

# Développement de packages R

*Sophie Baillargeon, Université Laval*

*2018-03-25*

## Table des matières

<b>Étape 1. Écrire les fonctions</b>	<b>1</b>
<b>Étape 2. Créer la bonne structure de fichiers</b>	<b>2</b>
Sous-dossier nommé <code>R</code>	3
Sous-dossier nommé <code>man</code>	3
Fichier nommé <code>DESCRIPTION</code>	3
Fichier nommé <code>NAMESPACE</code>	4
Autres fichiers et dossiers	4
<b>Étape 3. Écrire la documentation des fonctions et du package</b>	<b>4</b>
Comment écrire de la documentation avec <code>roxygen2</code> ?	5
Comment générer les fichiers <code>.Rd</code> et le fichier <code>NAMESPACE</code> ?	9
<b>Étape 4. Construire et vérifier le package</b>	<b>13</b>
Lancement des commandes de construction et de vérification en RStudio	14
Sous-étape a) Créer un projet RStudio avec notre package	14
Sous-étape b) Configurer les options de RStudio	14
Sous-étape c) Construire à partir du menu « Build » de RStudio	15
Lancement des commandes de construction et de vérification avec <code>devtools</code>	15
<b>Étape 5. Si désiré, partager le package</b>	<b>15</b>
<b>Étape 6. Au besoin, mettre à jour le package</b>	<b>16</b>
<b>Références</b>	<b>17</b>

---

Pour créer un package R, il faut d'abord développer une structure de fichiers similaires à ce qui se retrouve dans n'importe quel *package source*. Pour voir de quoi à l'air une telle structure de fichier, il suffit de :

- télécharger un *package source* à partir du CRAN (par exemple `ggplot2_2.2.1.tar.gz` sur <http://CRAN.R-project.org/package=ggplot2>),
- décompresser le fichier (il y a 2 étapes de décompression à effectuer).

L'exemple utilisé ici, soit le package `ggplot2`, est un gros package. Seuls certains de ses fichiers et dossiers sont obligatoires dans tout package.

Après avoir développé les fichiers sources, qui incluent du code et de la documentation, il faut construire le package. Nous verrons ici comment réaliser cette construction avec RStudio.

## Étape 1. Écrire les fonctions

Cette étape fait appel à ce que nous avons appris dans toutes les notes de cours précédentes, en particulier celles sur les [fonctions en R](#).

Lorsque nous créons des fonctions dans le but de les inclure dans un package, il faut garder en tête le chemin de recherche complet qu'utilisera R lors de l'exécution de ces fonctions. Ce chemin est le suivant :

1. l'environnement local d'exécution de la fonction,
2. l'environnement englobant de la fonction, soit l'espace de noms du package,
3. l'environnement des objets importés par le package,
4. l'environnement du package R de base,
5. l'environnement de travail,
6. les environnements de tous les packages chargés.

Nous savons déjà que, dans ce chemin de recherche, nous ne pouvons pas nous fier au contenu de l'environnement de travail, car il varie constamment en cours de session. Nous ne pouvons pas non plus nous fier aux packages chargés. En effet, la liste des packages chargés varie aussi en cours de session et d'un utilisateur à l'autre. Le bon fonctionnement des fonctions d'un package devrait dépendre d'un seul appel à la fonction `library`, celui servant à charger le package en question.

Ainsi, dans le corps des fonctions d'un package, nous pouvons toujours faire appel

- aux arguments et à des variables locales,
- à d'autres objets dans le package (publics ou privés) et
- à des objets du package R de base, qui est toujours inclus dans le chemin de recherche.

Si nous souhaitons faire appel à des objets autres, par exemple des fonctions provenant d'autres packages, il faudra faire le nécessaire pour inclure ces fonctions dans *l'environnement des objets importés par le package*.

Aussi, si notre but est éventuellement de rendre notre package public sur le CRAN, il faut respecter les politiques du CRAN : <http://cran.r-project.org/web/packages/policies.html>. La plus importante de ces politiques est que notre package doit passer le R CMD check `--as-cran` sans erreurs ni avertissements (<https://cran.r-project.org/doc/manuals/r-release/R-exts.html#Checking-packages>, nous en reparlons plus loin). Pour passer cette vérification sans problèmes, notre code ne devrait pas, entre autres :

- contenir des symboles non-ASCII (donc pas d'accents) :  
*afin de s'assurer que le code est fonctionnel sur n'importe quelle plateforme informatique ;*
- utiliser d'association partielle d'argument (donc nous devons utiliser le nom complet des arguments dans les appels de fonction) :  
*afin de s'assurer que le code demeure fonctionnel si des arguments sont ajoutés aux définitions des fonctions appelées dans le code (ces nouveaux arguments pourraient porter des noms qui entrent en conflit avec l'association partielle) ;*
- toujours utiliser comme valeurs logiques `TRUE` ou `FALSE` (donc pas de `T` ou `F`) :  
*parce qu'un objet R nommé `T` ou `F` peut être défini, écrasant du coup les définitions `T <- TRUE` et `F <- FALSE`, alors qu'il est impossible de créer un nouvel objet R nommé `TRUE` ou `FALSE` (mots-clés protégés).*

## Étape 2. Créer la bonne structure de fichiers

Un package, dans sa version source, est simplement un répertoire de fichiers compressés. Cependant, ce répertoire doit comprendre des fichiers et des sous-répertoires portant des noms précis.

