Se former au logiciel R : initiation et perfectionnement

François Rebaudo

2020-01-03

Table of Contents

# Préambule

## Remerciements

Je remercie tous les contributeurs qui ont participé à améliorer ce livre par leurs conseils, leurs suggestions de modifications et leurs corrections (par ordre alphabétique) :

## Contributeurs :  
Camila Benavides Frias (Bolivia)  
Marc Girondot (France ; UMR 8079 ESE)  
Susi Loza Herrera (Bolivia)  
Estefania Quenta Herrera (Bolivia)  
Baptiste Régnier (France)

Les versions gitbook, html et epub de ce livre utilisent les icônes open source de Font Awesome (<https://fontawesome.com>). La version PDF utilise les icônes issues du projet Tango disponibles depuis openclipart (<https://openclipart.org/>). Ce livre a été écrit avec le package R bookdown (<https://bookdown.org/>). Le code source est disponible sur GitHub (<https://github.com/frareb/myRBook_FR>). La version en ligne est hébergée et mise à jour grâce à Netlify (<http://myrbookfr.netlify.com/>).

Les diapositives du [module de formation IRD sur l’analyse de variance et le modèle linéaire sont disponibles ici](http://myrbookfr.netlify.com/myHtmls/France_Montpellier_2019/R00_links.html).

## Licence

Licence Creative Commons Attribution - Pas d’Utilisation Commerciale - Pas de Modification 3.0 France (CC BY-NC-ND 3.0 FR ; <https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>)

C’est un résumé (et non pas un substitut) de la licence.

**Vous êtes autorisé à :**

* Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats.
* L’Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

**Selon les conditions suivantes :**

* Attribution — Vous devez créditer l’Œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l’Oeuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l’Offrant vous soutient ou soutient la façon dont vous avez utilisé son Oeuvre.
* Pas d’Utilisation Commerciale — Vous n’êtes pas autorisé à faire un usage commercial de cette Oeuvre, tout ou partie du matériel la composant.
* Pas de modifications — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l’Oeuvre originale, vous n’êtes pas autorisé à distribuer ou mettre à disposition l’Oeuvre modifiée.
* Pas de restrictions complémentaires — Vous n’êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l’Oeuvre dans les conditions décrites par la licence.

**Notes :**

Vous n’êtes pas dans l’obligation de respecter la licence pour les éléments ou matériel appartenant au domaine public ou dans le cas où l’utilisation que vous souhaitez faire est couverte par une exception. Aucune garantie n’est donnée. Il se peut que la licence ne vous donne pas toutes les permissions nécessaires pour votre utilisation. Par exemple, certains droits comme les droits moraux, le droit des données personnelles et le droit à l’image sont susceptibles de limiter votre utilisation.

# Introduction

## Pourquoi se former à R

Parce que R s’est imposé comme un outil incontournable pour l’analyse et la gestion des données scientifiques (et des données en général), et qu’il devient dans ce contexte indispensable d’en maîtriser à minima les bases. Le succès de R n’est pas un hasard : R est un logiciel que tout le monde peut se procurer librement assurant ainsi la transparence et la reproductibilité des résultats scientifiques (sous réserve de respecter quelques règles que ce livre abordera). R repose aussi sur une communauté très active avec plusieurs milliers de modules complémentaires (packages) pour effectuer les analyses statistiques les plus pointues.

## Ce livre

L’objectif de ce livre est de fournir aux scientifiques, aux étudiants et aux personnes souhaitant s’initier à R une base solide pour mettre en œuvre leurs propres projets scientifiques et conduire avec rigueur la valorisation de leurs résultats. Il existe de nombreux livres dédiés à R, mais aucun ne couvre les éléments de base de ce langage dans un objectif de rendre les résultats scientifiques publiables et reproductibles. Ainsi, tout au long de ce livre nous nous efforcerons d’avoir non seulement un code fonctionnel pour la machine, mais aussi un code lisible et reproductible pour les humains.

Ce livre est né de la demande des étudiants des universités partenaires de l’IRD en Amérique du Sud que j’ai eu la chance de rencontrer et de former à R. Sa première version est donc rédigée en espagnol (il existe peu de documents de qualité sur R en espagnol). J’ai entamé sa traduction en français courant 2018 et aujourd’hui les deux versions coévoluent avec des contenus qui peuvent varier.

## Lectures complémentaires en français

* R pour les débutants, Emmanuel Paradis (<https://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf>)
* Introduction à la programmation avec R, Vincent Goulet (<https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf>)

# Premiers pas

## Installation de R

Le programme permettant l’installation du logiciel R peut être téléchargé depuis le site web de R : <https://www.r-project.org/>. Sur le site de R il faut au préalable choisir un miroir CRAN (serveur depuis lequel télécharger R ; sauf cas particulier le plus proche de sa localisation géographique), puis télécharger le fichier *base*. Les utilisateurs de Linux pourront préférer un sudo apt-get install r-base.

Le logiciel R peut être téléchargé depuis de nombreux serveurs du CRAN (Comprehensive R Archive Network) à travers le monde. Ces serveurs s’appellent des miroirs. Le choix du miroir est manuel.

## R comme calculatrice

Une fois le programme lancé, une fenêtre apparaît dont l’aspect peut varier en fonction de votre système d’exploitation (Figure 1). Cette fenêtre est dénommée la *console*. La première information que l’on peut trouver sur la console est la version de R utilisée. Il est recommandé de mettre à jour régulièrement sa version de R afin de bénéficier des dernières fonctionnalités. Au moment d’écrire ce livre, la version disponible est la 3.6.2 dénommée “Dark and Stormy Night”.

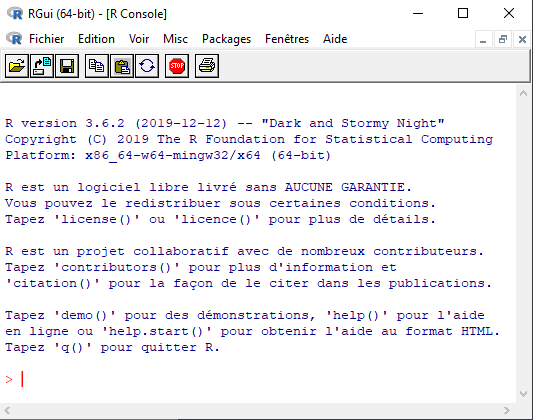


Figure 1: Capture d’écran de la console R sous Windows.

La console correspond à l’interface où va être interprété le code, c’est à dire à l’endroit depuis lequel le code va être transformé en langage machine, puis exécuté par l’ordinateur. Le résultat de cette exécution sera retransmis dans la console sous une forme lisible par des humains. Cela correspond à l’écran d’affichage d’une calculatrice (Figure 2). C’est de cette manière que R va être utilisé dans la suite de cette section.

Tout au long de ce livre, les exemples de code R apparaîtront sur fond en gris. Ils peuvent être copiés et collés directement dans la console, bien qu’il soit préférable de reproduire soit même les exemples dans la console (ou plus tard dans les scripts). Le résultat de ce qui est envoyé dans la console apparaîtra également sur fond en gris avec ## devant le code afin de bien faire la distinction entre le code et le résultat du code.

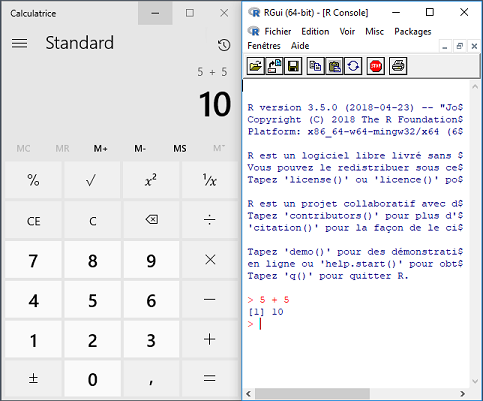


Figure 2: Capture d’écran de la console R sous Windows avec la calculatrice Windows.

### Les opérateurs arithmétiques

5 + 5

## [1] 10

Si nous écrivons 5 + 5 dans la console puis la touche du clavier Entrée, le résultat apparaît précédé du chiffre [1] entre crochets. Ce chiffre correspond au numéro du résultat (dans notre cas, il n’y a qu’un seul résultat ; nous reviendrons sur cet aspect plus tard). Nous pouvons également noter dans cet exemple l’utilisation d’espaces avant et après le signe +. Ces espaces ne sont pas nécessaires mais permettent au code d’être plus lisible par les humains (i.e., plus agréable à lire pour nous comme pour les personnes avec qui nous serons amenés à partager notre code). Les opérateurs arithmétiques disponibles sous R sont résumés dans la table 1.

Table 1: Opérateurs arithmétiques.

|  |  |  |  |
| --- | --- | --- | --- |
| Label | Opérateur | Exemple | Résultat |
| Addition | + | 5 + 5 | 10 |
| Soustraction | - | 5 - 5 | 0 |
| Multiplication | \* | 5\*5 | 25 |
| Division | / | 5/5 | 1 |
| Puissance | ^ | 5^5 | 3125 |
| Modulo | %% | 5 %% 5 | 0 |
| Quotient Décimal | %/% | 5 %/% 5 | 1 |

Classiquement, les multiplications et les divisions sont prioritaires sur les additions et les soustractions. Au besoin nous pouvons utiliser des parenthèses.

5 + 5 \* 2

## [1] 15

(5 + 5) \* 2

## [1] 20

L’opérateur modulo correspond au reste de la division euclidienne. Il est souvent utilisé en informatique par exemple pour savoir si un nombre est pair ou impair (un nombre modulo 2 va renvoyer 1 si il est impair et 0 si il est pair).

451 %% 2

## [1] 1

288 %% 2

## [1] 0

(5 + 5 \* 2) %% 2

## [1] 1

((5 + 5) \* 2) %% 2

## [1] 0

R intègre également certaines constantes comme pi (3.141593). Par ailleurs le signe infini est représenté par Inf.

pi

## [1] 3.141593

pi \* 5^2

## [1] 78.53982

1/0

## [1] Inf

Le *style* du code est important car le code est destiné à être lisible par nous plus tard et par d’autres personnes de manière générale. Pour avoir un style lisible il est recommandé de mettre des espaces avant et après les opérateurs arithmétiques (bonnes pratiques de programmation).

### Les opérateurs de comparaison

R est cependant bien plus qu’une simple calculatrice puisqu’il permet un autre type d’opérateurs : les opérateurs de comparaison. Ils servent comme leur nom l’indique à *comparer* des valeurs entre elles (Table 2).

Table 2: Opérateurs de comparaison.

|  |  |  |  |
| --- | --- | --- | --- |
| Label | Opérateur | Exemple | Résultat |
| plus petit que | < | 5 < 5 | FALSE |
| plus grand que | > | 5 > 5 | FALSE |
| plus petit ou égal à | <= | 5 <= 5 | TRUE |
| plus grand ou égal à | >= | 5 >= 5 | TRUE |
| égal à | == | 5 == 5 | TRUE |
| différent de | != | 5 != 5 | FALSE |

Par exemple si nous voulons savoir si un chiffre est plus grand qu’un autre, nous pouvons écrire :

5 > 3

## [1] TRUE

R renvoie la valeur TRUE si la comparaison est vraie et FALSE si la comparaison est fausse.

5 > 3

## [1] TRUE

2 < 1.5

## [1] FALSE

2 <= 2

## [1] TRUE

3.2 >= 1.5

## [1] TRUE

Nous pouvons combiner les opérateurs arithmétiques avec les opérateurs de comparaison.

(5 + 8) > (3 \* 45/2)

## [1] FALSE

Dans la comparaison (5 + 8) > (3 \* 45/2) les parenthèses ne sont pas nécessaires mais elles permettent au code d’être plus facile à lire.

Un opérateur de comparaison particulier est *égal à*. Nous verrons dans la section suivante que le signe = est réservé à un autre usage : il permet d’affecter une valeur à un objet. L’opérateur de comparaison *égal à* doit donc être différent, c’est pour cela que R utilise ==.

42 == 53

## [1] FALSE

58 == 58

## [1] TRUE

Un autre opérateur particulier est *différent de*. Il est utilisé avec *un point d’interrogation* suivi de *égal*, !=. Cet opérateur permet d’obtenir la réponse inverse à ==.

42 == 53

## [1] FALSE

42 != 53

## [1] TRUE

(3 + 2) != 5

## [1] FALSE

10/2 == 5

## [1] TRUE

R utilise TRUE et FALSE qui sont aussi des valeurs qui peuvent être testées avec les opérateurs de comparaison. Mais R attribue également une valeur à TRUE et FALSE :

TRUE == TRUE

## [1] TRUE

TRUE > FALSE

## [1] TRUE

1 == TRUE

## [1] TRUE

0 == FALSE

## [1] TRUE

TRUE + 1

## [1] 2

FALSE + 1

## [1] 1

(FALSE + 1) == TRUE

## [1] TRUE

La valeur de TRUE est de 1 et la valeur de FALSE est de 0. Si cela peut paraître étrange à ce stade, nous verrons plus tard comment utiliser cette information et toutes les fonctionnalités qu’elle permet.

R est aussi un langage relativement permissif, cela veut dire qu’il admet une certaine flexibilité dans la manière de rédiger le code. Débattre du bien-fondé de cette flexibilité sort du cadre de ce livre mais nous pourrons trouver dans du code R sur Internet ou dans d’autres ouvrages le raccourcis T pour TRUE et F pour FALSE.

T == TRUE

## [1] TRUE

F == FALSE

## [1] TRUE

T == 1

## [1] TRUE

F == 0

## [1] TRUE

(F + 1) == TRUE

## [1] TRUE

Bien que cette façon de se référer à TRUE et FALSE par T et F soit assez répandue, dans ce livre nous utiliserons toujours TRUE et FALSE afin que le code soit plus facile à lire. Encore une fois l’objectif d’un code est de non seulement être fonctionnel mais aussi d’être facile à lire et à relire.

### Les opérateurs logiques

Il existe un dernier type d’opérateur, les opérateurs logiques. Ils sont utiles pour combiner des opérateurs de comparaison (Table 3).

Table 3: Opérateurs logiques.

|  |  |
| --- | --- |
| Label | Operador |
| n’est pas | ! |
| et | & |
| ou | | |
| ou exclusif | xor() |

!TRUE

## [1] FALSE

!FALSE

## [1] TRUE

((3 + 2) == 5) & ((3 + 3) == 5)

## [1] FALSE

((3 + 2) == 5) & ((3 + 3) == 6)

## [1] TRUE

(3 < 5) & (5 < 5)

## [1] FALSE

(3 < 5) & (5 <= 5)

## [1] TRUE

L’opérateur logique xor() correspond à un *ou exclusif*. C’est à dire que l’un des deux **arguments** de la **fonction** xor() doit être vrai, mais pas les deux. Nous reviendrons plus tard sur les **fonctions** et leurs **arguments**, mais retenons que l’on identifie une fonction par ses parenthèses qui contiennent des arguments séparés par des virgules.

xor((3 + 2) == 5, (3 + 3) == 6)

## [1] FALSE

xor((3 + 2) == 5, (3 + 2) == 6)

## [1] TRUE

xor((3 + 3) == 5, (3 + 2) == 6)

## [1] FALSE

xor((3 + 3) == 5, (3 + 3) == 6)

## [1] TRUE

Il est recommandé que les virgules (,) soient suivies par un espace afin que le code soit plus agréable à lire. Cela fait beaucoup de recommandations sur le style du code qui peuvent sembler à ce stade superflues. Néanmoins il est important d’adopter de bonnes pratiques de programmation dès le début de notre apprentissage de R (il est toujours plus facile de prendre de bonnes habitudes que de devoir corriger de mauvaises habitudes par la suite).

### Aide sur les opérateurs

Le fichier d’aide en anglais sur les opérateurs arithmétiques peut être obtenu avec la commande ?'+'. Le fichier d’aide sur les opérateurs de comparaison peut être obtenu avec la commande ?'==' et celui sur les opérateurs logiques avec la commande ?'&'.

## La notion d’objet

Un aspect important de la programmation avec R, mais aussi de la programmation en général est la notion d’objet. Comme indiqué sur la page web de wikipedia (<https://fr.wikipedia.org/wiki/Objet_(informatique)>), en informatique, un objet est un *conteneur*, c’est à dire quelque chose qui va contenir de l’information. L’information contenue dans un objet peut être très variée, mais pour le moment nous allons contenir dans un objet le chiffre 5. Pour ce faire (et pour pouvoir le réutiliser par la suite), il nous faut donner un nom à notre objet. Avec R le nom des objets ne doit pas comprendre de caractères spéciaux comme *^$?|+()[]}{*, ne doit pas commencer par un chiffre ni contenir d’espaces. Le nom de l’objet doit être représentatif de ce qu’il contient, tout en étant ni trop court ni trop long. Imaginons que notre chiffre 5 corresponde au nombre de répétitions d’une expérience. Nous voudrions lui donner un nom faisant référence à *nombre* et à *répétition*, que nous pourrions réduire à *nbr* et *rep*, respectivement. Il existe plusieurs possibilités qui sont toutes assez répandues sous R :

* la séparation au moyen du caractère *tiret bas* : nbr\_rep
* la séparation au moyen du caractère *point* : nbr.rep
* l’utilisation de lettres minuscules : nbrrep
* le style *lowerCamelCase* consistant en un premier mot en minuscules et des suivants avec une majuscule : nbrRep
* le style *UpperCamelCase* consistant à mettre une majuscule au début de chacun des mots : NbrRep

Toutes ces formes de nommer un objet sont équivalentes. De manière générale il faut éviter les noms trop longs comme leNombreDeRepetitions ou trop courts comme nR, et les noms ne permettant pas d’identifier le contenu comme maVariable ou monChiffre, mais aussi a ou b…

Il existe différentes façons de définir un nom pour les objets que nous allons créer avec R. Dans ce livre nous utilisons le style *lowerCamelCase*. L’important n’est pas le choix du style mais la consistance dans son choix. L’objectif est d’avoir un code fonctionnel mais également un code facile et agréable à lire pour soi et pour les autres.

Maintenant que nous avons choisi un nom pour notre objet, il faut le créer et faire comprendre à R que notre objet doit contenir le chiffre 5. Il existe trois façons de créer un objet sous R et de lui assigner un contenu:

* avec le signe <-
* avec le signe =
* avec le signe ->

nbrRep <- 5  
nbrRep = 5  
5 -> nbrRep

Dans ce livre nous utiliserons toujours la forme <- par souci de consistance et aussi parce que c’est la forme la plus répandue.

nbrRep <- 5

Nous venons de créer un objet nbrRep et de lui affecter la valeur 5. Cet objet est désormais disponible dans notre environnement de calcul et peut donc être utilisé. Voici quelques exemples :

nbrRep + 2

## [1] 7

nbrRep \* 5 - 45/56

## [1] 24.19643

pi \* nbrRep^2

## [1] 78.53982

La valeur associée à notre objet nbrRep peut être modifiée de la même manière que lors de sa création :

nbrRep <- 5  
nbrRep + 2

## [1] 7

nbrRep <- 10  
nbrRep + 2

## [1] 12

nbrRep <- 5 \* 2 + 7/3  
nbrRep + 2

## [1] 14.33333

L’utilisation des objets prend tout son sens lorsque nous avons des opérations complexes à réaliser et rend le code plus agréable à lire et à comprendre.

(5 + 9^2 - 1/18) / (32 \* 45/8 + 3)

## [1] 0.4696418

terme01 <- 5 + 9^2 - 1/18  
terme02 <- 32 \* 45/8 + 3  
terme01 / terme02

## [1] 0.4696418

## Les scripts

R est un langage de programmation souvent dénommé *langage de script*. Cela fait référence au fait que la plupart des utilisateurs vont écrire des petits bouts de code plutôt que des programmes entiers. R peut être utilisé comme une simple calculatrice, et dans ce cas il ne sera pas nécessaire de conserver un historique des opérations qui ont été réalisées. Mais si les opérations à réaliser sont longues et complexes, il peut devenir nécessaire de pouvoir sauvegarder ce qui a été fait à un moment donné pour pouvoir poursuivre plus tard. Le fichier dans lequel seront conservées les opérations constitue ce que l’on appelle communément le script. Un script est donc un fichier contenant une succession d’informations compréhensibles par R et qu’il est possible d’exécuter.

### Créer un script et le documenter

Pour ouvrir un nouveau script il suffit de créer un fichier texte vide qui sera édité par un éditeur de texte comme le *bloc note* sous Windows ou Mac OS, ou encore *Gedit* ou même *nano* sous Linux. Par convention ce fichier prend l’extension “.r” ou plus souvent “.R”. C’est cette dernière convention qui sera utilisée dans ce livre. Depuis l’interface graphique de R il est possible de créer un nouveau script sous Mac OS et Windows via *fichier* puis *nouveau script* et *enregistrer sous*. Tout comme le nom des objets, le nom du script est important pour que nous puissions facilement identifier son contenu. Par exemple nous pourrions créer un fichier formRConceptsBase.R contenant les objets que nous venons de créer et les calculs effectués. Mais même avec des noms de variables et un nom de fichier bien définis, il sera difficile de se rappeler le sens de ce fichier sans une documentation accompagnant le script. Pour documenter un script nous allons utiliser des *commentaires*. Les commentaires sont des éléments qui seront identifiés par R et qui ne seront pas exécutés. Pour spécifier à R que nous allons faire un commentaire, il faut utiliser le caractère octothorpe (croisillon ; différent du caractère dièse) #. Les commentaires peuvent être insérés sur une nouvelle ligne ou en fin de ligne.

# création objet nombre de répétitions  
nbrRep <- 5 # commentaire de fin de ligne

Les commentaires peuvent aussi être utilisés pour qu’une ligne ne soit plus exécutée.

nbrRep <- 5  
# nbrRep + 5

Pour en revenir à la documentation du script, il est recommandé de commencer chacun de ses scripts par une brève description de son contenu, puis lorsque le script devient long, de le structurer en différentes parties pour faciliter sa lecture.

# ------------------------------------------------------------  
# Voici un script pour acquérir les concepts de base   
# avec R  
# date de création : 25/06/2018  
# auteur : François Rebaudo  
# ------------------------------------------------------------  
  
# [1] création de l'objet nombre de répétitions  
# ------------------------------------------------------------  
  
nbrRep <- 5  
  
# [2] calculs simples  
# ------------------------------------------------------------  
  
pi \* nbrRep^2

## [1] 78.53982

Pour aller plus loin sur le style de code, un guide de recommandations est disponible en ligne sur le site de tidyverse (en anglais ; <http://style.tidyverse.org/>). Nous verrons que ce livre suit les principales recommandations de ce site, mais pas toutes car notre préoccupation principale est d’avoir un code lisible et reproductible.

### Exécuter un script

Depuis que nous avons un script, nous ne travaillons plus directement dans la console. Or seule la console est capable d’interpréter le code R et de nous renvoyer les résultats que nous souhaitons obtenir. Pour l’instant la technique la plus simple consiste à copier-coller les lignes que nous souhaitons exécuter depuis notre script vers la console. A partir de maintenant nous n’allons plus utiliser les éditeurs de texte comme le bloc note mais des éditeurs spécialisés pour la confection de scripts R. C’est l’objet du chapitre suivant.

## Conclusion

Félicitations, nous avons atteint la fin de ce premier chapitre sur les éléments de base de R. Nous savons:

* Installer R
* Utiliser R comme une calculatrice
* Créez des **objets** et les utiliser pour les calculs arithmétiques, les comparaisons et les tests logiques
* Choisir des noms pertinents pour les objets
* Créer de nouveaux **scripts**
* Choisir un nom pertinent pour les fichiers de script
* Exécuter le code d’un script
* Documenter les scripts avec des **commentaires**
* Utiliser un style de code pour le rendre agréable à lire et facile à comprendre

# Choisir un environnement de développement

## Editeurs de texte et environnement de développement

Il existe de très nombreux éditeurs de texte, le chapitre précédent a permis d’en introduire quelques-uns parmi les plus simples comme le *bloc note* de Windows. Rapidement les limites de ces éditeurs ont rendu la tâche d’écrire un script fastidieuse. En effet, même en structurant son script avec des commentaires, il reste difficile de se repérer dans celui-ci. C’est là qu’interviennent les éditeurs de texte spécialisés qui vont permettre une écriture et une lecture agréable et simplifiée. L’éditeur de texte pour R certainement le plus répandu est *Rstudio*, mais il en existe bien d’autres. Faire une liste exhaustive de toutes les solutions disponibles sort du cadre de ce livre, ainsi nous nous focaliserons sur les trois solutions que j’utilise au quotidien que sont **Notepad++**, **Rstudio**, et **Geany**.

## RStudio



Figure 3: Logo RStudio.

### Installer RStudio

Le programme pour installer *Rstudio* se retrouve dans la partie *Products* du site web de *Rstudio* (<https://www.rstudio.com/>). Nous allons installer RStudio pour un usage local (sur notre ordinateur), donc la version qui nous intéresse est *Desktop*. Nous allons utiliser la version *Open Source* qui est gratuite. Ensuite il nous suffit de sélectionner la version qui correspond à notre système d’exploitation, de télécharger le fichier correspondant et de l’exécuter pour lancer l’installation. Nous pouvons conserver les options par défaut tout au long de l’installation.

### Un script avec RStudio

Nous pouvons alors ouvrir RStudio. Lors de la première ouverture, l’interface est divisée en deux avec à gauche la console R que nous avons vu au chapitre précédent (Figure 4). Pour ouvrir un nouveau script, nous allons dans le menu *File*, *New File*, *R script*. Par défaut ce fichier a comme nom *Untitled1*. Nous avons vu au chapitre précédent l’importance de donner un nom pertinent à nos scripts, c’est pourquoi nous allons le renommer *selecEnvDev.R*, dans le menu *File*, avec l’option *Save As…*. Nous avons pu noter que la partie gauche de RStudio est désormais séparée en deux, avec en bas de l’écran la console et en haut de l’écran le script.

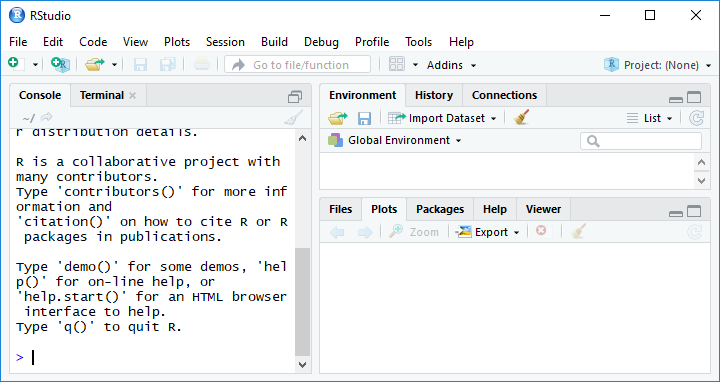


Figure 4: Capture d’écran de RStudio sous Windows : fenêtre par défaut.

Nous pouvons alors commencer l’écriture de notre script avec les commentaires décrivant ce que nous allons y trouver, et y ajouter un calcul simple. Une fois que nous avons recopié le code suivant, nous pouvons sauver notre script avec la commande CTRL + S ou en se rendant dans *File*, puis *Save*.

# ------------------------------------------------------------  
# Un script pour choisir son IDE  
# créé le 27/06/2018  
# modifié le 17/12/2019  
# François Rebaudo  
# ------------------------------------------------------------  
  
# [1] calcul simple  
# ------------------------------------------------------------  
nbrRep <- 5  
pi \* nbrRep^2

Pour exécuter notre script, il suffit de sélectionner les lignes que nous souhaitons exécuter et d’utiliser la combinaison de touches CTRL + ENTER. R va transformer notre code en langage compréhensible pour la machine (interprétation), puis l’exécuter. Le résultat de l’exécution est renvoyé sous forme lisible dans la console (Figure 5).

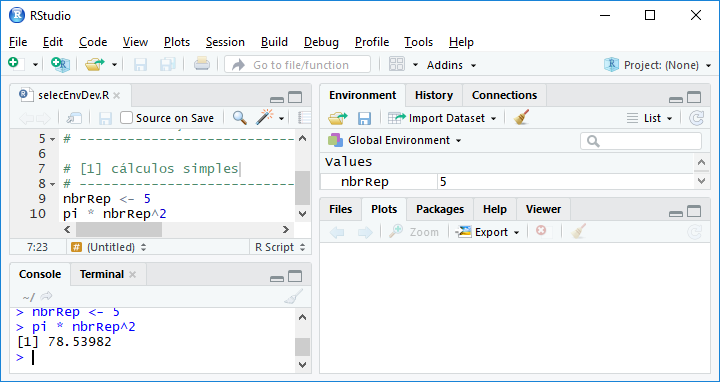


Figure 5: Capture d’écran de RStudio sous Windows : exécuter un script avec CTRL + ENTER.

Nous pouvons voir que par défaut dans la partie du script les commentaires apparaissent en vert, les chiffres en bleu, et le reste du code en noir. Dans la partie de la console ce qui a été exécuté apparaît en bleu et les résultats de l’exécution en noir. Nous pouvons également noter que dans la partie du code chaque ligne comporte un numéro correspondant au numéro de ligne à gauche sur fond gris. Il s’agit de la coloration syntaxique par défaut avec *RStudio.* Cette coloration syntaxique peut être modifiée en se rendant dans le menu *Tools*, *Global Options…*, *Appearance*, puis en choisissant un autre thème dans la liste *Editor theme:*. Nous allons choisir le thème *Cobalt*, puis *OK* (Figure 6).

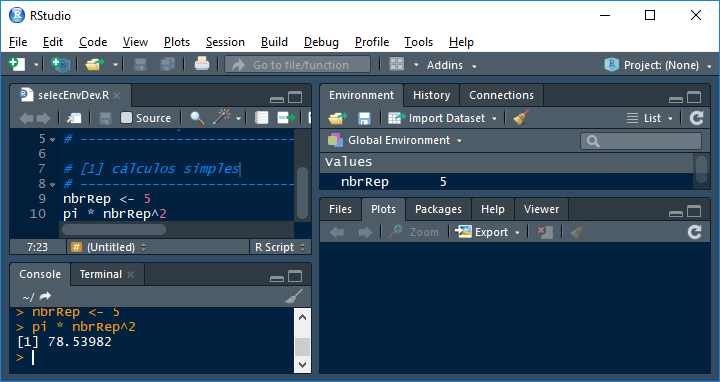


Figure 6: Capture d’écran de RStudio sous Windows : changer les paramètres de coloration syntaxique.

Nous savons comment créer un nouveau script, le sauvegarder, exécuter son contenu, et changer l’apparence de RStudio. Nous verrons les nombreux autres avantages de *RStudio* tout au long de ce livre car c’est l’environnement de développement qui sera utilisé. Nous serons néanmoins particulièrement vigilants à ce que tous les scripts développés dans ce livre s’exécutent de la même façon quel que soit l’environnement de développement utilisé.

## Notepad++ avec Npp2R



Figure 7: Logo Notepad++

### Installer Notepad++ (pour Windows uniquement)

Le programme pour installer *Notepad++* se trouve dans l’onglet *Downloads* (<https://notepad-plus-plus.org/download/>). Vous pouvez choisir entre la version 32-bit et 64-bit (64-bit si vous ne savez pas quelle version choisir). *Notepad++* seul est suffisant pour écrire un script, mais il est encore plus puissant avec *Notepad to R* (*Npp2R*) qui permet d’exécuter automatiquement nos script dans une console en local sur notre ordinateur ou à distance sur un serveur.

### Installer Npp2R

Le programme pour installer *Npp2R* est hébergé sur le site de Sourceforge (<https://sourceforge.net/projects/npptor/>). *Npp2R* doit être installé après *R* et après *Notepad++*.

### Un script avec Notepad++

Lors de la première ouverture *Notepad++* affiche un fichier vide *new 1* (Figure 8).

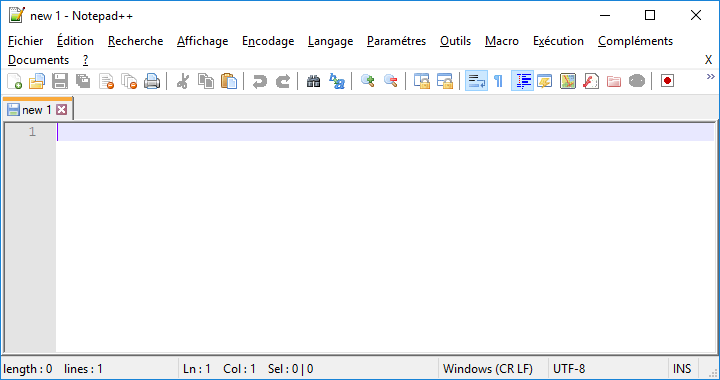


Figure 8: Capture d’écran de Notepad++ sous Windows : fenêtre par défaut.

Puisque nous avons déjà créé un script pour le tester avec RStudio, nous allons l’ouvrir à présent avec *Notepad++*. Dans *Fichier*, sélectionnons *Ouvrir…* puis choisir le script *selecEnvDev.R* créé précédemment. Une fois le script ouvert, allons dans *Langage*, puis *R*, et encore une fois *R*. La coloration syntaxique apparaît (Figure 9).

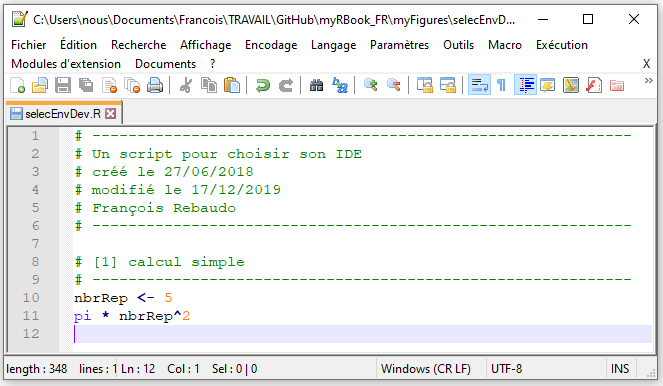


Figure 9: Capture d’écran de Notepad++ sous Windows : exécuter un script avec F8.

L’exécution du script ne peut se faire que si *Npp2R* est en cours d’exécution. Pour se faire il est nécessaire de lancer le programme *Npp2R* depuis l’invite de Windows. Un icône devrait apparaître en bas de votre écran. L’exécution automatique du code depuis Notepad++ se fait en sélectionnant le code à exécuter puis en utilisant la commande F8. Si la commande ne fonctionne pas et que vous venez d’installer Notepad++, il est peut être nécessaire de redémarrer votre ordinateur. Si la commande fonctionne, une nouvelle fenêtre va s’ouvrir aavec une console renvoyant le résultat de l’exécution des lignes souhaitées (Figure 10.

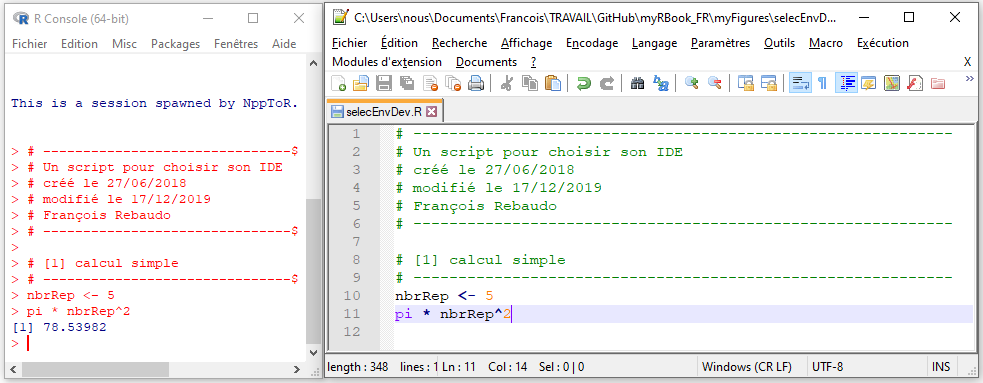


Figure 10: Capture d’écran de Notepad++ sous Windows : la console avec F8.

Comme pour *RStudio*, la coloration syntaxique peut être modifiée depuis le menu *Paramètres*, et un nouveau thème peut être sélectionné (par exemple *Solarized* dans la Figure 11)

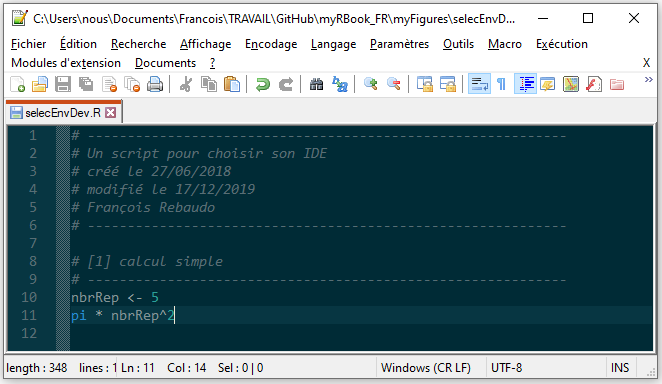


Figure 11: Capture d’écran de Notepad++ sous Windows : coloration syntaxique avec le thème Solarized.

Par rapport aux autres éditeurs de texte, *Notepad++* a l’avantage d’être très léger et offre une vaste gamme d’options pour personnaliser l’écriture du code.

## Geany



Figure 12: Logo Geany

### Installer Geany

Le programme pour l’installation de *Geany* se trouve sous l’onglet *Downloads* dans le menu de gauche *Releases* de la page web (<https://www.geany.org/>). Ensuite il suffit de télécharger l’exécutable pour Windows ou le dmg pour Mac OSX. Les utilisateurs de Linux préfèreront un sudo apt-get install geany (*Geany* est installé par défaut sur la plupart des distributions de Linux).

### Un script avec Geany

Lors de la première ouverture, comme pour *RStudio* et *Notepad++*, un fichier vide est créé (Figure 13).

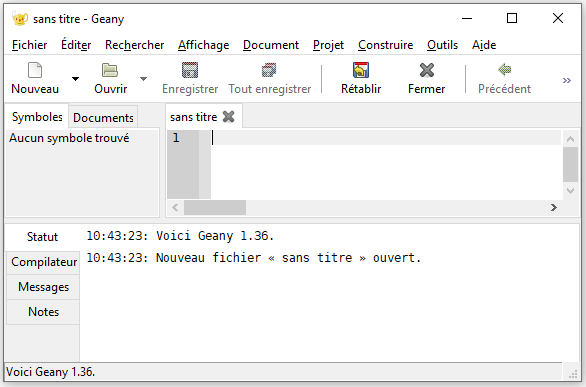


Figure 13: Capture d’écran de Geany sous Windows : fenêtre par défaut.

Nous pouvons ouvrir notre script avec *Fichier*, *Ouvrir* (Figure 14).

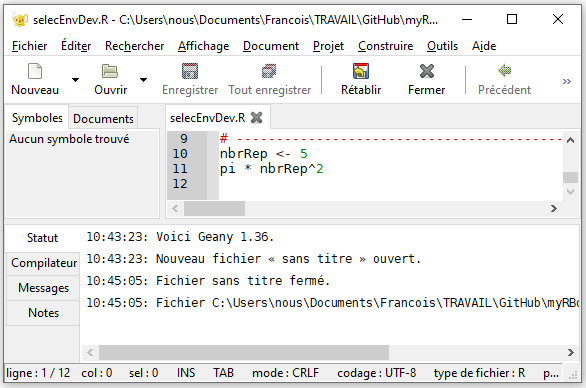


Figure 14: Capture d’écran de Geany sous Windows : ouvrir un script.

