: pages
  skip_cleanup: true
  token: $GITHUB_TOKEN
  on:
    branch: master
  local_dir: public
  repo: GitHubID/GitHubID.github.io
  target_branch: master
```

L'instruction `latex: false` ajoutée en début de fichier permet d'économiser le temps de l'installation d'une distribution LaTeX puisqu'un site web n'en a normalement pas besoin.

La mise en cache de `$HOME/bin` évite d'installer à chaque fois le générateur de site web Hugo.

Le script consiste à installer Hugo avec le package **blogdown** puis à construire le site (la commande `build_site()` est similaire à la commande `serve_site()` utilisée pendant le développement mais ne lance pas de serveur web pour afficher le résultat). Les arguments forcent la construction complète du site.

Le site est créé dans le dossier `public`. La commande `touch` crée un fichier vide appelé `.nojekyll` qui servira après publication à indiquer à GitHub que le site web créé ne doit pas être traité par son propre générateur de sites appelé Jekyll.

Tout utilisateur de GitHub dispose d'un dépôt par défaut dont le nom est `GitHubID.github.io`. Il n'a normalement qu'une seule branche `master`. Le contenu de la branche est un site web (pas sa source, qui doit être placée dans un dépôt différent) accessible à l'adresse https://GitHubID.github.io. Le déploiement copie le dossier `public` dans ce dépôt. Il ne garde pas de trace de l'historique (`skip_cleanup: true`) puisque le contrôle de source est déjà activé dans le dépôt contenant le code source du site.

Un fichier `DESCRIPTION` est obligatoire pour Travis qui y cherche les packages R à installer. Les premières lignes (Package, Title,...) ne sont pas utilisées par Travis mais permettent de respecter le format standard des fichiers `DESCRIPTION`.

```
Package: Home-Page
Title: Eric Marcon's Home Page
Version: 1.0.0
Authors@R: c(
  person("Eric", "Marcon", , "e.marcon@free.fr",
         c("aut", "cre"))
  )
URL: https://github.com/EricMarcon/Home-Page
Depends:
  R (>= 3.1.0)
Imports:
```

```
  blogdown,
  GoFKernel,
  igraph,
  scholar,
  tidyverse
```

Tous les packages utilisés dans les fichiers R Markdown du site doivent être déclarés dans le fichier `DESCRIPTION`.

Si le site web utilise des données en ligne qui justifient de le mettre à jour périodiquement, Travis peut être lancé tous les jours, toutes les semaines ou tous les mois en plus des reconstruction déclenchées par une modification du dépôt sur GitHub. Pour cela, afficher les propriétés du dépôt sur Travis-CI (voir la saisie du jeton GitHub en section 6.2) et ajouter une tâche programmée (*cron job*). Compléter puis cliquer sur "Add" :

- Branch : master ;
- Interval : selon les besoins ;
- Options : au choix, exécuter systématiquement ou ne pas le faire si le site a déjà été reconstruit dans les dernières 24 heures.

[4] https://EricMarcon.github.io/fr/publication/

Exemple : la page qui affiche la bibliométrie du site web[4] de l'auteur interroge Google Scholar pour afficher les citations des publications. Le site est mis à jour tous les mois pour que les statistiques soient à jour.

Packages R

Le script minimal est :

```
language: r
```

Travis s'appuie sur le fichier `DESCRIPTION` pour installer les packages nécessaires et lance la vérification du package avec l'option `--as-cran`. La vérification échoue si elle retourne une erreur, un avertissement ou une note.

Les packages supplémentaires, nécessaires aux actions de Travis en cas de succès, sont déclarés dans `r_packages` :

```
r_packages: # Packages à installer pour Travis
  - covr    # Interfaçage de Covecov
  - pkgdown # Création du site d'aide des packages R
```

La partie `after_success` du script permet d'effectuer des tâches seulement dans cette situation : typiquement appeler Codecov pour mesurer le taux de couverture du code et utiliser **pkgdown** pour créer le site web d'aide :