Nous pouvons créer cette structure de fichiers à l'aide de :

- la fonction `package.skeleton` du package `utils` ou
- la fonction `create` ou `setup` du package `devtools` ou
- les fonctionnalités de création de projets RStudio.

Cependant, nous pouvons aussi créer manuellement les dossiers et fichiers.

Peu importe la procédure utilisée, il faut d'abord avoir un dossier portant le nom du package.

Par exemple, si nous voulions créer un package nommé `distManhattan`, il faudrait d'abord créer un dossier portant ce nom sur notre ordinateur. Ce pourrait être par exemple le dossier `"C:/coursR/distManhattan"`.

Voici ce que doit contenir obligatoirement ce dossier :

- sous-dossier nommé `R`,
- sous-dossier nommé `man`,
- fichier nommé `DESCRIPTION`,
- fichier nommé `NAMESPACE`.

## Sous-dossier nommé `R`

Ce sous-dossier doit comprendre des fichiers de scripts `R` (portant l'extension `.R`) contenant le code de création des fonctions ainsi que les commentaires `roxygen2`, si c'est l'outil que nous utilisons pour générer la documentation (nous y reviendrons plus loin). Le développeur peut choisir les noms qu'il veut pour ces fichiers et il peut y répartir le code source comme il le veut. Évidemment, une bonne pratique est de répartir le code source de façon à ce que ce soit facile de s'y retrouver. Ainsi, les stratégies extrêmes « toutes les fonctions dans un seul fichier » et, à l'opposé, « un fichier par fonction » représentent rarement une bonne répartition du code, à moins que les fonctions soient peu nombreuses.

## Sous-dossier nommé `man`

Ce sous-dossier doit comprendre des fichiers sources de fiches d'aide `R` (portant l'extension `.Rd`). Nous allons voir à la prochaine étape comment générer ces fichiers à l'aide du package `roxygen2`.

## Fichier nommé `DESCRIPTION`

Il s'agit d'un fichier très important pour la construction du package. C'est un fichier court, comportant de l'information de base sur le package. Cependant, ce fichier doit respecter une syntaxe précise.

### Exemple :

```
Package: distManhattan
Version: 1.0.0
Date: 2018-03-25
License: GPL-3
Title: Distance de Manhattan
Description: Calcul de la distance de
            Manhattan entre deux points
Author: Sophie Baillargeon
Maintainer: Sophie Baillargeon
            <sophie.baillargeon@mat.ulaval.ca>
Imports: stats
LazyData: true
```

Ce fichier contient des champs nommés. Un champ débute par son nom (première lettre toujours majuscule), immédiatement suivi d'un deux-points et d'un espace. Vient ensuite la valeur fournie à ce champ. Les champs obligatoires sont les suivants : `Package`, `Version`, `License`, `Title`, `Description`, `Author`, et `Maintainer`. Les valeurs données aux champs peuvent s'étendre sur plus d'une ligne.

Voici quelques informations supplémentaires sur quelques-uns des champs obligatoires :

- `Package` : le nom du package fourni dans ce champ doit correspondre au nom du dossier contenant les fichiers sources du package,

- **Version** : il s'agit d'une séquence d'au minimum 2 (souvent 3) nombres entiers non négatifs séparés par un seul caractère . ou -,
- **Maintainer** : ce champ devrait contenir le nom de la personne à contacter pour toute question ou tout problème à rapporter concernant le package, suivi d'une adresse courriel valide placée entre les caractères < et >.

L'information complète et officielle concernant ce fichier peut être trouvée sur la page web suivante :

<http://cran.r-project.org/doc/manuals/r-release/R-exts.html#The-DESCRIPTION-file>

## Fichier nommé NAMESPACE

C'est un fichier qui permet de définir quels objets sont accessibles dans un package, c'est-à-dire exportés de l'espace de noms du package. Ce fichier permet aussi d'identifier quelles fonctions provenant d'autres packages sont utilisées dans le package, donc de définir le contenu de l'environnement des objets importés par le package. Nous allons voir comment utiliser **roxygen2** pour générer ce fichier.

## Autres fichiers et dossiers

Un package peut contenir plusieurs autres fichiers et sous-dossiers. Cependant, ces derniers ne sont pas obligatoires.

Autres dossiers parfois nécessaires :

- Sous-dossier nommé **data** : Si le package contient des jeux de données, ils se trouvent habituellement dans ce dossier. Ceux-ci sont typiquement dans un format de données R (**.rda**, **.RData** ou **.rds**).
- Sous-dossier nommé **src** : Si le package contient du code C, C++ ou Fortran, il doit se trouver dans ce dossier.
- Sous-dossier nommé **vignettes** : Si le package contient de la documentation autre que les fiches d'aide (par exemple un guide d'utilisateur), il devrait idéalement se trouver dans ce dossier.
- etc. : <http://cran.r-project.org/doc/manuals/r-release/R-exts.html#Package-subdirectories>

Autres fichiers utiles :

- Fichier nommé **NEWS** : Ce fichier décrit les modifications apportées à un package lors d'une mise à jour. Sa mise en forme n'est pas vraiment importante.
- etc. : <http://cran.r-project.org/doc/manuals/r-release/R-exts.html#Package-structure>

## Étape 3. Écrire la documentation des fonctions et du package

Documenter ses fonctions est en fait une étape intégrée au développement des fonctions. Cependant, il faut maintenant écrire la documentation dans un format qui produira correctement les fiches d'aide qui doivent être incluses dans le package.