Pour exécuter notre script, la version de *Geany* pour Windows ne dispose pas d’un terminal intégré, ce qui rend son utilisation limitée sous ce système d’exploitation. L’exécution d’un script peut se faire en ouvrant R dans une fenêtre à part et en copiant et collant les lignes à exécuter. Sous Linux et Mac OSX, il suffit d’ouvrir R dans le terminal situé dans la partie basse de la fenêtre de *Geany* avec la commande R. Nous pouvons ensuite paramétrer *Geany* pour qu’une combinaison de touches permette d’exécuter le code sélectionné (par exemple CTRL + R). Pour cela il faut tout d’abord autoriser l’envoi de sélection vers le terminal (send\_selection\_unsafe=true) dans le fichier geany.conf puis choisir la commande d’envoi vers le terminal (dans *Edit*, *Preferences*, *Combinations*). Pour changer le thème de *Geany*, il existe une collection de thèmes accessibles sur GitHub (<https://github.com/geany/geany-themes/>). Le thème peut ensuite être changé via le menu *View*, *Change color theme…* (un exemple avec le thème *Solarized*, Figure 15).

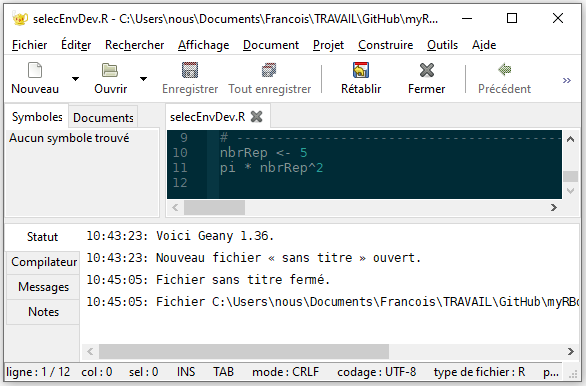


Figure 15: Capture d’écran de Geany sous Windows : changer les paramètres de coloration syntaxique.

## Autres solutions

Il existe beaucoup d’autres solutions, certaines spécialisées pour R comme **Tinn-R** (<https://sourceforge.net/projects/tinn-r/>), et d’autres plus généralistes pour la programmation comme **Atom** (<https://atom.io/>), **Sublime Text** (<https://www.sublimetext.com/>), **Vim** (<https://www.vim.org/>), **Gedit** (<https://wiki.gnome.org/Apps/Gedit>), **GNU Emacs** (<https://www.gnu.org/software/emacs/>), **Jupyter** (<http://jupyter.org>) ou encore **Brackets** (<http://brackets.io/>) et **Eclipse** (<http://www.eclipse.org/>).

## Conclusion

Félicitations, nous sommes arrivés au bout de ce chapitre sur environnements de développement pour utiliser R. Nous savons désormais :

* Installer *RStudio*, *Geany* ou *Notepad++*
* Reconnaître et choisir notre environnement préféré

A partir de maintenant nous allons pouvoir nous concentrer sur le langage de programmation R dans un environnement facilitant le travail de lecture et d’écriture du code. C’est un grand pas en avant pour maîtriser R.

# Les types de données

Nous avons vu précédemment comment créer un objet. Un objet est comme une boîte dans laquelle nous allons *stocker* de l’information. Jusqu’à présent nous n’avons stocké que des nombres mais dans ce chapitre nous allons voir qu’il est possible de stocker d’autres informations et nous allons nous attarder sur les types les plus courants. Dans ce chapitre nous allons utiliser des **fonctions** sur lesquelles nous reviendrons plus tard.

## Le type numeric

Le type numeric correspond à ce que nous avons fait jusqu’à présent, stocker des nombres. Il existe deux principaux types de nombres avec R: les nombres entiers (*integers*), et les nombres à virgule (*double*). Par défaut R considère tous les nombres comme des nombres à virgule et attribue le type double. Pour vérifier le type de données nous allons utiliser la fonction typeof() qui prend comme argument un objet (ou directement l’information que nous souhaitons tester). Nous pouvons également utiliser la fonction is.double() qui va renvoyer TRUE si le nombre est au format double et FALSE dans le cas contraire. La fonction générique is.numeric() va quant à elle renvoyer TRUE si l’objet est au format numeric et FALSE dans le cas contraire.

nbrRep <- 5  
typeof(nbrRep)

## [1] "double"

typeof(5.32)

## [1] "double"

is.numeric(5)

## [1] TRUE

is.double(5)

## [1] TRUE

Si nous voulons spécifier à R que nous allons travailler avec un nombre entier, alors il nous faut transformer notre nombre à virgule en nombre entier avec la fonction as.integer(). Nous pouvons également utiliser la fonction is.integer() qui va renvoyer TRUE si le nombre est au format integer et FALSE dans le cas contraire.

nbrRep <- as.integer(5)  
typeof(nbrRep)

## [1] "integer"

typeof(5.32)

## [1] "double"

typeof(as.integer(5.32))

## [1] "integer"

as.integer(5.32)

## [1] 5

as.integer(5.99)

## [1] 5

is.numeric(nbrRep)

## [1] TRUE

Nous voyons ici que transformer un nombre comme 5.99 au format integer va renvoyer uniquement la partie entière, soit 5.

is.integer(5)

## [1] FALSE

is.numeric(5)

## [1] TRUE

is.integer(as.integer(5))

## [1] TRUE

is.numeric(as.integer(5))

## [1] TRUE

La somme d’un nombre entier et d’un nombre à virgule renvoie un nombre à virgule.

sumIntDou <- as.integer(5) + 5.2  
typeof(sumIntDou)

## [1] "double"

sumIntInt <- as.integer(5) + as.integer(5)  
typeof(sumIntInt)

## [1] "integer"

Pour résumer, le type numeric contient deux sous-types, le type integer pour les nombres entiers et le type double pour les nombres à virgule. Par défaut R attribue le type double aux nombres.

Deux nombres particuliers doivent être considérés. Inf est un nombre de type double. Il désigne un nombre infini. Par exemple, 1/0 retourne Inf et aucun nombre ne peut être plus grand que Inf. NaN est une valeur qui signifie Not a Number : par exemple 0/0 retourne NaN. Cela peut être testé par is.nan(0/0). Attention à ne pas confondre avec NA qui désigne une donnée absente (voir plus loin).

Attention, il y a un piège à l’utilisation de la fonction is.integer(). Elle ne nous dit pas si le nombre est un entier mais s’il est du type integer. En effet, nous pouvons très bien stocker un nombre entier dans une variable de type double.

Les nombres stockés dans une variable integer sont codés sur 32 bits et peuvent prendre des valeurs comprises entre 0 et 2^32-1 = 4294967295. Il existe une autre façon d’indiquer à R qu’un nombre est un entier, en utilisant le suffixe L. Par exemple, écrire 5L revient à la même chose qu’écrire as.integer(5). L’origine de ce suffixe L date d’une époque où les ordinateurs utilisaient des mots de 16 bits, et 32 bits étaient bien un type Long. Maintenant les ordinateurs utilisent des mots de 64 bits et 32 bits est plutôt court !

Nous ne pouvons pas quitter cette section sans mentionner les fonctions [ceiling()](#l015round), [floor()](#l015round), [trunc()](#l015round) ou [round()](#l015round) qui retournent la partie entière d’un nombre mais le laisse au type double. Pour en savoir plus, nous pouvons utiliser [l’aide de R](#l015help) avec ?round. Nous reviendrons sur ces fonctions plus tard.

roundDou <- round(5.2)  
typeof(roundDou)

## [1] "double"

## Le type character

Le type character correspond au texte. Pour spécifier à R que l’information contenue dans un objet est au format texte, il faut utiliser les guillemets doubles ("), ou simples ('). Nous noterons que les guillemets doubles ou simples sont droits, par opposition au signe apostrophe (‘) ou encore aux signes correspondant aux ouvertures et fermetures de guillemets (« », ˮ et ˝ ̏).

myText <- "azerty"  
myText2 <- 'azerty'  
myText3 <- 'azerty uiop qsdfg hjklm'  
typeof(myText3)

## [1] "character"

Les guillemets doubles ou simples sont utiles si l’on souhaite mettre des guillemets dans notre texte. Nous pouvons également *échapper* un caractère spécial comme un guillemet grâce au signe backslash \.

myText <- "a 'ze' 'rt' y"  
myText2 <- 'a "zert" y'  
myText3 <- 'azerty uiop qsdfg hjklm'  
myText4 <- "qwerty \" azerty "  
myText5 <- "qwerty \\ azerty "

Par défaut lorsque nous créons un objet, son contenu n’est pas renvoyé par la console. Sur Internet ou dans de nombreux ouvrages nous pouvons retrouver le nom de l’objet sur une ligne pour renvoyer son contenu:

myText <- "a 'ze' 'rt' y"  
myText

## [1] "a 'ze' 'rt' y"

Dans ce livre nous n’utiliserons pas cette façon de faire et préfèrerons l’utilisation de la fonction print(), qui permet d’afficher dans la console le contenu d’un objet. Le résultat est le même mais le code est alors plus facile à lire et plus explicite sur ce qui est fait. D’autre part, nous verrons plus tard des cas où l’usage de print() sera indispensable au bon fonctionnement de nos scripts.

myText <- "a 'ze' 'rt' y"  
print(myText)

## [1] "a 'ze' 'rt' y"

nbrRep <- 5  
print(nbrRep)

## [1] 5

print(myText2)

## [1] "a \"zert\" y"

print(myText3)

## [1] "azerty uiop qsdfg hjklm"

print(myText4)

## [1] "qwerty \" azerty "

print(myText5)

## [1] "qwerty \\ azerty "

Nous pouvons également mettre des chiffres au format texte, mais il ne faut pas oublier de mettre des guillemets pour spécifier le type character ou utiliser la fonction as.character(). Une opération entre du texte et un nombre renvoie une erreur. Par exemple si l’on ajoute 10 à "5", R nous signale qu’un **argument** de la **fonction** + n’est pas de type numeric et que donc l’opération n’est pas possible. Nous ne pouvons pas non plus ajouter du texte à du texte, mais verrons plus tard comment *concaténer* deux *chaines de caractères*.

myText <- "qwerty"  
typeof(myText)

## [1] "character"

myText2 <- 5 # ceci n'est pas du texte  
typeof(myText2)

## [1] "double"

myText3 <- "5"  
typeof(myText3)

## [1] "character"

myText2 + 10

## [1] 15

as.character(5)

## [1] "5"

# myText3 + 10 # Error in myText3 + 10 : non-numeric argument to binary operator  
# "a" + "b" # Error in "a" + "b" : non-numeric argument to binary operator

Pour résumer, le type character permet la saisie de texte, nous pouvons le reconnaître grâce aux guillemets simples ou doubles.

## Le type factor

Le type factor correspond aux facteurs. Les facteurs sont un choix parmi une liste finie de possibilités. Par exemple les pays sont des facteurs car il y a une liste finie de pays dans le monde à un temps donné. Un facteur peut être défini avec la fonction factor() ou transformé en utilisant la fonction as.factor(). Comme pour les autres types de donnée nous pouvons utiliser la fonction is.factor() pour vérifier le type de donnée. Pour avoir la liste de toutes les possibilités, il existe la fonction levels() (cette fonction prendra plus de sens quand nous aurons abordé les types de conteneur de l’information).

factor01 <- factor("aaa")  
print(factor01)

## [1] aaa  
## Levels: aaa

typeof(factor01)

## [1] "integer"

is.factor(factor01)

## [1] TRUE

levels(factor01)

## [1] "aaa"

Un facteur peut être transformé en texte avec la fonction as.character() mais également en nombre avec as.numeric(). Lors de la transformation en nombre chaque facteur prend la valeur de sa position dans la liste des possibilités. Dans notre cas il n’y a qu’une seule possibilité donc la fonction as.numeric() va renvoyer 1:

factor01 <- factor("aaa")  
as.character(factor01)

## [1] "aaa"

as.numeric(factor01)

## [1] 1

## Le type logical

Le type logical correspond aux valeurs TRUE et FALSE (et NA) que nous avons déjà vu avec les opérateurs de comparaison.

aLogic <- TRUE  
print(aLogic)

## [1] TRUE

typeof(aLogic)

## [1] "logical"

is.logical(aLogic)

## [1] TRUE

aLogic + 1

## [1] 2

as.numeric(aLogic)

## [1] 1

as.character(aLogic)

## [1] "TRUE"

## A propos de NA

La valeur NA peut être utilisée pour spécifier l’absence de données ou les données manquantes. Par défaut NA est de type logical mais il peut être utilisé pour du texte, ou des nombres.

print(NA)

## [1] NA

typeof(NA)

## [1] "logical"

typeof(as.integer(NA))

## [1] "integer"

typeof(as.character(NA))

## [1] "character"

NA == TRUE

## [1] NA

NA == FALSE

## [1] NA

NA > 1

## [1] NA

NA + 1

## [1] NA

## Conclusion

Félicitations, nous sommes arrivés au bout de ce chapitre sur les types de données. Nous savons désormais :

* Reconnaître et faire des objets dans les principaux types de données
* Transformer les types de données d’un type à un autre

Ce chapitre un peu fastidieux est la base pour aborder le prochain chapitre sur les conteneurs des données.

# Les conteneurs de données

Jusqu’à présent nous avons fait des objets simples ne contenant qu’une seule valeur. Nous avons néanmoins pu voir qu’un objet avait différents attributs, comme sa valeur, mais aussi le type de donnée contenue. Maintenant nous allons voir qu’il existe différents types de conteneurs permettant de stocker plusieurs données.

## Le conteneur vector

Dans R, un vector est une combinaison de données avec la particularité que toutes les données contenues dans un vector sont du même type. Nous pouvons donc stocker plusieurs numeric ou character dans un vector, mais pas les deux. Le conteneur vector est important car c’est l’élément de base de R (R tire toute sa puissance des opérations sur les vector).

### Créer un vector

Pour créer un vector nous allons utiliser la fonction c() qui permet de combiner des éléments en un vector. Les éléments à combiner doivent être séparés par des virgules (avec un espace après chaque virgule afin de rester lisible).

miVec01 <- c(1, 2, 3, 4) # un vecteur de 4 éléments de type numeric ; double  
print(miVec01)

## [1] 1 2 3 4

typeof(miVec01)

## [1] "double"

is.vector(miVec01)

## [1] TRUE

La fonction is.vector() permet de vérifier le type de conteneur.

miVec02 <- c("a", "b", "c")   
print(miVec02)

## [1] "a" "b" "c"

typeof(miVec02)

## [1] "character"

is.vector(miVec02)

## [1] TRUE

miVec03 <- c(TRUE, FALSE, FALSE, TRUE)  
print(miVec03)

## [1] TRUE FALSE FALSE TRUE

typeof(miVec03)

## [1] "logical"

is.vector(miVec03)

## [1] TRUE

miVecNA <- c(1, NA, 3, NA, 5)  
print(miVecNA)

## [1] 1 NA 3 NA 5

typeof(miVecNA)

## [1] "double"

is.vector(miVecNA)

## [1] TRUE

miVec04 <- c(1, "a")  
print(miVec04)

## [1] "1" "a"

typeof(miVec04)

## [1] "character"

is.vector(miVec04)

## [1] TRUE

Si l’on combine différents types de données, par défaut R va chercher à transformer les éléments en un seul type. Si comme ici dans l’objet miVec04 nous avons des character et des numeric, R va transformer tous les éléments en character.

miVec05 <- c(factor("abc"), "def")  
print(miVec05)

## [1] "1" "def"

typeof(miVec05)

## [1] "character"

Ici nous combinons un élément de type factor et un élément de type character. Comme R n’accepte que des éléments de même type dans un vector, R transforme tous les éléments en type le plus simple (ici en character). Lors de la conversion de l’élément factor en character, R prend par défaut la valeur numérique associé à ce factor, c’est-à-dire 1. C’est pour cela que print(miVec05) renvoie un vector de deux éléments au format character, que sont "1" et "def". Pour transformer factor("abc") en type character, nous utiliserons as.character(factor("abc")) qui lui renvoie bien "abc".

miVec05b <- c(factor("abc"), factor("ghi"), "def")  
print(miVec05b)

## [1] "1" "1" "def"

miVec05c <- c(factor(c("abc", "ghi")), "def")  
print(miVec05c)

## [1] "1" "2" "def"

Nous pouvons voir ici la différence entre deux appels à factor pour un élément qui renvoie “1” et un appel à factor pour deux éléments qui renvoie “1” et “2” lors de la transformation en character par R.

miVec06 <- c(TRUE, "def")  
print(miVec06)

## [1] "TRUE" "def"

typeof(miVec06)

## [1] "character"

miVec07 <- c(factor("abc"), 55)  
print(miVec07)

## [1] 1 55

typeof(miVec07)

## [1] "double"

Ici miVec07 est simplifié au format numérique.

miVec08 <- c(TRUE, 55)  
print(miVec08)

## [1] 1 55

typeof(miVec08)

## [1] "double"

Dans ce dernier cas c’est le logical TRUE qui est transformé au format numérique.

Nous pouvons aussi combiner des objets existants au sein d’un vector.

miVec09 <- c(miVec02, "d", "e", "f")  
print(miVec09)

## [1] "a" "b" "c" "d" "e" "f"

miVec10 <- c("aaa", "aa", miVec09, "d", "e", "f")  
print(miVec10)

## [1] "aaa" "aa" "a" "b" "c" "d" "e" "f" "d" "e" "f"

miVec11 <- c(789, miVec01 , 564)  
print(miVec11)

## [1] 789 1 2 3 4 564

### Opérations sur un vector

Nous pouvons également effectuer des opérations sur un vector.

print(miVec01)

## [1] 1 2 3 4

miVec01 + 1

## [1] 2 3 4 5

miVec01 - 1

## [1] 0 1 2 3

miVec01 \* 2

## [1] 2 4 6 8

miVec01 /10

## [1] 0.1 0.2 0.3 0.4

Les opérations d’un vector sur un autre sont possibles, mais il faut veiller à ce que le nombre d’éléments d’un vector soit le même que l’autre, sinon R va effectuer le calcul en repartant du début du vector le plus court. Voici un exemple pour illustrer ce que R fait:

miVec12 <- c(1, 1, 1, 1, 1, 1, 1, 1, 1)  
print(miVec12)

## [1] 1 1 1 1 1 1 1 1 1

miVec13 <- c(10, 20, 30)  
print(miVec13)

## [1] 10 20 30

miVec12 + miVec13 # vecteurs de tailles différentes : attention au résultat

## [1] 11 21 31 11 21 31 11 21 31

miVec14 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)  
print(miVec14)

## [1] 10 20 30 40 50 60 70 80 90

miVec12 + miVec14 # les vecteurs sont de la même longueur

## [1] 11 21 31 41 51 61 71 81 91

miVec15 <- c(1, 1, 1, 1)  
print(miVec15)

## [1] 1 1 1 1

miVec15 + miVec13 # vecteurs de tailles différentes et non multiples

## Warning in miVec15 + miVec13: la taille d'un objet plus long n'est pas multiple  
## de la taille d'un objet plus court

## [1] 11 21 31 11

### Accéder aux valeurs d’un vector

Il souvent nécessaire de pouvoir accéder aux valeurs d’un vector, c’est à dire de récupérer une valeur ou un groupe de valeurs au sein d’un vector. Pour récupérer une valeur ou un groupe de valeurs dans un vector nous utilisons les crochets []. Entre les crochets, nous pouvons utiliser un numéro correspondant au numéro de l’élément dans le vector.

miVec20 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)  
miVec21 <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")  
print(miVec20)

## [1] 10 20 30 40 50 60 70 80 90

print(miVec21)

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"

print(miVec20[1])

## [1] 10

print(miVec21[3])

## [1] "c"

Nous pouvons aussi utiliser la combinaison de différents éléments (un autre vector).

print(miVec20[c(1, 5, 9)])

## [1] 10 50 90

print(miVec21[c(4, 3, 1)])

## [1] "d" "c" "a"

print(miVec21[c(4, 4, 3, 4, 3, 2, 5)])

## [1] "d" "d" "c" "d" "c" "b" "e"

Nous pouvons aussi sélectionner des éléments en utilisant un opérateur de comparaison ou un opérateur logique.

print(miVec20[miVec20 >= 50])

## [1] 50 60 70 80 90

print(miVec20[(miVec20 >= 50) & ((miVec20 < 80))])

## [1] 50 60 70

print(miVec20[miVec20 != 50])

## [1] 10 20 30 40 60 70 80 90

print(miVec20[miVec20 == 30])

## [1] 30

print(miVec20[(miVec20 == 30) | (miVec20 == 50)])

## [1] 30 50

print(miVec21[miVec21 == "a"])

## [1] "a"

Une autre fonctionnalité intéressante est de conditionner les éléments à sélectionner dans un vector en fonction d’un autre vector.

print(miVec21[miVec20 >= 50])

## [1] "e" "f" "g" "h" "i"

print(miVec21[(miVec20 >= 50) & ((miVec20 < 80))])

## [1] "e" "f" "g"

print(miVec21[miVec20 != 50])

## [1] "a" "b" "c" "d" "f" "g" "h" "i"

print(miVec21[miVec20 == 30])

## [1] "c"

print(miVec21[(miVec20 == 30) | (miVec20 == 50)])

## [1] "c" "e"

print(miVec21[(miVec20 == 30) | (miVec21 == "h")])

## [1] "c" "h"

Il est aussi possible d’exclure certains éléments plutôt que de les sélectionner.

print(miVec20[-1])

## [1] 20 30 40 50 60 70 80 90

print(miVec21[-5])

## [1] "a" "b" "c" "d" "f" "g" "h" "i"

print(miVec20[-c(1, 2, 5)])

## [1] 30 40 60 70 80 90

print(miVec21[-c(1, 2, 5)])

## [1] "c" "d" "f" "g" "h" "i"

Les éléments d’un vector peuvent aussi être sélectionnés sur la base d’un vector de type logical. Dans ce cas seuls les éléments avec une valeur TRUE seront sélectionnés.

miVec22 <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, TRUE)  
print(miVec21[miVec22])

## [1] "a" "b" "d" "f" "h" "i"

### Nommer les éléments d’un vector

Les éléments d’un vector peuvent être nommés pour pouvoir s’y référer par la suite et opérer une sélection. La fonction names() permet de récupérer les noms des éléments d’un vecteur.

miVec23 <- c(aaa = 10, bbb = 20, ccc = 30, ddd = 40, eee = 50)  
print(miVec23)

## aaa bbb ccc ddd eee   
## 10 20 30 40 50

print(miVec23["bbb"])

## bbb   
## 20

print(miVec23[c("bbb", "ccc", "bbb")])

## bbb ccc bbb   
## 20 30 20

names(miVec23)

## [1] "aaa" "bbb" "ccc" "ddd" "eee"

### Modifier les éléments d’un vector

Pour modifier un vecteur, nous opérons de la même façon que pour modifier un objet simple, avec le signe <- et l’élément ou les éléments à modifier entre crochets.

print(miVec21)

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"

miVec21[3] <- "zzz"  
print(miVec21)

## [1] "a" "b" "zzz" "d" "e" "f" "g" "h" "i"

miVec21[(miVec20 >= 50) & ((miVec20 < 80))] <- "qwerty"  
print(miVec21)

## [1] "a" "b" "zzz" "d" "qwerty" "qwerty" "qwerty" "h"   
## [9] "i"

print(miVec23)

## aaa bbb ccc ddd eee   
## 10 20 30 40 50

miVec23["ccc"] <- miVec23["ccc"] + 100  
print(miVec23)

## aaa bbb ccc ddd eee   
## 10 20 130 40 50

Nous pouvons aussi changer les noms associés aux éléments d’un vector.

print(miVec23)

## aaa bbb ccc ddd eee   
## 10 20 130 40 50

names(miVec23)[2] <- "bb\_bb"  
print(miVec23)

## aaa bb\_bb ccc ddd eee   
## 10 20 130 40 50

Nous pouvons faire bien plus avec un vector et reviendrons sur leur manipulations et les opérations lors du chapitre sur les fonctions.

## Le conteneur list

Le deuxième type de conteneur que nous allons introduire est le conteneur list, qui est également le deuxième conteneur après le type vector de par son importance dans la programmation avec R. Le conteneur de type list permet de stocker une **liste** d’éléments. Contrairement à ce que nous avons vu précédemment avec le type vector, les éléments du type list peuvent être différents (par exemple un vector de type numeric, puis un vector de type character). Les éléments du type list peuvent aussi être des conteneurs différents (par exemple un vector, puis une list). Le type de conteneur list prendra tout son sens lorsque nous aurons étudié les **boucles** et les **fonctions** de la famille apply.

### Créer une list

Pour créer une list nous allons utiliser la fonction list() qui prend comme argument des éléments (objets).

miList01 <- list()  
print(miList01)

## list()

miList02 <- list(5, "qwerty", c(4, 5, 6), c("a", "b", "c"))  
print(miList02)

## [[1]]  
## [1] 5  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c"

miList03 <- list(5, "qwerty", list(c(4, 5, 6), c("a", "b", "c")))  
print(miList03)

## [[1]]  
## [1] 5  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [[3]][[1]]  
## [1] 4 5 6  
##   
## [[3]][[2]]  
## [1] "a" "b" "c"

La fonction is.list() permet de tester si nous avons bien créer un objet de type list.

is.list(miList02)

## [1] TRUE

typeof(miList02)

## [1] "list"

### Accéder aux valeurs d’une list

Les éléments du conteneur list sont identifiables grâce aux double crochets [[ ]].

print(miList02)

## [[1]]  
## [1] 5  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c"

Dans l’objet miList02 de type list, il y a quatre éléments identifiables avec [[1]], [[2]], [[3]], et [[4]]. Chacun des éléments est de type vector de taille 1 et de type double pour le premier élément, de taille 1 et de type character pour le deuxième élément, de taille 3 et de type double pour le troisième élément, et de taille 3 et de type character pour le quatrième élément.

typeof(miList02)

## [1] "list"

print(miList02[[1]])

## [1] 5

typeof(miList02[[1]])

## [1] "double"

print(miList02[[2]])

## [1] "qwerty"

typeof(miList02[[2]])

## [1] "character"

print(miList02[[3]])

## [1] 4 5 6

typeof(miList02[[3]])

## [1] "double"

print(miList02[[4]])

## [1] "a" "b" "c"

typeof(miList02[[4]])

## [1] "character"

L’accès au deuxième élément du vector situé en quatrième position de la liste se fait donc avec miList02[[4]][2]. Nous utilisons un double crochet pour le quatrième élément de la list, puis un simple crochet pour le deuxième élément du vector.

print(miList02[[4]][2])

## [1] "b"

Comme une list peut contenir elle-même une ou plusieurs list, nous pouvons accéder à l’information recherchée en combinant les doubles crochets. L’objet miList04 est une list de deux éléments, les list miList02 et miList03. L’objet miList03 contient lui-même une list comme élément en troisième position. Pour accéder au premier élément du vector en première position de l’élément en troisième position du deuxième élément de la list miList04, nous pouvons utiliser miList04[[2]][[3]][[1]][1]. Il n’y a pas de limite quant à la profondeur des list mais dans la pratique il n’y que rarement besoin de faire des list de list de list.

miList04 <- list(miList02, miList03)  
print(miList04)

## [[1]]  
## [[1]][[1]]  
## [1] 5  
##   
## [[1]][[2]]  
## [1] "qwerty"  
##   
## [[1]][[3]]  
## [1] 4 5 6  
##   
## [[1]][[4]]  
## [1] "a" "b" "c"  
##   
##   
## [[2]]  
## [[2]][[1]]  
## [1] 5  
##   
## [[2]][[2]]  
## [1] "qwerty"  
##   
## [[2]][[3]]  
## [[2]][[3]][[1]]  
## [1] 4 5 6  
##   
## [[2]][[3]][[2]]  
## [1] "a" "b" "c"

print(miList04[[2]][[3]][[1]][1])

## [1] 4

Pour rendre concret l’exemple précédent, nous pouvons imaginer des espèces de foreurs de maïs (*Sesamia nonagrioides* et *Ostrinia nubilalis*), échantillonnées dans différents sites, avec différentes abondances à quatre dates. Ici nous allons donner des noms aux éléments des listes.

bddInsect <- list(Snonagrioides = list(site01 = c(12, 5, 8, 7), site02 = c(5, 23, 4, 41), site03 = c(12, 0, 0, 0)), Onubilalis = list(site01 = c(12, 1, 2, 3), site02 = c(0, 0, 0, 1), site03 = c(1, 1, 2, 3)))  
print(bddInsect)

## $Snonagrioides  
## $Snonagrioides$site01  
## [1] 12 5 8 7  
##   
## $Snonagrioides$site02  
## [1] 5 23 4 41  
##   
## $Snonagrioides$site03  
## [1] 12 0 0 0  
##   
##   
## $Onubilalis  
## $Onubilalis$site01  
## [1] 12 1 2 3  
##   
## $Onubilalis$site02  
## [1] 0 0 0 1  
##   
## $Onubilalis$site03  
## [1] 1 1 2 3

La lecture d’une ligne de code longue comme celle de la création de l’objet bddInsect est difficile à lire car la profondeur des éléments ne peut se déduire que grâce aux parenthèses. C’est pourquoi nous allons réorganiser le code pour lui donner plus de lisibilité grâce à l’**indentation**. L’indentation consiste à mettre l’information à des niveaux différents de telle manière que nous puissions rapidement identifier les différents niveaux d’un code. L’indentation se fait au moyen de la touche de tabulation du clavier. Nous reviendrons sur l’indentation avec plus de précisions lors du chapitre sur les **boucles**. Nous retiendrons pour le moment que si une ligne de code est trop longue, nous gagnons en lisibilité en passant à la ligne et que R va lire l’ensemble comme une seule ligne de code.

bddInsect <- list(  
 Snonagrioides = list(  
 site01 = c(12, 5, 8, 7),   
 site02 = c(5, 23, 4, 41),   
 site03 = c(12, 0, 0, 0)  
 ),   
 Onubilalis = list(  
 site01 = c(12, 1, 2, 3),   
 site02 = c(0, 0, 0, 1),   
 site03 = c(1, 1, 2, 3)  
 )  
)

Nous pouvons sélectionner les données d’abondance du deuxième site de la première espèce comme précédemment bddInsect[[1]][[2]], ou alternativement en utilisant les noms des éléments bddInsect$Snonagrioides$site02. Pour ce faire nous utilisons le signe $, ou alors le nom des éléments avec des guillemets simples ou doubles bddInsect[['Snonagrioides']][['site02']].

print(bddInsect[[1]][[2]])

## [1] 5 23 4 41

print(bddInsect$Snonagrioides$site02)

## [1] 5 23 4 41

print(bddInsect[['Snonagrioides']][['site02']])

## [1] 5 23 4 41

Comme pour les vecteurs nous pouvons récupérer les noms des éléments avec la fonction names().

names(bddInsect)

## [1] "Snonagrioides" "Onubilalis"

names(bddInsect[[1]])

## [1] "site01" "site02" "site03"

Lorsque nous utilisons les doubles crochets [[]] ou le signe $, R renvoie le contenu de l’élément sélectionné. Dans notre exemple les données d’abondance sont contenues sous la forme d’un vector, donc R renvoie un élément de type vector. Si nous souhaitons sélectionner un élément d’une list mais en conservant le format list, alors nous pouvons utiliser les crochets simples [].

print(bddInsect[[1]][[2]])

## [1] 5 23 4 41

typeof(bddInsect[[1]][[2]])

## [1] "double"

is.list(bddInsect[[1]][[2]])

## [1] FALSE

print(bddInsect[[1]][2])

## $site02  
## [1] 5 23 4 41

typeof(bddInsect[[1]][2])

## [1] "list"

is.list(bddInsect[[1]][2])

## [1] TRUE

L’utilisation des crochets simples [] est utile lorsque nous souhaitons récupérer plusieurs éléments d’une list. Par exemple pour sélectionner les abondances d’insectes des deux premiers sites de la première espèce, nous utiliserons bddInsect[[1]][c(1, 2)] ou alternativement bddInsect[[1]][c("site01", "site02")].

print(bddInsect[[1]][c(1, 2)])

## $site01  
## [1] 12 5 8 7  
##   
## $site02  
## [1] 5 23 4 41

print(bddInsect[[1]][c("site01", "site02")])

## $site01  
## [1] 12 5 8 7  
##   
## $site02  
## [1] 5 23 4 41

### Modification d’une list

Une list peut être modifiée de la même façon que pour le conteneur vector, c’est à dire en se référant avec des crochets à l’élément que nous souhaitons modifier.

print(miList02)

## [[1]]  
## [1] 5  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c"

miList02[[1]] <- 12  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c"

miList02[[4]] <- c("d", "e", "f")  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "d" "e" "f"

miList02[[4]] <- c("a", "b", "c", miList02[[4]], "g", "h", "i")  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"

miList02[[4]][5] <- "eee"  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 4 5 6  
##   
## [[4]]  
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"

miList02[[3]] <- miList02[[3]] \* 10 - 1  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 39 49 59  
##   
## [[4]]  
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"

miList02[[3]][2] <- miList02[[1]] \* 100  
print(miList02)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 39 1200 59  
##   
## [[4]]  
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"

print(bddInsect)

## $Snonagrioides  
## $Snonagrioides$site01  
## [1] 12 5 8 7  
##   
## $Snonagrioides$site02  
## [1] 5 23 4 41  
##   
## $Snonagrioides$site03  
## [1] 12 0 0 0  
##   
##   
## $Onubilalis  
## $Onubilalis$site01  
## [1] 12 1 2 3  
##   
## $Onubilalis$site02  
## [1] 0 0 0 1  
##   
## $Onubilalis$site03  
## [1] 1 1 2 3

bddInsect[['Snonagrioides']][['site02']] <- c(2, 4, 6, 8)  
print(bddInsect)

## $Snonagrioides  
## $Snonagrioides$site01  
## [1] 12 5 8 7  
##   
## $Snonagrioides$site02  
## [1] 2 4 6 8  
##   
## $Snonagrioides$site03  
## [1] 12 0 0 0  
##   
##   
## $Onubilalis  
## $Onubilalis$site01  
## [1] 12 1 2 3  
##   
## $Onubilalis$site02  
## [1] 0 0 0 1  
##   
## $Onubilalis$site03  
## [1] 1 1 2 3

Pour combiner deux list, il suffit d’utiliser la fonction c() que nous avions utilisée pour créer un vector.

miList0203 <- c(miList02, miList03)  
print(miList0203)

## [[1]]  
## [1] 12  
##   
## [[2]]  
## [1] "qwerty"  
##   
## [[3]]  
## [1] 39 1200 59  
##   
## [[4]]  
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"   
##   
## [[5]]  
## [1] 5  
##   
## [[6]]  
## [1] "qwerty"  
##   
## [[7]]  
## [[7]][[1]]  
## [1] 4 5 6  
##   
## [[7]][[2]]  
## [1] "a" "b" "c"

Un objet de type list peut être transformé en vector avec la fonction unlist() si le format des éléments de la list le permet (un vector ne peut contenir que des éléments du même type).

miList05 <- list("a", c("b", "c"), "d")  
print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b" "c"  
##   
## [[3]]  
## [1] "d"

miVec24 <- unlist(miList05)  
print(miVec24)

## [1] "a" "b" "c" "d"

miList06 <- list(c(1, 2, 3), c(4, 5, 6, 7), 8, 9, c(10, 11))  
print(miList06)

## [[1]]  
## [1] 1 2 3  
##   
## [[2]]  
## [1] 4 5 6 7  
##   
## [[3]]  
## [1] 8  
##   
## [[4]]  
## [1] 9  
##   
## [[5]]  
## [1] 10 11

miVec25 <- unlist(miList06)  
print(miVec25)

## [1] 1 2 3 4 5 6 7 8 9 10 11

Pour ajouter un élément à une list, nous pouvons utiliser la fonction c() ou alors les crochets doubles [[]].

print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b" "c"  
##   
## [[3]]  
## [1] "d"

miList05 <- c(miList05, "e")  
print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b" "c"  
##   
## [[3]]  
## [1] "d"  
##   
## [[4]]  
## [1] "e"

miList05[[5]] <- c("fgh", "ijk")  
print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b" "c"  
##   
## [[3]]  
## [1] "d"  
##   
## [[4]]  
## [1] "e"  
##   
## [[5]]  
## [1] "fgh" "ijk"

Pour supprimer un élément à une list, lune possibilité est d’attribuer la valeur NULL à l’élément à supprimer.

print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b" "c"  
##   
## [[3]]  
## [1] "d"  
##   
## [[4]]  
## [1] "e"  
##   
## [[5]]  
## [1] "fgh" "ijk"

miList05[[2]] <- NULL  
print(miList05)

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "d"  
##   
## [[3]]  
## [1] "e"  
##   
## [[4]]  
## [1] "fgh" "ijk"

## Le conteneur data.frame

Le conteneur data.frame peut être assimilé à un *tableau*. Il s’agit en réalité d’un cas particulier de list où tous les éléments de la list ont la même longueur.

### Créer un data.frame

Pour créer un data.frame nous allons utiliser la fonction data.frame() qui prend comme arguments les éléments du tableau que nous souhaitons créer. Les éléments sont de type vector et font tous la même taille. Nous pouvons donner un nom à chaque *colonne* (vector) de notre *tableau* (data.frame).

# création d'un data.frame   
miDf01 <- data.frame(  
 numbers = c(1, 2, 3, 4),   
 logicals = c(TRUE, TRUE, FALSE, TRUE),   
 characters = c("a", "b", "c", "d")  
)  
print(miDf01)

## numbers logicals characters  
## 1 1 TRUE a  
## 2 2 TRUE b  
## 3 3 FALSE c  
## 4 4 TRUE d

# création des vecteurs, puis du data.frame  
numbers <- c(1, 2, 3, 4)  
logicals <- c(TRUE, TRUE, FALSE, TRUE)  
characters <- c("a", "b", "c", "d")  
miDf01 <- data.frame(numbers, logicals, characters)  
print(miDf01)

## numbers logicals characters  
## 1 1 TRUE a  
## 2 2 TRUE b  
## 3 3 FALSE c  
## 4 4 TRUE d

### Accéder aux valeurs d’un data.frame

L’accès aux différentes valeurs d’un data.frame peut se faire de la même façon que pour un conteneur de type list.

print(miDf01$numbers) # vector

## [1] 1 2 3 4

print(miDf01[[1]]) # vector

## [1] 1 2 3 4

print(miDf01[1]) # list

## numbers  
## 1 1  
## 2 2  
## 3 3  
## 4 4

print(miDf01["numbers"]) # list

## numbers  
## 1 1  
## 2 2  
## 3 3  
## 4 4

print(miDf01[["numbers"]]) # vector

## [1] 1 2 3 4

Nous pouvons aussi utiliser une autre forme qui consiste à spécifier la (ou les) ligne suivie d’une virgule (et comme nous en avons pris l’habitude, d’un espace après la virgule), puis la (ou les) colonne entre crochets simples. Si l’information ligne ou colonne est omise, R affichera toutes les lignes ou toutes les colonnes. Là encore nous pouvons utiliser le numéro correspondant à un élément ou alors le nom de l’élément que nous souhaitons sélectionner.

myRow <- 2  
myCol <- 1  
print(miDf01[myRow, myCol])

## [1] 2

print(miDf01[myRow, ])

## numbers logicals characters  
## 2 2 TRUE b

print(miDf01[, myCol])

## [1] 1 2 3 4

myCol <- "numbers"  
print(miDf01[, myCol])

## [1] 1 2 3 4

Il est possible de sélectionner plusieurs lignes ou plusieurs colonnes.

print(miDf01[, c(1, 2)])

## numbers logicals  
## 1 1 TRUE  
## 2 2 TRUE  
## 3 3 FALSE  
## 4 4 TRUE

print(miDf01[c(2, 1), ])

## numbers logicals characters  
## 2 2 TRUE b  
## 1 1 TRUE a

Puisque chaque colonne est au format vector, nous pouvons également faire une sélection qui dépendra du contenu avec les opérateurs de comparaison et les opérateurs logiques.

miDfSub01 <- miDf01[miDf01$numbers > 2, ]  
print(miDfSub01)

## numbers logicals characters  
## 3 3 FALSE c  
## 4 4 TRUE d

miDfSub02 <- miDf01[(miDf01$logicals == TRUE) & (miDf01$numbers < 2), ]  
print(miDfSub02)

## numbers logicals characters  
## 1 1 TRUE a

miDfSub03 <- miDf01[(miDf01$numbers %% 2) == 0, ]  
print(miDfSub03)

## numbers logicals characters  
## 2 2 TRUE b  
## 4 4 TRUE d

miDfSub04 <- miDf01[((miDf01$numbers %% 2) == 0) | (miDf01$logicals == TRUE), ]  
print(miDfSub04)

## numbers logicals characters  
## 1 1 TRUE a  
## 2 2 TRUE b  
## 4 4 TRUE d

### Modifier un data.frame

Pour ajouter un élément à un data.frame, nous allons procéder comme pour un conteneur de type list. Il faudra veiller à ce que le nouvel élément soit de la même taille que les autres éléments de notre data.frame. Par défaut un nouvel élément dans un data.frame prend comme nom la lettre *V* suivie du numéro de la colonne. Nous pouvons changer les noms de colonne avec la fonction colnames(). Nous avons la possibilité de donner un nom aux lignes avec la fonction rownames()

newVec <- c(4, 5, 6, 7)  
miDf01[[4]] <- newVec  
print(miDf01)

## numbers logicals characters V4  
## 1 1 TRUE a 4  
## 2 2 TRUE b 5  
## 3 3 FALSE c 6  
## 4 4 TRUE d 7

print(colnames(miDf01))

## [1] "numbers" "logicals" "characters" "V4"

colnames(miDf01)[4] <- "newVec"  
print(miDf01)

## numbers logicals characters newVec  
## 1 1 TRUE a 4  
## 2 2 TRUE b 5  
## 3 3 FALSE c 6  
## 4 4 TRUE d 7

print(rownames(miDf01))

## [1] "1" "2" "3" "4"

rownames(miDf01) <- c("row1", "row2", "row3", "row4")  
print(miDf01)

## numbers logicals characters newVec  
## row1 1 TRUE a 4  
## row2 2 TRUE b 5  
## row3 3 FALSE c 6  
## row4 4 TRUE d 7

Une autre façon de d’ajouter une *colonne* à un data.frame est d’utiliser le symbole $ suivi du nom de la *colonne* à ajouter.

