Statistiques appliquées aux sciences

SCI 1018

Programmation avec R – notions générales

Marc J. Mazerolle

Institut de recherche sur les forêts, Université du Québec en Abitibi-Témiscamingue

Table des matières

1 INTRODUCTION **4**

2 LE PROJET R **4**

3 LA SYNTAXE **5**

4 À L'AIDE ! **6**

5 VECTEURS ET OPÉRATIONS DE BASE **8**

5.1 Créer un vecteur 8

5.2 Sélectionner des valeurs 10

5.3 Répéter des valeurs 11

5.4 Créer une série 12

5.5 Connaître les caractéristiques d'un vecteur 13

6 OPÉRATEURS MATHÉMATIQUES **14**

7 TESTS LOGIQUES **15**

8 MATRICES ET CALCULS MATRICIELS **21**

8.1 Créer une matrice 21

8.2 Sélectionner des valeurs 22

8.3 Connaître les caractéristiques d'une matrice 23

8.4 Opérations et manipulation de matrices 24

9 JEUX DE DONNÉES **26**

9.1 Créer un jeu de données 26

9.2 Connaître les caractéristiques de jeu de données 27

9.3 Accéder aux variables d'un jeu de données 29

9.4 Importer des fichiers de données 30

9.5 Ajouter une variable 33
9.6 Créer des sous-ensembles 34
9.7 Effectuer un tri 36
9.8 Créer des tableaux récapitulatifs des fréquences 37
9.9 Modifier la structure d'un jeu de données 41
9.10 Exporter un jeu de données 44

Index **48**

Notez bien. Le contenu théorique de chaque leçon est présenté dans un document comme celui-ci et intègre souvent du code en langage de programmation `R`. Nous sommes conscient que la convention en français est d'utiliser la virgule pour indiquer la décimale. Toutefois, nous utilisons systématiquement le point pour désigner la décimale dans le texte. Ce choix vient du fait que la syntaxe de `R` utilise le point comme décimale – à la fois pour la saisie de valeurs numériques et pour l'affichage des sorties d'analyses. Ainsi, l'usage du point uniformisera le texte et nous sommes d'avis que cette décision facilitera sa compréhension.

1 INTRODUCTION

Ce document a pour but d'illustrer les bases nécessaires afin de se familiariser avec `R`. Pour les personnes qui n'ont jamais rencontré cet environnement, nous recommandons très fortement de répéter les exemples afin de bien comprendre la syntaxe et les subtilités de la programmation en ce qui est devenu la *lingua franca* des statistiques. Dans ce document, le code `R` est distingué du reste du texte en utilisant une `police monospace comme celle-ci`. Aussi, vous remarquerez que le symbole `>` précède toujours les commandes illustrées dans les exemples du document. Ce symbole indique que les commandes pourraient être saisies dans la console `R` ou dans l'éditeur.

2 LE PROJET R

`R` est un langage de programmation ainsi qu'un logiciel développés spécifiquement pour les analyses statistiques et les graphiques. Il est le résultat des travaux de deux statisticiens-programmeurs de la Nouvelle-Zélande, Ross Ihaka et Robert Gentleman qui se sont inspirés des langages `S` et `Scheme` (http://fr.wikipedia.org/wiki/Scheme). D'ailleurs, ils ont nommé le langage `R` en référence à leur prénom (Ross et Robert), mais aussi en guise de clin d'oeil à son prédecesseur `S` (http://fr.wikipedia.org/wiki/S_(langage_de_programmation)).

Dès les débuts, Ihaka et Gentleman ont rendu le code source de leur langage disponible gratuitement à tous, mettant en pratique la philosophie que la qualité des graphiques et des analyses executés ne doivent pas être une fonction de l'épaisseur du portefeuille. Quiconque peut ainsi faire des analyses de très haut niveau peu importe sa situation financière. Le code source du langage étant ouvert, quiconque peut aller voir, s'il le désire, comment les fonctions ont été codées, les modifier pour son propre usage ou en créer de nouvelles pour des applications spécifiques, puis les rendre disponibles à tous par l'intermédiaire de banques de fonctions. Cette transparence et cet esprit de partage expliquent en partie le succès fulgurant de ce langage et la formation d'une large communauté d'utilisateurs dans le monde.