```
after_success:
  # Appel à Covecov
  - Rscript -e 'covr::codecov(type ="all")'
  # Dossier du site d'aide
  - mkdir docs
  # Tirer...
  - git fetch origin gh-pages:gh-pages
  # ...dans docs
  - git --work-tree=docs checkout gh-pages -- .
  # Construction du site
  - travis_wait 60 Rscript -e 'pkgdown::build_site()'
  # Fichier vide
  - touch docs/.nojekyll
```

La construction du site par **pkgdown** est plus compliqué que celle d'un livre produit par **bookdown** parce que deux versions coexistent si le package est hébergé sur CRAN : à la racine du dossier `docs`, la version de production correspondant à la version sur CRAN et, dans `docs/dev`, la version correspondant à la version en cours du package. Une seule est produite par Travis mais les deux doivent être conservées : il ne faut pas que la version de production, consultée par les utilisateurs du package, soit supprimée quand une version de développement est créée.

La dernière ligne du script consiste à créer un fichier vide nommé `.nojekyll` pour indiquer à GitHub que les pages web produites ne doivent pas être traitées comme le code source d'un site Jekyll, ce qui est le comportement par défaut des pages GitHub.

L'exemple suivant correspond au package **entropart**, déposé sur CRAN.

Fichier `.travis.yml` :

```
language: r
warnings_are_errors: true     # Comportement de CRAN
cache: packages

r_packages:     # Packages à installer pour Travis
  - covr        # Interfaçage de Covecov
  - pkgdown     # Pour la création du site d'aide

after_success:
  - Rscript -e 'covr::codecov(type ="all")'
  - mkdir docs
  - git fetch origin gh-pages:gh-pages
  - git --work-tree=docs checkout gh-pages -- .
  - travis_wait 60 Rscript -e 'pkgdown::build_site()'
  - touch docs/.nojekyll
```

```
deploy:
  provider: pages
  skip_cleanup: true
  keep_history: true
  token: $GITHUB_TOKEN
  on:
    branch: master
  local_dir: docs

notifications:
  email:
    recipients:
      - e.marcon@free.fr
```

Pour des paramétrages particuliers, la documentation complète de Travis peut être consultée[5].

[5] https://docs.travis-ci.com/user/languages/r/

Vérification de `.travis.yml`

Le débogage de la configuration de Travis est particulièrement laborieux parce que toute modification du fichier `.travis.yml` nécessite d'être validée et poussée pour déclencher Travis et éventuellement générer des erreurs. L'ensemble du processus peut durer plusieurs dizaines de minutes. Un outil de vérification en ligne[6] permet de valider au moins la syntaxe du fichier et donc de gagner du temps.

[6] https://config.travis-ci.com/explore

Problèmes transitoires

Les changements de version de R, de LaTeX, de pandoc… peuvent générer des erreurs dans l'intégration continue. Une bonne pratique consiste à adapter son fichier `.travis.yml` en s'inspirant de projets qui fonctionnent : il suffit de regarder leur code sur GitHub.

Parmi les problèmes les plus courants :

- pour les projets d'ouvrage, une version plus ancienne de pandoc sur Travis que sur le poste de travail empêche le bon tricot du document alors qu'il fonctionne parfaitement en local. La solution consiste à forcer la version de pandoc (elle devra être mise à jour régulièrement). Le début de fichier suivant force les versions de Linux et de pandoc :

```
language: r
os: linux
dist: xenial
pandoc_version: 2.7.3
```

- pour tous les projets, la défaillance d'un package importé qui ne s'installe pas correctement peut être contournée en

utilisant les versions binaires des packages fournies par RStudio. Les commandes suivantes, ajoutées avant les scripts, permettent de le faire :

```
# Packages binaires à partir de RStudio
before_install:
  - echo "options(repos =
     c(CRAN = 'https://packagemanager.rstudio.com/all/|>
     __linux__/xenial/latest'))" >> ~/.Rprofile.site
  - echo "options(HTTPUserAgent = paste0('R/',  |>
    getRversion(), ' R (', paste(getRversion(), |>
    R.version['platform'], R.version['arch'],   |>
    R.version['os']), ')'))" >> ~/.Rprofile.site
```

Ces contournements devront être réévalués régulièrement et supprimés quand ils ne seront plus utiles.

Ajouter des badges

Le succès du script sur Travis est visible sur GitHub en ajoutant un badge dans le fichier README.md, juste après le titre du fichier :

```
# Nom du projet

[![Travis-CI Build Status](https://travis-ci.org/ |>
  <GitHubID>/<Depot>.svg?branch=master)] |>
  (https://travis-ci.org/<GitHubID>/<Depot>)
```

Le taux de couverture mesuré par Codecov peut aussi être affiché par un badge :