# création d'un vecor  
newVec2 <- c(40, 50, 60, 70)  
# ajout du vetor dans le data.frame  
miDf01$newVec2 <- newVec2  
# ou alors directement :  
miDf01$newVec2 <- c(40, 50, 60, 70)  
# affichage du data.frame  
print(miDf01)

## numbers logicals characters newVec newVec2  
## row1 1 TRUE a 4 40  
## row2 2 TRUE b 5 50  
## row3 3 FALSE c 6 60  
## row4 4 TRUE d 7 70

Comme le type de conteneur data.frame est un cas particulier de list, la sélection et la modification se fait comme pour un conteneur de type list. Comme les éléments d’un data.frame sont de type vector, la sélection et la modification des éléments d’un data.frame se fait comme pour un conteneur de type vector.

miDf01$newVec2 <- miDf01$newVec2 \* 2  
print(miDf01)

## numbers logicals characters newVec newVec2  
## row1 1 TRUE a 4 80  
## row2 2 TRUE b 5 100  
## row3 3 FALSE c 6 120  
## row4 4 TRUE d 7 140

miDf01$newVec2 + miDf01$newVec

## [1] 84 105 126 147

miDf01$newVec2[2] <- 0  
print(miDf01)

## numbers logicals characters newVec newVec2  
## row1 1 TRUE a 4 80  
## row2 2 TRUE b 5 0  
## row3 3 FALSE c 6 120  
## row4 4 TRUE d 7 140

Un vector peut être transformé en data.frame avec la fonction as.data.frame().

print(newVec2)

## [1] 40 50 60 70

print(as.data.frame(newVec2))

## newVec2  
## 1 40  
## 2 50  
## 3 60  
## 4 70

is.data.frame(newVec2)

## [1] FALSE

is.data.frame(as.data.frame(newVec2))

## [1] TRUE

## Le conteneur matrix

Le conteneur matrix peut être vu comme un vector à deux dimensions (lignes et colonnes). Il correspond à une matrice en mathématiques, et ne peut contenir qu’un seul type de données (logical, numeric, character, …).

### Créer une matrix

Pour créer une matrix nous allons tout d’abord créer un vector, puis spécifier le nombre souhaité de lignes et de colonnes dans la fonction matrix().

vecForMatrix <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)  
miMat <- matrix(vecForMatrix, nrow = 3, ncol = 4)  
print(miMat)

## [,1] [,2] [,3] [,4]  
## [1,] 1 4 7 10  
## [2,] 2 5 8 11  
## [3,] 3 6 9 12

Il n’est pas nécessaire de spécifier le nombre de lignes nrow et le nombre de colonnes ncol. Si nous utilisons l’un ou l’autre de ces arguments, R va automatiquement calculer le nombre correspondant.

miMat <- matrix(vecForMatrix, nrow = 3)  
print(miMat)

## [,1] [,2] [,3] [,4]  
## [1,] 1 4 7 10  
## [2,] 2 5 8 11  
## [3,] 3 6 9 12

miMat <- matrix(vecForMatrix, ncol = 4)  
print(miMat)

## [,1] [,2] [,3] [,4]  
## [1,] 1 4 7 10  
## [2,] 2 5 8 11  
## [3,] 3 6 9 12

Nous observons que les différents éléments du vector initial sont renseignés par colonne. Si nous souhaitons renseigner la matrix par lignes alors il faut donner la valeur TRUE à l’argument byrow. Comme toujours nous pourrons consulter l’aide avec la fonction ? (?matrix) pour voir les différents arguments.

miMat <- matrix(vecForMatrix, nrow = 3, byrow = TRUE)  
print(miMat)

## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
## [2,] 5 6 7 8  
## [3,] 9 10 11 12

Nous pouvons également donner un nom aux lignes et aux colonnes de notre matrix lors de sa création avec l’argument dimnames qui prend comme valeur une list de deux éléments : le nom des lignes puis le nom des colonnes. Nous pouvons aussi changer le nom des lignes et des colonnes a posteriori avec les fonctions rownames() et colnames().

miMat <- matrix(  
 vecForMatrix,   
 nrow = 3,   
 byrow = TRUE,   
 dimnames = list(c("r1", "r2", "r3"), c("c1", "c2", "c3", "c4"))  
)  
print(miMat)

## c1 c2 c3 c4  
## r1 1 2 3 4  
## r2 5 6 7 8  
## r3 9 10 11 12

colnames(miMat) <- c("col1", "col2", "col3", "col4")  
rownames(miMat) <- c("row1", "row2", "row3")  
print(miMat)

## col1 col2 col3 col4  
## row1 1 2 3 4  
## row2 5 6 7 8  
## row3 9 10 11 12

Il est possible de créer une matrix à partir d’un data.frame avec la fonction as.matrix() sous réserve que le data.frame ne contienne que des éléments de même type (par exemple des éléments de type numeric).

vecForMat01 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)  
vecForMat02 <- vecForMat01 \* 10  
vecForMat03 <- vecForMat01 / 10  
dfForMat <- data.frame(vecForMat01, vecForMat02, vecForMat03)  
print(dfForMat)

## vecForMat01 vecForMat02 vecForMat03  
## 1 1 10 0.1  
## 2 2 20 0.2  
## 3 3 30 0.3  
## 4 4 40 0.4  
## 5 5 50 0.5  
## 6 6 60 0.6  
## 7 7 70 0.7  
## 8 8 80 0.8  
## 9 9 90 0.9  
## 10 10 100 1.0  
## 11 11 110 1.1  
## 12 12 120 1.2

is.matrix(dfForMat)

## [1] FALSE

as.matrix(dfForMat)

## vecForMat01 vecForMat02 vecForMat03  
## [1,] 1 10 0.1  
## [2,] 2 20 0.2  
## [3,] 3 30 0.3  
## [4,] 4 40 0.4  
## [5,] 5 50 0.5  
## [6,] 6 60 0.6  
## [7,] 7 70 0.7  
## [8,] 8 80 0.8  
## [9,] 9 90 0.9  
## [10,] 10 100 1.0  
## [11,] 11 110 1.1  
## [12,] 12 120 1.2

is.matrix(as.matrix(dfForMat))

## [1] TRUE

Nous pouvons aussi créer une matrix à partir d’un vector avec la fonction as.matrix() (matrice d’une seule colonne).

as.matrix(vecForMat01)

## [,1]  
## [1,] 1  
## [2,] 2  
## [3,] 3  
## [4,] 4  
## [5,] 5  
## [6,] 6  
## [7,] 7  
## [8,] 8  
## [9,] 9  
## [10,] 10  
## [11,] 11  
## [12,] 12

### Manipuler et faire des opérations sur une matrix

Toutes les opérations terme à terme sont possibles sur les matrix.

# opérations terme à terme  
miMat01 <- matrix(vecForMat01, ncol = 3)  
miVecOp <- c(1, 10, 100, 1000)  
miMat01 \* miVecOp

## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 20 60 100  
## [3,] 300 700 1100  
## [4,] 4000 8000 12000

miMat01 + miVecOp

## [,1] [,2] [,3]  
## [1,] 2 6 10  
## [2,] 12 16 20  
## [3,] 103 107 111  
## [4,] 1004 1008 1012

miMat01 / miMat01

## [,1] [,2] [,3]  
## [1,] 1 1 1  
## [2,] 1 1 1  
## [3,] 1 1 1  
## [4,] 1 1 1

miMat01 - 10

## [,1] [,2] [,3]  
## [1,] -9 -5 -1  
## [2,] -8 -4 0  
## [3,] -7 -3 1  
## [4,] -6 -2 2

Pour effectuer des opérations algébriques nous pouvons utiliser la fonction %\*%.

# opérations algébriques  
miVecConf <- c(1, 10, 100)  
miMat01 %\*% miVecConf

## [,1]  
## [1,] 951  
## [2,] 1062  
## [3,] 1173  
## [4,] 1284

miMat02 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)  
print(miMat02)

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

miMat02 %\*% miMat02

## [,1] [,2] [,3]  
## [1,] 30 66 102  
## [2,] 36 81 126  
## [3,] 42 96 150

La diagonale d’une matrix peut être obtenue avec la fonction diag(), et le déterminant d’une matrix avec la fonction det().

print(miMat02)

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

diag(miMat02)

## [1] 1 5 9

det(miMat02)

## [1] 0

Il est souvent utile de pouvoir transposer une matrix (colonnes en lignes ou lignes en colonnes). Pour cela il existe les fonctions aperm() ou t(). la fonction t() est plus générique et fonctionne aussi sur les data.frame.

aperm(miMat01)

## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
## [2,] 5 6 7 8  
## [3,] 9 10 11 12

t(miMat01)

## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
## [2,] 5 6 7 8  
## [3,] 9 10 11 12

### Accéder aux éléments d’une matrix

Comme pour un data.frame, nous pouvons accéder aux éléments d’une matrice en spécifiant un numéro de ligne et un numéro de colonne entre crochets simples [ ], et séparés par une virgule. Si i est le numéro de ligne et j le numéro de colonne, alors miMat01[i, j] renvoie l’élément situé à la ligne i et à la colonne j. miMat01[i, ] renvoie tous les éléments de la ligne i, et miMat01[, j] tous les éléments de la colonne j. Les sélections multiples sont possibles. Nous pouvons également accéder à un élément en fonction de sa position dans la matrice entre crochets simples [ ] en comptant par colonne puis par ligne. Dans notre exemple la valeur du dixième élément est 10.

i <- 2  
j <- 1  
print(miMat01[i, j])

## [1] 2

print(miMat01[i, ])

## [1] 2 6 10

print(miMat01[, j])

## [1] 1 2 3 4

print(miMat01[c(1, 2), c(2, 3)])

## [,1] [,2]  
## [1,] 5 9  
## [2,] 6 10

print(miMat01[10])

## [1] 10

## Le conteneur array

Le conteneur de type array est une généralisation du conteneur de type matrix. Là où le type matrix a deux dimensions (les lignes et les colonnes), le type array a un nombre indéfini de dimensions. Nous pouvons connaître le nombre de dimensions d’un array (et donc d’une matrix) avec la fonction dim().

dim(miMat01)

## [1] 4 3

### Créer un array

La création d’un array est similaire à celle d’une matrix avec une dimension supplémentaire.

miVecArr <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)  
miArray <- array(miVecArr, dim = c(3, 3, 2))  
print(miArray)

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

dim(miArray)

## [1] 3 3 2

is.array(miArray)

## [1] TRUE

miVecArr02 <- 10 \* miVecArr  
miArray02 <- array(c(miVecArr, miVecArr02), dim = c(3, 3, 2))  
print(miArray02)

## , , 1  
##   
## [,1] [,2] [,3]  
## [1,] 1 4 7  
## [2,] 2 5 8  
## [3,] 3 6 9  
##   
## , , 2  
##   
## [,1] [,2] [,3]  
## [1,] 10 40 70  
## [2,] 20 50 80  
## [3,] 30 60 90

dim(miArray02)

## [1] 3 3 2

is.array(miArray02)

## [1] TRUE

Nous pouvons donner des noms aux lignes et aux colonnes, mais aussi aux autres dimensions.

miArray02 <- array(  
 c(miVecArr, miVecArr02),   
 dim = c(3, 3, 2),   
 dimnames = list(  
 c("r1", "r2", "r3"),   
 c("c1", "c2", "c3"),   
 c("matrix1", "matrix2")  
 )  
)  
print(miArray02)

## , , matrix1  
##   
## c1 c2 c3  
## r1 1 4 7  
## r2 2 5 8  
## r3 3 6 9  
##   
## , , matrix2  
##   
## c1 c2 c3  
## r1 10 40 70  
## r2 20 50 80  
## r3 30 60 90

### Manipuler un array

La manipulation d’un array se fait de la même façon que pour une matrix. Pour accéder aux différents éléments d’un array, il suffit de spécifier la ligne i, la colonne j, et dans notre exemple, la matrix k.

i <- 2  
j <- 1  
k <- 1  
print(miArray02[i, j, k])

## [1] 2

print(miArray02[, j, k])

## r1 r2 r3   
## 1 2 3

print(miArray02[i, , k])

## c1 c2 c3   
## 2 5 8

print(miArray02[i, j, ])

## matrix1 matrix2   
## 2 20

## Conclusion

Félicitations ! Nous connaissons à présent les principaux types d’objets que nous allons utiliser avec R. Un objet se caractérise par ses attributs :

* le type de conteneur (vector, data.frame, matrix, array)
* le type de contenu de chacun des éléments (numeric, logical, character, …)
* la valeur de chacun des éléments (5, “qwerty”, TRUE, …)

Tous ces objets sont stockés temporairement dans l’environnement global de R (dans la RAM de notre ordinateur). Le prochain chapitre va traiter des fonctions, et mettra en lumière un des aspects qui rend R si puissant pour analyser et gérer nos données.

# Les fonctions

## Qu’est-ce qu’une fonction

Avec ce chapitre nous allons avoir un premier aperçu de la puissance de R grâce aux fonctions. Une fonction est un ensemble de lignes de code permettant d’exécuter une tâche particulière. Nous avons vu de nombreuses fonctions lors des précédents chapitres, la plus simple étant la fonction + permettant d’ajouter deux nombres entre eux, ou d’autres plus complexes comme c() ou data.frame() permettant de créer un vector ou un data.frame. Dans tous les cas une fonction se reconnait grâce aux parenthèses qui la suive dans laquelle nous allons renseigner des **arguments**. Les arguments correspondent aux informations que nous souhaitons transmettre à notre fonction pour qu’elle exécute la tâche que nous souhaitons réaliser.

Pour les fonctions les plus simples comme +, les parenthèses ont été supprimées pour que le code soit plus facile à lire, mais il s’agit bien d’une fonction qui peut s’utiliser avec des parenthèses si nous utilisons le signe + entre guillemets. Les arguments sont les nombres que nous souhaitons ajouter.

5 + 2

## [1] 7

'+'(5, 2)

## [1] 7

Dans ce chapitre nous allons nous focaliser sur les fonctions les plus courantes de façon à ce que ce chapitre soit consultable comme un dictionnaire. Il ne s’agit donc pas de tout apprendre par cœur mais bien de savoir que ces fonctions existent et de pouvoir consulter ce chapitre plus tard comme référence. Avec le temps et la pratique nous finirons par les connaître par cœur ! Il y a plus de 1000 fonctions à ce jour dans la version de base de R, et plus de 10000 packages complémentaires pouvant être installés, chacun contenant plusieurs dizaines de fonctions. Avant de nous lancer dans l’écriture d’une nouvelle fonction, il faudra toujours vérifier qu’elle n’existe pas déjà. Une bonne pratique pour un code durable est de privilégier les fonctions fournies de base avec R. R vient en effet avec une série de packages (Table 4). La liste à jour des packages fournis systématiquement avec R peut être consultée sur le site web de R dans la partie FAQ (<https://cran.r-project.org/doc/FAQ/R-FAQ.html#Add_002don-packages-in-R>).

Table 4: R et ses packages de base.

|  |  |
| --- | --- |
| package | description |
| base | Base R functions (and datasets before R 2.0.0). |
| compiler | R byte code compiler (added in R 2.13.0). |
| datasets | Base R datasets (added in R 2.0.0). |
| grDevices | Graphics devices for base and grid graphics (added in R 2.0.0). |
| graphics | R functions for base graphics. |
| grid | A rewrite of the graphics layout capabilities, plus some support for interaction. |
| methods | Formally defined methods and classes for R objects, plus other programming tools, as described in the Green Book. |
| parallel | Support for parallel computation, including by forking and by sockets, and random-number generation (added in R 2.14.0). |
| splines | Regression spline functions and classes. |
| stats | R statistical functions. |
| stats4 | Statistical functions using S4 classes. |
| tcltk | Interface and language bindings to Tcl/Tk GUI elements. |
| tools | Tools for package development and administration. |
| utils | R utility functions. |

## Les fonctions les plus courantes

Pour travailler avec les fonctions nous allons utiliser le jeu de données iris qui est inclus avec la version de base de R et qui correspond à la longueur et à la largeur des sépales et des pétales de différentes espèces d’iris. Le jeu de données est sous la forme d’un data.frame de 5 colonnes et de 150 lignes. Pour plus d’information sur le jeu de données iris nous pouvons consulter la documentation de R avec la fonction help(iris) (ou ?iris). L’accès à la documentation est l’objet de la section ci-dessous.

### L’accès à la documentation

#### help()

La fonction indispensable de R est celle permettant d’accéder à la documentation. Toutes les fonctions et tous les jeux de données de R possèdent une documentation. Nous pouvons accéder à la documentation avec la fonction help() ou en utilisant le raccourci ?.

help(matrix) # équivalent à ?matrix

La documentation est toujours structurée de la même manière. Tout d’abord nous avons le nom de la fonction recherchée (matrix), suivie entre accolades par le nom du package R dont la fonction dépend. Nous verrons comment installer des packages additionnels plus tard. Pour l’instant nous disposons de ceux fournis avec la version de base de R. Ici nous pouvons voir que la fonction matrix() dépend du package base (Table 4).

Ensuite nous pouvons voir le libellé de la fonction (Matrices), suivi des paragraphes Description, Usage, et Arguments. Parfois vient s’ajouter les paragraphes Details, Note, References, et See also. Le dernier paragraphe est Examples. La dernière ligne de la documentation permet de revenir à l’index du package dont dépend la fonction consultée.

Nous pouvons voir que le paragraphe Description indique ce que fait la fonction. Dans le cas de help(matrix), il y a trois fonctions qui sont présentées : matrix(), as.matrix(), et is.matrix().

# Description  
# matrix creates a matrix from the given set of values.  
# as.matrix attempts to turn its argument into a matrix.  
# is.matrix tests if its argument is a (strict) matrix.

Le paragraphe Usage explique comment utiliser la fonction et quels sont les valeurs par défaut éventuelles pour chacun des paramètres.

# Usage  
# matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,  
# dimnames = NULL)

La fonction matrix() peut prendre 5 arguments : data, nrow, ncol, byrow, et dimnames. Nous pouvons voir que par défaut une matrix sera composée d’une seule ligne et d’une seule colonne, et que les informations seront renseignées par colonne.

Le paragraphe Arguments détaille les valeurs et le type de conteneur de chacun des arguments de notre fonction. Par exemple nous pouvons voir que l’argument dimnames doit être de type list. C’est pourquoi nous avons utilisé ce format lors de la section sur les [matrix](#l014matrix).

# Arguments  
# data an optional data vector (including a list or expression vector).   
# Non-atomic classed R objects are coerced by as.vector and all   
# attributes discarded.  
# nrow the desired number of rows.  
# ncol the desired number of columns.  
# byrow logical. If FALSE (the default) the matrix is filled by columns,   
# otherwise the matrix is filled by rows.  
# dimnames A dimnames attribute for the matrix: NULL or a list of length 2   
# giving the row and column names respectively. An empty list is   
# treated as NULL, and a list of length one as row names. The   
# list can be named, and the list names will be used as names for   
# the dimensions.

Le paragraphe Details apporte des éléments complémentaires sur la fonction. Le paragraphe Examples procure des exemples reproductibles dans la console.

## Example of setting row and column names  
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,  
 dimnames = list(c("row1", "row2"),  
 c("C.1", "C.2", "C.3")))  
mdat

## C.1 C.2 C.3  
## row1 1 2 3  
## row2 11 12 13

Le nom des arguments n’est pas nécessaire pour qu’une fonction soit correctement interprétée par R. Néanmoins par soucis de clarté il est préférable d’utiliser le nom complet des arguments suivi du signe = pour que le code soit plus lisible.

# bon exemple  
mdat <- matrix(  
 c(1, 2, 3, 11, 12, 13),   
 nrow = 2, ncol = 3,   
 byrow = TRUE)  
# mauvais exemple  
mdat <- matrix(  
 c(1, 2, 3, 11, 12, 13),   
 nr = 2, nc = 3,   
 byr = T)  
# très mauvais exemple  
mdat <- matrix(c(1,2,3,11,12,13),2,3,T)

Le résultat de ces trois exemples est identique pour R. Dans le premier cas le code est facile à lire et à comprendre. Nous avons utilisé le nom complet des arguments, des espaces après les virgules, et des sauts de lignes pour séparer les arguments et faire en sorte que chaque ligne ne soit pas trop longue. Il sera facilement compréhensible par d’autres personnes appelées à lire notre code. Le deuxième exemple fait appel à des noms d’arguments incomplets. Il reste fonctionnel, mais porte à confusion. R est un langage permissif. Dans ce cas il va lire le nom des arguments, et utiliser le nom qui s’en rapproche le plus. Dans le troisième exemple le code est là aussi fonctionnel. Néanmoins il est difficile à lire car il n’y a ni espace, ni nom des arguments, et la valeur TRUE est remplacée par T. C’est bien sûr le premier exemple qui doit être utilisé pour améliorer la lisibilité et la transparence des analyses.

#### help.search()

La fonction help.search() ou ?? permet de rechercher une expression dans l’ensemble de la documentation. Elle est utile lorsque l’on cherche une fonctionnalité sans connaître le nom de la fonction sous R.

help.search("average")

La fonction help.search() renvoie vers une page contenant la liste des pages où l’expression a été retrouvée sous la forme nom-du-package::nom-de-la-fonction.

### Visualiser les données

#### str()

La fonction str() permet de visualiser la structure interne d’un objet, comme indiqué dans la documentation que nous pouvons consulter avec help(str).

str(iris)

## 'data.frame': 150 obs. of 5 variables:  
## $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
## $ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

La fonction str() renvoie le type d’objet (data.frame), le nombre d’observations (150), le nombre de variables (5), le nom de chacune des variables (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, et Species), le type de chacune des variables (num, Factor), et les premières valeurs de chacune des variables. C’est une fonction utile pour avoir un aperçu d’un jeu de données, mais aussi pour contrôler que les données sont du type voulu avant de procéder à des analyses statistiques.

#### head() et tail()

La fonction head() renvoie les premières valeurs d’un objet, et la fonction tail() les dernières valeurs d’un objet. Par défaut six valeurs sont retournées (l’argument n contrôle le nombre de valeurs à retourner).

head(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

tail(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 145 6.7 3.3 5.7 2.5 virginica  
## 146 6.7 3.0 5.2 2.3 virginica  
## 147 6.3 2.5 5.0 1.9 virginica  
## 148 6.5 3.0 5.2 2.0 virginica  
## 149 6.2 3.4 5.4 2.3 virginica  
## 150 5.9 3.0 5.1 1.8 virginica

head(iris, n = 2)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa

#### names()

Nous avons déjà vu la fonction names() qui permet à la fois de connaître le nom des éléments d’un objet, mais aussi d’assigner des noms aux éléments d’un objet comme une [matrix](#l014matrix), une [list](#l014list) ou un [data.frame](#l014dataframe).

names(iris)

## [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

irisCopy <- iris  
names(irisCopy) <- c("a", "b", "c", "d", "e")  
names(irisCopy)

## [1] "a" "b" "c" "d" "e"

#### cat() et print()

La fonction cat() permet d’afficher le contenu d’un objet alors que la fonction print() retourne la valeur d’un objet (avec la possibilité d’effectuer des opérations comme avec l’argument digits qui va permettre de fixer le nombre de chiffres significatifs).

cat(names(iris))

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

print(names(iris))

## [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

cat(iris[1, 1])

## 5.1

print(iris[1, 1])

## [1] 5.1

print(iris[1, 1], digits = 0)

## [1] 5

### Manipuler les données

#### rank()

La fonction rank() renvoie pour un ensemble d’éléments le numéro de la position de chacun des éléments. En cas d’éléments de même valeur, l’argument ties.method permet de faire un choix sur le classement. Comme pour toutes les fonctions, les détails sont présents dans la documentation. Le recours à la documentation doit devenir un réflexe pour chaque nouvelle fonction utilisée.

vecManip <- c(10, 20, 30, 70, 60, 50, 40)  
rank(vecManip)

## [1] 1 2 3 7 6 5 4

vecManip2 <- c(10, 20, 30, 10, 50, 10, 40)  
rank(vecManip2)

## [1] 2 4 5 2 7 2 6

rank(vecManip2, ties.method = "first")

## [1] 1 4 5 2 7 3 6

rank(vecManip2, ties.method = "min")

## [1] 1 4 5 1 7 1 6

print(iris[, 1])

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1  
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0  
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5  
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1  
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5  
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3  
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2  
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8  
## [145] 6.7 6.7 6.3 6.5 6.2 5.9

rank(iris[, 1], ties.method = "average")

## [1] 37.0 19.5 10.5 7.5 27.5 49.5 7.5 27.5 3.0 19.5 49.5 14.0  
## [13] 14.0 1.0 77.0 69.5 49.5 37.0 69.5 37.0 49.5 37.0 7.5 37.0  
## [25] 14.0 27.5 27.5 43.5 43.5 10.5 14.0 49.5 43.5 56.0 19.5 27.5  
## [37] 56.0 19.5 3.0 37.0 27.5 5.0 3.0 27.5 37.0 14.0 37.0 7.5  
## [49] 46.0 27.5 138.0 112.0 135.5 56.0 118.0 69.5 104.0 19.5 121.5 43.5  
## [61] 27.5 82.0 86.5 92.5 62.5 126.5 62.5 77.0 97.5 62.5 82.0 92.5  
## [73] 104.0 92.5 112.0 121.5 132.0 126.5 86.5 69.5 56.0 56.0 77.0 86.5  
## [85] 49.5 86.5 126.5 104.0 62.5 56.0 56.0 92.5 77.0 27.5 62.5 69.5  
## [97] 69.5 97.5 37.0 69.5 104.0 77.0 139.0 104.0 118.0 145.0 19.5 143.0  
## [109] 126.5 141.0 118.0 112.0 132.0 69.5 77.0 112.0 118.0 147.5 147.5 86.5  
## [121] 135.5 62.5 147.5 104.0 126.5 141.0 97.5 92.5 112.0 141.0 144.0 150.0  
## [133] 112.0 104.0 92.5 147.5 104.0 112.0 86.5 135.5 126.5 135.5 77.0 132.0  
## [145] 126.5 126.5 104.0 118.0 97.5 82.0

# help(rank)  
# ...  
# Usage  
# rank(x, na.last = TRUE,  
# ties.method = c("average", "first", "last",   
# "random", "max", "min"))

#### order()

La fonction order() retourne le numéro du réarrangement des éléments en fonction de leur position. Elle est très utile par exemple pour trier un data.frame en fonction d’une colonne.

print(vecManip2)

## [1] 10 20 30 10 50 10 40

rank(vecManip2)

## [1] 2 4 5 2 7 2 6

# il y a 3 éléments en première position : 1, 4 et 6 (10)  
# qui vont prendre avec rank la valeur 2. Puis vient  
# l'élément 2 (20) qui prend avec rank la valeur 4, ...  
order(vecManip2)

## [1] 1 4 6 2 3 7 5

# le premier élément est en position 1, le deuxième  
# élément en position 4, le troisième en position 6,   
# le quatrième en position 2, ...  
print(iris[, 1])

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1  
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0  
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5  
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1  
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5  
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3  
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2  
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8  
## [145] 6.7 6.7 6.3 6.5 6.2 5.9

rank(iris[, 1])

## [1] 37.0 19.5 10.5 7.5 27.5 49.5 7.5 27.5 3.0 19.5 49.5 14.0  
## [13] 14.0 1.0 77.0 69.5 49.5 37.0 69.5 37.0 49.5 37.0 7.5 37.0  
## [25] 14.0 27.5 27.5 43.5 43.5 10.5 14.0 49.5 43.5 56.0 19.5 27.5  
## [37] 56.0 19.5 3.0 37.0 27.5 5.0 3.0 27.5 37.0 14.0 37.0 7.5  
## [49] 46.0 27.5 138.0 112.0 135.5 56.0 118.0 69.5 104.0 19.5 121.5 43.5  
## [61] 27.5 82.0 86.5 92.5 62.5 126.5 62.5 77.0 97.5 62.5 82.0 92.5  
## [73] 104.0 92.5 112.0 121.5 132.0 126.5 86.5 69.5 56.0 56.0 77.0 86.5  
## [85] 49.5 86.5 126.5 104.0 62.5 56.0 56.0 92.5 77.0 27.5 62.5 69.5  
## [97] 69.5 97.5 37.0 69.5 104.0 77.0 139.0 104.0 118.0 145.0 19.5 143.0  
## [109] 126.5 141.0 118.0 112.0 132.0 69.5 77.0 112.0 118.0 147.5 147.5 86.5  
## [121] 135.5 62.5 147.5 104.0 126.5 141.0 97.5 92.5 112.0 141.0 144.0 150.0  
## [133] 112.0 104.0 92.5 147.5 104.0 112.0 86.5 135.5 126.5 135.5 77.0 132.0  
## [145] 126.5 126.5 104.0 118.0 97.5 82.0

order(iris[, 1])

## [1] 14 9 39 43 42 4 7 23 48 3 30 12 13 25 31 46 2 10  
## [19] 35 38 58 107 5 8 26 27 36 41 44 50 61 94 1 18 20 22  
## [37] 24 40 45 47 99 28 29 33 60 49 6 11 17 21 32 85 34 37  
## [55] 54 81 82 90 91 65 67 70 89 95 122 16 19 56 80 96 97 100  
## [73] 114 15 68 83 93 102 115 143 62 71 150 63 79 84 86 120 139 64  
## [91] 72 74 92 128 135 69 98 127 149 57 73 88 101 104 124 134 137 147  
## [109] 52 75 112 116 129 133 138 55 105 111 117 148 59 76 66 78 87 109  
## [127] 125 141 145 146 77 113 144 53 121 140 142 51 103 110 126 130 108 131  
## [145] 106 118 119 123 136 132

head(iris, n = 10)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa  
## 7 4.6 3.4 1.4 0.3 setosa  
## 8 5.0 3.4 1.5 0.2 setosa  
## 9 4.4 2.9 1.4 0.2 setosa  
## 10 4.9 3.1 1.5 0.1 setosa

# tri du data.frame iris en fonction de la colonne 1 :  
head(iris[order(iris[, 1]),], n = 10)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 14 4.3 3.0 1.1 0.1 setosa  
## 9 4.4 2.9 1.4 0.2 setosa  
## 39 4.4 3.0 1.3 0.2 setosa  
## 43 4.4 3.2 1.3 0.2 setosa  
## 42 4.5 2.3 1.3 0.3 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 7 4.6 3.4 1.4 0.3 setosa  
## 23 4.6 3.6 1.0 0.2 setosa  
## 48 4.6 3.2 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa

#### sort()

La fonction sort() permet de trier les éléments d’un objet. Elle ne permet pas de trier selon plusieurs variables comme c’est le cas avec order().

print(vecManip2)

## [1] 10 20 30 10 50 10 40

sort(vecManip2)

## [1] 10 10 10 20 30 40 50

print(iris[, 1])

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1  
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0  
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5  
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1  
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5  
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3  
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2  
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8  
## [145] 6.7 6.7 6.3 6.5 6.2 5.9

sort(iris[, 1])

## [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8 4.9 4.9  
## [19] 4.9 4.9 4.9 4.9 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.1 5.1 5.1 5.1  
## [37] 5.1 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4 5.4 5.5 5.5  
## [55] 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6 5.6 5.6 5.6 5.7 5.7 5.7 5.7 5.7 5.7 5.7  
## [73] 5.7 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.9 5.9 5.9 6.0 6.0 6.0 6.0 6.0 6.0 6.1  
## [91] 6.1 6.1 6.1 6.1 6.1 6.2 6.2 6.2 6.2 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3  
## [109] 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5 6.5 6.6 6.6 6.7 6.7 6.7 6.7  
## [127] 6.7 6.7 6.7 6.7 6.8 6.8 6.8 6.9 6.9 6.9 6.9 7.0 7.1 7.2 7.2 7.2 7.3 7.4  
## [145] 7.6 7.7 7.7 7.7 7.7 7.9

#### append()

Cette fonction permet d’ajouter un élément à un vector à une position déterminée par l’argument after. Cette fonction est aussi plus rapide que son alternative consistant à utiliser la fonction [c()](#l014vector). Cette différence sera importante pour les gros jeux de données.

print(vecManip2)

## [1] 10 20 30 10 50 10 40

append(vecManip2, 5)

## [1] 10 20 30 10 50 10 40 5

append(vecManip2, 5, after = 2)

## [1] 10 20 5 30 10 50 10 40

#### cbind() et rbind()

Les fonctions cbind() et rbind() permettent de combiner des éléments par colonne ou par ligne.

cbind(vecManip2, vecManip2)

## vecManip2 vecManip2  
## [1,] 10 10  
## [2,] 20 20  
## [3,] 30 30  
## [4,] 10 10  
## [5,] 50 50  
## [6,] 10 10  
## [7,] 40 40

rbind(vecManip2, vecManip2)

## [,1] [,2] [,3] [,4] [,5] [,6] [,7]  
## vecManip2 10 20 30 10 50 10 40  
## vecManip2 10 20 30 10 50 10 40

#### paste() et paste0()

Voilà deux fonctions que nous allons beaucoup utiliser par la suite. Les fonctions paste() et paste0() permettent de concaténer des chaines de caractère. La fonction paste0() est équivalente à paste() sans proposer de séparateur entre les éléments à concaténer. Elle est aussi plus rapide.

paste(1, "a")

## [1] "1 a"

paste0(1, "a")

## [1] "1a"

paste(1, "a", sep = "\_")

## [1] "1\_a"

paste0("prefix\_", vecManip2, "\_suffix")

## [1] "prefix\_10\_suffix" "prefix\_20\_suffix" "prefix\_30\_suffix" "prefix\_10\_suffix"  
## [5] "prefix\_50\_suffix" "prefix\_10\_suffix" "prefix\_40\_suffix"

paste(vecManip2, rank(vecManip2), sep = "\_")

## [1] "10\_2" "20\_4" "30\_5" "10\_2" "50\_7" "10\_2" "40\_6"

#### rev()

La fonction rev() renvoie les éléments d’un objet dans l’ordre inverse.

print(vecManip2)

## [1] 10 20 30 10 50 10 40

rev(vecManip2)

## [1] 40 10 50 10 30 20 10

#### %in%

La fonction %in% peut être assimilée à un [opérateur de comparaison](#l011opcomp). Cette fonction prend deux objets comme arguments et renvoie TRUE ou FALSE pour chacun des éléments du premier objet en fonction de leur présence ou absence dans le second objet. Pour accéder à la documentation de la fonction, il faut utiliser des guillemets help('%in%').

print(vecManip)

## [1] 10 20 30 70 60 50 40

print(vecManip2)

## [1] 10 20 30 10 50 10 40

vecManip %in% vecManip2

## [1] TRUE TRUE TRUE FALSE FALSE TRUE TRUE

vecManip2 %in% vecManip

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

### Fonctions mathématiques

Nous avons déjà vu les fonctions +, -, \*, /, ^, %% et autres [opérateurs arithmétiques](#l011opari). R possède également les fonctions mathématiques de base comme exponentielle exp(), racine carrée sqrt(), valeur absolue abs(), sinus sin(), cosinus cos(), tangente tan(), logarithme népérien log(), logarithme décimal log10(), arc cosinus acos(), arc sinus asin(), et arc tangente atan().

print(vecManip2)

## [1] 10 20 30 10 50 10 40

exp(vecManip2)

## [1] 2.202647e+04 4.851652e+08 1.068647e+13 2.202647e+04 5.184706e+21  
## [6] 2.202647e+04 2.353853e+17

sqrt(vecManip2)

## [1] 3.162278 4.472136 5.477226 3.162278 7.071068 3.162278 6.324555

abs(-vecManip2)

## [1] 10 20 30 10 50 10 40

sin(vecManip2)

## [1] -0.5440211 0.9129453 -0.9880316 -0.5440211 -0.2623749 -0.5440211 0.7451132

cos(vecManip2)

## [1] -0.8390715 0.4080821 0.1542514 -0.8390715 0.9649660 -0.8390715 -0.6669381

tan(vecManip2)

## [1] 0.6483608 2.2371609 -6.4053312 0.6483608 -0.2719006 0.6483608 -1.1172149

log(vecManip2)

## [1] 2.302585 2.995732 3.401197 2.302585 3.912023 2.302585 3.688879

log10(vecManip2)

## [1] 1.000000 1.301030 1.477121 1.000000 1.698970 1.000000 1.602060

acos(vecManip2/100)

## [1] 1.470629 1.369438 1.266104 1.470629 1.047198 1.470629 1.159279

asin(vecManip2/100)

## [1] 0.1001674 0.2013579 0.3046927 0.1001674 0.5235988 0.1001674 0.4115168

atan(vecManip2/100)

## [1] 0.09966865 0.19739556 0.29145679 0.09966865 0.46364761 0.09966865 0.38050638

### Statistiques descriptives

Nous pouvons également effectuer des statistiques descriptives très simplement à partir d’un jeu de données.