L'initiative des deux Néo-Zélandais a pris la forme d'un projet global auquel s'est greffée une équipe de programmeurs et de statisticiens qui mettent à jour régulièrement le logiciel, y apportent des améliorations et rendent le tout disponible gratuitement à partir de plus de 75 sites répartis sur les cinq continents. Un aspect important de `R` est qu'il est disponible sous différents systèmes d'exploitation, incluant MS-Windows, Mac, et GNU/Linux.

3 LA SYNTAXE

`R` est un langage de programmation orienté objet et interprété, ce qui signifie que toute information est gardée en mémoire sous forme d'objet pendant une session `R` tant que ce dernier n'est pas modifié ou éliminé (« orienté objet ») et qu'on n'a pas à compiler le programme avant de l'exécuter (« interprété »). On peut créer un objet en lui assignant un nom et de l'information. Une fois créé, cet objet existe dans la mémoire de `R` et peut prendre différentes formes comme une valeur numérique, du texte, un vecteur, une matrice, une fonction, un jeu de données ou le résultat d'une analyse.

Contrairement à d'autres logiciels, lorsqu'on crée un élément, il demeure accessible durant toute la session. Par exemple, après avoir réalisé une analyse et stocké les résultats dans un objet, on peut décider d'extraire les résidus du modèle, des valeurs prédites, un R^2 ou une autre valeur sans avoir à faire rouler à nouveau l'analyse. En effet, `R` garde en mémoire le résultat de l'analyse pour extraire les valeurs nécessaires. Conceptuellement, un objet est l'équivalent d'un contenant utilisé pour conserver des aliments dans un réfrigérateur dans votre milieu de travail – on peut mettre des étiquettes avec le nom du propriétaire sur le contenant afin de distinguer les différents contenants dans le réfrigérateur. C'est le même principe utilisé par `R` : on stocke de l'information dans un objet (le contenant) et le nom de cet objet (l'étiquette) permet à `R` de gérer l'information qu'on lui soumet.

Toutes les commandes ou les fonctions en `R` suivent la même stratégie : un nom suivi de parenthèses. Par exemple, la fonction `mean( )`, étonnamment, permet de calculer la moyenne