```
![codecov](https://codecov.io/github/<GitHubID>/
  <Depot>/branch/master/graphs/badge.svg)]
  (https://codecov.io/github/<GitHubID>/<Depot>)
```

Chapitre 7

Shiny

Shiny permet de publier sous la forme d'un site web une application interactive utilisant du code R. Le site peut fonctionner localement, sur le poste de travail d'un utilisateur qui le lance à partir de RStudio, ou en ligne, sur un serveur dédié exécutant Shiny Server[1].

[1] https://rstudio.com/products/shiny/download-server/

De façon basique, un formulaire permet de saisir les arguments d'un fonction et une fenêtre de visualisation d'afficher les résultats du calcul.

L'utilisation d'une application Shiny rend très simple l'exécution du code, y compris pour des utilisateurs étrangers à R, mais limite évidemment les possibilités.

7.1 Première application

Dans RStudio, créer une application avec le menu "File > New File > Shiny Web App…", saisir le nom de l'application "MonAppShiny" et sélectionner le dossier où la placer.

Le nom de l'application a servi à créer un dossier qu'il faut maintenant transformer en projet (menu des projets en haut à droite de RStudio, "New Project > Existing Directory", sélectionner le dossier de l'application).

Le fichier de l'application nommé `app.R` contient deux fonctions : `ui()` qui définit l'interface graphique et `server()` qui contient le code R à exécuter. L'application peut être lancée en cliquant sur "Run App" dans la fenêtre du code.

La correspondance entre la fenêtre affichée (figure 7.1) et le code de la fonction `ui()` est simple à voir :

- le titre de l'application est affiché par la fonction `titlePanel()` ;
- le curseur qui fixe le nombre de barres de l'histogramme est créé par `sliderInput()` ;

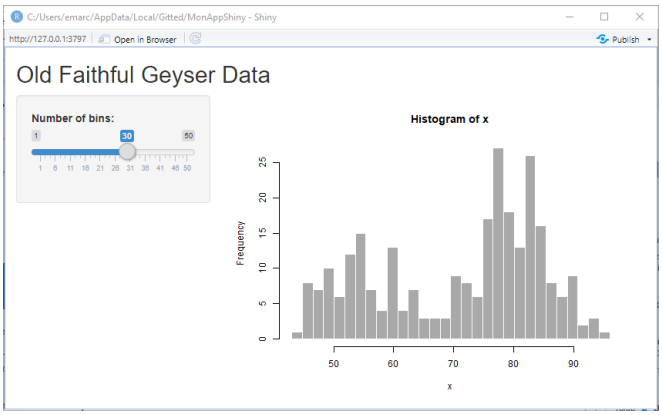


FIG. 7.1 : Application Shiny *Old Faithful Geyser Data.*

- la fonction `sidebarLayout()` fixe la disposition des éléments de la page, `sidebarPanel` pour les contrôles de saisie et `mainPanel()` pour l'affichage du résultat.

Le résultat est affiché par la fonction `plotOutput()` dont l'argument est le nom d'un élément de `output`, la variable remplie par la fonction `server()`.

Toute modification d'un élément de l'interface, précisément d'un élément affiché par une fonction dont le nom se termine par `Input()` (il en existe pour tous les types d'entrées, par exemple `textInput()`) de **Shiny** provoque l'exécution de `server()` et la mise à jour des éléments de `output`.

7.2 Application plus élaborée

Méthode de travail

Une application est créée en choisissant :

- une disposition de la fenêtre (*layout*) ;
- les contrôles de saisie des paramètres (*intput*) ;
- les contrôles d'affichage des résultats (*output*).

Le code pour traiter les entrées et produire les sorties est ensuite écrit dans `server()`.

Le tutoriel de RStudio[2] est très détaillé et doit être utilisé pour aller plus loin.

[2] https://shiny.rstudio.com/tutorial/

Exemple

Cette application simple utilise le package **scholar** pour interroger Google Scholar et obtenir les données bibliométriques d'un auteur à partir de son identifiant.

Le fichier `app.R` contient tout le code et est construit progressivement ici. L'application complète, avec des sorties

graphiques en plus de sa version simplifiée présentée ici est disponible sur GitHub[3].

[3] https://github.com/EricMarcon/bibliometrics

Le début du code consiste à préparer l'exécution de l'application en chargeant les packages nécessaires :