Ces fiches d'aide proviennent en fait de fichiers portant l'extension **.Rd**. Cependant, nous ne verrons pas comment éditer directement ces fichiers. Nous allons plutôt apprendre à faire ça en utilisant **roxygen2**. Le package **roxygen2** est un outil qui permet de générer des fichiers **.Rd**, ainsi que le fichier **NAMESPACE**, de façon automatique, à partir de commentaires intégrés au code.

L'utilisation de **roxygen2** comporte les avantages suivants :

- la documentation se situe dans le même fichier que le code, ce qui aide à se rappeler que la documentation doit être mise à jour si le code est modifié,
- la syntaxe **roxygen2** est un peu plus simple que la syntaxe des fichiers **.Rd**.

## Rappelons que dans un package il faut obligatoirement documenter :

- Les fonctions publiques :

Il est essentiel de faire des fiches d'aide pour ces fonctions, afin que tout utilisateur comprenne comment appeler correctement la fonction. Si un utilisateur a mal compris la documentation et fourni par erreur une valeur d'argument invalide en entrée, il est aussi souhaitable que la fonction retourne une erreur informative. Le code de ces fonctions publiques comporte donc typiquement de la **validation des arguments fournis en entrée**.

À l'opposé, les **fonctions privées ou internes** ne sont pas conçues pour être appelées par n'importe quel utilisateur. Ces fonctions ne sont pas exportées de l'espace de noms et elles ne sont pas documentées officiellement. Il est tout de même bon de documenter minimalement ces fonctions pour nous-mêmes, mais nous n'avons pas à produire de fiches d'aide pour elles. De plus, pour ne pas alourdir ces fonctions, elles comportent typiquement peu ou pas de validation d'arguments.

- Les jeux de données dans le répertoire **data** :

Chaque jeu de données dans le répertoire **data** d'un package doit être décrit dans une fiche d'aide pour expliquer son contenu.

- Les classes et méthodes S4 publiques :

Les packages exploitant le système de programmation orientée objet S4 doivent fournir des fiches d'aide pour les classes et méthodes S4 publiques.

<https://cran.r-project.org/doc/manuals/r-release/R-exts.html#Documenting-S4-classes-and-methods>

## Il est aussi recommandé de documenter :

- Les méthodes S3 pour des fonctions génériques :

Nous pouvons leur faire une fiche d'aide indépendante ou encore les documenter dans la même fiche d'aide que la fonction qui crée les objets de la classe en question.

- Le package lui-même :

Il est utile de créer une fiche d'aide présentant le package.

## Comment écrire de la documentation avec **roxygen2** ?

Il suffit d'insérer des « commentaires » **roxygen2** dans le code source des fonctions, donc dans les scripts situés dans le sous-dossier **R**. Un commentaire **roxygen2** débute par **#'**, ce qui le distingue d'un commentaire ordinaire, qui débute par **#**.

## Exemples :

D'abord, voici des exemples de scripts R contenant des commentaires **roxygen2**. Des explications se retrouvent après les exemples. Ces exemples poursuivent l'exemple de création d'une fonction qui calcule la distance de Manhattan entre deux points provenant des notes [Tests et exceptions en R](#).

```
#' Distance de Manhattan
#
#' Calcule la distance de Manhattan entre deux points
#
#' @param point1 Un vecteur numerique des coordonnees du premier point.
#' @param point2 Un vecteur numerique des coordonnees du deuxieme point.
#' @return une seule valeur : la distance de Manhattan entre
#'          \code{point1} et \code{point2}
#' @author Sophie Baillargeon
#' @export
#' @examples
#' Manhattan(point1 = c(0,-5), point2 = c(0,-15))
Manhattan <- function(point1, point2) {
  # Validation des arguments
  if (length(point1) != length(point2))
    stop("'point1' and 'point2' must have the same length")
  if (!is.null(dim(point1)) || !is.null(dim(point2)))
    warning("'point1' and 'point2' are treated as dimension 1 vectors")
  # Calculs
  out <- list(dist = sum(abs(point1 - point2)))
  # Sortie
  class(out) <- "Manhattan"
  return(out)
}
```

```

#' Distance de Manhattan
#'
#' Calcule la distance de Manhattan entre deux points
#'
#' Utilise la fonction \link[stats]{dist} du package \pkg{stats}.
#'
#' @param point1 Un vecteur numerique des coordonnees du premier point.
#' @param point2 Un vecteur numerique des coordonnees du deuxieme point.
#' @return \item{dist}{ la distance de Manhattan entre \code{point1} et \code{point2} }
#' @return \item{call}{ une copie de l'appel de la fonction }
#' @author Sophie Baillargeon
#' @export
#' @importFrom stats dist
#' @examples
#' Manhattan2(point1 = c(0,-5), point2 = c(0,-15))
Manhattan2 <- function(point1, point2) {
  call <- match.call()
  dist <- dist(rbind(point1, point2), method = "manhattan")
  out <- list(dist = as.vector(dist), call = call)
  class(out) <- "Manhattan"
  return(out)
}