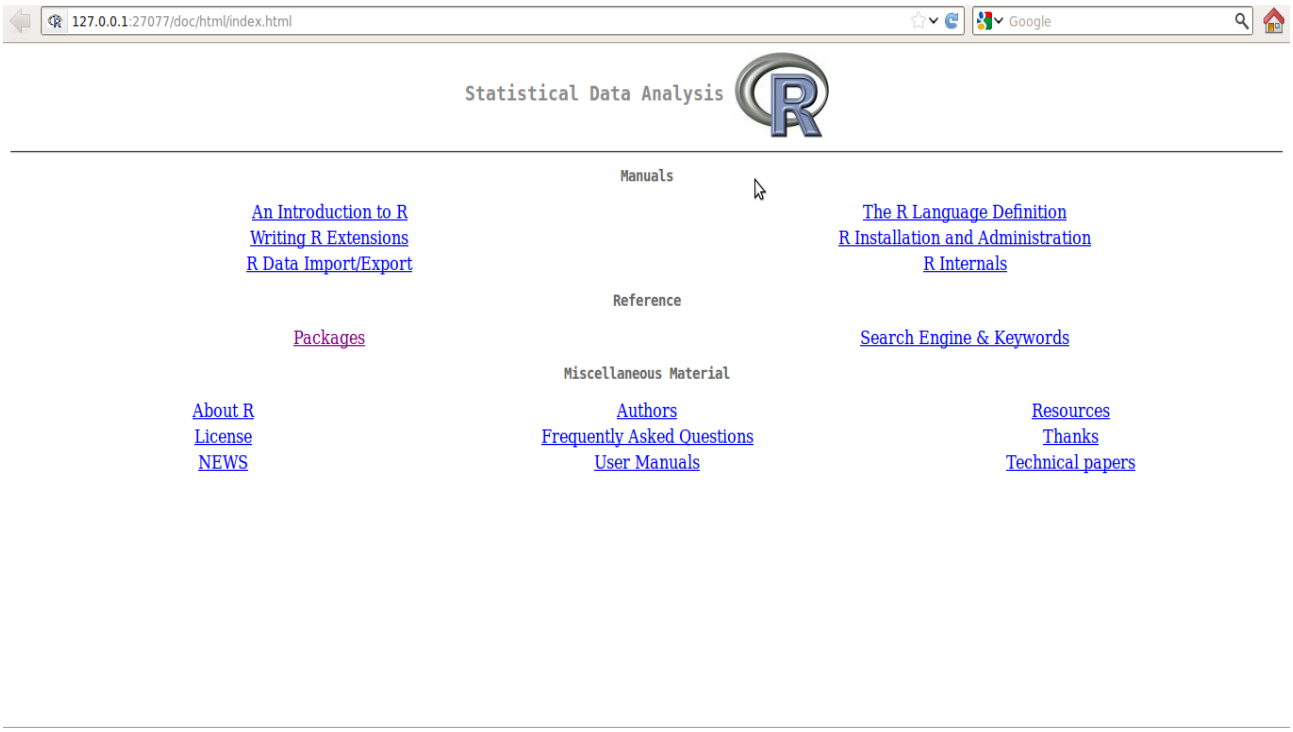

FIGURE 1 – Aide dans `R` avec `help.start( )`.

arithmétique. La plupart des fonctions possèdent des arguments séparés par des virgules. Dans notre exemple, la fonction `mean( )` possède un argument `'x = '` que nous utilisons pour spécifier la variable sur laquelle nous voulons calculer la moyenne. Toutefois, avant de continuer avec des exemples, voyons comment accéder à l'aide disponible dans `R`.

4 À L'AIDE !

Il existe plusieurs ressources pour assister les utilisateurs de `R`. Par exemple, la commande `help.start( )` génère une page `html` à partir de laquelle nous pouvons naviguer vers différents thèmes au moyen d'hyperliens (fig. 1).

La commande `help.search( )` permet de faire une recherche spécifique dans les fichiers d'aide de `R`. L'argument `apropos` permet de spécifier le sujet de la recherche. Par exemple,

la commande

```
> help.search(apropos = "generalized linear models")
```

retournera l'ensemble des fonctions qui traitent des modèles linéaires généralisés parmi les fonctions de R.

Afin d'accéder directement à l'aide d'une fonction, on peut utiliser la commande ? suivie du nom de la fonction. Toute la documentation des fonctions est fondée sur la même structure : la présentation de la syntaxe, des arguments, et des détails sur l'usage et le résultat de l'opération, de l'auteur de la fonction et des références. La plupart des fonctions finissent par des exemples qu'on peut reproduire soi-même : il suffit de copier-coller le code dans la console. C'est un excellent moyen de se familiariser avec une fonction et son utilisation.

```
> ?mean #retourne l'aide de la fonction mean( )
```

Un autre outil pratique est la fonction RSiteSearch( ) avec l'objet de la recherche entre guillements. Cette fonction cherche dans les pages d'aide de toutes les fonctions de R du site de R et n'est pas limité aux fonctions installées sur votre ordinateur. Après avoir exécuté la commande, une fenêtre de votre fureteur internet préféré s'ouvrira pour accéder aux résultats. Bien sûr, il est nécessaire d'avoir une connexion internet lors de l'exécution de cette fonction. Par exemple,

```
> RSiteSearch("error bars")
```

présentera comme résultat les pages de fonctions faisant mention de barres d'erreurs sur des graphiques. On peut ensuite modifier la recherche ou l'étendre à partir du moteur de recherche directement sur la page web.