```
# Prepare the application ####
# Load packages
library("shiny")
library("tidyverse")
```

Le code de l'application complète intègre une foncion pour installer les packages manquants, à n'exécuter que quand l'application est exécutée sur un poste de travail (sur un serveur, la gestion des packages n'est pas du ressort de l'application).

L'interface utilisateur est la suivante :

```
# UI ####
ui <- fluidPage(
  # Application title
  titlePanel("Bibliometrics"),

  sidebarLayout(
    sidebarPanel(
      helpText("Enter the Google Scholar ID of an author."),
      textInput("AuthorID", "Google Scholar ID", "4iLBmbUAAAAJ"),
      # End of input
      br(),
      # Display author's name and h
      uiOutput("name"),
      uiOutput("h")
    ),
    # Show plots in the main panel
    mainPanel(
      plotOutput("network"),
      plotOutput("citations")
    )
  )
)
```

La fenêtre de l'application est fluide, c'est-à-dire qu'elle se réorganise seule quand sa taille varie, et est composée d'un panneau latéral (pour la saisie et l'affichage de texte) et d'un panneau principal, pour l'affichage de graphiques.

Les éléments du panneau latéral sont :

- un texte d'aide : `helpText()` ;
- un champ de texte à saisir, `textInput()`, dont les arguments sont le nom, le texte affiché, et la valeur par défaut (l'identifiant d'un auteur) ;
- un saut de ligne : `br()` ;
- des contrôles de sortie au format HTML : `uiOutput()`, dont l'argument unique est le nom.

Le panneau principal contient deux contrôles de sortie graphiques, `plotOutput()` dont l'argument est aussi le nom.

Le code à exécuter pour traiter les entrées et produire les sorties est dans la fonction `server()`.

```
# Server logic ####
server <- function(input, output) {
  # Run the get_profile function only once ####
  # Store the author profile
  AuthorProfile <- reactiveVal()
  # Update it when input$AuthorID is changed
  observeEvent(input$AuthorID,
               AuthorProfile(get_profile(input$AuthorID)))

  # Output ####
  output$name <- renderUI({
    h2(AuthorProfile()$name)
  })

  output$h <-
    renderUI({
      a(href = paste0(
        "https://scholar.google.com/citations?user=",
        input$AuthorID),
        paste("h index:", AuthorProfile()$h_index),
        target = "_blank"
      )
    })

  output$citations <- renderPlot({
    get_citation_history(input$AuthorID)  %>%
      ggplot(aes(year, cites)) +
      geom_segment(aes(xend = year, yend = 0),
                   size = 1,
                   color =
                     'darkgrey') +
      geom_point(size = 3, color = 'firebrick') +
      labs(title = "Citations per year",
           caption = "Source: Google Scholar")
  })

  output$network <- renderPlot({
    ggplot() + geom_blank()
  })
}
```

Les informations nécessaires aux champs de sortie `$name` et `$h` (nom de l'auteur et indice h) sont obtenus par la fonction `get_profile()` du package **scholar**. Cette fonction effectue interroge la page web Google Scholar de l'auteur et extrait les valeurs du résultat : c'est une traitement lourd, qu'il vaut mieux n'exécuter qu'une seule fois plutôt que deux, dans les fonctions `renderUI()` chargées de calculer les valeurs de `output$h` et `output$name`.

Le code le plus simple pour le faire serait le suivant :

```
# Run the get_profile function only once #### Store the
# author profile
AuthorProfile <- get_profile(input$AuthorID)
```

La difficulté de la programmation d'une application Shiny est que tout calcul se référant à un élément de l'interface d'entrée doit être *réactif*. Si ce dernier code était exécuté, le message d'erreur suivant apparaît : "Operation not allowed without an

active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)".

En pratique, l'exécution du code est lancée par la modification d'un contrôle d'entrée (ici : `intput$AuthorID`). Le code faisant référence à un de ces contrôles doit être en permanence en attente d'une modification : il doit donc placé dans des fonctions particulières comme `renderPlot` dans l'application *Old Faithful Geyser Data* ou `renderUI()` ici. Le code suivant s'exécuterait sans erreur :