```

```
#' @rdname Manhattan
#' @param x Un objet produit par la fonction \code{Manhattan}, a afficher.
#' @param \dots D'autres arguments passes a d'autres methodes.
#' @export
print.Manhattan <- function(x, ...) {
  cat("Manhattan distance between 'point1' and 'point2' :", x$dist)
}
```

```
#' Distance de Manhattan
#
#' Distance de Manhattan entre deux points
#
#' \tabular{ll}{
#' Package: \tab distManhattan\cr
#' Type: \tab Package\cr
#' Version: \tab 1.0.0\cr
#' Date: \tab 2018-03-25\cr
#' License: \tab GPL-3\cr
#' }
#
#' @name distManhattan
#' @docType package
#' @author Sophie Baillargeon
#
#' Maintainer: Sophie Baillargeon <sophie.baillargeon@mat.ulaval.ca>
#' @keywords package
NULL

#' Points aleatoires
#
#' Coordonnes en deux dimensions de 10 points aleatoires.
#
#' @format Une matrice contenant 10 points designes par les
#'         coordonnees suivantes.
#' \describe{
#'   \item{X}{ coordonnee en X }
#'   \item{Y}{ coordonnee en Y }
#' }
#' @keywords datasets
#' @examples
#' Manhattan(point1 = points[4, ], point2 = points[8, ])
"points"
```

## Explications :

Pour les fonctions et les méthodes associées à des fonctions génériques, il suffit de mettre en entête au code source les commentaires **roxygen2** qui généreront la documentation. Pour le package et les jeux de données, il faut ajouter un fichier d'extension **.R** dans le dossier **R**. Ce fichier doit contenir les commentaires **roxygen2** pour documenter globalement le package, suivi de l'instruction **NULL**, et les commentaires pour documenter les jeux de données, chaque bloc suivi du nom du jeu de données sous forme de chaîne de caractères (par exemple **"points"**) .

La première phrase de ces commentaires doit être le **titre** de la fiche d'aide. Ce titre peut optionnellement être précédé du tag **@title**.

Cette phrase doit être suivie d'une ligne vide, puis du texte à mettre dans la section **Description**. Ce paragraphe peut optionnellement être précédé du tag **@description**.

Si nous souhaitons avoir une section **Details**, il faut mettre à la suite de ce texte une ligne vide. Tout le texte après cette ligne vide, mais avant les lignes débutant par un tag **roxygen2** formera la section **Details**. Ces paragraphes peuvent optionnellement être précédés du tag **@details**.

Ainsi, les deux documentations suivantes sont équivalentes :

```
#' Distance de Manhattan
#'
#' Calcule la distance de Manhattan entre deux points
#'
#' Utilise la fonction \code{\link[stats]{dist}} du package \pkg{stats}.
#'
#' @param point1 Un vecteur numerique des coordonnees du premier point.
#' @param point2 Un vecteur numerique des coordonnees du deuxieme point.
#' @return \item{dist}{ la distance de Manhattan entre \code{point1} et \code{point2} }
#' @return \item{call}{ une copie de l'appel de la fonction }
#' @author Sophie Baillargeon
#' @export
#' @importFrom stats dist
#' @examples
#' Manhattan2(point1 = c(0,-5), point2 = c(0,-15))
Manhattan2 <- function(point1, point2) {
  ... (code omis ici)
}
```

et

```
#' @title Distance de Manhattan
#' @description Calcule la distance de Manhattan entre deux points
#' @details Utilise la fonction \code{\link[stats]{dist}} du package \pkg{stats}.
#' @param point1 Un vecteur numerique des coordonnees du premier point.
#' @param point2 Un vecteur numerique des coordonnees du deuxieme point.
#' @return \item{dist}{ la distance de Manhattan entre \code{point1} et \code{point2} }
#' @return \item{call}{ une copie de l'appel de la fonction }
#' @author Sophie Baillargeon
#' @export
#' @importFrom stats dist
#' @examples
#' Manhattan2(point1 = c(0,-5), point2 = c(0,-15))
Manhattan2 <- function(point1, point2) {
  ... (code omis ici)
}
```



Les sections qui suivent le titre et la description, qui sont obligatoires, et les informations détaillées, s'il y lieu (cette section est optionnelle), débutent toutes par des tags. Il faut obligatoirement décrire, si la fonction en possède :

- les arguments en entrée avec le tag `@param` :  
il faut une description par argument, de la forme `@param nomArgument description`, où la description peut s'étendre sur plusieurs lignes ;
- la sortie avec le tag `@return` :  
si la fonction retourne une liste, il est recommandé d'avoir une description par élément de la liste, de la forme `@return \item{nomElementDeLaListe}{description}`, où la description peut encore une fois s'étendre sur plusieurs lignes.