Plusieurs ressources électroniques sont également disponibles sur une multitude de sites internet de chercheurs ou d'utilisateurs de R. Le site http://www.rseek.org peut faciliter cette recherche.

5 VECTEURS ET OPÉRATIONS DE BASE

R stocke l'information sous forme d'objets et ces derniers peuvent prendre différentes formes. Le premier type d'objet que nous verrons est le vecteur. Les éléments contenus dans un vecteur sont d'un seul type ou mode. Parmi les modes les plus courants, on compte, le mode numérique (`numeric`), entier (`integer`), caractère (`character`), facteur (`factor`) ou logique (`logical`).

5.1 Créer un vecteur

Le plus simple vecteur dans R consiste en une seule valeur (un scalaire). Comme premier exemple, créons un objet pour stocker le nombre de pistes (10) sur l'album *Folklore* du groupe 16 Horsepower [1]. Créons un vecteur que nous nommerons `Nbre.pistes` et qui prendra la valeur 10.

```
> Nbre.pistes <- 10
```

Notons le symbole `<-` (symbole inférieur suivi d'un tiret) qui est un opérateur d'assignation. En d'autres mots, la valeur de 10 est envoyée dans l'objet dénommé `Nbre.pistes`. Le symbole `=` peut aussi être utilisé pour assigner des valeurs, mais cela n'a pas toujours été le cas. Dans ce cours, nous utilisons le symbole classique d'assignation `<-`.

Rappelons aussi que l'on créé simplement en lui donnant un nom. Ce nom doit commencer par une lettre, laquelle peut être suivie de chiffres ou d'autres lettres. Par ailleurs, le nom de l'objet ne peut contenir d'espace ; on peut le remplacer au besoin par un point ou un trait souligné. C'est une bonne pratique de garder le nom des objets court et informatif (éviter toto1, toto2) et d'éviter d'attribuer des noms déja utilisés par des fonctions [2]. Pour reprendre à l'analogie des contenants dans un réfrigérateur, nous avons créé un contenant qui stocke la

1. Un groupe folk alternatif apprécié par l'auteur. Pour plus d'informations, voir http://fr.wikipedia.org/wiki/16_Horsepower.

2. Généralement, choisir un nom qui est déja utilisé par une fonction de R ne causera pas de problèmes, mais c'est préférable d'éviter cette situation.

valeur 10, et avons placé l'étiquette Nbre.pistes sur ce contenant afin de pouvoir facilement l'identifier par la suite.

Pour visualiser le contenu de l'objet, nous pouvons simplement saisir son nom dans la console (ou, préférablement, l'envoyer à R à partir de notre éditeur).

```
> Nbre.pistes
[1] 10
```

Remarquons aussi que R fait la distinction les minuscules et les majuscules. Si nous interrogeons R à propos de l'objet nbre.Pistes :

```
> nbre.Pistes
Erreur : objet 'nbre.Pistes' introuvable
```

il ne reconnaît pas l'objet. À noter aussi qu'on peut créer un objet en lui assignant les valeurs d'un autre objet. Par exemple, si nous désirons créer un nouvel objet qui prendra la même valeur que Nbre.pistes, nous pouvons écrire :

```
> le.nombre.de.pistes <- Nbre.pistes
> Nbre.pistes #l'objet original
> le.nombre.de.pistes #le nouvel objet créé
```

Si au lieu d'assigner une seule valeur à l'objet, nous désirons stocker le nombre de pistes de toute la discographie du groupe 16 Horsepower [3], nous pouvons procéder ainsi :

```
> Nbre.pistes <- c(6, 14, 14, 11, 11, 10, 20)
```

L'opérateur d'assignation <- indique que nous créons un objet, ici une variable numérique consituée des 7 valeurs à droite de l'opérateur. La fonction c( ) est une fonction qui combine les valeurs en un vecteur. Ces valeurs doivent être séparées par des virgules. La fonction c( ) est très souvent utilisée dans R pour combiner des éléments.