```
# Output ####
output$name <- renderUI({
    AuthorProfile <- get_profile(input$AuthorID)
    h2(AuthorProfile$name)
})
```

L'appel à la valeur du contrôle `input$AuthorID` a bien lieu dans une fonction réactive (mais `get_profile()` devrait être utilisé une deuxième fois dans le calcul de `output$h`, ce que nous voulons éviter). La fonction `h2(AuthorProfile$name)` produit du code HTML, un paragraphe de titre de niveau 2 dont la valeur est passée en argument.

Toutes les fonctions dont le nom commence par `render` dans le package **shiny** sont réactives, et chacune est destinée à produire un type de sortie différent, par exemple du texte (`renderText()`) ou du code HTML (`renderUI()`).

Si du code est nécessaire pour calculer des variables communes à plusieurs contrôles de sortie (`output$name` et `output$h`), il doit lui-même être réactif. Deux fonctions sont très utiles :

- `observeEvent()` surveille les changements d'un contrôle d'entrée et exécute du code quand ils se produisent ;
- `reactiveVal()` permet de définir une variable réactive, qui sera modifiée par le code de `observeEvent()` et entraînera à son tour l'exécution d'autres fonctions réactives qui utilisent sa valeur.

Le code optimal crée donc une variable réactive pour y stocker le résultat de l'interrogation de Google Scholar

```
# Store the author profile
AuthorProfile <- reactiveVal()
```

La variable réactive est vide à ce stade. Son utilisation est ensuite celle d'une fonction : `AuthorProfile(x)` lui attribue la valeur `x` et `AuthorProfile()`, sans argument, renvoie sa valeur. La fonction `observeEvent()` est déclenchée quand `input$AuthorID` est modifié et exécute le code passé en deuxième argument, ici la mise à jour de `AuthorProfile`.

```
# Update it when input$AuthorID is changed
observeEvent(input$AuthorID, AuthorProfile(get_profile(input$AuthorID)))
```

Enfin, les fonctions `renderUI()` qui fournissent les valeurs des contrôles de sortie utilisent la valeur de `AuthorProfile` :

```
# Output ####
output$name <- renderUI({
    h2(AuthorProfile()$name)
})
```

Remarquer les parenthèses de `AuthorProfile()`, variable réactive, par opposition à la syntaxe `AuthorProfile$name` pour une variable classique.

La valeur de `output$h` est un lien internet, `<a href=...` en HTML, écrit par la fonction `a()` du package **htmltools** utilisé par `renderUI()`.

```
output$h <- renderUI({
    a(href = paste0("https://scholar.google.com/citations?user=",
        input$AuthorID), paste("h index:", AuthorProfile()$h_index),
        target = "_blank")
})
```

Le lien est vers la page Google Scholar de l'auteur. La valeur affichée est son indice h. L'argument `target = "_blank"` indique que le lien doit être ouvert dans une nouvelle fenêtre du navigateur.

Le graphique `output$citations` est créé par la fonction réactive `renderPlot()`. Les données fournies par la fonction `get_citation_history()` de **scholar** (qui interroge l'API de Google Scholar) sont traitées par `ggplot()`.

Enfin, le graphique `output$network` est un graphique vide dans cette version simplifiée de l'application.

L'application complète reprend ce code en y ajoutant le traitement des erreurs dans le cas où le code de l'auteur n'existe par sur Google Scholar et le graphique du réseau des co-auteurs.

7.3 Hébergement

Une application Shiny n'est pas forcément hébergée par un serveur web : elle peut être exécutée sur les postes de travail des utilisateurs s'ils disposent de R.

Pour un usage plus large, un serveur dédié est nécessaire. Shinyapps.io[4] est un service de RStudio qui permet d'héberger gratuitement 5 applications Shiny avec un temps de fonctionnement maximal de 5 heures par mois.

[4] https://www.shinyapps.io/

Il faut tout d'abord ouvrir un compte sur le site, de préférence avec ses identifiants GitHub. Pour permettre la

gestion des applications en ligne directement depuis RStudio, il faut installer ensuite le package **rsconnect** et le paramétrer :