#### mean()

La fonction mean() renvoie la moyenne. Pour ignorer les valeurs manquantes NA, il faut donner la valeur TRUE à l’argument na.rm().

mean(iris[, 1])

## [1] 5.843333

vecManip3 <- c(1, 5, 6, 8, NA, 45, NA, 14)  
mean(vecManip3)

## [1] NA

mean(vecManip3, na.rm = TRUE)

## [1] 13.16667

#### sd()

La fonction sd() renvoie l’écart type.

sd(iris[, 1])

## [1] 0.8280661

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

sd(vecManip3)

## [1] NA

sd(vecManip3, na.rm = TRUE)

## [1] 16.16684

#### max() et min()

La fonction max() renvoie la valeur maximale et min() la valeur minimale.

max(iris[, 1])

## [1] 7.9

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

max(vecManip3)

## [1] NA

max(vecManip3, na.rm = TRUE)

## [1] 45

min(iris[, 1])

## [1] 4.3

min(vecManip3)

## [1] NA

min(vecManip3, na.rm = TRUE)

## [1] 1

#### quantile()

La fonction quantile() renvoie le quantile défini par l’argument probs.

quantile(iris[, 1])

## 0% 25% 50% 75% 100%   
## 4.3 5.1 5.8 6.4 7.9

quantile(iris[, 1], probs = c(0, 0.25, 0.5, 0.75, 1))

## 0% 25% 50% 75% 100%   
## 4.3 5.1 5.8 6.4 7.9

quantile(iris[, 1], probs = c(0, 0.1, 0.5, 0.9, 1))

## 0% 10% 50% 90% 100%   
## 4.3 4.8 5.8 6.9 7.9

#### summary()

La fonction summary() renvoie un résumé composé du minimum, premier quartile, médiane, moyenne, troisième quartile et maximum.

summary(iris[, 1])

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 4.300 5.100 5.800 5.843 6.400 7.900

#### median()

La fonction median() renvoie la médiane.

median(iris[, 1])

## [1] 5.8

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

median(vecManip3)

## [1] NA

median(vecManip3, na.rm = TRUE)

## [1] 7

#### length()

La fonction length() renvoie la taille d’un objet (nombre d’éléments).

length(iris[, 1])

## [1] 150

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

length(vecManip3)

## [1] 8

#### nrow() et ncol()

La fonction nrow() renvoie le nombre de lignes et la fonction ncol() le nombre de colonnes d’un objet.

nrow(iris)

## [1] 150

ncol(iris)

## [1] 5

#### round(), ceiling(), floor(), et trunc()

La fonction round() permet de sélectionner un certain nombre de décimales (0 par défaut)

round(5.56874258564)

## [1] 6

round(5.56874258564, digits = 2)

## [1] 5.57

La fonction ceiling() renvoie le plus petit nombre entier qui ne soit pas inférieur à la valeur renseignée.

ceiling(5.9999)

## [1] 6

ceiling(5.0001)

## [1] 6

La fonction floor() renvoie le plus grand nombre entier qui ne soit pas supérieur à la valeur renseignée.

floor(5.9999)

## [1] 5

floor(5.0001)

## [1] 5

La fonction trunc() renvoie la partie entière de la valeur renseignée.

trunc(5.9999)

## [1] 5

trunc(5.0001)

## [1] 5

#### rowSums() et colSums()

Les fonctions rowSums() et colSums() calculent la somme des lignes et des colonnes.

rowSums(iris[, c(1, 2, 3, 4)])

## [1] 10.2 9.5 9.4 9.4 10.2 11.4 9.7 10.1 8.9 9.6 10.8 10.0 9.3 8.5 11.2  
## [16] 12.0 11.0 10.3 11.5 10.7 10.7 10.7 9.4 10.6 10.3 9.8 10.4 10.4 10.2 9.7  
## [31] 9.7 10.7 10.9 11.3 9.7 9.6 10.5 10.0 8.9 10.2 10.1 8.4 9.1 10.7 11.2  
## [46] 9.5 10.7 9.4 10.7 9.9 16.3 15.6 16.4 13.1 15.4 14.3 15.9 11.6 15.4 13.2  
## [61] 11.5 14.6 13.2 15.1 13.4 15.6 14.6 13.6 14.4 13.1 15.7 14.2 15.2 14.8 14.9  
## [76] 15.4 15.8 16.4 14.9 12.8 12.8 12.6 13.6 15.4 14.4 15.5 16.0 14.3 14.0 13.3  
## [91] 13.7 15.1 13.6 11.6 13.8 14.1 14.1 14.7 11.7 13.9 18.1 15.5 18.1 16.6 17.5  
## [106] 19.3 13.6 18.3 16.8 19.4 16.8 16.3 17.4 15.2 16.1 17.2 16.8 20.4 19.5 14.7  
## [121] 18.1 15.3 19.2 15.7 17.8 18.2 15.6 15.8 16.9 17.6 18.2 20.1 17.0 15.7 15.7  
## [136] 19.1 17.7 16.8 15.6 17.5 17.8 17.4 15.5 18.2 18.2 17.2 15.7 16.7 17.3 15.8

colSums(iris[, c(1, 2, 3, 4)])

## Sepal.Length Sepal.Width Petal.Length Petal.Width   
## 876.5 458.6 563.7 179.9

#### rowMeans() et colMeans()

Les fonctions rowMeans() et colMeans() calculent la moyenne des lignes et des colonnes.

rowMeans(iris[, c(1, 2, 3, 4)])

## [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700 2.500  
## [13] 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675 2.350 2.650  
## [25] 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725 2.825 2.425 2.400  
## [37] 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675 2.800 2.375 2.675 2.350  
## [49] 2.675 2.475 4.075 3.900 4.100 3.275 3.850 3.575 3.975 2.900 3.850 3.300  
## [61] 2.875 3.650 3.300 3.775 3.350 3.900 3.650 3.400 3.600 3.275 3.925 3.550  
## [73] 3.800 3.700 3.725 3.850 3.950 4.100 3.725 3.200 3.200 3.150 3.400 3.850  
## [85] 3.600 3.875 4.000 3.575 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525  
## [97] 3.525 3.675 2.925 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575  
## [109] 4.200 4.850 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675  
## [121] 4.525 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025  
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875 4.550  
## [145] 4.550 4.300 3.925 4.175 4.325 3.950

colMeans(iris[, c(1, 2, 3, 4)])

## Sepal.Length Sepal.Width Petal.Length Petal.Width   
## 5.843333 3.057333 3.758000 1.199333

#### aggregate()

La fonction aggregate() permet de grouper les éléments d’un objet en fonction d’une valeur. L’argument by définit l’élément sur lequel est effectué le regroupement. Il doit être de type [list](#l014list).

aggregate(iris[, c(1, 2, 3, 4)], by = list(iris$Species), FUN = mean)

## Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width  
## 1 setosa 5.006 3.428 1.462 0.246  
## 2 versicolor 5.936 2.770 4.260 1.326  
## 3 virginica 6.588 2.974 5.552 2.026

aggregate(iris[, c(1, 2)], by = list(iris$Species), FUN = summary)

## Group.1 Sepal.Length.Min. Sepal.Length.1st Qu. Sepal.Length.Median  
## 1 setosa 4.300 4.800 5.000  
## 2 versicolor 4.900 5.600 5.900  
## 3 virginica 4.900 6.225 6.500  
## Sepal.Length.Mean Sepal.Length.3rd Qu. Sepal.Length.Max. Sepal.Width.Min.  
## 1 5.006 5.200 5.800 2.300  
## 2 5.936 6.300 7.000 2.000  
## 3 6.588 6.900 7.900 2.200  
## Sepal.Width.1st Qu. Sepal.Width.Median Sepal.Width.Mean Sepal.Width.3rd Qu.  
## 1 3.200 3.400 3.428 3.675  
## 2 2.525 2.800 2.770 3.000  
## 3 2.800 3.000 2.974 3.175  
## Sepal.Width.Max.  
## 1 4.400  
## 2 3.400  
## 3 3.800

#### range()

La fonction range() renvoie le minimum et le maximum.

range(iris[, 1])

## [1] 4.3 7.9

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

range(vecManip3)

## [1] NA NA

range(vecManip3, na.rm = TRUE)

## [1] 1 45

#### unique()

La fonction unique() renvoie les valeurs uniques d’un objet (sans les doublons).

unique(iris[, 1])

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.4 4.8 4.3 5.8 5.7 5.2 5.5 4.5 5.3 7.0 6.4 6.9 6.5  
## [20] 6.3 6.6 5.9 6.0 6.1 5.6 6.7 6.2 6.8 7.1 7.6 7.3 7.2 7.7 7.4 7.9

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

unique(vecManip3)

## [1] 1 5 6 8 NA 45 14

## Autres fonctions utiles

Nous ne pouvons aborder toutes les fonctions utiles, ici nous ne ferons qu’aborder certaines fonctions. Tout au long de ce livre de nouvelles fonctions seront utilisées. Lorsqu’une nouvelle fonction est utilisée, notre réflexe doit être toujours le même : **consulter la documentation** avec la fonction help().

### seq\_along()

La fonction seq\_along() permet de créer un vector de la taille de l’objet renseigné et prenant comme valeurs les chiffres de 1 à N (N correspondant aux nombres d’éléments de l’objet). Cette fonction nous servira beaucoup lors du chapitre sur les boucles.

print(vecManip3)

## [1] 1 5 6 8 NA 45 NA 14

seq\_along(vecManip3)

## [1] 1 2 3 4 5 6 7 8

### :

La fonction : permet de créer une séquence de a à b par pas de 1, avec a et b le début et la fin de la séquence souhaitée. Il a été difficile d’écrire les chapitres précédents sans y avoir recours tant cette fonction est utile.

5:10

## [1] 5 6 7 8 9 10

head(iris[, c(1, 2, 3, 4)])

## Sepal.Length Sepal.Width Petal.Length Petal.Width  
## 1 5.1 3.5 1.4 0.2  
## 2 4.9 3.0 1.4 0.2  
## 3 4.7 3.2 1.3 0.2  
## 4 4.6 3.1 1.5 0.2  
## 5 5.0 3.6 1.4 0.2  
## 6 5.4 3.9 1.7 0.4

head(iris[, 1:4]) # ;-)

## Sepal.Length Sepal.Width Petal.Length Petal.Width  
## 1 5.1 3.5 1.4 0.2  
## 2 4.9 3.0 1.4 0.2  
## 3 4.7 3.2 1.3 0.2  
## 4 4.6 3.1 1.5 0.2  
## 5 5.0 3.6 1.4 0.2  
## 6 5.4 3.9 1.7 0.4

miVec01 <- c(1, 2, 3, 4)  
miVec01 <- 1:4 # ;-)  
-10:12

## [1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8  
## [20] 9 10 11 12

5:-5

## [1] 5 4 3 2 1 0 -1 -2 -3 -4 -5

paste("X", 1:10, sep = "\_")

## [1] "X\_1" "X\_2" "X\_3" "X\_4" "X\_5" "X\_6" "X\_7" "X\_8" "X\_9" "X\_10"

### rep()

La fonction rep() permet de répéter des éléments.

miVec12 <- c(1, 1, 1, 1, 1, 1, 1, 1, 1)  
miVec12 <- rep(1, times = 9) # ;-)  
rep("Hola", times = 3)

## [1] "Hola" "Hola" "Hola"

rep(1:3, time = 3)

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

rep(1:3, length.out = 10)

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

rep(1:3, each = 3)

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

### seq()

La fonction seq() permet de créer une séquence personnalisée.

seq(from = 0, to = 1, by = 0.2)

## [1] 0.0 0.2 0.4 0.6 0.8 1.0

seq(from = 20, to = 10, length.out = 10)

## [1] 20.00000 18.88889 17.77778 16.66667 15.55556 14.44444 13.33333 12.22222  
## [9] 11.11111 10.00000

letters[seq(from = 1, to = 26, by = 2)]

## [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"

rep(seq(from = 1, to = 2, by = 0.5), times = 3)

## [1] 1.0 1.5 2.0 1.0 1.5 2.0 1.0 1.5 2.0

### getwd()

La fonction getwd() définit le répertoire de travail. Cela correspond à l’endroit relatif à partir duquel R se positionne pour identifier les fichiers. Ce concept prendra son sens lorsque nous verrons comment importer et exporter des données.

getwd()

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRBook\_FR"

### setwd()

La fonction setwd() permet de définir un nouveau répertoire de travail.

oldWd <- getwd()  
print(oldWd)

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRBook\_FR"

setwd("..")  
getwd()

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub"

setwd(oldWd)  
getwd()

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRBook\_FR"

### list.files()

La fonction list.files() permet de faire la liste de tous les fichiers présents dans le répertoire de travail.

list.files(pattern = "(html)$") # html

## [1] "google\_analytics\_FR.html"

list.files(pattern = "(pdf)$") # pdf

## character(0)

### ls()

Tout comme la fonction list.files() permet de faire la liste de tous les fichiers présents dans le répertoire de travail, la fonction ls() permet de faire la liste de tous les objets présents dans l’environnement de travail de R.

ls()

## [1] "aLogic" "bddInsect" "characters" "contrib" "dfForMat"   
## [6] "factor01" "i" "irisCopy" "j" "k"   
## [11] "logicals" "mdat" "miArray" "miArray02" "miDf01"   
## [16] "miDfSub01" "miDfSub02" "miDfSub03" "miDfSub04" "miList01"   
## [21] "miList02" "miList0203" "miList03" "miList04" "miList05"   
## [26] "miList06" "miMat" "miMat01" "miMat02" "miVec01"   
## [31] "miVec02" "miVec03" "miVec04" "miVec05" "miVec05b"   
## [36] "miVec05c" "miVec06" "miVec07" "miVec08" "miVec09"   
## [41] "miVec10" "miVec11" "miVec12" "miVec13" "miVec14"   
## [46] "miVec15" "miVec20" "miVec21" "miVec22" "miVec23"   
## [51] "miVec24" "miVec25" "miVecArr" "miVecArr02" "miVecConf"   
## [56] "miVecNA" "miVecOp" "msg" "myCol" "myRow"   
## [61] "myText" "myText2" "myText3" "myText4" "myText5"   
## [66] "nbrRep" "newVec" "newVec2" "numbers" "oldWd"   
## [71] "opAriDf" "roundDou" "sumIntDou" "sumIntInt" "tabPkgDf"   
## [76] "terme01" "terme02" "vecForMat01" "vecForMat02" "vecForMat03"   
## [81] "vecForMatrix" "vecManip" "vecManip2" "vecManip3"

zzz <- "a new object"  
ls()

## [1] "aLogic" "bddInsect" "characters" "contrib" "dfForMat"   
## [6] "factor01" "i" "irisCopy" "j" "k"   
## [11] "logicals" "mdat" "miArray" "miArray02" "miDf01"   
## [16] "miDfSub01" "miDfSub02" "miDfSub03" "miDfSub04" "miList01"   
## [21] "miList02" "miList0203" "miList03" "miList04" "miList05"   
## [26] "miList06" "miMat" "miMat01" "miMat02" "miVec01"   
## [31] "miVec02" "miVec03" "miVec04" "miVec05" "miVec05b"   
## [36] "miVec05c" "miVec06" "miVec07" "miVec08" "miVec09"   
## [41] "miVec10" "miVec11" "miVec12" "miVec13" "miVec14"   
## [46] "miVec15" "miVec20" "miVec21" "miVec22" "miVec23"   
## [51] "miVec24" "miVec25" "miVecArr" "miVecArr02" "miVecConf"   
## [56] "miVecNA" "miVecOp" "msg" "myCol" "myRow"   
## [61] "myText" "myText2" "myText3" "myText4" "myText5"   
## [66] "nbrRep" "newVec" "newVec2" "numbers" "oldWd"   
## [71] "opAriDf" "roundDou" "sumIntDou" "sumIntInt" "tabPkgDf"   
## [76] "terme01" "terme02" "vecForMat01" "vecForMat02" "vecForMat03"   
## [81] "vecForMatrix" "vecManip" "vecManip2" "vecManip3" "zzz"

### rm()

La fonction rm() permet de supprimer un objet présent dans l’environnement de travail de R.

rm(zzz)  
ls()

## [1] "aLogic" "bddInsect" "characters" "contrib" "dfForMat"   
## [6] "factor01" "i" "irisCopy" "j" "k"   
## [11] "logicals" "mdat" "miArray" "miArray02" "miDf01"   
## [16] "miDfSub01" "miDfSub02" "miDfSub03" "miDfSub04" "miList01"   
## [21] "miList02" "miList0203" "miList03" "miList04" "miList05"   
## [26] "miList06" "miMat" "miMat01" "miMat02" "miVec01"   
## [31] "miVec02" "miVec03" "miVec04" "miVec05" "miVec05b"   
## [36] "miVec05c" "miVec06" "miVec07" "miVec08" "miVec09"   
## [41] "miVec10" "miVec11" "miVec12" "miVec13" "miVec14"   
## [46] "miVec15" "miVec20" "miVec21" "miVec22" "miVec23"   
## [51] "miVec24" "miVec25" "miVecArr" "miVecArr02" "miVecConf"   
## [56] "miVecNA" "miVecOp" "msg" "myCol" "myRow"   
## [61] "myText" "myText2" "myText3" "myText4" "myText5"   
## [66] "nbrRep" "newVec" "newVec2" "numbers" "oldWd"   
## [71] "opAriDf" "roundDou" "sumIntDou" "sumIntInt" "tabPkgDf"   
## [76] "terme01" "terme02" "vecForMat01" "vecForMat02" "vecForMat03"   
## [81] "vecForMatrix" "vecManip" "vecManip2" "vecManip3"

## Quelques exercices

Voici quelques exercices pour se perfectionner dans l’usage des fonctions et en apprendre de nouvelles grâce à la documentation. Certains exercices sont difficiles, nous pourrons y revenir plus tard.

### Séquences

#### Reproduisons les séquences suivantes :

-3 -4 -5 -6 -7 -8 -9 -10 –11

-3 -1 1 3 5 7 9 11

3.0 3.2 3.4 3.6 3.8 4.0

20 18 16 14 12 10 8 6

“a” “f” “k” “p” “u” “z”

“a” “a” “a” “a” “a” “f” “f” “f” “f” “f” “k” “k” “k” “k” “k” “p” “p” “p” “p” “p” “u” “u” “u” “u” “u” “z” “z” “z” “z” “z”

#### Solutions possibles (car il y a toujours plusieurs solutions) :

-3:-11

## [1] -3 -4 -5 -6 -7 -8 -9 -10 -11

seq(from = -3, to = 11, by = 2)

## [1] -3 -1 1 3 5 7 9 11

seq(from = 3.0, to = 4.0, by = 0.2)

## [1] 3.0 3.2 3.4 3.6 3.8 4.0

letters[seq(from = 1, to = 26, by = 5)]

## [1] "a" "f" "k" "p" "u" "z"

letters[rep(seq(from = 1, to = 26, by = 5), each = 5)]

## [1] "a" "a" "a" "a" "a" "f" "f" "f" "f" "f" "k" "k" "k" "k" "k" "p" "p" "p" "p"  
## [20] "p" "u" "u" "u" "u" "u" "z" "z" "z" "z" "z"

### Statistiques descriptives

Dans le jeu de données iris, combien de valeurs de largeur de sépales sont supérieures à 3 ? Entre 2.8 et 3.2 ?

Comment peut-on visualiser la distribution des données (fonction table()) ?

Quelles sont les 10 valeurs les plus petites ?

Comment calculer un intervalle contenant 90% des valeurs ?

Si la distribution des données était Normale, quelle serait la valeur théorique de cet intervalle de 90% (fonction qnorm()) ?

Solutions :

length(iris$Sepal.Width[iris$Sepal.Width > 3])

## [1] 67

length(iris$Sepal.Width[iris$Sepal.Width > 2.8 &   
 iris$Sepal.Width < 3.2])

## [1] 47

table(iris$Sepal.Width)

##   
## 2 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4   
## 1 3 4 3 8 5 9 14 10 26 11 13 6 12 6 4 3 6 2 1   
## 4.1 4.2 4.4   
## 1 1 1

table(round(iris$Sepal.Width))

##   
## 2 3 4   
## 19 106 25

irisSepWCopy <- iris$Sepal.Width  
irisSepWCopy <- irisSepWCopy[order(irisSepWCopy)]  
head(irisSepWCopy, n = 10)

## [1] 2.0 2.2 2.2 2.2 2.3 2.3 2.3 2.3 2.4 2.4

quantile(irisSepWCopy, probs = c(0.05, 0.95))

## 5% 95%   
## 2.345 3.800

qnorm(  
 p = c(0.05, 0.95),   
 mean = mean(irisSepWCopy),   
 sd = sd(irisSepWCopy)  
)

## [1] 2.340397 3.774270

## Ecrire une fonction

Lorsque nous reproduisons plusieurs fois les mêmes opérations, le code devient fastidieux à écrire, et plus difficile à maintenir car si nous devons effectuer une modification, il faudra la répéter chaque fois que nous l’avons utilisée. C’est un signe indiquant la nécessité de recourir à une **fonction**. Dans l’exemple qui suit, il est fastidieux de modifier le code si nous souhaitons ajouter +45 au lieu de +20 à chaque ligne.

35 + 20

## [1] 55

758 + 20

## [1] 778

862 + 20

## [1] 882

782 + 20

## [1] 802

Comme pour les fonctions de base de R, nos fonctions vont avoir un **nom**, et des **arguments**. Comme pour les noms des objets et les noms des fichiers, il est important de bien choisir le nom de notre fonction (cf. [section sur les objets](#l011object)). Pour créer une fonction nous allons utiliser la fonction function() qui prend comme arguments les arguments de notre fonction. La fonction va retourner le résultat souhaité. Par défaut le résultat renvoyé est le dernier utilisé, mais il est préférable de l’expliciter avec la fonction return(). La fonction suivante addX prend comme argument x et renvoie x + 20.

addX <- function(x){  
 return(x + 20)  
}

Notre code devient :

addX(35)

## [1] 55

addX(758)

## [1] 778

addX(862)

## [1] 882

addX(782)

## [1] 802

Si nous souhaitons modifier le code pour ajouter 45 plutôt que 20, il suffit alors de modifier la fonction addX().

addX <- function(x){  
 return(x + 45)  
}  
addX(35)

## [1] 80

addX(758)

## [1] 803

addX(862)

## [1] 907

addX(782)

## [1] 827

Ici nous aurions pu utiliser le format vector pour éviter les répétitions, mais ce n’est pas toujours possible.

c(35, 758, 862, 782) + 20

## [1] 55 778 882 802

Voyons cette fonction qui va compter le nombre de consonnes et de voyelles en minuscule dans un mot. Tout d’abord nous allons séparer toutes les lettres avec la fonction strsplit (nous pouvons consulter l’aide pour en savoir plus sur cette fonction). Ensuite nous allons compter les voyelles et les consonnes avec la fonction [length()](#l015length). Pour avoir la liste des lettres nous allons utiliser la constante letters (consulter l’aide).

countVowelConso <- function(word){  
 wordSplit <- strsplit(word, split = "")[[1]]  
 vowels <- c("a", "e", "i", "o", "u", "y")  
 numVowel <- length(wordSplit[wordSplit %in% vowels])  
 consonants <- letters[!letters %in% vowels]  
 numConso <- length(wordSplit[wordSplit %in% consonants])  
 return(c(numVowel, numConso))  
}

Nous pouvons maintenant utiliser notre fonction.

countVowelConso(word = "qwertyuiop azertyuiop")

## [1] 11 9

Cette fonction peut être modifiée en affichant un message plus explicite. Même si en général ce genre de message est à éviter pour ne pas surcharger les fonctions, il peut être utile pour vérifier que tout se déroule correctement (nous le supprimerons ensuite).

countVowelConso <- function(word){  
 wordSplit <- strsplit(word, split = "")[[1]]  
 vowels <- c("a", "e", "i", "o", "u", "y")  
 numVowel <- length(wordSplit[wordSplit %in% vowels])  
 consonants <- letters[!letters %in% vowels]  
 numConso <- length(wordSplit[wordSplit %in% consonants])  
 print(paste0("Il y a ", numVowel, " voyelles et ",   
 numConso, " consonnes dans le mot '", word, "'."))  
 return(c(numVowel, numConso))  
}  
countVowelConso(word = "qwertyuiop azertyuiop")

## [1] "Il y a 11 voyelles et 9 consonnes dans le mot 'qwertyuiop azertyuiop'."

## [1] 11 9

Par contre si nous utilisons countVowelConso(word = 5), une erreur va être renvoyée car notre fonction attend un objet de type character. De manière générale il est recommandé de gérer les erreurs renvoyées par nos fonctions afin que notre code soit plus facile à débugger. Ici nous allons simplement vérifier que l’argument est de type character, dans un vector de taille 1. Nous allons aussi commenter notre fonction pour rapidement retrouver ce qu’elle réalise (commentaire inséré sur la première ligne, que l’on retrouve aussi parfois sur la dernière ligne des fonctions).

countVowelConso <- function(word){ # compte les voyelles et les consonnes  
 if(is.vector(word) & is.character(word) & length(word) == 1){  
 wordSplit <- strsplit(word, split = "")[[1]]  
 vowels <- c("a", "e", "i", "o", "u", "y")  
 numVowel <- length(wordSplit[wordSplit %in% vowels])  
 consonants <- letters[!letters %in% vowels]  
 numConso <- length(wordSplit[wordSplit %in% consonants])  
 return(c(numVowel, numConso))  
 } else {  
 print(paste0("Erreur dans la fonction countVowelConso, ",   
 "argument 'word' incorrect (", word, ")"))  
 }  
}   
countVowelConso(word = "qwertyuiop azertyuiop")

## [1] 11 9

countVowelConso(word = 5)

## [1] "Erreur dans la fonction countVowelConso, argument 'word' incorrect (5)"

Avec R comme pour tout langage de programmation, pour un problème il existe toujours de multiples solutions. Nous nous souvenons de la section sur les types de données ([type de données logical](#l013logi)), ainsi que de la section sur les [opérateurs de comparaison](#l011opcomp) que la valeur de TRUE est de 1 et la valeur de FALSE est de 0. Nous venons de voir ci-dessus que la fonction [%in%](#l015in) renvoie TRUE ou FALSE pour chacun des éléments du premier objet en fonction de leur présence ou absence dans le second objet. Notre fonction aurait pu donc se passer de la fonction length() pour le comptage des voyelles et des consonnes et utiliser la fonction sum().

countVowelConsoAlt <- function(word){ # compte les voyelles et les consonnes  
 if(is.vector(word) & is.character(word) & length(word) == 1){  
 wordSplit <- strsplit(word, split = "")[[1]]  
 vowels <- c("a", "e", "i", "o", "u", "y")  
 numVowel <- sum(wordSplit %in% vowels)  
 consonants <- letters[!letters %in% vowels]  
 numConso <- sum(wordSplit %in% consonants)  
 return(c(numVowel, numConso))  
 } else {  
 print(paste0("Erreur dans la fonction countVowelConso, ",   
 "argument 'word' incorrect (", word, ")"))  
 }  
}   
countVowelConsoAlt(word = "qwertyuiop azertyuiop")

## [1] 11 9

Il n’y a pas de solution optimale dans l’absolu, tout dépend des objectifs recherchés. La première solution est peut être plus facile à comprendre, et la seconde peut-être plus rapide en terme de vitesse d’exécution (même en répétant l’utilisation de la fonction 100000 fois, le gain de temps est presque nul dans notre cas).

system.time(replicate(n = 100000, countVowelConso(word = "qwertyuiop azertyuiop")))

## user system elapsed   
## 2.28 0.00 2.28

system.time(replicate(n = 100000, countVowelConsoAlt(word = "qwertyuiop azertyuiop")))

## user system elapsed   
## 2.34 0.00 2.35

Une fonction peut avoir des valeurs par défaut pour ses arguments. C’est le cas de la plupart des fonctions existantes. Par défaut, notre fonction va désormais compter le nombre de voyelles et de consonnes dans le mot qwerty (les parenthèses restent nécessaires même en l’absence d’arguments).

countVowelConsoAlt <- function(word = "qwerty"){ # compte les voyelles et les consonnes  
 if(is.vector(word) & is.character(word) & length(word) == 1){  
 wordSplit <- strsplit(word, split = "")[[1]]  
 vowels <- c("a", "e", "i", "o", "u", "y")  
 numVowel <- sum(wordSplit %in% vowels)  
 consonants <- letters[!letters %in% vowels]  
 numConso <- sum(wordSplit %in% consonants)  
 return(c(numVowel, numConso))  
 } else {  
 print(paste0("Erreur dans la fonction countVowelConso, ",   
 "argument 'word' incorrect (", word, ")"))  
 }  
}   
countVowelConsoAlt()

## [1] 2 4

R compte de nombreuses fonctions, donc avant de vous lancer dans l’écriture d’une nouvelle fonction, il faut toujours vérifier que celle-ci n’existe pas déjà soit dans la version de base de R, soit dans des **packages** développés par la communauté des utilisateurs. Pour cela nous pouvons utiliser l’aide et la fonction ??, mais aussi notre navigateur Internet.

## Autres fonctions développées par la communauté des utilisateurs : les packages

Un package est un ensemble de fichiers que l’on va ajouter à R pour pouvoir utiliser des fonctions (ou des jeux de données) que d’autres personnes ont développés. Il y a à ce jour plus de 10000 packages sur les serveurs de R (CRAN ; <https://cran.r-project.org/web/packages/>), plus de 1000 sur les serveurs de BioConductor (pour l’analyse génomique), et plusieurs centaines sur GitHub (il existe d’autres serveurs populaires comme R-forge ou encore Omegahat). Chaque package permet de mettre à disposition des fonctions pour à peu près tout faire… Il peut donc être difficile de trouver le package adapté à ce que nous souhaitons réaliser, et il est important de consacrer du temps sa recherche, et de tester plusieurs solutions.

Pour utiliser un package il nous faut tout d’abord l’**installer**, puis le **charger** dans notre session de R.

### Installer un package

Une fois notre package sélectionné, nous pouvons le télécharger et l’installer avec la fonction install.packages() qui prend comme argument le nom du package entre guillemets (la fonction tolère l’absence de guillemets mais il est préférable de les utiliser pour que le code soit plus lisible). Certains packages sont installés par défaut avec R, c’est le cas par exemple de stats (qui est aussi chargé par défaut).

install.packages("stats") # R statistical functions

L’installation d’un package est à réaliser une seule fois, ensuite le package est sur notre ordinateur.

### Charger un package

Pour pouvoir utiliser les fonctions d’un package, nous devons le charger dans notre session de R. Il y a tellement de packages disponibles que R ne va pas charger par défaut tous ceux que nous avons installé, mais seulement ceux dont nous allons avoir besoin pour notre étude en cours. Pour charger un package nous utilisons la fonction library().

library("stats")

Le chargement du package est à réaliser à chaque fois que nous souhaitons exécuter notre code, il fait donc partie intégrante de notre script.

### Portabilité du code

Nous venons de voir que l’installation d’un package est à faire une seule fois par ordinateur, et que par contre le chargement d’un package est à réaliser pour chaque nouvelle session de R. Si l’on change d’ordinateur ou si l’on partage un script avec un collègue, il peut donc y avoir des erreurs à l’exécution liées à l’absence de l’installation d’un package. Pour pallier à ce problème, il est recommandé d’utiliser une fonction qui va vérifier si les packages nécessaires à l’exécution d’un script sont installés, si besoin les installer, puis les charger. Il existe de nombreuses fonctions pour faire cela sur Internet. La solution que nous proposons ici est un mélange adapté de différentes sources. Il n’est pas nécessaire de comprendre les détails de ce script pour le moment, mais simplement de comprendre ce qu’il fait. Voici un exemple pour les packages stats et graphics qui sont deux packages déjà présents avec la version de base de R, mais nous pouvons essayer avec tous les packages disponibles sur le CRAN, dont la liste se trouve ici : <https://cran.r-project.org/web/packages/available_packages_by_name.html>.

pkgCheck <- function(packages){  
 for(x in packages){  
 try(if(!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)){  
 stop()  
 }  
 })  
 }  
}  
pkgCheck(c("stats", "graphics"))

Alternativement nous pouvons utiliser la fonction .packages() pour lister les packages disponibles sur le CRAN par ordre alphabétique.

head(.packages(all.available = TRUE), n = 30)

## [1] "abind" "acepack" "ade4" "agricolae" "AlgDesign"   
## [6] "alr4" "ape" "askpass" "assertthat" "backports"   
## [11] "base64enc" "bdsmatrix" "BH" "bibtex" "bindr"   
## [16] "bindrcpp" "BioFTF" "bitops" "bold" "bookdown"   
## [21] "boot" "brew" "broom" "ca" "callr"   
## [26] "car" "carData" "caret" "cartography" "caTools"

La fonction pkgCheck() assure la **portabilité** de nos scripts : ils fonctionneront sur tous les ordinateurs sans avoir à effectuer de modification. Ainsi nos scripts pourront par exemple être joints à nos articles scientifiques et assurer ainsi la **reproductibilité** de nos résultats.

## Conclusion

Félicitations ! Nous savons à présent ce qu’est une fonction, comment chercher de l’aide sur une fonction, et même écrire ses propres fonctions. Nous savons aussi qu’il existe de nombreuses fonctions développées par la communauté des utilisateurs de R au sein de packages que nous savons installer et charger, et s’assurer de la portabilité de nos script d’un ordinateur à un autre (important pour la reproductibilité des résultats). Le prochain chapitre va s’intéresser à la lecture et à l’écriture de fichiers car bien souvent, nos données sont sur des fichiers de texte ou de tableurs.

# Importer et exporter des données

## Lire des données depuis un fichier

### Transformer ses données au format TXT ou CSV

Il existe de nombreuses façons de lire le contenu d’un fichier avec R. Cependant nous nous focaliserons sur la lecture des fichiers TXT et CSV qui sont les plus communs et les plus fiables. A de rares exceptions près tous les fichiers de données peuvent très facilement être transformés aux formats TXT et CSV. C’est la pratique à préférer pour une analyse des données avec R. Dans tous les autres cas et pour couvrir l’ensemble des possibilités avec R, nous pourrons nous référer au manuel de référence sur l’import et l’export de données sur le site web de R (en anglais ; <https://cran.r-project.org/manuals.html>).

Concrètement, depuis Microsoft Excel, il suffit d’aller dans *Fichier*, puis *Enregistrer sous*, de sélectionner l’endroit où nous souhaitons sauvegarder notre fichier (cf. chapitre suivant sur la gestion d’un projet R) puis dans la fenêtre de sauvegarde changer le *Type* depuis XLSX vers CSV. Depuis LibreOffice Calc, il suffit d’aller dans *Fichier*, puis *Enregistrer sous*, puis de sélectionner le *Type* CSV. Il est important de savoir que le format CSV ne supporte pas la mise en forme des fichiers tableurs avec par exemple des couleurs, et que le fichier CSV ne contient qu’un seul onglet. Si nous avons un fichier tableur avec plusieurs onglets, il faudra sauvegarder autant de fichiers CSV que d’onglets.

CSV vient de l’anglais *Comma-separated values* (<https://fr.wikipedia.org/wiki/Comma-separated_values>), et représente des données de tableur au format texte séparées par des virgules (ou des points virgules suivant les pays). Un fichier CSV pourra toujours s’ouvrir avec notre logiciel de tableur, mais aussi avec un simple éditeur de texte comme le bloc-notes de Windows ou encore avec Notepad++. Il est d’ailleurs préférable d’ouvrir les fichiers CSV avec un éditeur de texte car les tableurs ont la fâcheuse tendance à vouloir modifier automatiquement les fichiers CSV et cela a pour conséquence de les rendre difficiles à lire.

Une fois le fichier TXT ou CSV obtenu, la lecture du contenu depuis R est facile, même si elle demande un peu de rigueur.

### Lire un fichier CSV

C’est la source d’erreur la plus commune chez les débutants en R. C’est pourquoi il est important de lire et de relire ce chapitre et celui sur la gestion d’un projet R avec beaucoup d’attention.

Commençons par préciser que R travaille dans un répertoire défini par défaut. Les utilisateurs de Rstudio ou autre [environnement de développement](#IDE) spécialisés pour R seront tentés d’utiliser les options disponibles via les menus pour définir leur répertoire de travail ou pour charger le contenu d’un fichier. Dans ce livre ces techniques ne seront jamais utilisées car elles ne permettent pas la reproductibilité des résultats. Un script doit toujours pouvoir fonctionner pour tous les systèmes d’exploitation et quel que soit [l’environnement de développement](#IDE) de l’utilisateur.

Le répertoire de travail par défaut peut être obtenu avec la fonction [getwd()](#l015getwd) et spécifié avec la fonction [setwd()](#l015setwd).

oldWd <- getwd()  
print(oldWd)

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRBook\_FR"

setwd("..")  
getwd()

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub"

setwd(oldWd)  
getwd()

## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRBook\_FR"

Nous avons donc quatre options :

* nous pouvons lire le contenu d’un fichier en indiquant à R son chemin complet (à proscrire pour la reproductibilité des résultats, en effet le chemin complet varie d’un ordinateur à un autre)
* nous pouvons lire le contenu d’un fichier en indiquant à R son chemin relatif
* nous pouvons déplacer le fichier dans le répertoire de travail de R par défaut
* nous pouvons modifier le répertoire de travail de R pour qu’il corresponde à l’emplacement de notre fichier (avec son chemin relatif)

Un exemple de chemin complet serait :

* “/home/myName/myFile.csv” sous un environnement UNIX
* “C:/users/myName/myFile.csv” sous un environnement Windows (attention, sous R nous utilisons / et non pas \ comme c’est le cas par défaut sous Windows)

Un chemin relatif serait :

* myName/myFiles.csv

Pour naviguer dans les chemins relatifs nous pouvons utiliser “..” qui permet de remonter dans le répertoire source. Par exemple si le répertoire de travail est myScripts et que l’arborescence de mes fichiers est ainsi :

## -myProject  
## |-myFiles  
## |-|-data01.csv  
## |-|-data02.csv  
## |-myScripts  
## |-|-myFirstScript.R

Le chemin relatif vers le fichier “data01.csv” serait “../myFiles/data01.csv”.

Donc pour lire le contenu du fichier data01.csv, nous allons privilégier l’option 2 (lire le contenu d’un fichier en indiquant à R son chemin relatif) ou l’option 4 (modifier le répertoire de travail de R pour qu’il corresponde à l’emplacement de notre fichier). Dans ce dernier cas :

myWD <- "../myFiles/"  
setwd(myWd)  
getwd() # pour vérifier que nous sommes dans le bon répertoire  
list.files() # pour vérifier que le fichier se trouve bien ici

L’erreur la plus commune :

## Error in setwd("../myFiles/") :   
## impossible de changer de répertoire de travail

Cela veut dire que le répertoire n’existe pas (il faut vérifier que la syntaxe est correcte et que le répertoire existe bien avec ce chemin).

Une fois le répertoire de travail correctement défini ou le chemin relatif vers le fichier correctement établi, nous pouvons lire le fichier avec la fonction read.table(). Certains utilisent la fonction read.csv() mais ce n’est qu’un cas particulier de la fonction read.table().

myWD <- "../myFiles/"  
setwd(myWd)  
read.table(file = "data01.csv")

ou alternativement :

read.table(file = "../myFiles/data01.csv")

Si le chemin n’est pas correctement renseigné ou si le fichier de données n’existe pas, R renverra l’erreur suivante :

## Error in file(file, "rt") : impossible d'ouvrir la connexion  
## De plus : Warning message:  
## In file(file, "rt") :  
## impossible d'ouvrir le fichier '../myFiles/data01.csv' : No such file or directory

Si tout va bien, R affiche le contenu du fichier “data01.csv”. Attention aux utilisateurs de Windows car par défaut le nom des fichiers apparaît sans leur extension… Ainsi lorsque nous naviguons dans les répertoires avec l’explorateur de fichiers, il n’y a pas de fichier “data01.csv” mais uniquement un fichier “data01”. Il est indispensable de remédier à ce problème pour éviter les erreurs. Pour ce faire il suffit d’ouvrir les ‘Options de l’Explorateur de fichiers’ via la touche ‘Windows’, puis dans l’onglet ‘Affichage’, de vérifier que l’option ‘Masquer les extensions des fichiers dont le type est connu’ est bien décochée.

En consultant l’aide sur la fonction read.table(), nous pouvons voir qu’elle possède de nombreux arguments. Les principaux sont les suivants :

* header = FALSE : est-ce que le fichier contient des noms de colonnes ? Si oui alors il faut changer la valeur pour header = TRUE
* sep = "" : comment sont séparées les données de la table ? Dans un fichier CSV il s’agit de la virgule ou du point-virgule, donc à changer pour sep = "," ou sep = ";"
* dec = "." : quel est le séparateur des nombres décimaux ? Si c’est la virgule alors il faut changer pour dec = ","

Avec ces trois arguments la plupart des fichiers pourront être lus sans problème. En cas de besoin l’aide est faite pour être consultée, et elle est très complète.

La fonction read.table() renvoie le contenu du fichier sous forme d’une data.frame. Pour pouvoir utiliser le contenu du fichier nous allons donc stocker la data.frame dans un objet.

myWD <- "../myFiles/"  
setwd(myWd)  
data01 <- read.table(file = "data01.csv")  
str(data01) # vérifier le format des données  
head(data01) # vérifier les premières données

Voici un exemple où la première ligne est ignorée avec l’argument skip = 1.