Maintenant, observons ce que R retourne avec la commande Nbre.pistes

3. Pour les albums lancés en Amérique du Nord avant la désintégration du groupe en 2005.

```
> Nbre.pistes
[1]  6 14 14 11 11 10 20
```

Nous voyons le nombre de pistes de chaque album : le vecteur initial a été remplacé par les nouvelles valeurs que nous avons spécifiées. On remarque que les valeurs sont précédées par `[1]`. Le chiffre entre `[ ]` donne l'indice de l'observation qui la succède (la valeur 6 est la première). Pour des vecteurs beaucoup plus longs, `R` retournera l'indice à chaque nouvelle ligne.

Le même principe s'applique pour créer une chaîne de caractères. Dans le même ordre d'idées, nous pourrions créer un objet pour stocker les noms des albums de 16 Horsepower auxquels nous faisions référence dans `Nbre.pistes`.

```
> Albums <- c("16 Horsepower", "Sackcloth 'n' Ashes", "Low Estate",
              "Secret South", "Hoarse", "Folklore", "Olden")
> Albums
[1] "16 Horsepower"       "Sackcloth 'n' Ashes"
[3] "Low Estate"          "Secret South"
[5] "Hoarse"              "Folklore"
[7] "Olden"
```

5.2 Sélectionner des valeurs

On peut sélectionner certaines valeurs à partir d'un vecteur à l'aide de `[ ]`. Par exemple, pour choisir la deuxième valeur du vecteur `Nbre.pistes`, on peut écrire :

```
> Nbre.pistes[2]
[1] 14
```

Pour extraire les trois premières valeurs du vecteur, on peut utiliser :

```
> Nbre.pistes[1:3]
[1]  6 14 14
```

L'opérateur `:` crée une séquence, ici de 1 à 3. On peut aussi sélectionner des valeurs non-consécutives en utilisant la fonction `c( )` :

```
> Nbre.pistes[c(1, 3, 5)]
[1]  6 14 11
```

Dans l'exemple ci-haut, on a extrait les première, troisième et cinquième valeurs du vecteur. Il est aussi possible d'exclure des valeurs en précédant les indices du symbole `-` à l'intérieur des `[ ]`.

```
> Nbre.pistes[-1]
[1] 14 14 11 11 10 20
> Nbre.pistes[-c(1,3,6)]
[1] 14 11 11 20
```

5.3 Répéter des valeurs

La fonction `rep( )` permet de répéter une ou plusieurs valeurs. Par exemple, pour répéter la valeur 5 vingt fois :

```
> y20 <- rep(x = 5, times = 20)
> y20
 [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

La fonction `rep( )` inclut deux arguments : `x` spécifie la valeur à répéter, tandis que `times` indique le nombre de répétitions. On peut aussi répéter des caractères.

```
> y20.caracteres <- rep(x = c("neuf", "vieux"), times = 20)
> y20.caracteres
 [1] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
 [8] "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux"
[15] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
[22] "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux"
[29] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
[36] "vieux" "neuf"  "vieux" "neuf"  "vieux"
```

Plusieurs fonctions de base peuvent être appliquées à une multitude de scénarios et c'est l'un des attraits de la programmation en `R`. Ici, nous avons utilisé à nouveau la fonction `c( )` pour créer un vecteur à 2 valeurs, `neuf` et `vieux`, et nous avons répété ce vecteur 20 fois. Il est donc important d'apprendre quelques fonctions de base qui reviendront souvent dans le cours.

5.4 Créer une série

La création d'une série ou séquence de valeurs avec un pas spécifique est réalisée avec la fonction `seq( )`.