```
rsconnect::setAccountInfo(name = "prenom.nom", token = "xxx",
    secret = "<SECRET>")
```

Le code exact, avec le nom d'uilisateur et le jeton à utiliser, sont affichés sur la page d'accueil de Shinyapps.io : cliquer sur "Show Secret", copier le code et le coller dans la console de RStudio pour l'exécuter. Un bouton *Publish* est disponible juste à droite du bouton *Run App*. Cliquer dessus et valider la publication (figure 7.2).

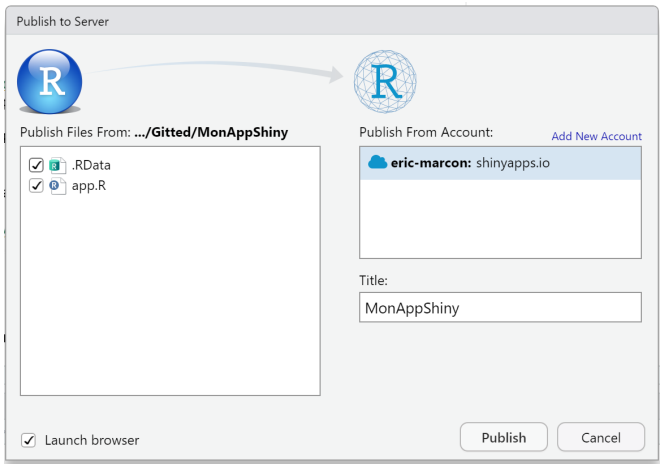


Fig. 7.2 : Application Shiny *Old Faithful Geyser Data*.

L'application est maintenant accessible à l'adresse https://prenom-nom.shinyapps.io/MonAppShiny/

L'application "Bibliometrics" ne fonctionne pas sur Shinyapps.io parce que la façon dont le package **Scholar** interroge Google Scholar n'est pas supportée. La plupart des applications Shiny fonctionnent sans difficulté, tant qu'elles ne nécessitent pas de fonctionnalités réseau complexes.

Chapitre 8

Enseigner avec R

R, RStudio et GitHub fournissent des outils pour enseigner.

Le package **learnr** permet de réaliser des tutoriels interactifs.

On verra aussi comment utiliser les salles de classe GitHub (*GitHub Classrooms*) qui permettent de diffuser à une classe (une liste d'étudiants disposant d'un compte GitHub) un modèle de dépôt (un début de projet R) que chaque étudiant devra développer et publier. Les outils de la salle de classe permettent d'évaluer le travail fourni assez simplement.

8.1 learnr

learnr permet de rendre interactifs les bouts de code de n'importe quel document produit par R Markdown en HTML, en les transformant en applications Shiny. La documentation sur le site de RStudio[1] est très claire et ne sera pas reprise ici : nous verrons seulement comment commencer et comment diffuser les tutoriels.

[1] https://rstudio.github.io/learnr/

Premier tutoriel

Utiliser comme pour tous les documents le menu "File > New File > RMarkdown…" et créer un nouveau document à partir d'un modèle "Interactive Tutorial". L'assistant crée un dossier du nom choisi, à transformer en projet R et passer sous contrôle de source, comme pour tous les documents vus précédemment (voir section 4.3).

Pour exécuter le tutoriel, cliquer sur le bouton *Run Document* qui se trouve à place habituelle du bouton *Tricoter*.

Les tutoriels peuvent inclure des exercices, qui sont des bouts de code avec l'option `exercise=TRUE`. Ces exercices sont affichés sous la forme d'une fenêtre de code modifiable et exécutable par l'utilisateur. Des indices peuvent être donnés[2], un bouton ajouté pour afficher la solution, une limite de temps peut être fixée[3], et

[2] https://rstudio.github.io/learnr/exercises.html#Hints_and_Solutions

[3] https://rstudio.github.io/learnr/exercises.html#Time_Limits

le code comme son résultat peuvent être comparés à une valeur attendue[4].

Des quizz[5] peuvent être ajoutés, sous la forme de questionnaires à choix multiples ou uniques.

La progression de l'utilisateur dans le tutoriel (code saisi, réponses aux questions…) est sauvegardée par **learnr** sur le poste de travail. Un tutoriel peut être arrêté puis repris sans perte de données. En revanche, il n'y a pas de moyen simple de récupérer ces données pour une évaluation par le formateur par exemple.

Diffusion

Les tutoriels peuvent être diffusés en copiant les fichiers ou en indiquant aux utilisateurs de cloner les projets GitHub qui les contiennent.

Ils peuvent aussi être hébergés sur Shinyapps.io (voir section 7.3).

Enfin, ils peuvent être inclus dans un package[6].

[4] https://rstudio.github.io/learnr/exercises.html#Exercise_Checking

[5] https://rstudio.github.io/learnr/questions.html

[6] https://rstudio.github.io/learnr/publishing.html#R_Package

8.2 GitHub Classrooms

GitHub Classrooms permet de diffuser à un public étudiant des dépôts GitHub à modifier et de contrôler le résultat. Les applications sont aussi bien l'apprentissage de R que la production de documents, pour un travail personnel ou un examen par exemple.

Inscription

Pour commencer à utiliser l'outil, il faut ouvrir un compte. Sur le site de GitHub Classrooms[7], cliquer sur "Sign in" et utiliser son compte GitHub pour s'authentifier.

[7] https://classroom.github.com/

Organisations

L'étape suivant consiste à créer une organisation GitHub. Une organisation GitHub contient essentiellement des membres (titulaires d'un compte GitHub) et des dépôts accessibles à l'adresse https://github.com/Organisation/Depot.

La façon la plus simple de travailler consiste à créer une organisation par cours mais d'autres approches sont possibles dans des structures utilisant intensivement l'outil. L'organisation crée pour l'exemple est ici "Cours-R"[8].

Une adresse de messagerie est nécessaire (utiliser la même que celle de son compte GitHub) et l'organisation doit être déclarée comme appartenant à son compte personnel.