# Affcihage du fichier brut (10 premières lignes)  
readLines("myFiles/E05C13.csv", n = 10)

## [1] "ï»¿\"Titre de tracÃ© : E05C13\""   
## [2] "\"#\",\"Date Heure, GMT-04:00\",\"Temp., Â°C (LGR S/N: 10362176, SEN S/N: 10362176)\""  
## [3] "1,11/12/15 23:00:00,4.973"   
## [4] "2,11/12/15 23:30:00,4.766"   
## [5] "3,11/13/15 00:00:00,4.844"   
## [6] "4,11/13/15 00:30:00,4.844"   
## [7] "5,11/13/15 01:00:00,5.076"   
## [8] "6,11/13/15 01:30:00,5.282"   
## [9] "7,11/13/15 02:00:00,5.308"   
## [10] "8,11/13/15 02:30:00,5.385"

# lecture du fichier avec R  
bdd <- read.table("myFiles/E05C13.csv", skip = 1, header = TRUE,   
 sep = ",", dec = ".", stringsAsFactors = FALSE)  
colnames(bdd) <- c("id", "date", "temp")  
head(bdd)

## id date temp  
## 1 1 11/12/15 23:00:00 4.973  
## 2 2 11/12/15 23:30:00 4.766  
## 3 3 11/13/15 00:00:00 4.844  
## 4 4 11/13/15 00:30:00 4.844  
## 5 5 11/13/15 01:00:00 5.076  
## 6 6 11/13/15 01:30:00 5.282

tail(bdd)

## id date temp  
## 32781 32781 09/25/17 21:00:00 7.091  
## 32782 32782 09/25/17 21:30:00 6.914  
## 32783 32783 09/25/17 22:00:00 6.813  
## 32784 32784 09/25/17 22:30:00 6.611  
## 32785 32785 09/25/17 23:00:00 6.331  
## 32786 32786 09/25/17 23:30:00 5.385

str(bdd)

## 'data.frame': 32786 obs. of 3 variables:  
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...  
## $ date: chr "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...  
## $ temp: num 4.97 4.77 4.84 4.84 5.08 ...

### Lire un fichier texte

La fonction la plus simple pour lire un fichier contenant du texte est readLines(). Voici un exemple avec le fichier \_bookdown.yml de ce livre, que l’on retrouve sur GitHub.

readmeGitHub <- "https://raw.githubusercontent.com/frareb/myRBook\_FR/master/\_bookdown.yml"  
readLines(readmeGitHub)

## [1] "---" "book\_filename: \"myRBook\_FR\""  
## [3] "output\_dir: public" "language: "   
## [5] " ui:" " chapter\_name: \" \""   
## [7] "delete\_merged\_file: true" "---"

Il existe aussi la fonction scan() qui va renvoyer l’ensemble des mots séparés par des espaces. Nous pouvons consulter l’aide pour plus d’information.

scan(readmeGitHub, what = "character")

## [1] "---" "book\_filename:" "myRBook\_FR"   
## [4] "output\_dir:" "public" "language:"   
## [7] "ui:" "chapter\_name:" " "   
## [10] "delete\_merged\_file:" "true" "---"

## Exporter ou charger des données pour R

Il est parfois utile de pouvoir sauvegarder un objet R pour pouvoir le réutiliser plus tard. C’est le cas par exemple lorsque le temps de calcul pour arriver à un résultat est très long, ou alors lorsque l’on souhaite libérer de l’espace dans la RAM. Pour ce faire il existe la fonction save() qui prend comme argument principal le nom du ou des objets que nous voulons sauvegarder.

L’objet sauvé va être stocké dans un fichier. Par convention, il est bon de donner le nom d’extension .RData aux fichiers contenant des objets R, de préférer un seul objet par fichier, et de donner le nom de l’objet comme nom de fichier.

# Création de l'objet myObject  
myObject <- 5  
# Vérification de l'existence de l'objet  
ls(pattern = "myObject")

## [1] "myObject"

# Sauvegarde de l'objet dans un fichier RData  
save(myObject, file = "myFiles/myObject.RData")  
# Suppression de l'objet dans R  
rm(myObject)  
# Vérification de la suppression  
ls(pattern = "myObject")

## character(0)

Lors de la session suivante de R ou si nous avons à nouveau besoin de l’objet sauvegardé dans le fichier “myObject.RData”, nous pouvons le recharger avec la fonction load().

ls(pattern = "myObject")

## character(0)

load("myFiles/myObject.RData")  
ls(pattern = "myObject")

## [1] "myObject"

print(myObject)

## [1] 5

## Exporter des données

Le meilleur moyen de communiquer vos résultats ou vos données est de transmettre vos scripts et vos fichiers de données. Parfois ce n’est pas possible ou pas adapté, et il peut être utile d’exporter ses données dans un fichier texte ou CSV. Pour ce faire il existe la fonction générique write() et la fonction write.table() pour les data.frame.

Par exemple nous allons créer une data.frame avec les numéros de 1 à 26 et les lettres correspondantes, puis les sauver dans un fichier CSV, puis lire les données contenues dans ce fichier.

# Création de l'objet dfLetters  
dfLetters <- data.frame(num = 1:26, letters = letters)  
# Ecriture du contenu dans un fichier CSV  
write.table(dfLetters, file = "myFiles/dfLetters.csv",   
 sep = ",", col.names = TRUE, row.names = FALSE)  
# Suppression de l'objet dfLetters  
rm(dfLetters)  
# Lecture du fichier CSV  
read.table(file = "myFiles/dfLetters.csv", header = TRUE, sep = ",")

## num letters  
## 1 1 a  
## 2 2 b  
## 3 3 c  
## 4 4 d  
## 5 5 e  
## 6 6 f  
## 7 7 g  
## 8 8 h  
## 9 9 i  
## 10 10 j  
## 11 11 k  
## 12 12 l  
## 13 13 m  
## 14 14 n  
## 15 15 o  
## 16 16 p  
## 17 17 q  
## 18 18 r  
## 19 19 s  
## 20 20 t  
## 21 21 u  
## 22 22 v  
## 23 23 w  
## 24 24 x  
## 25 25 y  
## 26 26 z

## Conclusion

Félicitations ! Nous savons désormais comment lire des données contenues dans un fichier texte ou CSV, sauver et charger des données RData, et écrire dans un fichier. Rappelons que l’erreur la plus commune chez les débutants en R est la lecture des fichiers de données et l’organisation des fichiers. C’est pourquoi ce chapitre est à lire et à relire sans modération.

# Algorithmique

## Tests logiques avec if

Si nous voulons effectuer une opération différente en fonction d’une condition, nous pouvons mettre en place un test logique du type **SI** *ceci* **ALORS** *faire cela* **SINON** *faire cela*. Avec R cela va se traduire par la fonction if(cond) cons.expr else alt.expr comme indiqué dans l’aide de la fonction.

myVar <- 2  
if(myVar < 3) print("myVar < 3")

## [1] "myVar < 3"

if(myVar < 3) print("myVar < 3") else print("myVar > 3")

## [1] "myVar < 3"

Lorsque il y a plusieurs lignes de codes à exécuter en fonction du test logique, ou simplement pour rendre le code plus facile à lire, nous allons utiliser plusieurs lignes avec les acolades {} et en utilisant l’indentation.

myVar <- 2  
myResult <- 0  
if(myVar < 3){  
 print("myVar < 3")  
 myResult <- myVar + 10  
} else {  
 print("myVar > 3")  
 myResult <- myVar - 10  
}

## [1] "myVar < 3"

print(myResult)

## [1] 12

Dans cet exemple nous définissons une variable myVar. Si cette variable est inférieure à 3 alors la variable myResult prend comme valeur myVar + 10, et dans le cas contraire myResult prend comme valeur myVar - 10.

Nous avions déjà vu l’utilisation du test logique if dans le [chapitre sur les fonctions](#l015function). Nous avions alors testé si la variable entrée comme argument dans notre fonction était bien de type character.

myVar <- "qwerty"  
if(is.character(myVar)){  
 print("ok")  
} else {  
 print("error")  
}

## [1] "ok"

Nous pouvons aussi imbriquer les tests logiques les uns dans les autres.

myVar <- TRUE  
if(is.character(myVar)){  
 print("myVar: character")  
} else {  
 if(is.numeric(myVar)){  
 print("myVar: numeric")  
 } else {  
 if(is.logical(myVar)){  
 print("myVar: logical")  
 } else {  
 print("myVar: ...")  
 }  
 }  
}

## [1] "myVar: logical"

Il est aussi possible de stipuler plusieurs conditions, comme nous l’avons vu lors du chapitre sur les [opérateurs de comparaison](#l011opcomp).

myVar <- 2  
if(myVar > 1 & myVar < 50){  
 print("ok")  
}

## [1] "ok"

Dans cet exemple myVar est au format numeric donc la première condition (>1), et la seconde condition (<50) sont toutes les deux vérifiables. Par contre si nous affectons une variable de type character à myVar alors R va transformer 0 et 10 en objets de type character et tester si myVar > "1" puis si myVar < "50" en se basant sur un tri par ordre alphabétique. Dans l’exemple suivant "azerty" n’est pas situé par ordre alphabétique entre "1" et "50", par contre c’est le cas de "2azerty".

myVar <- "azerty"  
limInit <- 1  
limEnd <- 50  
if(myVar > limInit & myVar < limEnd){  
 print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))  
} else {  
 print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))  
}

## [1] "azerty not between 1 and 50."

myVar <- "2azerty"  
if(myVar > limInit & myVar < limEnd){  
 print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))  
} else {  
 print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))  
}

## [1] "2azerty is between 1 and 50."

Donc ce que nous voudrions faire est de tester si myVar est bien au format numeric puis uniquement si c’est le cas de tester les conditions suivantes.

myVar <- "2azerty"  
if(is.numeric(myVar)){  
 if(myVar > limInit & myVar < limEnd){  
 print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))  
 } else {  
 print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))  
 }  
} else {  
 print(paste0("Object ", myVar, " is not numeric"))  
}

## [1] "Object 2azerty is not numeric"

Parfois, nous pouvons avoir besoin de tester une première condition puis une seconde condition uniquement si la première se vérifie dans un même test. Par exemple, pour un site nous voudrions savoir si il y a une seule espèces et tester si son abondance est supérieure à 10. Imaginons un jeu de données avec sous forme de vecteur les abondances. Nous allons tester le nombre d’espèces avec la fonction [length()](#l015length).

mySpecies <- c(15, 14, 20, 12)  
if(length(mySpecies) == 1 & mySpecies > 10){  
 print("ok!")  
}  
## Warning message:  
## In if (length(mySpecies) == 1 & mySpecies > 10) { :  
## the condition has length > 1 and only the first element will be used

R renvoie une erreur car il ne peut pas au sein d’un test logique avec if() vérifier la seconde condition. En effet, mySpecies > 10 renvoie TRUE TRUE TRUE TRUE TRUE. Nous pouvons séparer le code en deux conditions :

mySpecies <- c(15, 14, 20, 12)  
if(length(mySpecies) == 1){  
 if(mySpecies > 10){  
 print("ok!")  
 }  
}

Une alternative plus élégante consiste à spécifier à R de vérifier la seconde condition uniquement si la première est vraie. Pour cela nous pouvons utiliser && à la place de &.

mySpecies <- c(15, 14, 20, 12)  
if(length(mySpecies) == 1 && mySpecies > 10){  
 print("ok!")  
}  
mySpecies <- 15  
if(length(mySpecies) == 1 && mySpecies > 10){  
 print("ok!")  
}

## [1] "ok!"

mySpecies <- 5  
if(length(mySpecies) == 1 && mySpecies > 10){  
 print("ok!")  
}

Avec & R va vérifier toutes les conditions, et avec && R va prendre chaque condition l’une après l’autre et poursuivre uniquement si elle se vérifie. Cela peut paraître anecdotique mais il est bon de connaître la différence entre & et && car nous les rencontrons souvent dans les codes disponibles sur Internet ou dans les packages.

## Tests logiques avec switch

La fonction switch() est une variante de if() qui est utile lorsque nous avons de nombreuses options possibles sur une même expression. L’exemple suivant montre comment transformer un code utilisant if() en switch().

x <- "aa"  
if(x == "a"){  
 result <- 1  
}  
if(x == "aa"){  
 result <- 2  
}  
if(x == "aaa"){  
 result <- 3  
}  
if(x == "aaaa"){  
 result <- 4  
}  
print(result)

## [1] 2

x <- "aa"  
switch(x,   
 a = result <- 1,  
 aa = result <- 2,  
 aaa = result <- 3,  
 aaaa = result <- 4)  
print(result)

## [1] 2

## La boucle for

En programmation, lorsque nous sommes amenés à répéter plusieurs fois la même ligne de code, c’est un signe indiquant qu’il faut utiliser une **boucle**. Une boucle est une manière d’itérer sur un ensemble d’objets (ou sur les éléments d’un objet), et de répéter une opération. Imaginons une data.frame avec des mesures de terrain à deux dates.

bdd <- data.frame(date01 = rnorm(n = 100, mean = 10, sd = 1),   
 date02 = rnorm(n = 100, mean = 10, sd = 1))  
print(head(bdd))

## date01 date02  
## 1 9.324726 9.387889  
## 2 10.246147 9.994158  
## 3 11.384945 9.772913  
## 4 9.083354 12.319240  
## 5 8.097771 10.538057  
## 6 10.879128 9.532593

Nous voudrions quantifier la différence entre la première et la deuxième date, puis mettre un indicateur pour savoir si cette différence est petite ou grande, par exemple avec un seuil arbitraire de 3. Donc pour chaque ligne nous pourrions faire :

bdd$dif <- NA  
bdd$isDifBig <- NA  
  
bdd$dif[1] <- sqrt((bdd$date01[1] - bdd$date02[1])^2)  
bdd$dif[2] <- sqrt((bdd$date01[2] - bdd$date02[2])^2)  
bdd$dif[3] <- sqrt((bdd$date01[3] - bdd$date02[3])^2)  
# ...  
bdd$dif[100] <- sqrt((bdd$date01[100] - bdd$date02[100])^2)  
  
if(bdd$dif[1] > 3){  
 bdd$isDifBig[1] <- "big"  
}else{  
 bdd$isDifBig[1] <- "small"  
}  
if(bdd$dif[2] > 3){  
 bdd$isDifBig[2] <- "big"  
}else{  
 bdd$isDifBig[2] <- "small"  
}  
if(bdd$dif[3] > 3){  
 bdd$isDifBig[3] <- "big"  
}else{  
 bdd$isDifBig[3] <- "small"  
}  
# ...  
if(bdd$dif[100] > 3){  
 bdd$isDifBig[100] <- "big"  
}else{  
 bdd$isDifBig[100] <- "small"  
}

Cette façon de faire serait extrêmement fastidieuse à réaliser, et même presque impossible à réaliser si la table contenait 1000 ou 100000 lignes. Il pourrait sembler logique de vouloir itérer sur les lignes de notre data.frame pour obtenir les nouvelles colonnes. Nous allons le réaliser même si ce n’est pas la solution que nous allons retenir par la suite.

Nous allons utiliser une boucle for(). La boucle for() va itérer sur les éléments d’un objet que nous allons donner en argument. Par exemple voici une boucle qui pour tous les chiffres de 3 à 9 va calculer leur valeur au carré. La valeur courante du chiffre est symbolisé par un objet qui peut prendre le nom que nous souhaitons (ici cela sera i).

for(i in c(3, 4, 5, 6, 7, 8, 9)){  
 print(i^2)  
}

## [1] 9  
## [1] 16  
## [1] 25  
## [1] 36  
## [1] 49  
## [1] 64  
## [1] 81

Que nous pouvons améliorer en utilisant [la fonction :](#l0152points).

for(i in 3:9){  
 print(i^2)  
}

Le choix du nom i est ici arbitraire, nous aurions pu choisir myVarFor de la même façon :

for(myVarFor in 3:9){  
 print(myVarFor^2)  
}

La boucle for() peut itérer sur tous les types d’éléments.

nChar <- c("a", "z", "e", "r", "t", "y")  
for(i in nChar){  
 print(i)  
}

## [1] "a"  
## [1] "z"  
## [1] "e"  
## [1] "r"  
## [1] "t"  
## [1] "y"

Revenons à notre cas. Nous allons itérer sur le nombre de lignes de notre data.frame bdd. Avant cela nous allons créer les colonnes dif et isDifBig avec les valeurs NA. Ensuite nous allons utiliser [la fonction nrow()](#l015nrow) pour connaître le nombre de lignes.

bdd$dif <- NA  
bdd$isDifBig <- NA  
for(i in 1:nrow(bdd)){  
 bdd$dif[i] <- sqrt((bdd$date01[i] - bdd$date02[i])^2)  
 if(bdd$dif[i] > 3){  
 bdd$isDifBig[i] <- "big"  
 }else{  
 bdd$isDifBig[i] <- "small"  
 }  
}  
print(head(bdd, n = 20))

## date01 date02 dif isDifBig  
## 1 9.324726 9.387889 0.063163220 small  
## 2 10.246147 9.994158 0.251988954 small  
## 3 11.384945 9.772913 1.612031888 small  
## 4 9.083354 12.319240 3.235886251 big  
## 5 8.097771 10.538057 2.440285846 small  
## 6 10.879128 9.532593 1.346535194 small  
## 7 9.528947 10.420441 0.891494151 small  
## 8 10.816466 10.637327 0.179139074 small  
## 9 8.498380 8.648135 0.149754671 small  
## 10 9.446408 10.081401 0.634992556 small  
## 11 10.205497 10.213789 0.008292681 small  
## 12 9.455219 8.632372 0.822846120 small  
## 13 8.864346 10.019548 1.155201748 small  
## 14 11.199604 10.444301 0.755302816 small  
## 15 10.961770 9.388242 1.573528223 small  
## 16 10.201865 10.961367 0.759502067 small  
## 17 8.248711 9.984930 1.736219166 small  
## 18 10.541660 9.231123 1.310537058 small  
## 19 8.595878 10.642203 2.046325099 small  
## 20 10.137996 10.194916 0.056919322 small

En pratique ce n’est pas la meilleure façon de réaliser cet exercice car il s’agit ici de simples calculs sur des vecteurs contenus dans une data.frame. R est particulièrement puissant pour effectuer des opérations sur des vecteurs. Lorsque cela est possible il faut donc toujours privilégier les opérations sur les vecteurs. Ici notre code devient :

bdd$dif <- sqrt((bdd$date01 - bdd$date02)^2)  
bdd$isDifBig <- "small"  
bdd$isDifBig[bdd$dif > 3] <- "big"  
print(head(bdd, n = 20))

## date01 date02 dif isDifBig  
## 1 9.324726 9.387889 0.063163220 small  
## 2 10.246147 9.994158 0.251988954 small  
## 3 11.384945 9.772913 1.612031888 small  
## 4 9.083354 12.319240 3.235886251 big  
## 5 8.097771 10.538057 2.440285846 small  
## 6 10.879128 9.532593 1.346535194 small  
## 7 9.528947 10.420441 0.891494151 small  
## 8 10.816466 10.637327 0.179139074 small  
## 9 8.498380 8.648135 0.149754671 small  
## 10 9.446408 10.081401 0.634992556 small  
## 11 10.205497 10.213789 0.008292681 small  
## 12 9.455219 8.632372 0.822846120 small  
## 13 8.864346 10.019548 1.155201748 small  
## 14 11.199604 10.444301 0.755302816 small  
## 15 10.961770 9.388242 1.573528223 small  
## 16 10.201865 10.961367 0.759502067 small  
## 17 8.248711 9.984930 1.736219166 small  
## 18 10.541660 9.231123 1.310537058 small  
## 19 8.595878 10.642203 2.046325099 small  
## 20 10.137996 10.194916 0.056919322 small

La plupart des exemples que l’on peut trouver sur Internet à propos de la boucle for() peuvent être remplacés par des opérations sur les vecteurs. Voici quelques exemples adaptés de plusieurs sources :

# tester si des nombres sont pairs  
# [1] FOR  
x <- sample(1:100, size = 20)  
count <- 0  
for (val in x) {  
 if(val %% 2 == 0){  
 count <- count + 1  
 }  
}  
print(count)

## [1] 8

# [2] VECTOR  
sum(x %% 2 == 0)

## [1] 8

# calculer des carrés  
# [1] FOR  
x <- rep(0, 20)  
for (j in 1:20){  
 x[j] <- j^2  
}  
print(x)

## [1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361  
## [20] 400

# [2] VECTOR  
(1:20)^2

## [1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361  
## [20] 400

# répéter un lancer de dés et faire la moyenne  
# [1] FOR  
ntrials = 1000  
trials = rep(0, ntrials)  
for (j in 1:ntrials){  
 trials[j] = sample(1:6, size = 1)  
}  
mean(trials)

## [1] 3.529

# [2] VECTOR  
mean(sample(1:6, ntrials, replace = TRUE))

## [1] 3.529

C’est un bon exercice de parcourir les nombreux exemples disponibles sur Internet sur la boucle for() et de tenter de les transformer en opérations vectorielles. Cela nous permet d’acquérir les bons réflexes de programmation avec R. La boucle for() reste très utile pour par exemple lire plusieurs fichiers et traiter l’information qu’ils contiennent de la même façon, faire des graphiques, ou encore lorsque les opérations vectorielles deviennent fastidieuses. Imaginons une matrice de 10 colonnes et 100 lignes. Nous voulons la somme de chaque ligne (nous verrons plus loin comment faire avec la fonction apply()).

myMat <- matrix(sample(1:100, size = 1000, replace = TRUE), ncol = 10)  
# VECTOR  
sumRow <- myMat[, 1] + myMat[, 2] + myMat[, 3] + myMat[, 4] +   
 myMat[, 5] + myMat[, 6] + myMat[, 7] + myMat[, 8] +   
 myMat[, 9] + myMat[, 10]  
print(sumRow)

## [1] 390 266 519 607 523 522 679 417 506 431 524 571 379 557 647 439 595 543  
## [19] 568 422 547 446 414 581 484 501 501 638 647 503 461 489 645 537 499 418  
## [37] 645 334 557 393 405 585 418 517 504 436 524 577 471 572 339 396 462 455  
## [55] 651 498 590 624 604 465 422 340 527 488 463 598 544 563 378 465 626 473  
## [73] 610 373 494 474 570 591 452 611 580 534 498 407 531 576 612 506 597 596  
## [91] 680 560 394 480 626 508 686 366 436 541

# FOR  
sumRow <- rep(NA, times = nrow(myMat))  
for(j in 1:nrow(myMat)){  
 sumRow[j] <- sum(myMat[j, ])  
}  
print(sumRow)

## [1] 390 266 519 607 523 522 679 417 506 431 524 571 379 557 647 439 595 543  
## [19] 568 422 547 446 414 581 484 501 501 638 647 503 461 489 645 537 499 418  
## [37] 645 334 557 393 405 585 418 517 504 436 524 577 471 572 339 396 462 455  
## [55] 651 498 590 624 604 465 422 340 527 488 463 598 544 563 378 465 626 473  
## [73] 610 373 494 474 570 591 452 611 580 534 498 407 531 576 612 506 597 596  
## [91] 680 560 394 480 626 508 686 366 436 541

En conclusion, il est recommandé de ne pas utiliser la boucle for() avec R chaque fois que cela est possible, et nous verrons dans ce chapitre des alternatives comme les boucles de la famille apply().

## La boucle while

La boucle while(), contrairement à la boucle for(), signifie *TANT QUE*. Tant qu’une condition n’est pas remplie, la boucle va continuer à s’exécuter. Attention car en cas d’erreur, nous pouvons facilement programmer des boucles qui ne terminent jamais ! Cette boucle est moins courante que la boucle for(). Prenons un exemple :

i <- 0  
while(i < 10){  
 print(i)  
 i <- i + 1  
}

## [1] 0  
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6  
## [1] 7  
## [1] 8  
## [1] 9

Dans cet exemple, la variable i a comme valeur 0. TANT QUE i < 10, nous allons afficher i. Pour que cette boucle se termine, il ne faut pas oublier de modifier la valeur de i, c’est ce qui est fait avec la ligne i <- i + 1. Lorsque la condition i < 10 ne se vérifie plus, la boucle s’arrête.

La boucle while() est très utile pour créer des scripts qui vont effectuer des calculs sur des variables dont la valeur évoluent dans le temps. Par exemple imaginons un nombre entre 0 et 10000 et un générateur aléatoire qui va essayer de déterminer la valeur de ce nombre. Si nous souhaitons limiter les tentatives de R à 2 secondes, nous pouvons écrire le script suivant (qui devrait marcher à tous les coups sur un ordinateur de bureau classique pouvant facilement effectuer 35000 essais en 2 secondes) :

myNumber <- sample(x = 10000, size = 1)  
myGuess <- sample(x = 10000, size = 1)  
startTime <- Sys.time()  
numberGuess <- 0  
while(Sys.time() - startTime < 2){  
 if(myGuess == myNumber){  
 numberGuess <- numberGuess + 1  
 print("Number found !")  
 print(paste0("And I have plenty of time left: ",   
 round(2 - as.numeric(Sys.time() - startTime), digits = 2),   
 " sec"))  
 break  
 }else{  
 myGuess <- sample(x = 10000, size = 1)  
 numberGuess <- numberGuess + 1  
 }  
}

## [1] "Number found !"  
## [1] "And I have plenty of time left: 1.51 sec"

Dans ce script nous générons un nombre aléatoire à deviner avec la fonction sample(), et chaque essai avec la même fonction sample(). Ensuite nous utilisons la fonction Sys.time() (avec un S majuscule à Sys), pour connaître l’heure de début de la boucle. Tant que la différence entre chaque itération de la boucle et l’heure de démarrage est inférieur à 2 secondes, la boucle while() va vérifier si le bon nombre a été deviné dans le test logique avec if() puis si c’est le cas nous informer que le nombre a été trouvé, nous dire le temps restant avant les deux secondes, puis terminer la boucle avec le mot clef break sur lequel nous reviendrons. En bref, break permet de sortir d’une boucle. Si le nombre n’a pas été deviné, la boucle fait un autre essai avec la fonction sample().

Plus concrètement nous pourrions imaginer des algorithmes pour explorer un espace de solutions face à un problème avec un temps limité pour y parvenir. La boucle while() peut aussi être pratique pour qu’un script ne s’exécute que lorsque un fichier issu d’un autre programme devient disponible… En pratique la boucle while() reste peu utilisée avec R, car ce genre de script qui manipule des informations système (temps, fichiers, …) est plus communément réalisé avec d’autres langages comme Python.

## La boucle repeat

La boucle repeat() permet de répéter une opération sans condition à vérifier. Pour sortir de cette boucle il faut donc obligatoirement utiliser le mot clef break.

i <- 1  
repeat{  
 print(i^2)  
 i <- i + 1  
 if(i == 5){  
 break  
 }  
}

## [1] 1  
## [1] 4  
## [1] 9  
## [1] 16

Si nous reprenons l’exemple précédent, nous pouvons utiliser une boucle repeat() pour le répéter 5 fois.

numTry <- 0  
repeat{  
 myNumber <- sample(x = 10000, size = 1)  
 myGuess <- sample(x = 10000, size = 1)  
 startTime <- Sys.time()  
 numberGuess <- 0  
 while(Sys.time() - startTime < 2){  
 if(myGuess == myNumber){  
 numberGuess <- numberGuess + 1  
 print(round(as.numeric(Sys.time() - startTime), digits = 3))  
 break  
 }else{  
 myGuess <- sample(x = 10000, size = 1)  
 numberGuess <- numberGuess + 1  
 }  
 }  
 numTry <- numTry + 1  
 if(numTry == 5){break}  
}

## [1] 0.195  
## [1] 0.157  
## [1] 0.455  
## [1] 0.122  
## [1] 0.115

Comme la boucle while(), la boucle repeat() reste peu utilisée avec R.

## next et break

Nous avons déjà vu le mot clef break qui permet de sortir de la boucle en cours. Par exemple si nous cherchons le premier nombre après 111 qui soit divisible par 32 :

myVars <- 111:1000  
for(myVar in myVars){  
 if(myVar %% 32 == 0){  
 print(myVar)  
 break  
 }  
}

## [1] 128

Même si nous avons vu que dans la pratique nous pouvons éviter la boucle for() avec une opération sur les vecteurs. Avec une opération sur les vecteurs nous pourrions facilement calculer tous les nombres entre 111 et 1000 qui sont divisibles par 32.

(111:1000)[111:1000 %% 32 == 0]

## [1] 128 160 192 224 256 288 320 352 384 416 448 480 512 544 576 608 640 672 704  
## [20] 736 768 800 832 864 896 928 960 992

Le premier est 128, que nous pouvons récupérer avec [1].

(111:1000)[111:1000 %% 32 == 0][1]

## [1] 128

Le mot clef next permet quant à lui de passer à l’itération suivante d’une boucle si une certaine condition est remplie. Par exemple si nous voulons imprimer les lettres de l’alphabet sans les voyelles :

for(myLetter in letters){  
 if(myLetter %in% c("a", "e", "i", "o", "u", "y")){  
 next  
 }  
 print(myLetter)  
}

## [1] "b"  
## [1] "c"  
## [1] "d"  
## [1] "f"  
## [1] "g"  
## [1] "h"  
## [1] "j"  
## [1] "k"  
## [1] "l"  
## [1] "m"  
## [1] "n"  
## [1] "p"  
## [1] "q"  
## [1] "r"  
## [1] "s"  
## [1] "t"  
## [1] "v"  
## [1] "w"  
## [1] "x"  
## [1] "z"

Encore une fois nous pourrions éviter la boucle for() avec :

letters[! letters %in% c("a", "e", "i", "o", "u", "y")]

## [1] "b" "c" "d" "f" "g" "h" "j" "k" "l" "m" "n" "p" "q" "r" "s" "t" "v" "w" "x"  
## [20] "z"

En conclusion, si nous utilisons les boucles, alors les mots clefs next et break sont souvent très utiles, mais chaque fois que cela est possible il vaut mieux avoir recours à des opérations sur les vecteurs. Lorsque cela n’est pas possible de travailler sur les vecteurs, il est préférable d’utiliser les boucles de la famille apply qui sont le sujet de la prochaine section.

## Les boucles de la famille apply

### apply

La fonction apply() permet d’appliquer une fonction à tous les éléments d’un array ou d’une matrix. Par exemple si nous souhaitons connaître la somme de chaque ligne d’une matrix de 10 colonnes et 100 lignes :

myMat <- matrix(sample(1:100, size = 1000, replace = TRUE), ncol = 10)  
apply(X = myMat, MARGIN = 1, FUN = sum)

## [1] 519 382 496 438 462 519 421 496 498 557 567 645 231 443 475 617 605 480  
## [19] 349 384 647 621 512 401 373 482 568 595 577 510 482 480 624 536 337 464  
## [37] 587 420 522 521 407 573 506 573 463 427 538 358 418 559 556 390 507 537  
## [55] 474 499 497 427 427 590 504 528 490 665 569 589 512 320 417 575 416 545  
## [73] 424 620 469 616 648 581 649 342 607 514 523 349 544 579 564 531 625 407  
## [91] 558 782 512 393 322 547 382 701 467 517

Si nous souhaitons connaître la médiane de chaque colonne, l’expression devient :

apply(X = myMat, MARGIN = 2, FUN = median)

## [1] 50.5 48.5 50.0 52.5 53.0 49.5 53.5 50.5 45.5 50.5

L’argument X correspond à l’objet sur lequel la boucle apply va itérer. L’argument MARGIN correspond à la dimension à prendre en compte (1 pour les lignes, et 2 pour les colonnes). L’argument FUN correspond à la fonction à appliquer. Sur un objet de type array, l’argument MARGIN peut prendre autant de valeurs que de dimensions. Dans cet exemple MARGIN = 1 correspond à la moyenne de chaque ligne - dimension 1 - (toutes dimensions confondues), MARGIN = 2 correspond à la moyenne de chaque colonne - dimension 2 - (toutes dimensions confondues), et MARGIN = 3 correspond à la moyenne de chaque dimension 3. Ci-dessous chaque calcul est réalisé de deux manières différentes pour en expliciter le fonctionnement.

# création de la matrix  
myArr <- array(sample(1:100, size = 1000, replace = TRUE), dim = c(10, 20, 5))  
# Calcul avec APPLY  
apply(X = myArr, MARGIN = 1, FUN = mean)

## [1] 49.33 51.62 52.75 51.16 53.76 53.42 48.03 49.24 52.35 51.09

# Calcul "à la main"  
(apply(myArr[,,1], 1, mean) + apply(myArr[,,2], 1, mean) +   
 apply(myArr[,,3], 1, mean) + apply(myArr[,,4], 1, mean) +   
 apply(myArr[,,5], 1, mean))/5

## [1] 49.33 51.62 52.75 51.16 53.76 53.42 48.03 49.24 52.35 51.09

# Calcul avec APPLY  
apply(X = myArr, MARGIN = 2, FUN = mean)

## [1] 51.86 50.18 52.94 46.36 50.52 48.08 54.00 44.06 58.34 52.84 49.94 51.28  
## [13] 53.34 47.10 54.72 50.98 60.08 54.04 51.14 43.70

# Calcul "à la main"  
(apply(myArr[,,1], 2, mean) + apply(myArr[,,2], 2, mean) +   
 apply(myArr[,,3], 2, mean) + apply(myArr[,,4], 2, mean) +   
 apply(myArr[,,5], 2, mean))/5

## [1] 51.86 50.18 52.94 46.36 50.52 48.08 54.00 44.06 58.34 52.84 49.94 51.28  
## [13] 53.34 47.10 54.72 50.98 60.08 54.04 51.14 43.70

# Calcul avec APPLY  
apply(X = myArr, MARGIN = 3, FUN = mean)

## [1] 48.490 53.820 54.585 49.360 50.120

# Calcul "à la main"  
c(mean(myArr[,,1]), mean(myArr[,,2]), mean(myArr[,,3]),   
 mean(myArr[,,4]), mean(myArr[,,5]))

## [1] 48.490 53.820 54.585 49.360 50.120

Nous pouvons aussi calculer la moyenne pour chaque valeur de ligne et de colonne (la fonction itère alors sur la dimension 3) :

apply(X = myArr, MARGIN = c(1, 2), FUN = mean)

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]  
## [1,] 42.6 43.8 58.6 47.4 59.2 46.4 56.8 53.8 54.6 53.8 53.2 34.2 69.8  
## [2,] 39.4 58.2 45.8 48.8 47.4 46.2 59.4 40.6 46.0 61.4 67.0 71.2 45.4  
## [3,] 69.8 54.2 42.0 58.8 43.2 46.0 54.2 54.4 77.8 47.6 40.4 69.2 48.2  
## [4,] 40.4 65.6 70.0 44.8 39.8 55.2 43.8 48.4 71.2 62.4 32.6 42.2 40.6  
## [5,] 47.8 48.6 48.6 53.2 62.6 44.0 51.2 40.6 55.2 69.6 49.0 50.4 62.8  
## [6,] 50.6 52.6 62.0 49.2 62.2 57.0 60.0 40.0 54.8 42.6 62.8 43.2 61.8  
## [7,] 71.2 46.0 36.2 36.4 41.6 42.2 45.0 50.8 57.0 51.2 46.8 36.2 53.4  
## [8,] 31.8 53.4 58.6 48.0 55.2 27.4 44.8 15.0 57.0 53.4 40.4 61.6 33.6  
## [9,] 62.8 32.6 44.2 54.6 43.0 56.8 55.0 39.2 44.6 45.8 61.6 45.6 57.8  
## [10,] 62.2 46.8 63.4 22.4 51.0 59.6 69.8 57.8 65.2 40.6 45.6 59.0 60.0  
## [,14] [,15] [,16] [,17] [,18] [,19] [,20]  
## [1,] 54.4 57.4 49.6 37.6 31.0 39.6 42.8  
## [2,] 37.4 51.0 48.6 71.8 31.8 76.8 38.2  
## [3,] 44.8 34.4 42.2 65.8 35.6 68.2 58.2  
## [4,] 34.0 61.4 54.0 57.2 74.8 32.6 52.2  
## [5,] 42.8 57.4 34.2 77.2 75.8 67.0 37.2  
## [6,] 52.4 70.0 68.4 53.8 61.6 43.0 20.4  
## [7,] 74.2 45.8 48.0 46.0 40.2 38.6 53.8  
## [8,] 32.6 55.0 69.6 73.8 58.8 62.0 52.8  
## [9,] 51.0 68.2 41.2 67.6 78.0 48.2 49.2  
## [10,] 47.4 46.6 54.0 50.0 52.8 35.4 32.2

### lapply

Comme indiqué dans la documentation, lapply() renvoie une liste de même longueur que X, chaque élément résultant de l’application de FUN à l’élément correspondant de X. Si X est une list contenant des vector et que nous cherchons à obtenir la moyenne de chacun des éléments de la list, nous pouvons utiliser la fonction lapply() :

# Création de l'objet myList  
myList <- list(  
 a = sample(1:100, size = 10),   
 b = sample(1:100, size = 10),   
 c = sample(1:100, size = 10),   
 d = sample(1:100, size = 10),   
 e = sample(1:100, size = 10)  
)  
print(myList)

## $a  
## [1] 31 68 81 44 84 50 27 79 55 33  
##   
## $b  
## [1] 84 32 85 44 55 64 50 27 51 8  
##   
## $c  
## [1] 56 15 59 63 46 70 40 90 66 10  
##   
## $d  
## [1] 74 52 53 68 76 80 61 72 22 77  
##   
## $e  
## [1] 19 77 73 21 49 65 17 24 56 68

# Fonction moyenne appliquée à chacun des  
# éléments de la list  
lapply(myList, FUN = mean)

## $a  
## [1] 55.2  
##   
## $b  
## [1] 50  
##   
## $c  
## [1] 51.5  
##   
## $d  
## [1] 63.5  
##   
## $e  
## [1] 46.9

Cela revient à une boucle for() itérant sur les élément de l’objet de type list.

for(i in 1:length(myList)){  
 mean(myList[[i]])  
}

Cependant, la boucle for() ne renvoie pas de résultat. Pour l’obtenir, nous devons au préalable créer un objet qui va contenir notre résultat. A chaque itération de la boucle for(), le résultat sera affecté à l’élément correspondant de notre objet.

myResult <- list(a = NULL, b = NULL, c = NULL,   
 d = NULL, e = NULL)  
for(i in 1:length(myList)){  
 myResult[[i]] <- mean(myList[[i]])  
}  
print(myResult)

## $a  
## [1] 55.2  
##   
## $b  
## [1] 50  
##   
## $c  
## [1] 51.5  
##   
## $d  
## [1] 63.5  
##   
## $e  
## [1] 46.9

Nous comprenons alors que la fonction lapply() est bien plus performante car elle va à la fois permettre de procéder à notre tâche tout en fournissant un résultat qui peut être stocké directement dans un objet (ou pas comme c’est le cas ici).

Comme pour la fonction apply(), nous pouvons passer des arguments supplémentaires à la fonction lapply() en les ajoutant à la suite de la fonction. C’est par exemple utile si notre list contient ces valeurs manquantes NA et que nous voulons les ignorer pour le calcul des moyennes (avec l’argument na.rm = TRUE).