Il est aussi essentiel de mettre les tags pour l'écriture du `NAMESPACE` :

- le tag `@export` exporte une fonction du `NAMESPACE` (donc la rend publique),
- le tag `@importFrom` assure l'importation des fonctions provenant d'autres packages qui sont utilisées dans le code de notre package.

Nous pouvons ajouter les informations suivantes :

- les noms des auteurs avec le tag `@author`,
- des références avec le tag `@references`,
- des liens vers les fiches d'aide de fonction en lien avec la fonction documentée avec le tag `@seealso`,
- etc. : voir les vignettes du package `roxygen2` <https://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html>.

Pour jumeler des fiches d'aide, il faut utiliser le tag `@rdname`. Dans l'exemple, la fonction `Manhattan` et la méthode `print.Manhattan` sont documentées dans la même fiche d'aide grâce à ce tag.

Nous pouvons contrôler le nom de la fiche d'aide avec le tag `@name`. Sans ce tag, la fiche d'aide porte le nom de la fonction ou le nom de jeu de données qui suit le bloc de documentation `roxygen2`. Pour une documentation de package, c'est la l'instruction `NULL` qui suit le bloc de documentation. Il faut donc obligatoirement, dans ce cas, un tag `@name`. Il faut aussi spécifier le type de documentation avec le tag `@docType`.

Il est recommandé de toujours mettre des exemples dans une fiche d'aide. Dans la documentation `roxygen2`, ceux-ci doivent être ajoutés à la fin du bloc de documentation, après le tag `@examples`.

Dans les commentaires `roxygen2`, il est possible de formater du texte en utilisant les tags de mises en forme acceptés dans les fichiers `.Rd` (par exemple `\code{}`, `\pkg{}`, `\item{}`, `\link{}`, `\dots`, etc. <https://cran.r-project.org/web/packages/roxygen2/vignettes/formatting.html>).

## Comment générer les fichiers `.Rd` et le fichier `NAMESPACE` ?

Pour générer les fichiers `.Rd` et le fichier `NAMESPACE`, il suffit de lancer la commande `roxygenize` du package `roxygen2`. La commande peut être lancée directement dans la console. Par exemple

```
roxygenize("C:/coursR/distManhattan")
```

Elle peut aussi être lancée par l'intermédiaire d'un menu de RStudio (voir prochaine section).

Dans l'exemple présenté précédemment, les commentaires `roxygen2` des fichiers `"Manhattan.R"`, `"Manhattan2.R"`, `"print.Manhattan.R"` et `"distManhattan.R"`, du sous-dossier `"C:/coursR/distManhattan/R"`, ont produit les fichiers suivants :

Fichier "C:/coursR/distManhattan/NAMESPACE" :

---

```
S3method(print,Manhattan)
export(Manhattan)
export(Manhattan2)
importFrom(stats,dist)
```

---

Fichier "C:/coursR/distManhattan/man/Manhattan.Rd" :

---

```
\name{Manhattan}
\alias{Manhattan}
\alias{print.Manhattan}
\title{Distance de Manhattan}
\usage{
Manhattan(point1, point2)

\method{print}{Manhattan}(x, ...)
}
\arguments{
\item{point1}{Un vecteur numerique des coordonnees du premier point.}

\item{point2}{Un vecteur numerique des coordonnees du deuxieme point.}

\item{x}{Un objet produit par la fonction \code{Manhattan}, a afficher.}

\item{\dots}{D'autres arguments passes a d'autres methodes.}
}
\value{
une seule valeur : la distance de Manhattan entre
\code{point1} et \code{point2}
}
\description{
Calcule la distance de Manhattan entre deux points
}
\examples{
Manhattan(point1 = c(0,-5), point2 = c(0,-15))
}
\author{
Sophie Baillargeon
}
```

---

Fichier "C:/coursR/distManhattan/ma/Manhattan2.Rd" :

---

```
\name{Manhattan2}
\alias{Manhattan2}
\title{Distance de Manhattan}
\usage{
Manhattan2(point1, point2)
}
\arguments{
\item{point1}{Un vecteur numerique des coordonnees du premier point.}

\item{point2}{Un vecteur numerique des coordonnees du deuxieme point.}
}
\value{
\item{dist}{ la distance de Manhattan entre \code{point1} et \code{point2} }

\item{call}{ une copie de l'appel de la fonction }
}
\description{
Calcule la distance de Manhattan entre deux points
}
\details{
Utilise la fonction \code{\link[stats]{dist}} du package \code{stats}.
}
\examples{
Manhattan2(point1 = c(0,-5), point2 = c(0,-15))
}
\author{
Sophie Baillargeon
}
```

---

Fichier "C:/R/coursR/distManhattan/man/distManhattan.Rd" :

---

```
\docType{package}
\name{distManhattan}
\alias{distManhattan}
\alias{distManhattan-package}
\title{Distance de Manhattan}
\description{
Distance de Manhattan entre deux points
}
\details{
\table{ll}{
Package: \tab distManhattan\cr
Type: \tab Package\cr
Version: \tab 1.0.0\cr
Date: \tab 2018-03-25\cr
License: \tab GPL-3\cr
}
}
\author{
Sophie Baillargeon

Maintainer: Sophie Baillargeon <sophie.baillargeon@mat.ulaval.ca>
}
\keyword{package}
```