```
> y.seq <- seq(from = 1, to = 10, by = 1)
> y.seq
 [1]  1  2  3  4  5  6  7  8  9 10
```

Les arguments de la fonction sont assez explicites : `from` indique le départ de la série, `to` indique la fin, et `by` spécifie le pas. On peut facilement créer une séquence qui décline.

```
> y.seq.decl <- seq(from = 10, to = 1, by = -1)
```

La valeur de l'argument `by` n'est pas limitée aux entiers.

```
> y.seq.01 <- seq(from = 1, to = 5, by = 0.1)
> y.seq.01
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
[15] 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
[29] 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0
```

La fonction `seq( )` possède également un argument `length.out` qui permet d'obtenir le nombre de valeurs sans indiquer le pas.

```
> y.seq.length <- seq(from = 1, to = 100, length.out = 12)
> y.seq.length
 [1]   1  10  19  28  37  46  55  64  73  82  91 100
```

5.5 Connaître les caractéristiques d'un vecteur

Afin de s'informer sur la nature d'un objet, on peut utiliser la fonction `class( )`.

```
> class(Nbre.pistes)
[1] "numeric"
> class(Albums)
[1] "character"
```

On remarque que `R` identifie `Nbre.pistes` comme un vecteur numérique alors que `Albums` est identifié comme un vecteur de caractères. `R` possède aussi plusieurs fonctions logiques nous dévoilant les caractéristiques d'un vecteur.

```
> is.numeric(Nbre.pistes)
[1] TRUE
> is.character(Albums)
[1] TRUE
```

La fonction `length( )` est particulièrement utile pour déterminer le nombre d'observations.

```
> length(Nbre.pistes)
[1] 7
> length(y.seq.01)
[1] 41
```

6 OPÉRATEURS MATHÉMATIQUES

R utilise les opérateurs mathématiques conventionnels tels que l'addition (+), la soustraction (-), la multiplication (*) et la division (/) et respecte les conventions de priorité des opérations. Ainsi, dans une équation, la multiplication et la division ont priorité sur l'addition et la soustraction.

```
> 30 + 10 - 2 * 3
[1] 34
```

Ceci étant dit, c'est une bonne idée de regrouper certains termes lorsque les calculs sont particulièrement longs pour éviter les problèmes. Lors d'opérations mathématiques impliquant deux vecteurs de différentes tailles, les valeurs du vecteur le plus court sont recyclées ou réutilisées. Ce comportement est un aspect important de la programmation en `R`.

```
> Nbre.pistes + Nbre.pistes
[1] 12 28 28 22 22 20 40
```

Les deux vecteurs ci-dessus sont de la même taille et l'opération s'effectue un élément à la fois. Toutefois, dans l'exemple suivant, on calcule la différence entre `Nbre.pistes` (7 valeurs) et 24 (1 valeurs).

```
> Nbre.pistes - 24
```

```
[1] -18 -10 -10 -13 -13 -14  -4
```

Ici, la valeur 24 est recyclée et est soustraite de chaque valeur du vecteur `Nbre.pistes`. On observe le même comportement pour la multiplication de `Nbre.pistes` et `y.seq` créé plus tôt.

```
> Nbre.pistes * y.seq
 [1]   6  28  42  44  55  60 140  48 126 140
```

Puisque `Nbre.pistes` comporte 7 valeurs et que `y.seq` en compte 10, les premières valeurs de `Nbre.pistes` seront réutilisées avec les dernières valeurs de `y.seq` pour compléter le calcul. On remarque le message d'avertissement de `R` à propos de l'objet le plus long qui n'est pas un multiple de l'objet le plus court (10 n'est pas un mutiple de 7, mais est un multiple de 5 et de 2).

Le tableau 1 montre plusieurs fonctions mathématiques de base. Les fonctions trigonométriques sont également disponibles en `R` (tableau 2).

7 TESTS LOGIQUES

Une série d'opérateurs logiques sont disponibles en `R`, le résultat étant `TRUE` ou `FALSE`. Les opérateurs logiques les plus intuitifs sont `<`, `<=`, `>`, `>=`. L'opérateur `==` teste si deux éléments sont exactement égaux, alors que `!=` teste si deux éléments sont différents. Pour comparer deux vecteurs de valeurs logiques, on peut utiliser l'opérateur `&` qui déterminera l'intersection des deux vecteurs, c'est-à-dire, les cas où les tests sont vrais pour les deux simultanément. L'opérateur `|` (ou logique) permet de déterminer l'union des deux vecteurs (les cas où les tests sont vrais pour l'un ou l'autre).