# Création de l'objet myList  
myList <- list(  
 a = sample(c(1:5, NA), size = 10, replace = TRUE),   
 b = sample(c(1:5, NA), size = 10, replace = TRUE),   
 c = sample(c(1:5, NA), size = 10, replace = TRUE),   
 d = sample(c(1:5, NA), size = 10, replace = TRUE),   
 e = sample(c(1:5, NA), size = 10, replace = TRUE)  
)  
print(myList)

## $a  
## [1] 1 NA 1 5 2 4 2 5 3 5  
##   
## $b  
## [1] NA 2 1 5 1 5 3 5 2 1  
##   
## $c  
## [1] 3 2 4 2 3 3 1 NA 5 2  
##   
## $d  
## [1] 1 2 4 3 1 3 1 1 5 2  
##   
## $e  
## [1] NA 4 4 NA 3 4 2 NA 2 2

lapply(myList, FUN = mean)

## $a  
## [1] NA  
##   
## $b  
## [1] NA  
##   
## $c  
## [1] NA  
##   
## $d  
## [1] 2.3  
##   
## $e  
## [1] NA

lapply(myList, FUN = mean, na.rm = TRUE)

## $a  
## [1] 3.111111  
##   
## $b  
## [1] 2.777778  
##   
## $c  
## [1] 2.777778  
##   
## $d  
## [1] 2.3  
##   
## $e  
## [1] 3

Pour plus de lisibilité ou si plusieurs opérations sont à réaliser au sein de l’argument FUN, nous pouvons utiliser l’écriture suivante :

lapply(myList, FUN = function(i){  
 mean(i, na.rm = TRUE)  
})

## $a  
## [1] 3.111111  
##   
## $b  
## [1] 2.777778  
##   
## $c  
## [1] 2.777778  
##   
## $d  
## [1] 2.3  
##   
## $e  
## [1] 3

Par exemple si nous souhaitons obtenir i^2 si la moyenne est supérieure à 3, et i^3 sinon :

lapply(myList, FUN = function(i){  
 m <- mean(i, na.rm = TRUE)  
 if(m > 3){  
 return(i^2)   
 }else{  
 return(i^3)  
 }  
})

## $a  
## [1] 1 NA 1 25 4 16 4 25 9 25  
##   
## $b  
## [1] NA 8 1 125 1 125 27 125 8 1  
##   
## $c  
## [1] 27 8 64 8 27 27 1 NA 125 8  
##   
## $d  
## [1] 1 8 64 27 1 27 1 1 125 8  
##   
## $e  
## [1] NA 64 64 NA 27 64 8 NA 8 8

### sapply

La fonction sapply() est une version modifiée de la fonction lapply() qui effectue la même opération mais en renvoyant le résultat sous un format simplifié lorsque c’est possible.

lapply(myList, FUN = function(i){  
 mean(i, na.rm = TRUE)  
})

## $a  
## [1] 3.111111  
##   
## $b  
## [1] 2.777778  
##   
## $c  
## [1] 2.777778  
##   
## $d  
## [1] 2.3  
##   
## $e  
## [1] 3

sapply(myList, FUN = function(i){  
 mean(i, na.rm = TRUE)  
})

## a b c d e   
## 3.111111 2.777778 2.777778 2.300000 3.000000

La fonction sapply() est intéressante pour récupérer par exemple le nième élément de chacun des éléments d’une list. La fonction qui est appelée pour faire cela est '[['.

sapply(myList, FUN = '[[', 2)

## a b c d e   
## NA 2 2 2 4

### tapply

La fonction tapply() permet d’appliquer une fonction en prenant comme élément à itérer une variable existante. Imaginons des informations sur des espèces représentées par des lettres majuscules (e.g., A, B, C) et des valeurs de performances à différentes localisations.

species <- sample(LETTERS[1:10], size = 1000, replace = TRUE)  
perf1 <- rnorm(n = 1000, mean = 10, sd = 0.5)  
perf2 <- rlnorm(n = 1000, meanlog = 10, sdlog = 0.5)  
perf3 <- rgamma(n = 1000, shape = 10, rate = 0.5)  
dfSpecies <- data.frame(species, perf1, perf2, perf3)  
print(head(dfSpecies, n = 10))

## species perf1 perf2 perf3  
## 1 B 10.606103 44425.576 11.140060  
## 2 E 10.641796 24183.836 12.014162  
## 3 F 10.363385 31580.038 13.961420  
## 4 B 10.163289 29614.367 17.790957  
## 5 I 10.266055 21946.675 20.958287  
## 6 H 9.405259 15754.040 9.301748  
## 7 I 10.052885 42516.656 14.289287  
## 8 C 9.566599 11304.199 25.199471  
## 9 I 9.770159 6752.326 7.823470  
## 10 F 9.598935 22765.329 33.245761

Nous pouvons facilement obtenir un résumé des performances pour chaque espèce avec la fonction tapply() et la fonction summary().

tapply(dfSpecies$perf1, INDEX = dfSpecies$species, FUN = summary)

## $A  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.816 9.729 10.078 9.993 10.255 11.423   
##   
## $B  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.393 9.750 10.101 10.042 10.355 11.134   
##   
## $C  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.400 9.698 10.014 10.014 10.297 11.269   
##   
## $D  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.952 9.654 9.975 10.014 10.396 11.308   
##   
## $E  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.811 9.773 10.083 10.128 10.488 11.488   
##   
## $F  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.910 9.624 9.984 9.971 10.293 11.118   
##   
## $G  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.483 9.817 10.092 10.064 10.354 11.355   
##   
## $H  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 9.063 9.629 9.951 9.990 10.328 11.135   
##   
## $I  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.200 9.757 9.989 10.020 10.396 11.284   
##   
## $J  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 8.970 9.575 9.959 10.005 10.380 11.230

Nous pouvons aussi obtenir la valeur moyenne de chacune des performances en combinant une fonction sapply() avec la fonction tapply()et en utilisant la fonction mean().

sapply(2:4, FUN = function(i){  
 tapply(dfSpecies[,i], INDEX = dfSpecies$species, FUN = mean)  
})

## [,1] [,2] [,3]  
## A 9.992865 25813.78 19.80805  
## B 10.042099 26221.12 19.80784  
## C 10.014167 24150.18 19.78282  
## D 10.013754 23973.87 19.36892  
## E 10.127837 23840.73 20.73716  
## F 9.970511 24257.74 21.24172  
## G 10.063579 24493.89 19.38913  
## H 9.990043 25125.72 20.25617  
## I 10.019617 23751.32 19.77898  
## J 10.004819 24605.83 19.95974

La fonction sapply() est utilisée pour spécifier la colonne (la colonne 2 correspond à perf1, la colonne 3 correspond à perf2, et la colonne 4 correspond à perf3). La colonne prend alors le nom de variable i. Ensuite, pour chaque colonne dfSpecies[,i], nous pouvons utiliser la fonction tapply() pour calculer la moyenne par espèce. Voici l’équivalent avec des boucles for().

for(i in 2:4){  
 for(spp in levels(dfSpecies$species)){  
 mean(dfSpecies[,i][dfSpecies$species == spp])  
 }  
}

Et si nous voulons le résultat sous forme d’une data.frame, le code devient beaucoup plus long et fastidieux (le code suivant correspond à une solution parmi d’autres). Là encore nous voyons toute la puissance des boucles de la famille apply.

for(i in 2:4){  
 myPerf <- rep(NA, length(levels(dfSpecies$species)))  
 for(spp in seq\_along(levels(dfSpecies$species))){  
 myPerf[spp] <- mean(  
 dfSpecies[,i][dfSpecies$species == levels(dfSpecies$species)[spp]])  
 }  
 assign(x = paste0("perf", i - 1), value = myPerf)  
}  
myDF <- data.frame(perf1, perf2, perf3)  
rownames(myDF) <- levels(dfSpecies$species)  
print(myDF)

## perf1 perf2 perf3  
## A 9.992865 25813.78 19.80805  
## B 10.042099 26221.12 19.80784  
## C 10.014167 24150.18 19.78282  
## D 10.013754 23973.87 19.36892  
## E 10.127837 23840.73 20.73716  
## F 9.970511 24257.74 21.24172  
## G 10.063579 24493.89 19.38913  
## H 9.990043 25125.72 20.25617  
## I 10.019617 23751.32 19.77898  
## J 10.004819 24605.83 19.95974

### mapply

La fonction mapply() est une version de la fonction sapply()qui utilise de multiples arguments. Par exemple si nous avons une liste de deux éléments 1:5 et 5:1 et que nous souhaitons ajouter 10 au premier élément et 100 au deuxième élément :

mapply(  
 FUN = function(i, j){i+j},   
 i = list(1:5, 5:1),   
 j = c(10, 100))

## [,1] [,2]  
## [1,] 11 105  
## [2,] 12 104  
## [3,] 13 103  
## [4,] 14 102  
## [5,] 15 101

## Conclusion

Félicitations, nous sommes arrivés au bout de ce chapitre sur l’algorithmique. Retenons ce message clef : dès qu’une opération doit être réalisée plus de deux fois dans un script en répétant du code qui a été déjà écrit, c’est un signe qui doit nous conduire à l’utilisation d’une boucle. Pour autant, chaque fois que cela est possible, il est recommandé de ne pas utiliser les boucles traditionnelles for(), while(), et repeat(), mais de préférer les opérations sur les vecteurs ou encore les boucles de la famille apply. Cela peut être assez difficile à intégrer au début mais nous verrons que nos scripts seront plus faciles à maintenir et à lire, et beaucoup plus performants si nous suivons ces habitudes.

# Gestion d’un projet avec R

Maintenant Que nous avons vu les concepts de base de R, il nous reste à aborder un élément déterminant pour le bon déroulement de nos activités scientifiques avec R : la gestion de projet. Cela consiste à intégrer ses développements dans un environnement et avec une logique visant à faciliter son travail et donc augmenter son efficacité. Il s’agit ici que d’une façon de faire parmis les inombrables possibilités, à adapter pour chacun et chacune.

## Gestion des fichiers et des répertoires de travail

Entre les fichiers d’entrée (i.e., les fichiers qui contiennent nos données brutes), les fichiers de sortie (e.g., avec la fonction write()), les graphiques (prochain chapitre), et les nombreux scripts associés à un projet de recherche, un minimum d’organisation s’impose pour pouvoir être efficace et reprendre rapidement son projet en cours. La solution la plus simple consiste à structurer son environnement de travail en dossiers en fonction de chaque catégorie de fichiers. Par exemple avec un dossier “myProject” pour le projet en cours, contenant lui-même les dossiers “myFiles” pour les fichiers d’entrée, un dossier “myScripts” pour les fichiers script R, et un dossier “myOutputs” pour les fichiers de sortie (e.g., les graphiques et les analyses).

## -myProject  
## |-myFiles  
## |-|-data01.csv  
## |-|-data02.csv  
## |-myScripts  
## |-|-myFirstScript.R  
## |-myOutputs  
## |-|-dataOut01.csv  
## |-|-figure01.pdf

## Gestion des versions de script

Le travail sur un script est itératif : même si les objectifs sont définis dès le départ, nous allons retravailler certaines parties pour obtenir par exemples des informations supplémentaires, ou encore pour optimiser telle ou telle fonction, ou encore rendre généralisable un script pour le communiquer à la communauté scientifique ou tout simplement à un collègue. Parfois ce que nous allons voir comme une amélioration va au final se révéler être une erreur, et le retour au script initial peut être difficile. Il faut donc gérer des versions.

Dans la plupart des laboratoire il y a des services de gestion des versions, les plus connus étant GIT (<https://git-scm.com/>) et Subversion (<https://subversion.apache.org/>). Lorsque GIT ou Subversion sont disponibles il est recommandé de les utiliser. Si nous n’avons pas accès à ces services il existe des services gratuits en ligne comme GitHub (<https://github.com/> ; ce livre utilise GitHub). Il existe de nombreuses autres solutions comme GitLab (<https://about.gitlab.com/>), Bitbucket (<https://bitbucket.org/>), SourceForge (<https://sourceforge.net/>), GitKraken (<https://www.gitkraken.com/>), ou encore Launchpad (<https://launchpad.net/>).

L’utilisation de ces différents services de gestion des versions sort du cadre de ce livre. Pour le débutant ou pour les projets ne nécessitant pas un travail collaboratif sur les scripts, une alternative consiste à gérer ses versions manuellement. Par exemple une solution consiste à ajouter un numéro à la fin de son nom de fichier de script (e.g., “myFirstScript\_01.R”). Dès qu’une modification importante est apportée à ce script, il suffira alors de le sauver avec un nouveau nom (e.g., “myFirstScript\_02.R”) et de placer l’ancien script dans un dossier d’archive pour ne pas encombrer l’espace de travail et risquer des erreurs. En cas de problème, nous pourrons ainsi facilement retourner au script antérieur et reprendre notre travail.

## -myProject  
## |-myFiles  
## |-|-data01.csv  
## |-|-data02.csv  
## |-myScripts  
## |-|-myFirstScript04.R  
## |-|-ARCHIVES  
## |-|-|-myFirstScript01.R  
## |-|-|-myFirstScript02.R  
## |-|-|-myFirstScript03.R  
## |-myOutputs  
## |-|-dataOut01.csv  
## |-|-figure01.pdf

## Gestion de la documentation

La documentation de son code est essentielle pour pouvoir facilement reprendre un travail ou communiquer son travail avec ses collègues et la communauté scientifique. Un code bien documenté sera compréhensible par un plus grand nombre et donc utilisé d’avantage. Il est donc important d’adopter de bonnes techniques.

Nous avons déjà vu qu’il y avait plusieurs façons d’écrire son code avec R car c’est un langage assez permissif. Le premier pas vers un code lisible et reproductible est donc d’adopter un style de code clair, cohérent, et… fait pour les humains ! Car même si notre code a vocation à être exécuter par les machines, il doit rester compréhensible pour soit et toutes les personnes qui seront amenées à le consulter. Il s’agit par exemple de mettre des espaces après les virgules, ou encore d’utiliser l’indentation. Bien sûr la lisibilité du code doit être à balancer avec l’optimisation du code pour les grands jeux de données, mais dans la plupart des cas nous pouvons associer un code clair et optimisé. Donc la première étape de la documentation et de sa gestion est tout d’abord de rédiger son code en pensant aux personnes qui vont le lire et le reproduire.

La deuxième étape est de commenter son code. Les commentaires sont indispensables lorsque nous privilégions du code optimisé pour la performance mais qui perd en lisibilité. Les commentaires sont souvent superflus si le code est bien rédigé et les objets et les fonctions bien nommés. Cela veut dire qu’il ne faut pas utiliser les commentaires pour expliquer un code mal rédigé, mais dès le début bien rédiger son code. Les commentaires sont utiles pour apporter des éléments de contexte (e.g., choix d’une méthode plutôt qu’une autre au regard de la littérature). La place des commentaires peut être en fin de ligne ou sur des lignes à part.

Pour un petit projet R il est indispensable que chaque script commence par une description de son contenu pour que nous puissions rapidement savoir de quoi il traite. C’est ce que nous avions fait au début de ce livre :

# ------------------------------------------------------------  
# Voici un script pour acquérir les concepts de base   
# avec R  
# date de création : 25/06/2018  
# auteur : François Rebaudo  
# ------------------------------------------------------------  
  
# [1] création de l'objet nombre de répétitions  
# ------------------------------------------------------------  
  
nbrRep <- 5  
  
# [2] calculs simples  
# ------------------------------------------------------------  
  
pi \* nbrRep^2

Ici les commentaires qui suivent l’en-tête ne sont pas nécessaires car le nom de l’objet se comprend de lui même. Notre fichier devient :

# ------------------------------------------------------------  
# Voici un script pour acquérir les concepts de base   
# avec R  
# date de création : 25/06/2018  
# auteur : François Rebaudo  
# ------------------------------------------------------------  
  
nbrRep <- 5  
pi \* nbrRep^2

Pour un gros projet avec de nombreuses fonctions destinées à être utilisées par d’autres usagers, il est préférable que la documentation du code soit à part, dans un fichier d’aide spécifique. C’est le cas de tous les packages R ! Pour gérer la documentation d’un package (et donc de toutes les fonctions), là encore il existe de nombreuses possibilités. La plus répandue consiste à utiliser la package R roxigen2. Sans entrer dans les détails, voici quelques examples issus de la documentation du package.

#' Add together two numbers  
#'  
#' @param x A number  
#' @param y A number  
#' @return The sum of \code{x} and \code{y}  
#' @examples  
#' add(1, 1)  
#' add(10, 1)  
add <- function(x, y) {  
 x + y  
}

#' Sum of vector elements.  
#'  
#' `sum` returns the sum of all the values present in its arguments.  
#'  
#' This is a generic function: methods can be defined for it directly  
#' or via the [Summary()] group generic. For this to work properly,  
#' the arguments `...` should be unnamed, and dispatch is on the  
#' first argument.  
sum <- function(..., na.rm = TRUE) {}

Cela permet d’écrire la documentation de chaque fonction à côté de la focntion. Le package roxigen2 va ensuite générer à partir de ces commentaires un document d’aide accessible avec la fonction '?'. A moins que nous écrivions un nouveau package, les commentaires simples suffiront, et le développement d’un package sort du cadre de ce livre.

## Conclusion

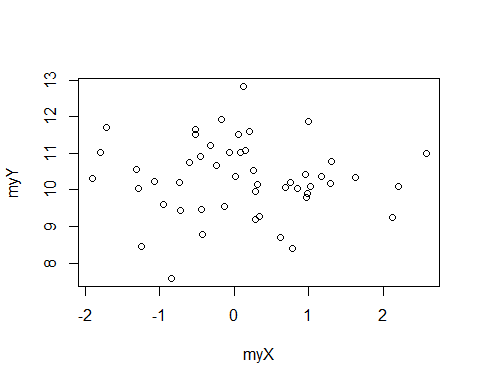
Félicitations ! Ce chapitre marque la fin de la première partie de ce livre. Nous avons désormais les bases pour mener à bien nos projets avec R. Dans la prochaine partie nous allons voir les graphiques et comment faire des figures dans le cadre d’articles scientifiques.

# Graphiques simples

## plot

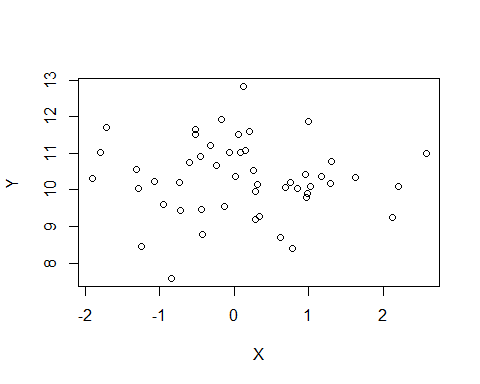
Le premier type de graphique que nous allons voir est le nuage de points. Dans un nuage de points, chaque point est représenté par sa valeur en x et en y. La fonction permettant de faire un nuage de points est plot().

myX <- rnorm(50, mean = 0, sd = 1)  
myY <- rnorm(50, mean = 10, sd = 1)  
plot(x = myX, y = myY)



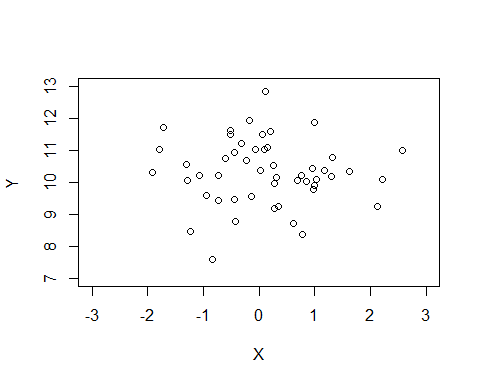
Comme pour tous les types de graphiques, nous pouvons ajouter une légende sur l’axe des x et des y.

plot(x = myX, y = myY,   
 xlab = "X", ylab = "Y")



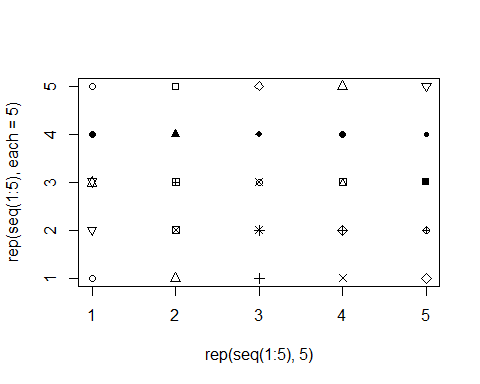
Nous pouvons aussi définir les limites des axes en X et en Y.

plot(x = myX, y = myY,   
 xlab = "X", ylab = "Y",   
 xlim = c(-3, 3), ylim = c(7, 13))

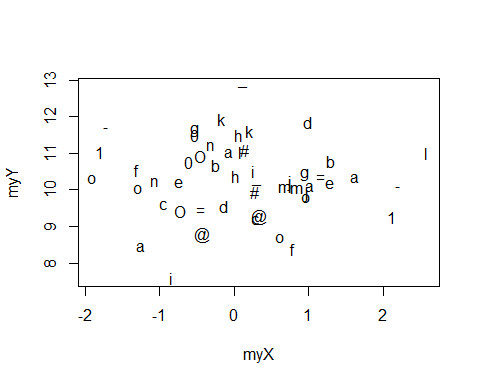


Le type de point peut être défini avec l’argument pch qui peut prendre un caractère ou un chiffre de 1 à 25.

plot(x = rep(seq(1:5), 5), y = rep(seq(1:5), each = 5),  
 pch = 1:25)

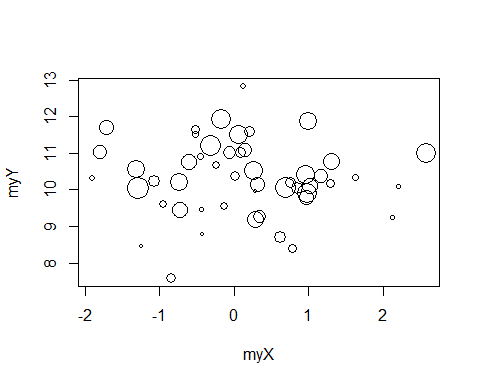


plot(x = myX, y = myY,   
 pch = c("a", "@", "#", "1", "=", "-", "\_", "o", "O", "0", letters[1:15]))



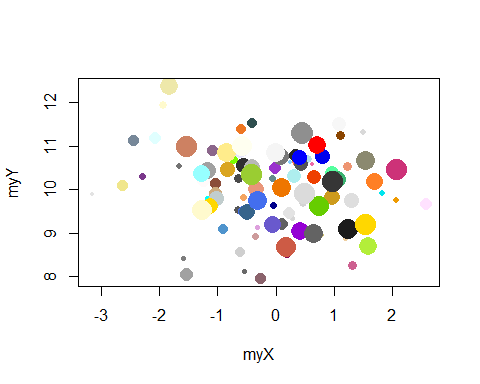
La taille des points peut se définir avec l’argument cex.

plot(x = myX, y = myY,   
 cex = seq(from = 0.5, to = 3, length.out = 50))



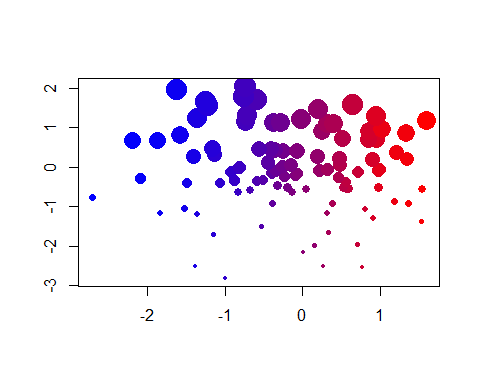
La couleur des points peut se définir avec l’argument col. Nous reviendrons sur les couleurs dans un prochain chapitre.

myX <- rnorm(100, mean = 0, sd = 1)  
myY <- rnorm(100, mean = 10, sd = 1)  
plot(x = myX, y = myY,   
 cex = seq(from = 0.5, to = 3, length.out = 100),  
 pch = 16,  
 col = sample(colors(), 100))



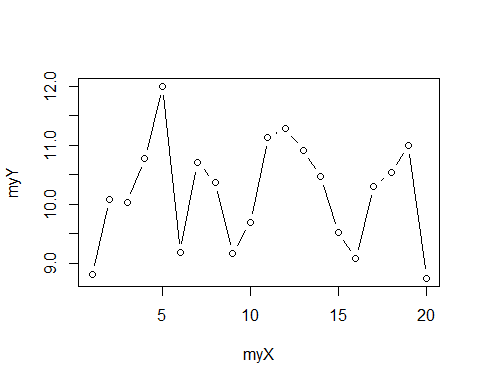
Pour représenter nos points la couleur et la taille des points peuvent représenter des informations complémentaires. Ici nous allons représenter par un gradient de taille la variable myY et par un gradient de couleur la variable myX.

myX <- rnorm(100)  
myY <- rnorm(100)  
dfGraph <- data.frame(myX, myY)  
dfGraph <- dfGraph[order(dfGraph$myX),]  
dfGraph$myCol <- colorRampPalette(c("blue", "red"))(100)  
dfGraph <- dfGraph[order(dfGraph$myY),]  
dfGraph$myCex <- seq(from = 0.5, to = 3, length.out = 100)  
plot(x = dfGraph$myX, y = dfGraph$myY,   
 cex = dfGraph$myCex, pch = 16, col = dfGraph$myCol,   
 xlab = "", ylab = "")



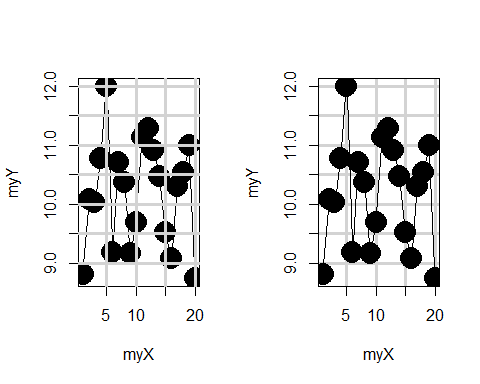
R offre la possibilité de relier les points des nuages de points de différentes façons. Les différentes options sont disponibles dans l’aide de la fonction plot() et plot.default().

myX <- 1:20  
myY <- rnorm(20, mean = 10, sd = 1)  
plot(x = myX, y = myY,   
 type = 'b') # 'p', 'l', 'b', 'c', 'o', 'h', 's', 'S', 'n'



Une dernière option très utile est l’argument panel.first qui permet de réaliser une opération graphique sur une couche située en dessous de notre graphique. Voici un exemple illustratif avec une grille réalisée avec et sans panel.first. La grille se fait grâce à la fonction grid(). Pour mettre les graphiques côte à côte nous allons utiliser mfrow.

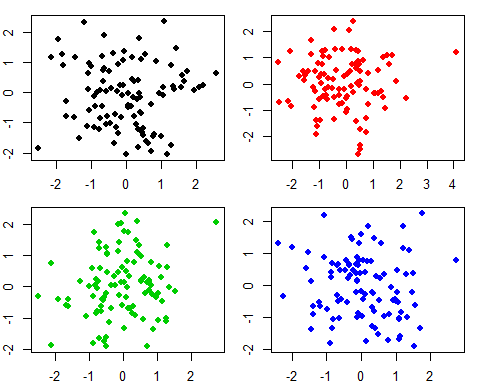
par(mfrow = c(1, 2))  
plot(x = myX, y = myY,   
 type = 'b', pch = 16, cex = 3)   
grid(lwd = 3, lty = 1)  
plot(x = myX, y = myY,   
 type = 'b', pch = 16, cex = 3,   
 panel.first = grid(lwd = 3, lty = 1))



par(mfrow = c(1, 1))

La fonction par() permet d’accéder aux paramètres graphiques. Parmi ces paramètres il y a mfrow qui permet de diviser l’espace graphique comme une matrice. mfrow prend comme arguments un vecteur numérique de taille 2 : le premier élément correspond au nombre de lignes et le deuxième élément au nombre de colonnes. Le paramètre mar permet de contrôler les marges en bas, à gauche, en haut et à droite, respectivement, au moyen d’un vecteur numérique de taille 4. Après avoir modifié les paramètres graphiques par défaut il est recommandé de les réinitialiser pour que cela n’affecte pas les graphiques à venir. Les valeurs par défaut de mfrow sont c(1, 1) et mar = c(4, 4, 4, 4). Nous pouvons remettre ces valeurs par défaut comme ci-dessus en redéfinissant chacun des paramètres. Nous pouvons également enregistrer au préalable les valeurs courantes (dans un objet op), puis les modifier pour les besoins de nos graphiques, puis ensuite rappeler les valeurs contenues dans l’objet op. Ici nous utilisons lapply pour réaliser rapidement quatre graphiques.

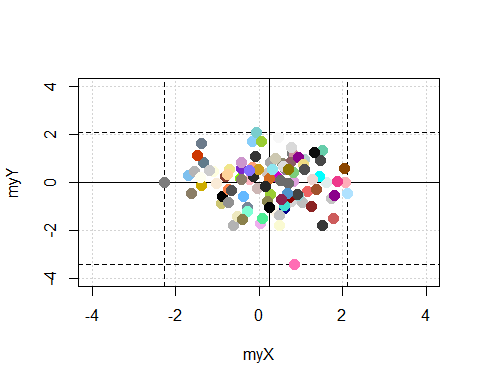
op <- par(no.readonly = TRUE)  
par(mfrow = c(2, 2), mar = c(2, 2, 1, 1))  
graph4 <- lapply(1:4, function(i){  
 plot(x = rnorm(100),   
 y = rnorm(100),   
 col = i, pch = 16)  
})



par(op)

Il est souvent utile de faire figurer des lignes verticales ou horizontales pour représenter des valeurs particulières. Ces lignes peuvent être ajoutées avec la fonction abline().

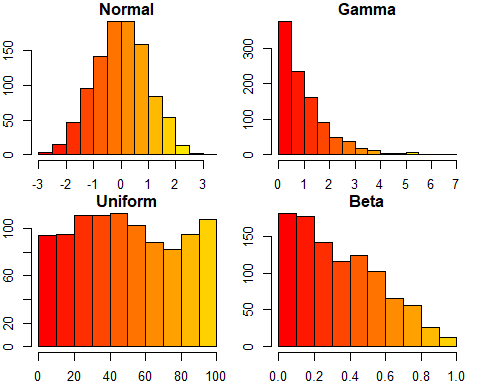
myX <- rnorm(100)  
myY <- rnorm(100)  
plot(x = myX, y = myY,   
 xlim = c(-4, 4), ylim = c(-4, 4),   
 pch = 16, cex = 1.5,   
 col = sample(colors(), size = 100),  
 panel.first = {  
 grid()  
 abline(v = c(min(myX), max(myX)), lty = 2)  
 abline(h = c(min(myY), max(myY)), lty = 2)  
 abline(v = mean(myX), lty = 1)  
 abline(h = mean(myY), lty = 1)  
})



## hist

Pour faire un histogramme nous utilisons la fonction hist(). C’est une fonction graphique utile pour visualiser rapidement la distribution d’un jeu de données.

op <- par(no.readonly = TRUE)  
par(mfrow = c(2, 2), mar = c(2, 2, 1, 1))  
myX <- list(  
 rnorm(1000),  
 rgamma(1000, shape = 1),  
 sample(1:100, size = 1000, replace = TRUE),  
 rbeta(1000, shape1 = 1, shape2 = 2)  
)  
myTitle <- c("Normal", "Gamma", "Uniform", "Beta")  
tr <- lapply(1:4, function(i){  
 hist(myX[[i]],   
 col = heat.colors(15),   
 main = myTitle[i]  
 )  
})

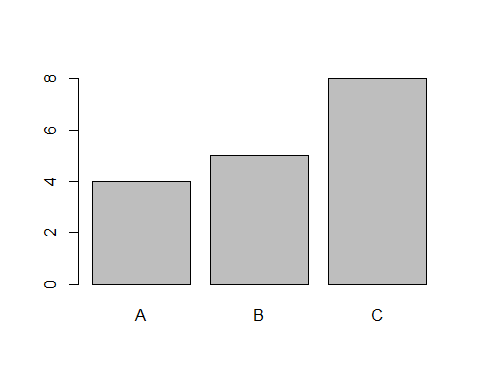


par(op)

## barplot

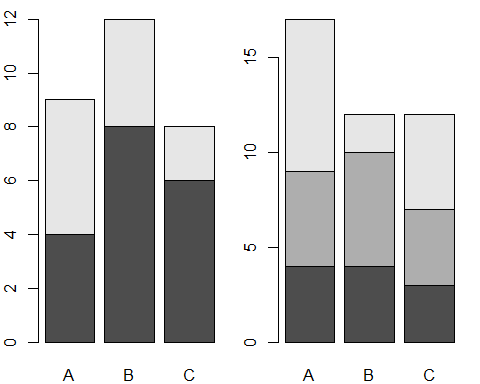
Le graphique en barres se fait au moyen de la fonction barplot().

myX <- c(4, 5, 8)  
barplot(myX, names.arg = c("A", "B", "C"))



Quand l’objet envoyé à cette fonction est un vector alors la fonction barplot() renvoie un graphique en barres simples. Quand c’est une matrix alors les barres sont multiples.

op <- par(no.readonly = TRUE)  
par(mfrow = c(1, 2), mar = c(2, 2, 1, 1))  
myX <- matrix(c(4, 5, 8, 4, 6, 2), nrow = 2)  
barplot(myX, names.arg = c("A", "B", "C"))  
myX <- matrix(c(4, 5, 8, 4, 6, 2, 3, 4, 5), nrow = 3)  
barplot(myX, names.arg = c("A", "B", "C"))



par(op)

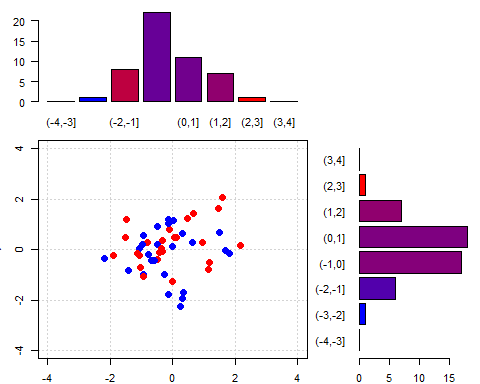
La fonction barplot() peut aussi être utilisée pour représenter l’équivalent d’un histogramme. Cela peut être utile pour représenter la distribution d’une variable en fonction de l’axe des abscisses et l’axe des ordonnées. Dans l’exemple qui suit nous avons n points tirés au hasard dans une loi normale de paramètres mean = 0 et sd = 1 (myX <- rnorm(n)). Ces points sont soit représentés en bleu, soit représentés en rouge (la couleur bleue est codée avec la valeur 4 et la couleur rouge avec la valeur 2, nous en reparlerons dans un prochain chapitre). Le tirage au hasard de la couleur se fait avec la fonction sample() (myCol <- sample(c(4, 2), size = n, replace = TRUE)). Ici nous voulons représenter un nuage de points avec les points en rouge ou en bleu, puis pour l’axe des abscisses et l’axe des ordonnées, des histogrammes pour voir la répartition des points et un gradient de couleur du bleu au rouge en fonction de la proportion des points de couleur dans chaque catégorie. Le gradient de couleur avec 100 valeurs entre bleu et rouge se réalise avec la fonction colorRampPalette() (myColors <- colorRampPalette(c("blue", "red"))(100)).

Pour faire l’histogramme nous allons découper les données avec la fonction cut() en spécifiant que nous souhaitons des séparations faites entre -4 et 4 par pas de 1 (myYCut <- cut(myY, breaks = -4:4)). Pour avoir le compte du nombre de points dans chacune des catégories et pour chaque couleur il nous suffit d’utiliser la fonction table() (myYCutCol <- table(myCol, myYCut)). Dans cette table la première ligne correspond à la première couleur rencontrée dans le jeu de données et la deuxième ligne à l’autre couleur. C’est pourquoi il nous faut modifier le tirage aléatoire des couleurs pour que la première ligne corresponde toujours au bleu et la deuxième ligne au rouge : myCol <- c(2, sample(c(4, 2), size = (n - 1), replace = TRUE)).

Ensuite nous pouvons calculer la proportion de rouge en divisant la première ligne par la somme des deux lignes que nous allons représenter en pourcentage en multipliant par 100 : myXCutCol[1,] / (myXCutCol[1,] + myXCutCol[2,]) \* 100. Pour que ce chiffre corresponde à une couleur nous n’allons conserver que sa partie entière avec la fonction round(). Si le pourcentage est de zéro ou si le résultat n’est pas possible du fait d’une division par zéro alors il nous faut le remplacer par 1 afin que cela corresponde à une couleur dans notre gradient qui va de 1 à 100 (xCol[is.na(xCol) | xCol == 0] <- 1).

Il ne nous reste plus qu’à organiser l’espace graphique avec la fonction layout() qui prend comme argument une matrice dont les valeurs et leur position vont correspondre à l’agencement des différents graphiques que nous souhaitons réaliser. Le graphique 1 correspond au barplot du haut, le graphique 2 au nuage de points et le graphique 3 au barplot de droite.

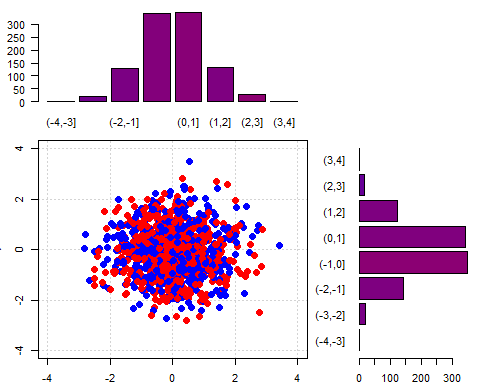
n <- 50  
myX <- rnorm(n)  
myY <- rnorm(n)  
myCol <- c(2, sample(c(4, 2), size = (n - 1), replace = TRUE))  
myColors <- colorRampPalette(c("blue", "red"))(100)  
myYCut <- cut(myY, breaks = -4:4)  
myXCut <- cut(myX, breaks = -4:4)  
myYCutCol <- table(myCol, myYCut)  
myXCutCol <- table(myCol, myXCut)  
xCol <- round(  
 myXCutCol[1,] / (myXCutCol[1,] + myXCutCol[2,]) \* 100  
)  
xCol[is.na(xCol) | xCol == 0] <- 1  
yCol <- round(  
 myYCutCol[1,] / (myYCutCol[1,] + myYCutCol[2,]) \* 100  
)  
yCol[is.na(yCol) | yCol == 0] <- 1  
op <- par(no.readonly = TRUE)  
par(mar = c(2, 3, 1, 1))  
layout(matrix(c(1, 1, 0,   
 2, 2, 3,   
 2, 2, 3), ncol = 3, byrow = TRUE))  
barplot(table(myXCut), las = 1, col = myColors[xCol])  
plot(x = myX, y = myY, col = myCol, pch = 16,   
 xlim = c(-4, 4), ylim = c(-4, 4), cex = 1.5,   
 panel.first = grid())  
barplot(table(myYCut), las = 1, horiz = TRUE, col = myColors[yCol])



par(op)

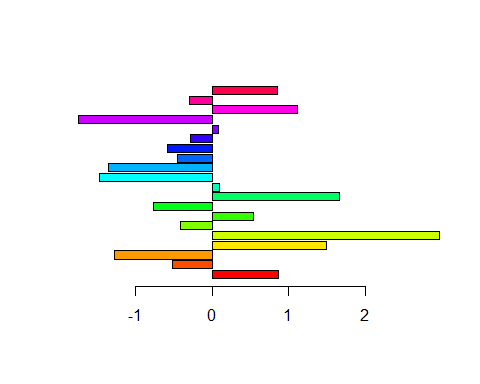
Nous pouvons ensuite intégrer ce script dans une fonction pour par exemple étudier l’effet de la variable n.

graphBarplotCol <- function(n){  
 myX <- rnorm(n)  
 myY <- rnorm(n)  
 myCol <- c(2, sample(c(4, 2), size = (n - 1), replace = TRUE))  
 myColors <- colorRampPalette(c("blue", "red"))(100)  
 myYCut <- cut(myY, breaks = -4:4)  
 myXCut <- cut(myX, breaks = -4:4)  
 myYCutCol <- table(myCol, myYCut)  
 myXCutCol <- table(myCol, myXCut)  
 xCol <- round(  
 myXCutCol[1,] / (myXCutCol[1,] + myXCutCol[2,]) \* 100  
 )  
 xCol[is.na(xCol) | xCol == 0] <- 1  
 yCol <- round(  
 myYCutCol[1,] / (myYCutCol[1,] + myYCutCol[2,]) \* 100  
 )  
 yCol[is.na(yCol) | yCol == 0] <- 1  
 op <- par(no.readonly = TRUE)  
 par(mar = c(2, 3, 1, 1))  
 layout(matrix(c(1, 1, 0,   
 2, 2, 3,   
 2, 2, 3), ncol = 3, byrow = TRUE))  
 barplot(table(myXCut), las = 1, col = myColors[xCol])  
 plot(x = myX, y = myY, col = myCol, pch = 16,   
 xlim = c(-4, 4), ylim = c(-4, 4), cex = 1.5,   
 panel.first = grid())  
 barplot(table(myYCut), las = 1, horiz = TRUE, col = myColors[yCol])  
 par(op)  
}  
graphBarplotCol(n = 1000)



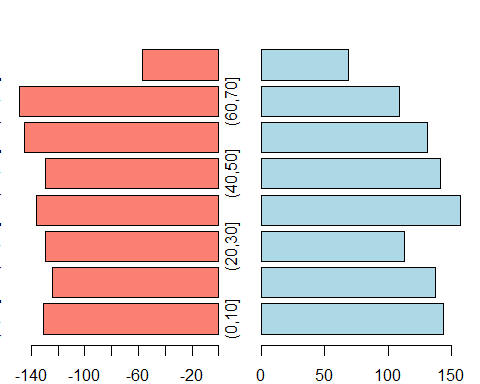
Un barplot peut bien sûr prendre des valeurs positives ou négatives.

barplot(rnorm(20), horiz = TRUE, col = rainbow(20))



Le barplot peut aussi être utilisé pour faire une pyramide des âges (il existe des fonctions pour réaliser des pyramides des âges, ici l’objectif est pédagogique).

gender <- data.frame(  
 m = cut(sample(1:75, 1000, replace = TRUE),   
 breaks = seq(from = 0, to = 80, by = 10)),   
 f = cut(sample(1:75, 1000, replace = TRUE),   
 breaks = seq(from = 0, to = 80, by = 10))  
)  
op <- par(no.readonly = TRUE)  
par(mfrow = c(1, 2), mar = c(2, 1, 2, 1))  
barplot(-table(gender$f), horiz = TRUE, col = "salmon")  
barplot(table(gender$m), horiz = TRUE, col = "lightblue")



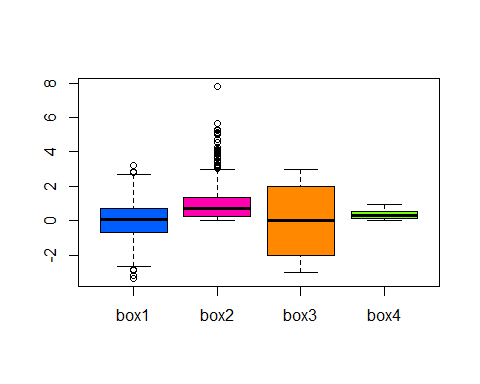
par(op)

## boxplot

Les boxplot ou boîtes à moustache sont des graphiques très courant avec R car ils donnent un bon apperçu d’un jeu de données en représentant les valeurs extrèmes (“outliers”), la médiane, les quartiles, les minimums et les maximums.