---

Fichier "C:/coursR/distManhattan/man/points.Rd" :

---

```
\docType{data}
\name{points}
\alias{points}
\title{Points aleatoires}
\format{Une matrice contenant 10 points designes par les
        coordonnees suivantes.}
\describe{
  \item{X}{ coordonnee en X }
  \item{Y}{ coordonnee en Y }
}
\usage{
points
}
\description{
Coordonnees en deux dimensions de 10 points aleatoires.
}
\examples{
Manhattan(point1 = points[4, ], point2 = points[8, ])
}
\keyword{datasets}
```

---

Ces fichiers utilisent une syntaxe inspirée de la syntaxe de LaTeX. J'ai inclus ici ces fichiers seulement pour vous montrer de quoi ils avaient l'air. Lorsque nous utilisons **roxygen2**, nous n'avons pas besoin d'éditer directement ces fichiers, donc nous n'avons pas besoin de comprendre leur syntaxe. La syntaxe de **roxygen2** est plus simple.

## Étape 4. Construire et vérifier le package

Une fois les étapes 1 à 3 complétées, nous sommes prêts à construire le fichier compressé du package à partir du dossier contenant les fichiers source. En fait, nous allons aussi à cette étape vérifier le dossier afin de s'assurer qu'il produira un package conforme.

Ces tâches se réalisent avec les utilitaires :

- R CMD `check`,
- R CMD `build`,
- R CMD `INSTALL`.

Ces utilitaires s'exploitent par des commandes soumises dans le *terminal* sous Unix / Linux ou Mac OS X / OS X / macOS et dans une fenêtre *invite de commandes* sous Windows. Dans ces notes, nous verrons comment utiliser un outil qui soumet ces commandes à notre place, sans que nous aillions besoin d'ouvrir le terminal ou la fenêtre d'invite de commande.

Les utilitaires R CMD `check`, `build` et `INSTALL` sont inclus dans l'installation de base de R (dans le package `utils`). Cependant, ils nécessitent des outils supplémentaires pour fonctionner : les outils de développement de logiciel GNU, incluant un compilateur C/C++. Voici comment s'assurer que ces outils sont installés sur notre ordinateur, selon son système d'exploitation (plus de détails sur la page <https://support.rstudio.com/hc/en-us/articles/200486498-Package-Development-Prerequisites>) :

- Windows : il faut installer les « Rtools », téléchargeable sur la page web <https://cran.r-project.org/bin/windows/Rtools/> (plus de détails dans le [Guide d'installation ou de mise à jour de R et RStudio](#)) ;
- Mac OS X / OS X / macOS : il faut installer les « Apple Xcode developer tools », disponible gratuitement sur le « App Store », s'ils ne sont pas déjà installés (souvent installés par défaut) ;
- Unix / Linux : il faut s'assurer d'avoir installé R accompagné des ses « development tools » (`r-base-dev`).

Le package `devtools` comporte une fonction pour tester si tout le nécessaire au développement de package est installé et fonctionne correctement (source : <http://r-pkgs.had.co.nz/intro.html#intro-get>).

```
library(devtools)
has_devel()
```

Si la sortie générée par cet appel à la fonction `has_devel` se termine par `TRUE`, alors tout est fonctionnel.

De plus, une des commandes R pour le développement de packages, soit R CMD `check`, a besoin d'une installation de LaTeX pour tester la création de la documentation des packages en format PDF. Pour développer des packages, vous avez donc le choix entre :

- installer LaTeX sur votre ordinateur :
  - une version gratuite pour Windows est MiKTeX (<https://miktex.org/download>),
  - une version gratuite pour Mac OS X / OS X / macOS est MacTeX (<http://www.tug.org/mactex/>),
  - les systèmes Unix / Linux viennent habituellement par défaut avec une distribution de LaTeX ;
- omettre la création de la documentation PDF lors de la soumission de la commande R CMD `check` grâce à l'option `--no-manual` (plus d'informations à venir en temps opportun).

## Lancement des commandes de construction et de vérification en RStudio

**RStudio** peut [lancer pour nous les commandes de construction et de vérification de packages](#). Le logiciel rend le processus vraiment plus simple que de lancer les commandes manuellement dans le *terminal* ou l'*invite de commandes*. Nous apprendrons donc seulement comment construire et vérifier un package avec **RStudio**.

### Sous-étape a) Créer un projet RStudio avec notre package

Il faut tout d'abord créer un projet RStudio avec notre package. Pour ce faire, nous pouvons procéder comme suit :

- ouvrir le menu « File » et sélectionner « New Project... » (il y a aussi un bouton dans la barre de RStudio en haut à droite, nommé à l'origine « Project : (None) » qui ouvre un menu contenant aussi l'élément « New Project... »),
- sélectionner « Existing Directory »,
- sélectionner le dossier portant le nom du package et contenant les fichiers sources,
- cliquer sur « Create Project ».