Si l'organisation n'est pas visible sur la page de GitHub Classrooms, cliquer sur "Grant us access".

[8] https://github.com/Cours-R

Nouvelle salle de classe

Une salle de classe (*classroom*) est peuplée d'étudiants qui recevront des tâches(*assignments*) à exécuter.

Cliquer sur *New Classroom*. Sélectionner l'organisation en charge de l'administration de la salle de classe.

Saisir le nom de la salle de classe : une bonne pratique est de la préfixer par le nom du cours et d'ajouter le nom de la session, par exemple "Cours-R-2020-EdGuyane".

Ne pas ajouter de collaborateurs (ce sera possible plus tard), et saisir éventuellement la liste des étudiants (un nom par ligne, possible plus tard aussi). La classe est créée.

Toutes les salles de classe sont visibles depuis la page d'accueil de GitHub Classrooms[9]. Cliquer sur un nom pour en ouvrir une. Le bouton *Settings* permet de changer son nom ou de la supprimer. Le bouton *TAs and Admins* permet d'ajouter des collaborateurs, c'est-à-dire d'autres utilisateurs GitHub qui pourront administrer la salle de classe.

[9] https://classroom.github.com/classrooms

Le bouton *Students* permet d'ajouter des étudiants. La liste de nom est libre, sans format obligatoire. Cliquer sur *Create Roster* pour l'activer. Les noms doivent ensuite être liés à des comptes GitHub : ce travail peut être fait par l'administrateur ou par les étudiants eux-mêmes quand ils recevront la première tâche à effectuer. Chaque étudiant doit avoir un compte sur GitHub.

Préparer un modèle de dépôt

Une tâche est un dépôt GitHub à modifier. Par exemple[10], créer un dépôt contenant un projet R avec un fichier Markdown décrivant le travail à faire et éventuellement une partie du code nécessaire pour y parvenir, les autres fichiers du modèle R Markdown utilisé et un fichier de données. Si le modèle est par exemple Memo (section 4.3), le script `GitHubPages.R` est fourni.

[10] https://github.com/EricMarcon/Cours-R-Memo/settings

Ouvrir les propriétés du dépôt sur GitHub et cocher la case *Template Repository* pour en faire un modèle.

Assigner une tâche

Ouvrir une salle de classe et cliquer sur *New Assignment*.

Saisir un titre explicite pour les étudiants, une date limite optionnelle et choisir *Individual Assignment".

Par défaut, le nom de la tâche sert de préfixe pour le nom des dépôts des étudiants mais il peut être remplacé par un préfixe choisi. Quand les étudiants rendront leur travail, tous les dépôts de toutes les tâches seront stockés dans l'organisation.

Le dépôt crée sur le compte de chaque étudiant peut être privé ou public, selon que l'on souhaite que les étudiants puissent voir le travail des autres ou non. Donner le droit d'administration et

rendre le site public si les étudiants doivent pouvoir activer les pages GitHub pour présenter le résultat de leur travail. Cliquer sur *Continue*.

Sélectionner le dépôt modèle (*starter code*) puis cliquer sur *Continue* puis *Create Assignment*.

La nouvelle tâche est créée. Elle est associée à un lien d'invitation qu'il faut copier et envoyer aux étudiants. Quand ils cliqueront sur le lien, ils atteindront une page GitHub qui leur permettra d'associer leur compte à un nom de la liste (aucun contrôle n'est possible : le premier connecté peut s'associer à n'importe quel nom). Ils pourront ensuite créer un nouveau projet RStudio à partir du dépôt GitHub créé automatiquement par GitHub Classrooms, modifier ce projet selon les consignes de travail et le pousser sur GitHub. Le dépôt se trouve sur le compte de l'organisation à laquelle la classe est reliée, et est suffixé par l'identifiant GitHub de l'étudiant.

Contrôler le travail des étudiants

Il est possible d'afficher chaque dépôt crée par les étudiants à partir de la page de la tâche sur GitHub Classrooms. Si le travail à produire est un document rédigé, demander aux étudiants de le placer dans les pages GitHub du dépôt pour le lire directement en ligne.

L'assistant GitHub Classrooms[11] permet de télécharger en une fois tous les dépôts des étudiants pour les corriger sur son poste de travail.

[11] https://classroom.github.com/assistant

Chapitre 9

Conclusion

L'environnement de travail de R et RStudio permet de produire tous types de documents avec un langage unique.

L'objectif de reproductibilité des résultats est atteint en intégrant les traitements statistiques et la rédaction. Le travail collaboratif est permis par l'utilisation systématique du contrôle de source et de GitHub. La présentation des résultats est assurée par les pages GitHub et des modèles de documents couvrant la majorité des besoins.

Pour les pauses, R fournit même quelques jeux dans le package **fun**, dont le célèbre démineur :