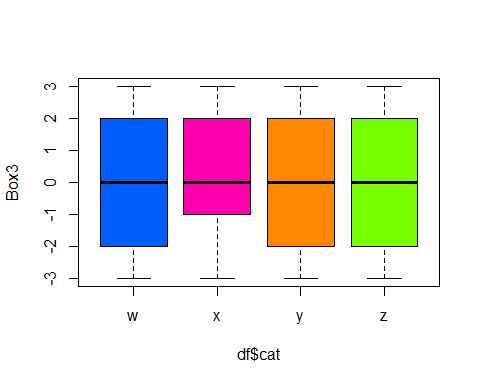
La fonction boxplot() s’applique à un ou plusieurs vector.

df <- data.frame(  
 box1 = rnorm(1000),   
 box2 = rgamma(1000, shape = 1),   
 box3 = sample(-3:3, size = 1000, replace = TRUE),  
 box4 = rbeta(1000, shape1 = 1, shape2 = 2)  
)  
boxplot(df, col = c(rgb(0, 94, 255, maxColorValue = 255),   
 rgb(255, 0, 174, maxColorValue = 255),   
 rgb(255, 136, 0, maxColorValue = 255),   
 rgb(119, 255, 0, maxColorValue = 255)))

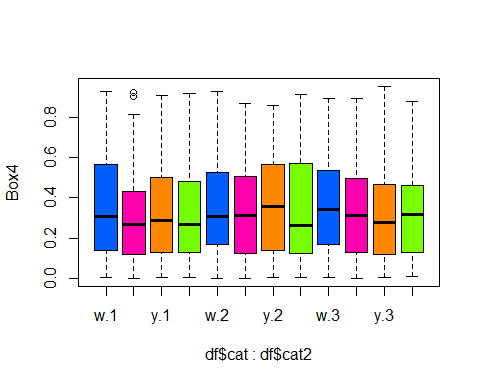


Si une variable est de type factor, la fonction boxplot() permet facilement de représenter chaque catégorie. Cela fonctionne aussi avec les variables numériques mais il faut veiller à ne pas avoir trop de valeurs différentes pour que le graphique reste lisible.

df$cat <- sample(c("w", "x", "y", "z"), size = 1000, replace = TRUE)  
boxplot(df$box3 ~ df$cat, col = c(rgb(0, 94, 255, maxColorValue = 255),   
 rgb(255, 0, 174, maxColorValue = 255),   
 rgb(255, 136, 0, maxColorValue = 255),   
 rgb(119, 255, 0, maxColorValue = 255)), ylab = "Box3")

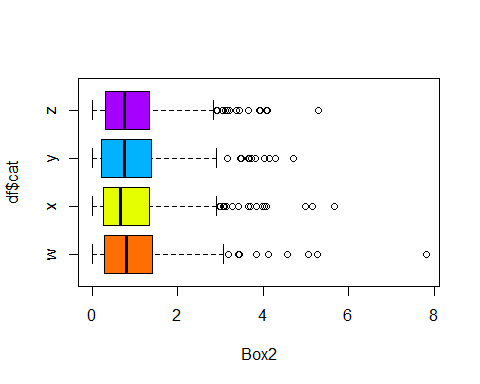


df$cat2 <- sample(1:3, size = 1000, replace = TRUE)  
boxplot(df$box4 ~ df$cat\*df$cat2, col = c(  
 rgb(0, 94, 255, maxColorValue = 255),   
 rgb(255, 0, 174, maxColorValue = 255),   
 rgb(255, 136, 0, maxColorValue = 255),   
 rgb(119, 255, 0, maxColorValue = 255)), ylab = "Box4")



Le boxplot peut être représenté horizontalement ou verticalement.

df$cat <- sample(c("w", "x", "y", "z"), size = 1000, replace = TRUE)  
boxplot(df$box2 ~ df$cat, horizontal = TRUE,   
 col = c(rgb(255, 110, 0, maxColorValue = 255),   
 rgb(230, 255, 0, maxColorValue = 255),   
 rgb(0, 178, 255, maxColorValue = 255),   
 rgb(166, 0, 255, maxColorValue = 255)), xlab = "Box2")



## Autres graphiques

Il existe de nombreux autres graphiques mais ceux que nous venons de voir constituent la base. Pour plus d’information et d’idées pour représenter vos données nous pouvons consulter le très beau site <https://www.data-to-viz.com/> ou encore la gallerie de graphiques R <https://www.r-graph-gallery.com/> (en anglais ; la plupart des graphiques sont réalisés avec le package ggplot2 que nous verrons plus tard). Pour plus d’idées vous pouvez aussi utiliser la demonstration du package graphics en utilisant la commande demo("graphics") (la touche “Entrée” permet d’afficher les graphiques).

## Conclusion

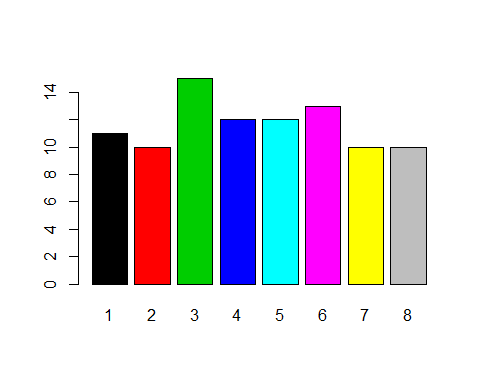
Félicitations, nous sommes arrivés à la fin de ce chapitre sur les graphiques simples ! Nous savons désormais réaliser les principaux graphiques plot(), hist(), barplot(), et boxplot(). Tout au long de ce chapitre nous avons utilisé différentes couleurs et différentes façons de représenter les couleurs : il est temps de formaliser l’utilisation et la gestion des couleurs. C’est le sujet du prochain chapitre !

# La gestion des couleurs

Nous avons vu différentes manières d’utiliser les couleurs : avec leur nom (e.g., "salmon"), avec un numéro de 1 à 8, avec la fonction rgb() (pour “red”, “green”, “blue”), et avec la fonction colors(). Il en existe d’autres mais celles-ci sont les principales.

L’utilisation des numéros de 1 à 8 correspond au noir, rouge, vert, bleu, cyan, magenta, jaune et gris. Cette utilisation est pratique pour visualiser rapidement nos résultats mais donne globalement des graphiques visuellement moyens. Ces couleurs sont plutôt à éviter pour communiquer nos graphiques.

barplot(sample(10:15, 8, replace = TRUE), col = 1:8, names.arg = 1:8)



## colors()

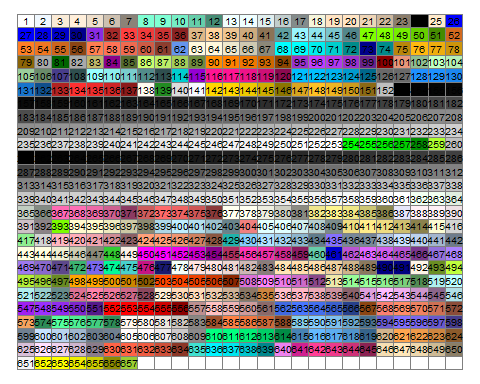
Pour choisir des couleurs plus agréables et mettant plus en avant nos résultats, une option consiste à les choisir dans la liste des couleurs pré-enregistrées dans R. Nous pouvons accéder à la liste des couleurs avec la fonction colors()

head(colors(), n = 20)

## [1] "white" "aliceblue" "antiquewhite" "antiquewhite1"  
## [5] "antiquewhite2" "antiquewhite3" "antiquewhite4" "aquamarine"   
## [9] "aquamarine1" "aquamarine2" "aquamarine3" "aquamarine4"   
## [13] "azure" "azure1" "azure2" "azure3"   
## [17] "azure4" "beige" "bisque" "bisque1"

Nous pouvons utiliser ces couleurs avec leur nom (e.g., “white”, “azure3”), ou alors avec leur numéro (e.g., “white” = colors()[1], “azure3” = colors()[16]).

# adapted from http://www.r-graph-gallery.com/42-colors-names/  
op <- par(no.readonly = TRUE)  
par(mar = c(0, 0, 0, 0))  
plot(0, type = "n", xlim = c(0, 1), ylim = c(0, 1),   
 axes = FALSE, xlab = "", ylab = "")  
numRow <- 26  
numCol <- 26  
rect(  
 xleft = rep((0:(numCol - 1)/numCol), numRow),   
 ybottom = sort(rep((0:(numRow - 1)/numRow),numCol), decreasing = TRUE),  
 xright = rep((1:numCol/numCol), numRow),  
 ytop = sort(rep((1:numRow/numRow), numCol), decreasing = TRUE),  
 border = grey(0.5),   
 col = colors()[seq(1, numRow\*numCol)])  
myLabels <- c(as.character(1:657), rep("", numRow\*numCol - 657))  
text(  
 x = rep((0:(numCol - 1)/numCol), numRow) + 0.02,  
 y = sort(rep((0:(numRow - 1)/numRow), numCol), decreasing = TRUE) + 0.02,  
 labels = myLabels,   
 cex = 0.6)



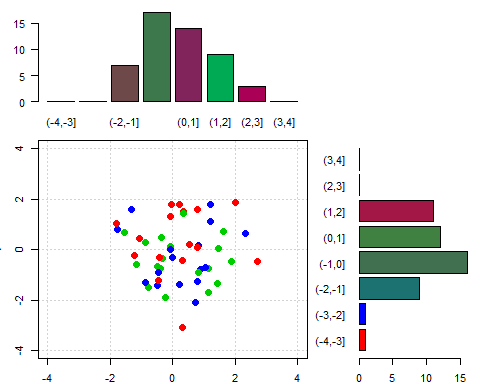
par(op)

## rgb()

Une autre option consiste à construire ses propres couleurs ave la fonction rgb() qui prend comme argument la quantité de rouge, de de vert, et de bleu. Par défaut ces valeurs sont comprises entre 0 et 1. Ce réglage par défaut peut être modifié avec l’arguement maxColorValue pour par exemple avoir des valeurs entre 0 et 255 (maxColorValue = 255 ; norme pour la représentation des couleurs RGB).

Nous allons reprendre notre fonction permettant de représenter la distribution de points dans un nuage de points au moyen de barplot avec cette fois trois couleurs de points (rouge, vert, bleu), et des barplot dont la couleur correspondra à la quantité de chaque couleur avec la fonction rgb().

graphBarplotCol <- function(n){  
 myX <- rnorm(n)  
 myY <- rnorm(n)  
 myCol <- c(2, 3, 4, sample(2:4, size = (n - 3), replace = TRUE))  
 myYCut <- cut(myY, breaks = -4:4)  
 myXCut <- cut(myX, breaks = -4:4)  
 myYCutCol <- table(myCol, myYCut)  
 myXCutCol <- table(myCol, myXCut)  
 rColX <- myXCutCol[1,] / (myXCutCol[1,] + myXCutCol[2,] +   
 myXCutCol[3,])  
 gColX <- myXCutCol[2,] / (myXCutCol[1,] + myXCutCol[2,] +   
 myXCutCol[3,])  
 bColX <- myXCutCol[3,] / (myXCutCol[1,] + myXCutCol[2,] +   
 myXCutCol[3,])  
 rColX[is.na(rColX)] <- 0  
 gColX[is.na(gColX)] <- 0  
 bColX[is.na(bColX)] <- 0  
 rColY <- myYCutCol[1,] / (myYCutCol[1,] + myYCutCol[2,] +   
 myYCutCol[3,])  
 gColY <- myYCutCol[2,] / (myYCutCol[1,] + myYCutCol[2,] +   
 myYCutCol[3,])  
 bColY <- myYCutCol[3,] / (myYCutCol[1,] + myYCutCol[2,] +   
 myYCutCol[3,])  
 rColY[is.na(rColY)] <- 0  
 gColY[is.na(gColY)] <- 0  
 bColY[is.na(bColY)] <- 0  
 op <- par(no.readonly = TRUE)  
 par(mar = c(2, 3, 1, 1))  
 layout(matrix(c(1, 1, 0,   
 2, 2, 3,   
 2, 2, 3), ncol = 3, byrow = TRUE))  
 barplot(table(myXCut), las = 1, col = rgb(rColX, gColX, bColX))  
 plot(x = myX, y = myY, col = myCol, pch = 16,   
 xlim = c(-4, 4), ylim = c(-4, 4), cex = 1.5,   
 panel.first = grid())  
 barplot(table(myYCut), las = 1, horiz = TRUE,   
 col = rgb(rColY, gColY, bColY))  
 par(op)  
}  
graphBarplotCol(n = 50)



Avec la fonction rgb() nous pouvons donc représenter 256^3 couleurs, soit 167 777 216 couleurs différentes. Notre objectif reste cependant de faire des graphiques agréables à lire et qui mettent bien en valeurs nos résultats scientifiques. Il faut donc choisir les couleurs adéquates au regard de notre objectif. C’est pour cela que nous allons utiliser des palettes.

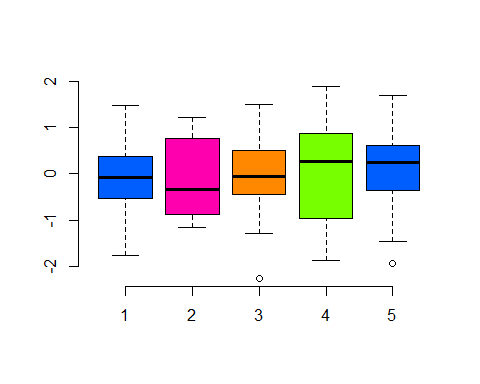
## Palettes

Les palettes sont des jeux de couleurs représentées sous forme d’un vector avec les couleurs au format hexadécimal (valeur renvoyée par la fonction rgb() par exemple).

myPal <- c(  
 rgb(0, 94, 255, maxColorValue = 255),   
 rgb(255, 0, 174, maxColorValue = 255),   
 rgb(255, 136, 0, maxColorValue = 255),   
 rgb(119, 255, 0, maxColorValue = 255))  
print(myPal)

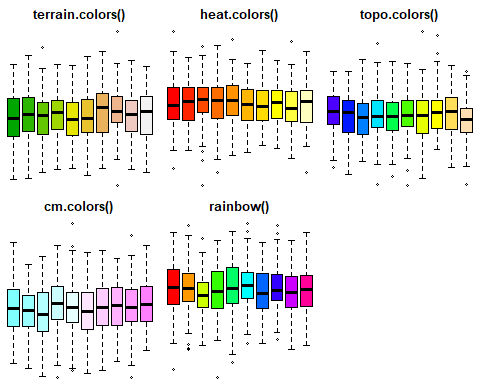
## [1] "#005EFF" "#FF00AE" "#FF8800" "#77FF00"

boxplot(matrix(rnorm(100), ncol = 5), col = myPal, axes = FALSE)  
axis(1)  
axis(2)



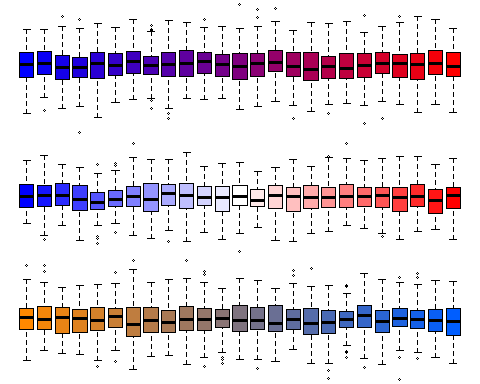
Il existe des palettes inclues dans R : terrain.colors(), heat.colors(), topo.colors(), cm.colors(), rainbow().

op <- par(no.readonly = TRUE)  
par(mfrow = c(2, 3), mar = c(0, 0, 2, 0))  
boxplot(matrix(rnorm(1000), ncol = 10), main = "terrain.colors()",   
 col = terrain.colors(10), axes = FALSE)  
boxplot(matrix(rnorm(1000), ncol = 10), main = "heat.colors()",   
 col = heat.colors(10), axes = FALSE)  
boxplot(matrix(rnorm(1000), ncol = 10), main = "topo.colors()",   
 col = topo.colors(10), axes = FALSE)  
boxplot(matrix(rnorm(1000), ncol = 10), main = "cm.colors()",   
 col = cm.colors(10), axes = FALSE)  
boxplot(matrix(rnorm(1000), ncol = 10), main = "rainbow()",   
 col = rainbow(10), axes = FALSE)  
par(op)



Il existe aussi une fonction colorRampPalette() qui permet de créer un dégradé de couleur.

op <- par(no.readonly = TRUE)  
par(mfrow = c(3, 1), mar = c(0, 0, 0, 0))  
boxplot(matrix(rnorm(2500), ncol = 25),   
 col = colorRampPalette(c('blue', 'red'))(25), axes = FALSE)  
boxplot(matrix(rnorm(2500), ncol = 25),   
 col = colorRampPalette(c('blue', 'white', 'red'))(25), axes = FALSE)  
boxplot(matrix(rnorm(2500), ncol = 25),   
 col = colorRampPalette(c(rgb(255, 136, 0, maxColorValue = 255),   
 rgb(0, 94, 255, maxColorValue = 255)))(25), axes = FALSE)



par(op)

Nous pouvons aussi créer nos propres palettes en utilisant des sites web dédiés à la sélection des couleurs comme <http://paletton.com/> ou <https://coolors.co/> (il en existe bien d’autres), puis les utiliser dans R en recopiant dans un vecteur les valeurs héxadécimales ou rgb.

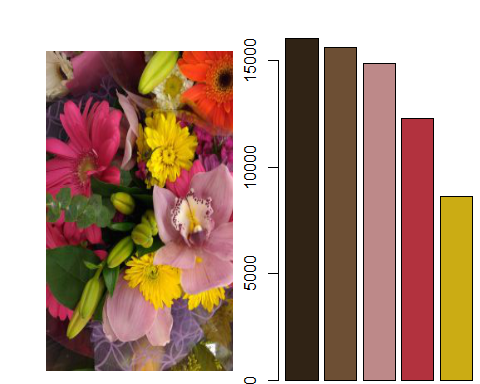
R est un language de programmation très puissant. Nous pouvons imaginer de nombreuses façons de créer automatiquement des palettes en fonction de critères variés. Par exemple nous pouvons importer une image dont les teintes nous semblent pertinentes, puis extraire les informations de chacun des points pour ensuite sélectionner les couleurs dominantes via un regroupement par cluster. C’est ce que fait la fonction suivante.

Tout d’abord nous allons charger les packages raster, rgdal et jpeg qui vont nous servir à manipuler notre image sous R.

pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("raster")  
pkgCheck("rgdal")  
pkgCheck("jpeg")

Ensuite nous allons utiliser la fonction kmeans() pour effectuer des groupes de couleurs en utilisant les valeurs RGB de chacun des points de notre image. Ici nous avons deux méthodes possibles, la première utilise la fonction kmeans() pour les trois valeurs RGB, et la seconde utilise la fonction kmeans() pour chaque valeur RGB individuellement (cette seconde fonction donne une palette qui pourra être assez éloignée des couleurs de l’image de départ).

createPal <- function(photo, met = 1, graph = TRUE, k = 9){  
 if(met == 1){  
 colR <- getValues(raster(photo, band = 1))  
 colG <- getValues(raster(photo, band = 2))  
 colB <- getValues(raster(photo, band = 3))  
 kMeans <- kmeans(data.frame(colR, colG, colB), centers = k)  
 kCol <- rgb(kMeans$centers, maxColorValue = 255)[order(table(  
 kMeans$cluster), decreasing = TRUE)]  
 if(graph == TRUE){  
 op <- par(no.readonly = TRUE)  
 par(mfrow = c (1, 2), mar = c(0, 2, 2, 0))  
 myJpg <- readJPEG("./myFiles/photoKmeans.jpg", native = TRUE)  
 plot(0:1, 0:1, type = "n", ann = FALSE, axes = FALSE)  
 rasterImage(myJpg, 0, 0, 1, 1)  
 barplot(table(kMeans$cluster)[order(table(kMeans$cluster),   
 decreasing = TRUE)], col = kCol, names.arg = NA)  
 par(op)  
 }  
 return(kCol)  
 } else {  
 if(met == 2){  
 kColR <- kmeans(x = getValues(raster(photo, band = 1)),   
 centers = k)  
 kColG <- kmeans(x = getValues(raster(photo, band = 2)),   
 centers = k)  
 kColB <- kmeans(x = getValues(raster(photo, band = 3)),   
 centers = k)  
 kCol <- (rgb(kColR$centers, kColG$centers, kColB$centers,  
 maxColorValue = 255))  
 if(graph == TRUE){  
 op <- par(no.readonly = TRUE)  
 par(mfrow = c (1, 2), mar = c(0, 2, 2, 0))  
 myJpg <- readJPEG("./myFiles/photoKmeans.jpg", native = TRUE)  
 plot(0:1, 0:1, type = "n", ann = FALSE, axes = FALSE)  
 rasterImage(myJpg, 0, 0, 1, 1)  
 # par(mar = c(0, 0, 0, 0))  
 plot(x = 1:k, y = rep(1, k), ylim = c(0, 1),   
 xlim = c(0, k), axes = FALSE, xlab = "",   
 ylab = "", type = "n")  
 for(i in 1:k){  
 polygon(x = c(i-1, i, i, i-1), y = c(0, 0, 1, 1),   
 col = kCol[i])  
 text(x = i - 0.5, y = 0.5,   
 labels = as.character(kCol[i]), srt = 90)  
 }  
 par(op)  
 }  
 return(kCol)  
 } else {  
 print(paste0("No method ", met, "."))  
 return(rgb(0, 0, 0))  
 }  
 }  
}  
  
myPalMet1 <- createPal(photo = "./myFiles/photoKmeans.jpg",   
 met = 1, graph = TRUE, k = 5)

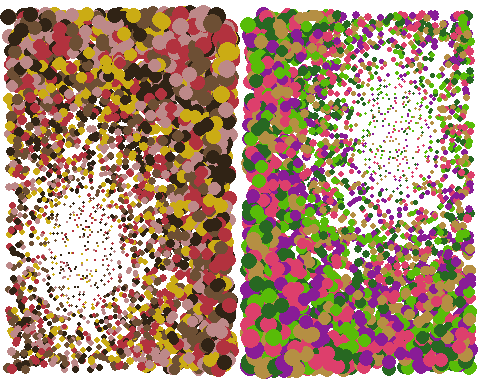


myPalMet2 <- createPal(photo = "./myFiles/photoKmeans.jpg",   
 met = 2, graph = TRUE, k = 5)



La fonction nous renvoie les couleurs de la palette avec un graphique en barres représentant le nombre de points de l’image dans chacun des groupes de couleurs. Nous pouvons désormais utiliser notre nouvelle palette pour réaliser nos graphiques.

makeImpact <- function(myPal, numP = 300, impact = 0.33, multCex = 3){  
 myX <- sample(0:1000, size = numP, replace = TRUE)/1000  
 myY <- sample(0:1000, size = numP, replace = TRUE)/1000  
 distImpact <- sqrt((myX - impact)^2 + (myY - impact)^2)  
 dfXY <- data.frame(myX, myY, distImpact)  
 plot(x = dfXY$myX, y = dfXY$myY, axes = FALSE,   
 xlab = "", ylab = "", cex = dfXY$distImpact\*multCex,   
 col = myPal, pch = 16)  
}  
  
op <- par(no.readonly = TRUE)  
par(mfrow = c (1, 2), mar = c(0, 0, 0, 0))  
makeImpact(myPal = myPalMet1, numP = 3000, impact = 0.33)  
makeImpact(myPal = myPalMet2, numP = 3000, impact = 0.66)



par(op)

## Conclusion

Félicitations ! C’est la fin de ce chapitre sur la gestion des couleurs. Nous savons désormais comment utiliser les couleurs et les palettes, et comment guider le choix des couleurs pour mettre en valeur nos résultats. Dans le prochain chapitre nous allons voir quelques exemples de packages graphiques et les dernières tendances comme les graphiques dynamiques.

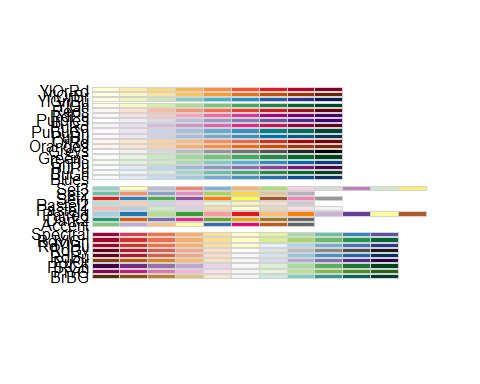
# Les packages graphiques

## les packages de palettes

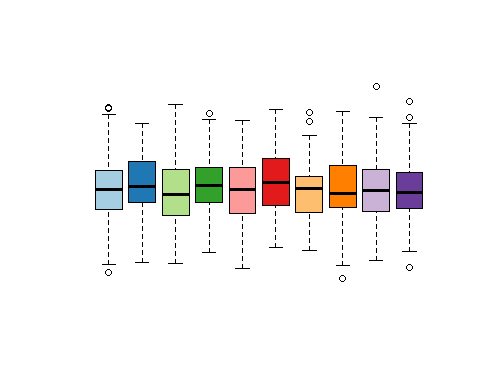
### RColorBrewer

Le package RColorBrewer est un package populaire qui contient des palettes complémentaires à celle disponibles dans la version de base de R. Une fois le package installé, il suffit d’appeler les palettes pour les utiliser. Voici les palettes disponibles et un exemple d’utilisation.

pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("RColorBrewer")  
display.brewer.all()



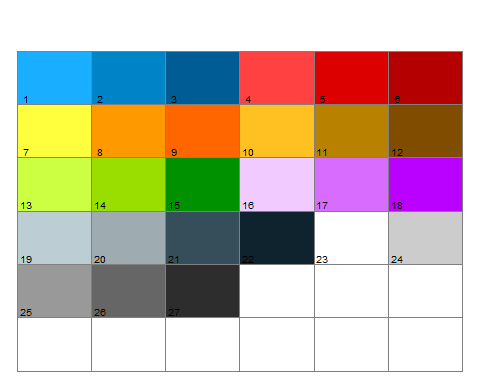
boxplot(matrix(rnorm(1000), ncol = 10),   
 col = brewer.pal(10, "Paired"), axes = FALSE)



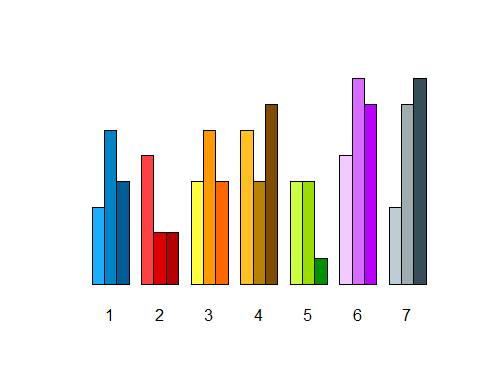
### palettesForR

Le package palettesForR est un autre package contenant des palettes prêtes à l’emploi, issues des projets ‘Gimp’ et ‘Inkscape’. Comme pour RColorBrewer, il suffit d’appeler les palettes pour les utiliser. Les nombreuses palettes disponibles sont listées dans l’aide du package. Voici un exemple d’utilisation.

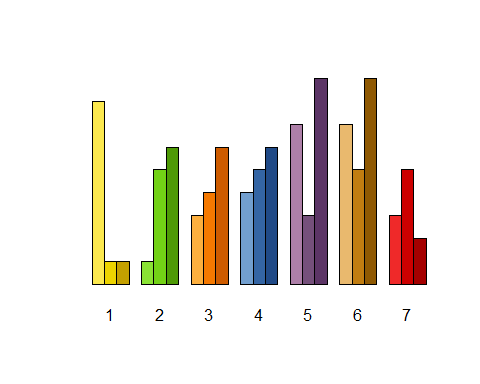
pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("palettesForR")  
showPalette(Echo\_gpl)



groupTest <- sample(1:3, size = 100, replace = TRUE)   
valueTest <- sample(1:7, size = 100, replace = TRUE)  
tableTest <- table(groupTest, valueTest)  
barplot(tableTest,   
 col = Echo\_gpl, axes = FALSE, beside = TRUE)



groupTest <- sample(1:3, size = 100, replace = TRUE)   
valueTest <- sample(1:7, size = 100, replace = TRUE)  
tableTest <- table(groupTest, valueTest)  
barplot(tableTest,   
 col = Tango\_gpl, axes = FALSE, beside = TRUE)



### Les autres packages

Il existe de très nombreux packages contenant des palettes. Par exemple :

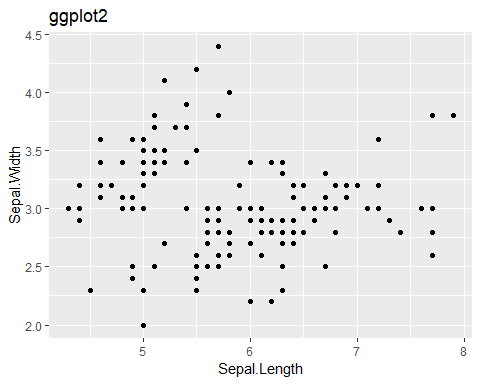
* viridis (<https://CRAN.R-project.org/package=viridis>)
* jcolors (<https://CRAN.R-project.org/package=jcolors>)
* scico (<https://CRAN.R-project.org/package=scico>)
* …

## ggplot2 package

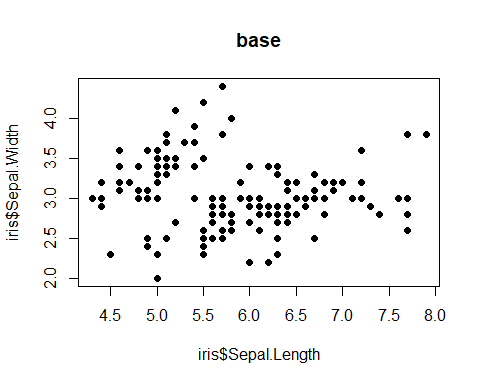
Le package ggplot2 est une alternative aux fonctions de base de R pour réaliser des graphiques. Il repose sur “La Grammaire des Graphiques” de Leland Wilkinson et permet de réaliser des graphiques sous forme de couches, avec en général un rendu esthétique supérieur aux graphiques réalisés avec les fonctions R de base. Est-ce qu’il faut pour autant oublier ce qui a été vu jusqu’à présent et se concentrer sur l’utilisation de ggplot2 ? Heureusement que non ! Si pour explorer un jeu de données et en sortir les tendances principales ggplot2 s’avère parfois plus puissant, nos graphiques ne viennent jamais seuls et sont accompagnés d’analyses statistiques rendant nécessaire un travail souvent poussé sur la gestion des données. Une fois nos hypothèses de travail testées statistiquement, il devient facile de réaliser nos graphiques quel que soit leur niveau de complexité (avec les fonctions de base ou avec ggplot2). Par ailleurs nous verrons dans le chapitre suivant que depuis le graphique jusqu’à la figure dans l’article scientifique, il y a une série de traitements à effectuer et la manipulation des paramètres esthétiques peut se faire indépendamment de R. Donc ggplot2 est un package intéressant car il apporte une alternative avec une autre philosophie dans la construction des graphiques, mais il ne se substitue pas à ce que nous venons d’apprendre jusqu’à présent. Dans la pratique nous pourrons utiliser l’un ou l’autre en fonction des données et des manipulations que nous souhaitons en faire.

Pour revenir à ggplot2, commençons par un exemple avec les données iris.

pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("ggplot2")  
data(iris)  
# ggplot2  
p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width))  
p + geom\_point() + ggtitle("ggplot2")

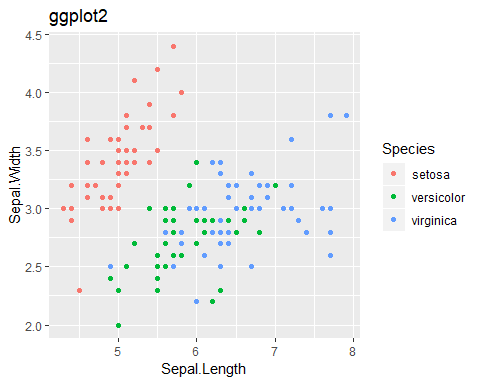


# base  
plot(x = iris$Sepal.Length, y = iris$Sepal.Width,   
 main = "base", pch = 16)

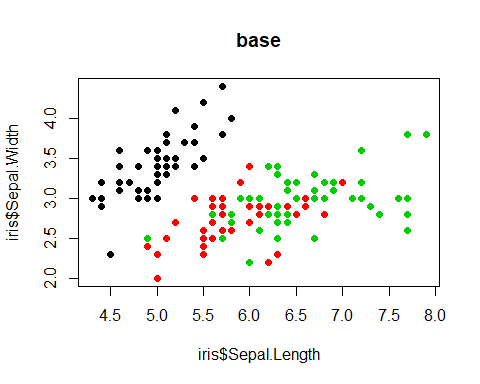


Maintenant séparons l’information en fonction de l’espèce de fleur.

# ggplot2  
p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, colour = Species))  
p + geom\_point() + ggtitle("ggplot2")

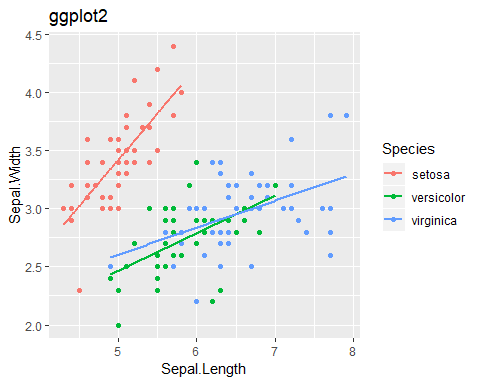


# base  
plot(x = iris$Sepal.Length, y = iris$Sepal.Width,   
 main = "base", pch = 16, col = iris$Species)

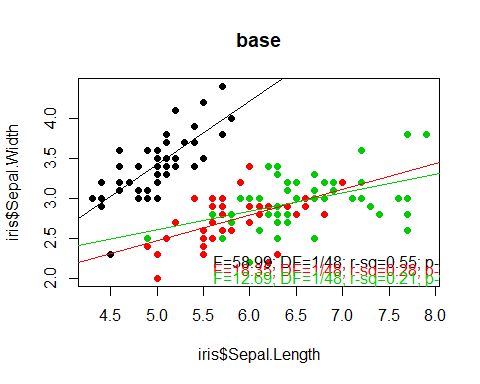


Il semble y avoir une relation entre largeur et longueur des sépales par espèces.

# linear regressions  
lmFits <- lapply(1:3, function(i){  
 fitSp1 <- lm(iris$Sepal.Width[as.numeric(iris$Species) == i] ~   
 iris$Sepal.Length[as.numeric(iris$Species) == i])  
 fStat1 <- summary(fitSp1)$fstatistic  
 rSq1 <- summary(fitSp1)$r.squared  
 pVal1 <- summary(fitSp1)$coefficients[2, 4]  
 stat1 <- paste0("F=", round(fStat1[1], digits = 2),   
 "; DF=", fStat1[2], "/", fStat1[3], "; r-sq=", round(rSq1, digits = 2),   
 "; p-val=", round(pVal1, digits = 6))  
 return(list(fitSp1, stat1))  
})  
# ggplot2  
p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, colour = Species))  
p <- p + geom\_point() + ggtitle("ggplot2") + stat\_smooth(method = "lm", se = FALSE)  
p <- p + annotate(geom = "text", x = 6, y = 2.250, label = lmFits[[1]][[2]], colour = 2)  
p <- p + annotate(geom = "text", x = 6, y = 2.125, label = lmFits[[2]][[2]], colour = 3)  
p <- p + annotate(geom = "text", x = 6, y = 2.000, label = lmFits[[3]][[2]], colour = 4)  
p



# base  
plot(x = iris$Sepal.Length, y = iris$Sepal.Width,   
 main = "base", pch = 16, col = iris$Species)  
abline(lmFits[[1]][[1]], col = 1)  
abline(lmFits[[2]][[1]], col = 2)  
abline(lmFits[[3]][[1]], col = 3)  
text(x = 5.5, y = 2.2, labels = lmFits[[1]][[2]], pos = 4)  
text(x = 5.5, y = 2.1, labels = lmFits[[2]][[2]], pos = 4, col = 2)  
text(x = 5.5, y = 2.0, labels = lmFits[[3]][[2]], pos = 4, col = 3)



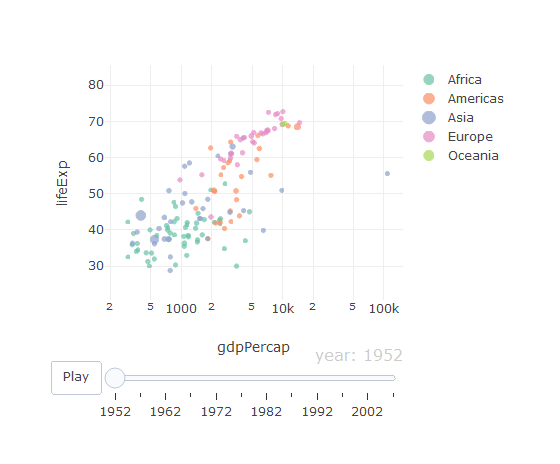
Nous pouvons voir sur ces exemples que les graphiques avec ggplot2 commencent par un appel de la fonction ggplot(), dans laquelle le premier argument data correspond à nos données (typiquement une data.frame), et le deuxième argument aes() correspond aux informations que nous souhaitons utiliser. Par convention cette information est stockée dans un objet p. Nous allons ensuite ajouter des couches supplémentaires en utilisant +.