Le projet sera créé et ouvert. Ça ajoute des fichiers dans le dossier de notre package. Nous ne nous préoccupons pas de ces fichiers. Ils sont automatiquement ignorés lors de la construction du package avec RStudio.

**Note :** Si nous sélectionnons « New Directory » plutôt que « Existing Directory », puis « R package », RStudio crée un squelette de dossier de fichiers source de package.

### Sous-étape b) Configurer les options de RStudio

Voici quelques configurations de RStudio que je conseille d'utiliser.

#### Options globales :

(à modifier une seule fois)

- par le menu « Tools > Global Options... »,
- dans « Packages », décocher « Cleanup output after successful R CMD check ».

Les dossiers générés par la commande **R CMD check** ne seront ainsi pas effacés et nous pourrons, par exemple, aller y récupérer la version PDF de la documentation du package.

#### Options du projet :

(à modifier pour chaque nouveau projet)

- par le menu « Tools > Project Options... » ;
- dans « Build Tools » :
  - cocher « Generate documentation with Roxygen »,
  - si le menu de configuration ne s'ouvre pas automatiquement, cliquez sur « Configure », puis assurez-vous que les options suivantes soient cochées :
    - \* « Use roxygen to generate » : « Rd files » et « NAMESPACE »,
    - \* « Automatically roxygenize when running » : tout cocher.

Avec ces configurations, la majorité des commandes de construction et de vérification de package lancées par le menu « Build » (voir ci-dessous) vont d'abord soumettre la commande **roxygenize** sur le dossier du package avant de faire leur travail. Ainsi, les fichiers **.Rd** de documentation et le fichier **NAMESPACE** seront mis à jour à chaque lancement d'une de ces commandes

Nous pouvons aussi soumettre la commande **roxygenize** par le menu « **Build > Document** ».

**Note :** Si vous n'avez pas de compilateur LaTeX sur votre ordinateur, ajoutez l'option suivante à `R CMD check` : `--no-manual`.

### Sous-étape c) Construire à partir du menu « Build » de RStudio

Il est préférable de toujours d'abord s'assurer que le package passe sans erreur ou avertissements problématiques la vérification faite par la commande `R CMD check`. Ensuite, nous pouvons construire le package, soit dans sa version source ou binaire. Voici comment faire tout ça facilement en RStudio.

- Pour vérifier le package :  
menu « **Build > Check Package** » (lance en fait la commande `R CMD check`).
- Pour créer le package source (qui est aussi la version Unix / Linux) :  
menu « **Build > Build Source Package** » (lance en fait la commande `R CMD build`)  
→ un fichier nommé "`nomPackage_numeroVersion.tar.gz`" (dans notre exemple "`distManhattan_1.0.0.tar.gz`") sera créé dans le dossier contenant le dossier des fichiers sources du package (donc un niveau plus haut).
- Pour créer le package binaire (si nous travaillons sous Windows ou Mac OS X / OS X / macOS) :  
menu « **Build > Build Binary Package** » (lance en fait la commande `R CMD INSTALL --build`)  
→ un fichier nommé "`nomPackage_numeroVersion.zip`" (sous Windows) ou "`nomPackage_numeroVersion.tgz`" (sous Mac OS X / OS X / macOS) sera créé dans le dossier contenant le dossier des fichiers sources du package.

La commande « **Install & Restart** » (anciennement « **Build and Reload** ») est pratique en cours de travail. Elle permet de construire le package dans le bon format pour notre système d'exploitation, l'installer (donc remplacer l'ancienne installation par la nouvelle) et charger de nouveau le package (avec la commande `library`).

### Lancement des commandes de construction et de vérification avec devtools

Nous ne couvrirons pas cette option ici, mais les commandes de construction et de vérification d'un package peuvent aussi être soumises à l'aide de fonctions du package `devtools` (<https://github.com/hadley/devtools>). Comme les utilitaires de RStudio, ces fonctions permettent de soumettre des commandes `R CMD` sans passer nous-mêmes par le terminal ou l'invite de commandes. Avec `devtools`, tout se réalise via des commandes soumises dans la console R.

## Étape 5. Si désiré, partager le package

Notre package est maintenant prêt à être utilisé. Si nous souhaitons le partager avec le grand public, nous pouvons le rendre disponible sur le CRAN. Pour ce faire, il faut d'abord s'assurer de respecter les politiques du CRAN : <http://cran.r-project.org/web/packages/policies.html>. Comme mentionné précédemment, il faut donc que notre package passe le `R CMD check --as-cran` sans erreurs ni avertissements.

Une fois s'être assuré de respecter les politiques du CRAN, nous pouvons soumettre notre package au CRAN en ligne par l'intermédiaire de l'interface web suivante :

<https://cran.r-project.org/submit.html>

Il suffit de suivre les instructions.

## Étape 6. Au besoin, mettre à jour le package

Lors de mise à jour d'un package, il faut :

### Sous-étape a) Incrémenter le numéro de version :

Cette incrémentation doit être effectuée dans le fichier `DESCRIPTION`, ainsi que dans les commentaires `roxygen2` générant la fiche d'aide du package.