```
# Installation du package
install.packages("fun")
# Ouverture d'une fenêtre X et exécution
if (interactive()) {
    if (.Platform$OS.type == "windows")
        x11() else x11(type = "Xlib")
    fun::mine_sweeper()
}
```

Ce document n'a pas pour objectif d'être exhaustif sur les possibilités de R mais plutôt de présenter une méthode de travail et des moyens simples de l'appliquer rapidement. On se reportera aux ouvrages plus détaillés cités dans le texte pour approfondir tel ou tel point.

Ce document est mis à jour régulièrement en fonction de l'évolution des outils disponibles.

Bibliographie

Barnier, J. (2020). *Introduction à R et au tidyverse* (cf. p. 11).

Gandrud, C. (2015). *Reproducible Research with R and R Studio*. 2e éd. Chapman et Hall/CRC (cf. p. ix).

Gillespie, C. et R. Lovelace (2016). *Efficient R Programming*. O'Reilly Media (cf. p. 11).

Wickham, H. (2010). « A Layered Grammar of Graphics ». In : *Journal of Computational and Graphical Statistics* 19.1, p. 3-28 (cf. p. 17).

— (2014). *Advanced R*. Chapman et Hall/CRC (cf. p. 11).

— (2015). *R Packages*. 1st. O'Reilly Media, Inc. (cf. p. 93).

— (2017). *ggplot2 : Elegant Graphics for Data Analysis*. 2nd. Springer. doi : 10.1007/978-3-319-24277-4 (cf. p. 18).

Wickham, H. et G. Grolemund (2016). *R for Data Science*. O'Reilly Media (cf. p. 11, 18).

Xie, Y. (2015). *Dynamic Documents with R and knitr*. 2nd. Boca Raton, Florida : Chapman et Hall/CRC (cf. p. 63).

Xie, Y., J. Allaire et G. Grolemund (2018). *R Markdown : The Definitive Guide*. Boca Raton, Florida : Chapman et Hall/CRC (cf. p. 64).

Résumé : Cet ouvrage propose une organisation du travail autour de R et RStudio pour, au-delà des statistiques, rédiger des documents efficacement avec R Markdown, aux formats variés (mémos, articles scientifiques, mémoires d'étudiants, livres, diaporamas), créer son site web et des applications R en ligne (Shiny), produire des packages et utiliser R pour l'enseignement.

Unité Mixte de Recherche
ECOlogie
des
FOrêts
de
Guyane