Dans les couches nous pouvons ajouter des aspects géométriques (le type de graphique, par exemple geom\_point()), des statistiques (par exemple stat\_smooth()), des annotations (par exemple annotate()), et bien d’autres choses liées aux axes, aux couleurs, … La documentation complète (en anglais) peut être consultée à l’adresse <https://ggplot2.tidyverse.org/> (fiche de résumé : <https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>). De nombreuses extensions à ggplot2 sont disponibles à l’adresse <http://www.ggplot2-exts.org/gallery/>.

## Les graphiques interactifs et dynamiques avec Plotly

Plotly est un package permettant de réaliser des graphiques interactifs et dynamiques. Cela peut être particulièrement utile pour les résultats ayant vocation à être diffusées sur Internet. Le package s’installe comme tous les autres avec install.packages("plotly"). Le package est gratuit et open source.

Cet exemple a été copié depuis le livre de Carson Sievert (<https://plotly-book.cpsievert.me>). Le code qui a permis de faire ce graphique est sous licence Creative Commons Attribution-NonCommercial-NoDerivs 3.0 United States License (Carson Sievert ; <https://creativecommons.org/licenses/by-nc-nd/3.0/us/>).



## Conclusion

Ce chapitre nous a permis de survoler d’autres options graphiques et en particulier les packages ggplot2 et plotly. Des livres spécifiques (en anglais) couvrent tous les aspects de ces packages, ici l’objectif est de savoir que ces options existent pour y avoir recours si besoin. Les sites web “Data to Viz” et “r-graph gallery” (<https://www.data-to-viz.com> ; <https://www.r-graph-gallery.com/>) permettront de se faire une idée des possibilités offertes par R quant aux représentations graphiques. Le chapitre suivant traite des processus nécessaires pour transformer un graphique R en une figure publiable dans un article scientifique. A très bientôt !

# Du graphique à la figure dans un article scientifique

Une fois nos résultats obtenus, il est primordial de les publier. Ce n’est qu’en publiant nos résultats que nous participons à la science. Cela peut se faire via des communications dans des congrès, via des posters, ou encore (et c’est le plus fréquent), via des articles scientifiques.

Dans les articles scientifiques, les résultats graphiques prennent la forme de *figures* qui sont souvent le fruit de un ou plusieurs graphiques. Ces figures suivent des critères bien précis dictés par la revue. Prenons cet exemple adapté et traduit du guide de la revue PLoS :

*Les fichiers doivent être au format TIFF (avec compression LZW et une seule couche), ou EPS. Les dimensions doivent être de 789 à 2250 pixels et une hauteur maximum de 2625 pixels pour une résolution de 300 dpi (entre 6.68 et 19.05 cm de large - maximum 13.2 cm pour un alignement sur une colonne - et maximum 22.23 cm de haut). La police doit être Arial, Times, ou Symbol et d’une taille entre 8 et 12 points.*

Il est difficile de respecter toutes ces contraintes avec R. C’est pourquoi nous allons utiliser des logiciels tiers pour transformer nos graphiques en figures. Nous allons utiliser Inkscape pour la mise en page et Gimp pour la transformation dans les formats requis.

Quant on prend une photo, chaque pixel prend une valeur qui va définir la couleur du pixel, donc en zoomant sur une photo on va voir apparaître les pixels (nous perdons en netteté). C’est une image *matricielle*. Dans une image *vectorielle*, les éléments sont codés sous forme de segments, nous ne perdons pas en qualité car chaque élément conserve ses coordonées (Figure 16). L’avantage des images vectorielles est que l’on peut modifier les éléments de l’image sans perdre en qualité. C’est ce que nous allons faire avec Inkscape. L’avantage des images matricielles est que l’on peut choisir entre de très nombreux formats (dont TIFF). C’est ce que nous allons faire avec Gimp.

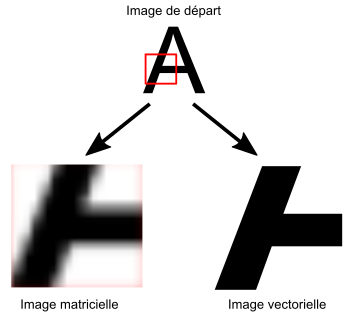


Figure 16: Image matricielle et vectorielle.

## Inkscape



Figure 17: Logo Inkscape (<https://inkscape.org>).

Inkscape est un logiciel de dessin vectoriel disponible sous Windows, Mac OS X et GNU/Linux. C’est un logiciel libre et open source sous licence GPL. Nous pouvons le télécharger à l’adresse suivante <https://inkscape.org>. Inkscape est déjà installé par défaut sur de nombreuses distributions de Linux.

Expliciter toutes les fonctionnalités de Inkscape sort du cadre de ce livre, mais nous pourrons trouver de nombreux tutoriels en lignes pour rapidement maîtriser cet outil. Brièvement, pour respecter le guide de construction d’une figure d’une revue, nous allons tout d’abord construire notre figure *brute* sous R, puis l’exporter au format PDF à l’aide de la fonction pdf().

pdf("maFigure.pdf")  
 plot(0)  
dev.off()

Ensuite il nous suffira d’importer notre pdf sous Inkscape pour le modifier. Par défaut le PDF sera *groupé* en un seul élément. Pour accéder aux sous éléments de notre figure, nous pouvons degrouper l’élément avec le raccourci Ctrl+Shift+g. Il est conseillé d’ajouter un fond blanc à notre figure pour éviter la transparence (un simple rectangle blanc sans bordure fera l’affaire). Une fois notre figure prête à être exportée, il nous faut la sauvegarder dans un fichier PNG avec le raccourci Ctrl+Shift+e, et passer à The Gimp.

## The Gimp



Figure 18: Logo Gimp (<https://www.gimp.org/>).

Gimp est un logiciel de dessin matriciel disponible sous Windows, Mac OS X et GNU/Linux. C’est un logiciel libre et open source sous licence GPL. Nous pouvons le télécharger à l’adresse suivante <https://www.gimp.org/>. Gimp signifie “GNU Image Manipulation Program”. Gimp est déjà installé par défaut sur de nombreuses distributions de Linux.

Nous allons importer notre fichier PNG puis tout simplement l’exporter dans un autre format avec le raccourci Ctrl+Shift+e. Une fenêtre nous proposera alors de choisir le nouveau format, par exemple TIFF avec une compression LZW. Notre figure est alors prête pour soumission.

## Table de référence

Voici une Table de référence (Table 5) avec quelques revues illustrant la diversité des formats pour les figures scientifiques.

Table 5: Table de référence pour la construction des figures.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Revue | Largeur | Hauteur | Format | Resolution | Police | Taille |
| PLoS | 6.68-19.05 | max 22.23 | TIFF, EPS | 300-600 dpi | Arial, Times, Symbol | 9-12 |
| PNAS | 8.7; 11.4; 17.8 | max 22.5 | TIF, EPS | 300 dpi | Arial, Helvetica, Times, Symbol, Mathematical Pi, European Pi | min 6-8 |
| Science | 9; 12.7; 18.4 | NA | AI, EPS, PDF, TIF, JPEG | 300 dpi | Helvetica | panel 10, axis 6-9, font 5 |

# Analyser des données de datalogger de température

Dans les études de biologie, d’écologie ou d’agronomie, nous utilisons fréquemment des données de température provenant de dataloggers. Dans cette étude de cas, nous verrons comment analyser ces données en utilisant les données de température de l’altiplano bolivien près de la ville de El Alto. La première étape consiste à transformer les données du datalogger en un format facile à lire pour R. Nous utiliserons un fichier CSV et la fonction [read.table()](#import). Le fichier peut être téléchargé à partir du site Web du livre sur GitHub (<https://github.com/frareb/myRBook_FR/blob/master/myFiles/E05C13.csv>), ou alors lu directement par R depuis sa source (<https://raw.githubusercontent.com/frareb/myRBook_FR/master/myFiles/E05C13.csv>).

# Lecture du fichier en local après téléchargement :   
bdd <- read.table("myFiles/E05C13.csv", skip = 1, header = TRUE,   
 sep = ",", dec = ".", stringsAsFactors = FALSE)  
# Lecture du fichier depuis GitHub :   
# bdd <- read.table("https://raw.githubusercontent.com/frareb/myRBook\_FR/master/myFiles/E05C13.csv",   
# skip = 1, header = TRUE, sep = ",", dec = ".", stringsAsFactors = FALSE)  
colnames(bdd) <- c("id", "date", "temp")  
head(bdd)

## id date temp  
## 1 1 11/12/15 23:00:00 4.973  
## 2 2 11/12/15 23:30:00 4.766  
## 3 3 11/13/15 00:00:00 4.844  
## 4 4 11/13/15 00:30:00 4.844  
## 5 5 11/13/15 01:00:00 5.076  
## 6 6 11/13/15 01:30:00 5.282

tail(bdd)

## id date temp  
## 32781 32781 09/25/17 21:00:00 7.091  
## 32782 32782 09/25/17 21:30:00 6.914  
## 32783 32783 09/25/17 22:00:00 6.813  
## 32784 32784 09/25/17 22:30:00 6.611  
## 32785 32785 09/25/17 23:00:00 6.331  
## 32786 32786 09/25/17 23:30:00 5.385

str(bdd)

## 'data.frame': 32786 obs. of 3 variables:  
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...  
## $ date: chr "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...  
## $ temp: num 4.97 4.77 4.84 4.84 5.08 ...

Nous pouvons voir que la date est au format character et qu’elle contient la date avec le mois, le jour et l’année séparés par /, puis vient un espace et l’heure avec des heures de 0 à 24, minutes et secondes, séparés par : (exemple: 11/12/15 23:00:00 pour le 12 novembre 2015 à 11 heures du soir). Nous allons séparer les informations en plusieurs objets. Séparons d’abord la date de l’heure. Pour cela, nous utiliserons la fonction strsplit() en utilisant l’espace entre la date et l’heure comme séparateur.

strsplit("11/12/15 23:00:00", split = " ")

## [[1]]  
## [1] "11/12/15" "23:00:00"

Comme l’indiquent les doubles crochets, la fonction strsplit() renvoie un objet au format list. Nous voulons le vecteur qui correspond au premier élément de la liste, donc nous allons ajouter [[1]].

strsplit("11/12/15 23:00:00", split = " ")[[1]]

## [1] "11/12/15" "23:00:00"

Le premier élément du vector est la date. Pour avoir toutes les dates, nous allons faire une boucle avec la fonction sapply().

bddDay <- sapply(strsplit(bdd[, 2], split = " "), "[[", 1)  
head(bddDay)

## [1] "11/12/15" "11/12/15" "11/13/15" "11/13/15" "11/13/15" "11/13/15"

Ensuite, nous aurons besoin des dates dans le format Date. pour cela nous allons transformer l’objet au format Date avec la fonction as.Date().

bddDay <- as.Date(sapply(strsplit(bdd[, 2], split = " "), "[[", 1), format = "%m/%d/%y")  
head(bddDay)

## [1] "2015-11-12" "2015-11-12" "2015-11-13" "2015-11-13" "2015-11-13"  
## [6] "2015-11-13"

Nous allons maintenant ajouter l’objet bddDay à notre data.frame bdd. Ensuite, nous allons vérifier que cette nouvelle colonne dans notre data.frame est bien au format Date avec la fonction str()

bdd$day <- bddDay  
str(bdd)

## 'data.frame': 32786 obs. of 4 variables:  
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...  
## $ date: chr "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...  
## $ temp: num 4.97 4.77 4.84 4.84 5.08 ...  
## $ day : Date, format: "2015-11-12" "2015-11-12" ...

Si l’information concernant la date n’est pas suffisante, nous pouvons ajouter l’heure. Dans ce cas c’est le format POSIX qu’il faudra utiliser avec la fonction as.POSIXct(). De la même manière nous allons ajouter l’information dans notre objet bdd et vérifier le format avec la fonction str().

bddPosix <- as.POSIXct(bdd$date, format = "%m/%d/%y %H:%M:%S")  
head(bddPosix)

## [1] "2015-11-12 23:00:00 CET" "2015-11-12 23:30:00 CET"  
## [3] "2015-11-13 00:00:00 CET" "2015-11-13 00:30:00 CET"  
## [5] "2015-11-13 01:00:00 CET" "2015-11-13 01:30:00 CET"

bdd$posix <- bddPosix  
str(bdd)

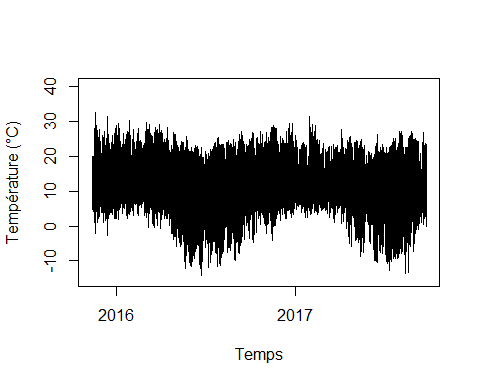
## 'data.frame': 32786 obs. of 5 variables:  
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...  
## $ date : chr "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...  
## $ temp : num 4.97 4.77 4.84 4.84 5.08 ...  
## $ day : Date, format: "2015-11-12" "2015-11-12" ...  
## $ posix: POSIXct, format: "2015-11-12 23:00:00" "2015-11-12 23:30:00" ...

Dans les fonctions as.Date() et as.POSIXct(), nous avons utilisé l’argument format qui correspond au format dans lequel l’information sur la date est codée. Voici les éléments les plus courants pour le formatage de la date :

|  |  |
| --- | --- |
| code | Valeur |
| %a | jour de la semaine (abréviation) |
| %A | jour de la semaine |
| %b | mois (abréviation) |
| %B | mois |
| %d | jour du mois (decimal) |
| %j | jour de l’année (decimal) |
| %m | mois (decimal) |
| %y | année avec deux chiffres |
| %Y | année |
| %U | semaine de l’année en commençant par dimanche (decimal) |
| %W | semaine de l’année en commençant par lundi (decimal) |
| %H | heure 24 |
| %I | heure 12 |
| %M | minute |
| %S | seconde |

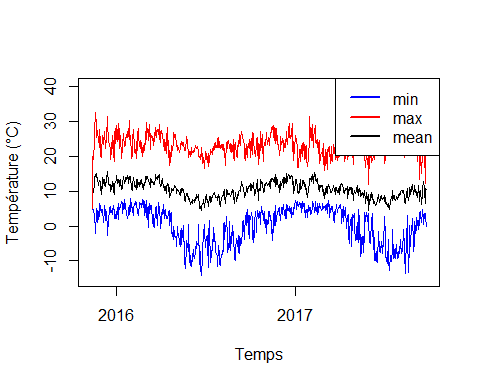
Nous pouvons visualiser les données avec la fonction plot().

plot(x = bdd$day, y = bdd$temp,   
 type = 'l', ylim = c(-15, 40),   
 xlab = "Temps", ylab = "Température (°C)")



Nous pouvons simplifier les informations en ne calculant que les températures minimales, moyennes et maximales avec la fonction tapply().

tempDayMean <- tapply(bdd$temp, INDEX = bdd$day, FUN = mean)  
tempDayMin <- tapply(bdd$temp, INDEX = bdd$day, FUN = min)  
tempDayMax <- tapply(bdd$temp, INDEX = bdd$day, FUN = max)  
plot(x = as.Date(names(tempDayMean), format = "%Y-%m-%d"),   
 y = tempDayMean, type = 'l', ylim = c(-15, 40),   
 xlab = "Temps", ylab = "Température (°C)")  
points(as.Date(names(tempDayMin), format = "%Y-%m-%d"),   
 y = tempDayMin, type = 'l', col = 4)  
points(as.Date(names(tempDayMax), format = "%Y-%m-%d"),   
 y = tempDayMax, type = 'l', col = 2)  
legend("topright", legend = c("min", "max", "mean"),   
 lty = 1, lwd = 2, col = c(4, 2, 1))



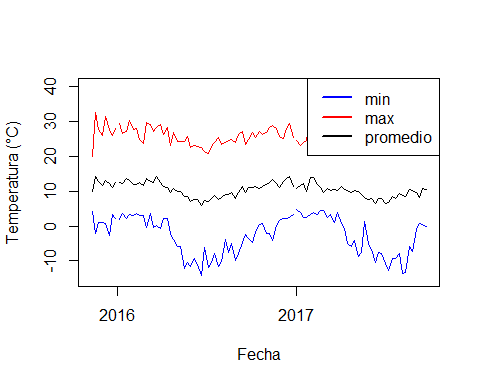
Nous pouvons très facilement représenter le même information par semaine. Il suffit de changer l’argument INDEX pour une date en semaine. Voici un exemple de format par semaine :

anoSem <- format(bdd$posix, format = "%Y-%W")  
head(anoSem)

## [1] "2015-45" "2015-45" "2015-45" "2015-45" "2015-45" "2015-45"

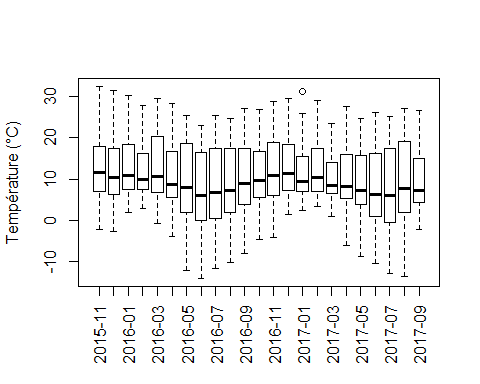
Et voici le graphique par semaine :

tempWeekMean <- tapply(bdd$temp,   
 INDEX = format(bdd$posix, format = "%Y-%W-1"), FUN = mean)  
tempWeekMin <- tapply(bdd$temp,   
 INDEX = format(bdd$posix, format = "%Y-%W-1"), FUN = min)  
tempWeekMax <- tapply(bdd$temp,   
 INDEX = format(bdd$posix, format = "%Y-%W-1"), FUN = max)  
plot(x = as.Date(names(tempWeekMean), format = "%Y-%W-%u"),   
 y = tempWeekMean, type = 'l', ylim = c(-15, 40),   
 xlab = "Fecha", ylab = "Temperatura (°C)")  
points(x = as.Date(names(tempWeekMin), format = "%Y-%W-%u"),   
 y = tempWeekMin, type = 'l', col = 4)  
points(x = as.Date(names(tempWeekMax), format = "%Y-%W-%u"),   
 y = tempWeekMax, type = 'l', col = 2)  
legend("topright", legend = c("min", "max", "promedio"),   
 lty = 1, lwd = 2, col = c(4, 2, 1))



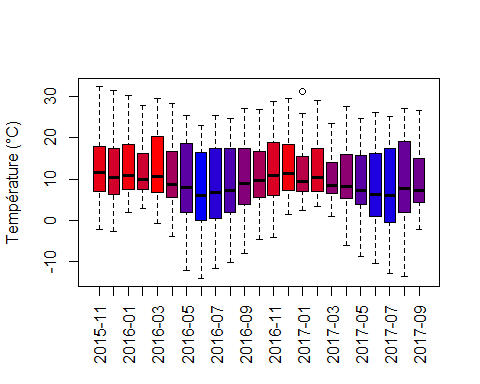
Dans ce type de représentation nous perdons l’information sur la variabilité des températures par semaine. En focntion des objectifs il pourra être intéressant de préférer une représentation sous forme de boxplot (ici un boxplot par mois en formattant la date avec "%Y-%m").

boxplot(bdd$temp ~ format(bdd$posix, format = "%Y-%m"), las = 3,  
 xlab = "", ylab = "Température (°C)")



Pour comparer les différentes valeurs de température, il peut être utile d’attribuer une couleur à chaque boxplot. Par exemple, nous pouvons utiliser un gradient de couleur du bleu au rouge en utilisant la valeur de la température moyenne.

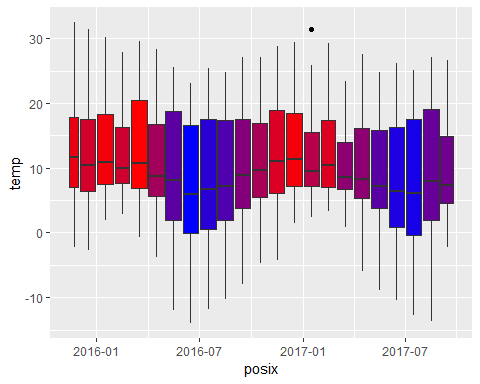
tempMonthMean <- tapply(bdd$temp,   
 INDEX = format(bdd$posix, format = "%Y-%m"), FUN = mean)  
myCol <- colorRampPalette(c("blue", "red"))(101)  
tempMeanDayPos <- round(  
 (tempMonthMean - min(tempMonthMean)) /   
 (max(tempMonthMean) - min(tempMonthMean))\*100) + 1  
boxplot(bdd$temp ~ format(bdd$posix, format = "%Y-%m"), las = 3,   
 col = myCol[tempMeanDayPos],  
 xlab = "", ylab = "Température (°C)")



Et pour celles et ceux qui utilisent ggplot2 :

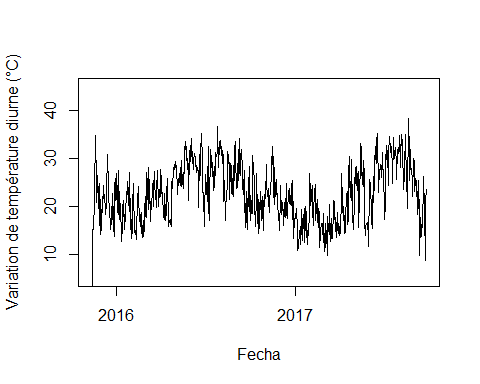
pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("ggplot2")  
  
tempMonthMean <- tapply(bdd$temp,   
 INDEX = format(bdd$posix, format = "%Y-%m"), FUN = mean)  
myCol <- colorRampPalette(c("blue", "red"))(101)  
tempMeanDayPos <- round(  
 (tempMonthMean - min(tempMonthMean)) /   
 (max(tempMonthMean) - min(tempMonthMean))\*100) + 1  
p01 <- ggplot(data = bdd,   
 aes(  
 x = posix,   
 y = temp,   
 group = format(bdd$posix, format = "%Y-%m"))) +   
 geom\_boxplot(outlier.colour = "black", fill = myCol[tempMeanDayPos])  
p01

## Warning: Removed 4 rows containing missing values (stat\_boxplot).



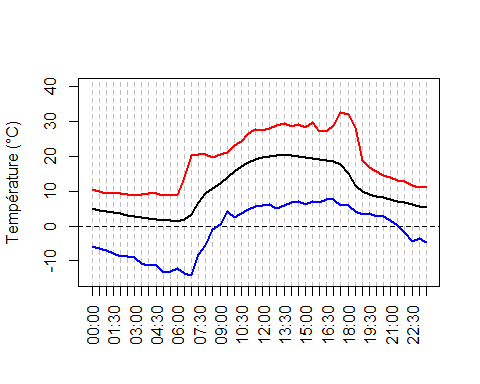
Nous pouvons également calculer la différence entre la température maximale et la température minimale (variation de la température diurne).

tempDayTR <- tempDayMax - tempDayMin  
plot(x = as.Date(names(tempDayMean), format = "%Y-%m-%d"),   
 y = tempDayTR, type = 'l', ylim = c(5, 45),   
 xlab = "Fecha", ylab = "Variation de température diurne (°C)")



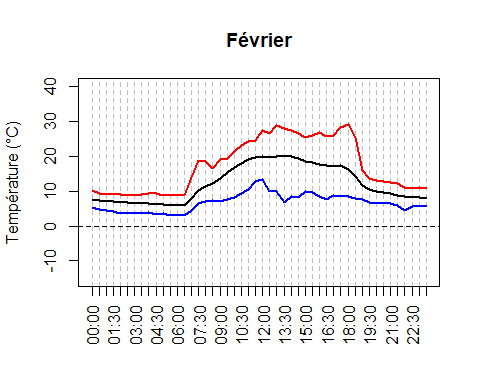
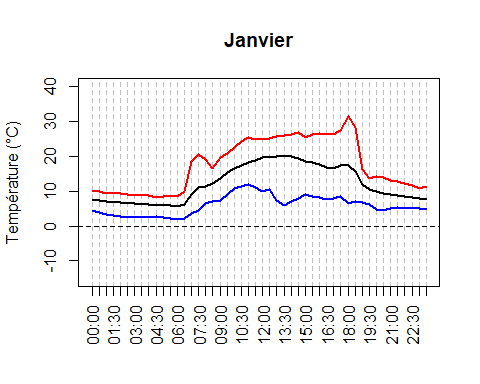
Une autre possibilité est de regrouper les données pour avoir la température moyenne des heures de la journée avec la fonction aggregate().

tempHourMean <- aggregate(x = bdd$temp,   
 by = list(format(bdd$posix, format = "%H:%M")), FUN = mean)  
tempHourMin <- aggregate(x = bdd$temp,   
 by = list(format(bdd$posix, format = "%H:%M")), FUN = min)  
tempHourMax <- aggregate(x = bdd$temp,   
 by = list(format(bdd$posix, format = "%H:%M")), FUN = max)  
hours <- seq(from = 0, to = 23.5, by = 0.5)  
plot(x = hours,   
 y = tempHourMean[, 2], type = 'l', ylim = c(-15, 40),   
 xlab = "", ylab = "Température (°C)", lwd = 2,   
 xaxt = "n", panel.first = {  
 abline(v = hours, col = "gray", lty = 2)  
 abline(h = 0, lty = 2)   
 })  
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)  
points(x = hours, y = tempHourMin[, 2], type = 'l', col = 4, lwd = 2)  
points(x = hours, y = tempHourMax[, 2], type = 'l', col = 2, lwd = 2)



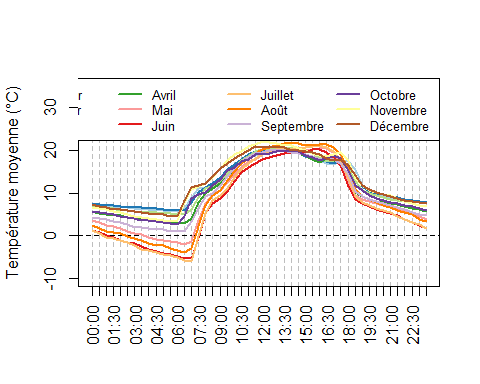
Nous pouvons également calculer les températures des heures de la journée pour chaque mois.

meses <- c("Janvier", "Février", "Mars", "Avril", "Mai", "Juin",   
 "Juillet", "Août", "Septembre", "Octobre", "Novembre", "Décembre")  
hours <- seq(from = 0, to = 23.5, by = 0.5)  
bddMonth <- format(bdd$day, format = "%m")  
tempDayEachMonth <- lapply(sort(unique(bddMonth)), function(myMonth){  
 bddX <- bdd[bddMonth == myMonth, ]  
 tempHourMean <- aggregate(x = bddX$temp, by = list(format(bddX$posix, format = "%H:%M")), FUN = mean)  
 tempHourMin <- aggregate(x = bddX$temp, by = list(format(bddX$posix, format = "%H:%M")), FUN = min)  
 tempHourMax <- aggregate(x = bddX$temp, by = list(format(bddX$posix, format = "%H:%M")), FUN = max)  
 return(data.frame(tempHourMean, tempHourMin, tempHourMax))  
})  
  
# for (i in seq\_along(tempDayEachMonth)){ # para todos los meses  
for (i in 1:2){ # solo para Enero y Febrero   
 plot(x = hours, y = tempDayEachMonth[[i]][, 2],   
 type = 'l', ylim = c(-15, 40),   
 xlab = "", ylab = "Température (°C)", lwd = 2,   
 main = meses[i],  
 xaxt = "n", panel.first = {  
 abline(v = hours, col = "gray", lty = 2)  
 abline(h = 0, lty = 2)   
 })  
 axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)  
 points(x = hours, y = tempDayEachMonth[[i]][, 4],   
 type = 'l', col = 4, lwd = 2)  
 points(x = hours, y = tempDayEachMonth[[i]][, 6],   
 type = 'l', col = 2, lwd = 2)  
}

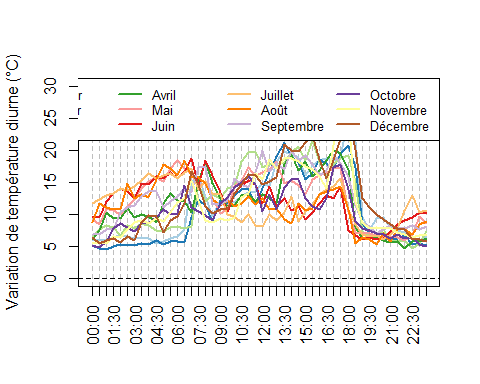


Ou les regrouper dans un même graphique, ainsi que la variation diurne des températures pour chaque mois.

plot(x = hours, y = tempDayEachMonth[[1]][, 2], type = 'n', ylim = c(-10, 35),  
 xlab = "", ylab = "Température moyenne (°C)",  
 xaxt = "n",   
 panel.first = {  
 abline(v = hours, col = "gray", lty = 2)  
 abline(h = 0, lty = 2)   
 })  
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)  
myColors <- c("#A6CEE3", "#1F78B4", "#B2DF8A", "#33A02C", "#FB9A99",   
 "#E31A1C", "#FDBF6F", "#FF7F00", "#CAB2D6", "#6A3D9A", "#FFFF99",   
 "#B15928")  
for (i in seq\_along(tempDayEachMonth)){  
 points(x = hours,   
 y = tempDayEachMonth[[i]][, 2],   
 type = 'l', col = myColors[i], lwd = 2)  
}  
legend("topright", ncol = 4, legend = meses, col = myColors,   
 lty = 1, lwd = 2, cex = 0.8)



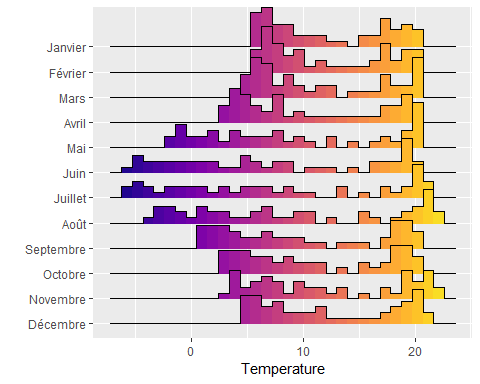
plot(x = hours, y = tempDayEachMonth[[1]][, 2], type = 'n', ylim = c(0, 30),  
 xlab = "", ylab = "Variation de température diurne (°C)",  
 xaxt = "n",   
 panel.first = {  
 abline(v = hours, col = "gray", lty = 2)  
 abline(h = 0, lty = 2)   
 })  
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)  
myColors <- c("#A6CEE3", "#1F78B4", "#B2DF8A", "#33A02C", "#FB9A99",   
 "#E31A1C", "#FDBF6F", "#FF7F00", "#CAB2D6", "#6A3D9A", "#FFFF99",   
 "#B15928")  
for (i in seq\_along(tempDayEachMonth)){  
 points(x = hours,   
 y = tempDayEachMonth[[i]][, 6] - tempDayEachMonth[[i]][, 4],   
 type = 'l', col = myColors[i], lwd = 2)  
}  
legend("topright", ncol = 4, legend = meses, col = myColors,   
 lty = 1, lwd = 2, cex = 0.8)



Nous pouvons aussi représenter les températures journalières avec des graphiques de type “ridgeline” et le package ggplot2 (<https://www.data-to-viz.com/graph/ridgeline.html>).

pkgCheck <- function(x){   
 if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 }  
}  
pkgCheck("ggplot2")  
pkgCheck("ggridges")  
pkgCheck("viridis")  
meanTemps <- unlist(lapply(tempDayEachMonth, "[[", 2))  
labelMonth <- rep(meses, each = 48)  
dfTemps <- data.frame(month = labelMonth, value = meanTemps,   
 stringsAsFactors = FALSE)  
dfTemps$month <- factor(dfTemps$month, levels = rev(meses))  
p <- ggplot(data = dfTemps, aes(y = month, x = value, fill = ..x..))  
p <- p + geom\_density\_ridges\_gradient(stat="binline")  
p <- p + scale\_fill\_viridis(name = "Temp. [°C]", option = "C")  
p <- p + xlab("Temperature") + ylab("") +  
 theme(  
 legend.position="none",  
 panel.spacing = unit(0.1, "lines"),  
 strip.text.x = element\_text(size = 8)  
 )   
p

## `stat\_binline()` using `bins = 30`. Pick better value with `binwidth`.



# Obtenir le numéro WOS d’un article scientifique à partir de son numéro DOI

Il peut être intéressant d’obtenir le numéro WOS d’un article scientifique. Ce numéro est cependant fastidieux à obtenir, d’autant plus si nous souhaitons le récupérer pour une liste d’articles ! Par chance *The Kitchin Research Group* dans leur blog de juin 2015 (<http://kitchingroup.cheme.cmu.edu/blog/2015/06/08/Getting-a-WOS-Accession-number-from-a-DOI/>) propose une méthode pour récupérer le numéro WOS à partir du numéro DOI. C’est cette méthode que nous allons utiliser avec R et le package httr. En bref, cette méthode consite à interroger le site web du WOS à partir du numéro DOI. Le site web du WOS va répondre en spécifiant qu’il faut se connecter pour accéder à l’article. Il y a pour cela une redirection vers une page web dont l’URL contient le numéro WOS. Il suffit alors d’extraire le numéro WOS de l’URL de la page web. **Si nous utilisons un proxy pour accéder au site web du WOS, il faut donc le désactiver pour que la méthode fonctionne**.

Tout d’abord si cela n’est pas déjà fait il faut installer le package httr avec install.packages("httr"), puis le charger avec library("httr"). Une autre solution consiste à utiliser la fonction suivante qui va vérifier si le package est installé puis le charger (il existe de nombreuses déclinaisons de cette fonction sur internet, il s’agit ici d’un mélange de multiples sources).

pkgCheck <- function(packages){  
 for(x in packages){  
 try(if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 })  
 }  
}  
pkgCheck("httr")

La liste des numéros DOI est la suivante (contenu dans un vector) :

myDOIs <- c("10.1111/2041-210X.12935", "10.1007/s13355-017-0480-5")  
print(myDOIs)

## [1] "10.1111/2041-210X.12935" "10.1007/s13355-017-0480-5"

Pour chaque DOI, nous allons interroger le site web du WOS, récupérer l’URL de redirection, puis récupérer le numéro WOS. Nous allons donc faire une boucle sur notre vector contenant les DOI. Nous utilisons la fonction seq\_along() qui va prendre comme valeurs les éléments d’une séquence de 1 à la taille de l’objet myDOIs, soit : i = 1, puis i = 2.

for(i in seq\_along(myDOIs)){  
 # ...  
}

Dans la boucle, le DOI que nous allons traiter est donc myDOIs[i] que nous allons appeler myDOI. La page d’interrogation du WOS correspond à la concaténation de l’URL du WOS avec le numéro WOS (l’URL est présentée sur plusieurs lignes pour respecter la largeur de page de ce livre).

for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
}

Maintenant nous allons utiliser la fonction GET() du package httr pour récupérer l’URL.

for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
 r <- GET(myWebPage)  
 urlWOS <- r[[1]]  
}

Il se peut que pour un article, il n’y ait pas de numéro WOS correspondant. Pour que notre script ne soit pas arrêté en cas d’erreur il faut donc **gérer cette exception**. Nous allons créer un objet tryExtract qui va prendre comme valeur une chaîne de caractères vide "". Ensuite nous allons essayer avec la fonction try() d’extraire le numéro WOS avec la fonction substr(). Le numéro WOS se situe depuis le caractère numéro 117 jusqu’au caractère numéro 131 de l’URL.

for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
 r <- GET(myWebPage)  
 urlWOS <- r[[1]]  
 tryExtract <- ""  
 try(tryExtract <- substr(x = urlWOS, start = 117, stop = 131), silent = TRUE)  
}

Nous pouvons ensuite vérifier que l’extraction correspond bien à un numéro en utilisant une **expression régulière**. Ici nous allons simplement vérifier que l’extraction ne contient que des chiffres. Dans le cas contraire tryExtract reprendra sa valeur initiale "".

for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
 r <- GET(myWebPage)  
 urlWOS <- r[[1]]  
 tryExtract <- ""  
 try(tryExtract <- substr(x = urlWOS, start = 117, stop = 131), silent = TRUE)  
 if(!grepl(pattern = '^[0-9]\*$', x = tryExtract)){tryExtract <- ""}  
}

Le résultat est ensuite stocké dans un vector créé au préalable et appelé vecWOS.

vecWOS <- vector()  
for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
 r <- GET(myWebPage)  
 urlWOS <- r[[1]]  
 tryExtract <- ""  
 try(tryExtract <- substr(x = urlWOS, start = 117, stop = 131), silent = TRUE)  
 if(!grepl(pattern = '^[0-9]\*$', x = tryExtract)){tryExtract <- ""}  
 vecWOS <- append(vecWOS, tryExtract)  
}

Nous pouvons alors créer un objet de type data.frame qui va contenir les numéros DOI et les numéros WOS, et éventuellement l’exporter dans un fichier CSV.

dfDOIWOS <- data.frame(DOI = myDOIs, WOS = vecWOS)  
write.csv(dfDOIWOS, file = "dfDOIWOS.csv", row.names = FALSE)

Le résultat est le suivant (non exécuté car la procédure d’interrogation avec la fonction GET() est très lente : **si nous souhaitons travailler sur un liste de plusieurs dizaines ou centaines d’articles, plusieurs heures seront nécessaires** avant d’obtenir le résultat).

# DOI WOS  
# 1 10.1111/2041-210X.12935 000429421800031  
# 2 10.1007/s13355-017-0480-5 000400381400016

Voici le code complet :

pkgCheck <- function(packages){  
 for(x in packages){  
 try(if (!require(x, character.only = TRUE)){  
 install.packages(x, dependencies = TRUE)  
 if(!require(x, character.only = TRUE)) {  
 stop()  
 }  
 })  
 }  
}  
pkgCheck("httr")  
  
myDOIs <- c("10.1111/2041-210X.12935", "10.1007/s13355-017-0480-5")  
  
vecWOS <- vector()  
for(i in seq\_along(myDOIs)){  
 myDOI <- myDOIs[i]  
 myWebPage <- paste0(  
 "http://ws.isiknowledge.com/",   
 "cps/openurl/service?url\_ver=Z39.88-2004",   
 "&rft\_id=info:doi/", myDOI)  
 r <- GET(myWebPage)  
 urlWOS <- r[[1]]  
 tryExtract <- ""  
 try(tryExtract <- substr(x = urlWOS, start = 117, stop = 131), silent = TRUE)  
 if(!grepl(pattern = '^[0-9]\*$', x = tryExtract)){tryExtract <- ""}  
 vecWOS <- append(vecWOS, tryExtract)  
}  
  
dfDOIWOS <- data.frame(DOI = myDOIs, WOS = vecWOS)  
write.csv(dfDOIWOS, file = "dfDOIWOS.csv", row.names = FALSE)

La boucle for() pourrait être remplacée par une boucle sapply() pour gagner en temps d’exécution. Un gain serait également possible en effectuant une parallélisation sur cette boucle. Pour information, voici un exemple de temps d’exécution renvoyé par la fonction system.time() et microbenchmark::microbenchmark() :

# system.time()  
# user system elapsed  
# 0.10 0.01 35.00  
  
# microbenchmark()  
# Unit: seconds  
# expr min lq mean median uq max neval  
# myFun() 36.58966 36.58966 36.58966 36.58966 36.58966 36.58966 1

Nous venons de faire un script qui permet à partir d’une liste de numéros DOI d’obtenir automatiquement les numéros WOS.