Rappelons qu'un numéro de version est une séquence d'au minimum 2 (souvent 3) nombres entiers non négatifs séparés par un seul caractère `.` ou `-`. Ces nombres ne sont pas contraints à être compris entre 0 et 9.

Voici les règles que j'ai personnellement choisi de suivre dans la numérotation des versions de mes packages. Je forme mes numéros de version de 3 nombres séparés par un point. En cours de développement, soit avant d'avoir une version que je considère suffisamment testée, j'utilise un 0 comme premier nombre dans mon numéro de version (par exemple 0.9.12). Lorsque je juge mon package assez fiable pour être rendu disponible à plus grande échelle qu'à l'interne, je change le premier nombre dans le numéro de version pour 1, ce qui fait retomber à zéro les nombres suivants. La première version officielle porte donc le numéro de version 1.0.0.

Ensuite, voici comment je fais évoluer mes numéros de version :

- Lors d'une mise à jour majeure (beaucoup de nouvelles fonctionnalités) : le premier nombre de la numérotation est incrémenté de 1, les nombres subséquents retombent à 0.
- Lors d'une mise à jour mineure (seulement quelques fonctionnalités pas trop importantes ajoutées ou importants bogues réglés) : le premier nombre est inchangé, par contre le deuxième est incrémenté de 1 et le dernier retombe à 0.
- Si je corrige seulement quelques bogues sans changer du tout les fonctionnalités : j'incrémente de 1 le troisième nombre de la numérotation, sans modifier les deux premiers.

La page Wikipédia [https://en.wikipedia.org/wiki/Software\\_versioning](https://en.wikipedia.org/wiki/Software_versioning) traite de ce sujet et propose d'autres règles.

### Sous-étape b) Faire les mises à jour dans le code :

Si nous ajoutons des fonctionnalités ou corrigeons un bogue, le code va nécessairement être modifié.

### Sous-étape c) Documenter les modifications :

Les commentaires `roxygen2` produisant les fiches d'aide doivent être mis à jour de façon à refléter les modifications apportées au code. Une correction d'un bogue ne nécessite pas toujours de mise à jour des fiches d'aide, mais un ajout de fonctionnalités en nécessite toujours.

Je conseille vivement aussi de documenter les mises à jour dans un fichier `NEWS`. Ce fichier est un point de repère pour un utilisateur d'un package qui souhaite identifier ce qui a changé lors d'une mise à jour, donc ce qui pourrait affecter son utilisation du package.

Il n'y a pas de consensus en R à propos de comment rédiger le fichier `NEWS`, ni de l'endroit où le placer dans les fichiers source. Nous pouvons observer, notamment, les pratiques suivantes :

- fichier `NEWS` en format texte simple (dont le nom de fichier ne porte pas d'extension), placé dans le dossier principal (au même niveau que les fichiers `DESCRIPTION` et `NAMESPACE`) ;
- fichier `NEWS.md` en format Markdown, placé dans le dossier principal ;
- fichier `NEWS` en format texte simple (dont le nom de fichier ne porte pas d'extension), placé dans le sous-dossier `inst` ;
- fichier `NEWS.Rd` en format `.Rd`, placé dans le sous-dossier `inst`.



La première des pratiques est probablement la plus répandue.

Voici à quoi pourrait ressembler un fichier `NEWS` en format texte simple pour l'exemple du package `distManhattan`, après l'avoir mis à jour.

Fichier `"C:/coursR/distManhattan/NEWS"` :

---

Changements dans `distManhattan` version 1.1.0 (2018-03-27)

- \* modifications des fonctions `Manhattan` et `Manhattan2` afin qu'elles puissent calculer la distance entre plusieurs points
- \* ajout d'une methode `plot` pour un objet de classe `"Manhattan"`

Changements dans `distManhattan` version 1.0.0 (2018-03-25)

- \* premiere version de `distManhattan`
- 

Si nous installions la version 1.1.0 du package et que nous chargions le package en R avec la commande `library`, nous pourrions afficher son fichier `NEWS` dans la console avec la commande suivante :

```
news(package = "distManhattan")
```

Nous pouvons aussi voir le fichier `NEWS` des packages distribués sur le CRAN directement sur leur page web du CRAN.

### Sous-étape d) Reconstruire et revérifier le package

Un nouveau fichier compressé sera produit, portant le nom du package accompagné du nouveau numéro de version.

## Références

- R Core Team (2018). *Writing R Extensions*. R Foundation for Statistical Computing. Chapitre 4. <http://cran.r-project.org/doc/manuals/r-release/R-exts.pdf>
- Wickham, H. (2015). *R packages*. O'Reilly Media, Inc. URL <http://r-pkgs.had.co.nz/>  
– documentation avec `roxygen2` : <http://r-pkgs.had.co.nz/man.html>
- Hadley Wickham, Peter Danenberg and Manuel Eugster (2017). `roxygen2` : In-Line Documentation for R. R package version 6.0.1. <https://CRAN.R-project.org/package=roxygen2>

Pour le développement en utilisant le package `devtools` :

<https://www.rstudio.com/wp-content/uploads/2015/06/devtools-cheatsheet.pdf>