```
> Nbre.pistes
[1]  6 14 14 11 11 10 20
```

Tableau 1 – Fonctions mathématiques de base courantes en R.

`min(x)`	valeur minimale parmi les éléments de `x`
`max(x)`	valeur maximale parmi les éléments de `x`
`range(x)`	étendue des éléments de `x`
`rank(x)`	le rang des éléments de `x` (ième observation en ordre croissant)
`round(x, digits)`	arrondit les éléments de `x` à `digits` décimales
`sum(x)`	somme des éléments de `x`
`cumsum(x)`	renvoie le vecteur dont le ième élément est la somme de `x[1]` à `x[i]`
`prod(x)`	produit des éléments de `x`
`cumprod(x)`	renvoie le vecteur dont le ième élément est le produit de `x[1]` à `x[i]`
`abs(x)`	valeur absolue des éléments de `x`
`sqrt(x)`	racine carrée des éléments de `x`
`log2(x)`	logarithme en base 2 des éléments de `x`
`log10(x)`	logarithme en base 10 des éléments de `x`
`log(x, base)`	logarithme en base `base` des éléments de `x`
`log(x)`	logarithme naturel des éléments de `x` équivaut à `log(x, base = exp(1))`
`exp(x)`	renvoie la valeur de `e` élevée à la puissance `x`
`x^(y)`	la valeur de `x` élévée à la puissance `y`

Tableau 2 – Fonctions trigonométriques courantes en R.

`cos(x)`	cosinus en radians des éléments de `x`
`sin(x)`	sinus en radians des éléments de `x`
`tan(x)`	tangente en radians des éléments de `x`
`acos(x)`	arc-cosinus en radians des éléments de `x`
`asin(x)`	arc-sinus en radians des éléments de `x`
`atan(x)`	arc-tangente en radians des éléments de `x`

```
> Nbre.pistes >= 10
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> Nbre.pistes != 10
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
> Albums == "Olden" #spécifier les caractères entre " "
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
> vec.log1 <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
> vec.log2 <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
> vec.log1 & vec.log2 #deux tests vrais pour la valeur 1
[1]  TRUE FALSE FALSE FALSE FALSE
> vec.log1 | vec.log2 #test vrai pour valeurs 1, 2, 3, 4
[1]  TRUE  TRUE  TRUE  TRUE FALSE
```

Les fonctions `all( )` et `any( )` déterminent si toutes ou au moins l'une des valeurs répondent au test logique, respectivement. Pour comparer deux vecteurs entre eux, on utilisera plutôt les fonctions `all.equal( )` ou `identical( )`.

```
> any(Nbre.pistes > 11)
[1] TRUE
> all(Nbre.pistes == 11)
[1] FALSE
> all.equal(Nbre.pistes, Nbre.pistes)
[1] TRUE
> Nbre <- rep(10, 7) #on crée un autre vecteur
> Nbre
[1] 10 10 10 10 10 10 10
> all.equal(Nbre.pistes, Nbre)
```

```
[1] "Mean relative difference: 0.31579"
> identical(Nbre.pistes, Nbre)
[1] FALSE
```

Le dernier appel à `all.equal( )` nous indique que les deux vecteurs ne sont pas identiques puisqu'il y a une différence moyenne entre les deux de 0.31579. La fonction `which( )` est aussi apparentée aux tests logiques. Elle permet de déterminer lesquelles des valeurs répondent à une certaine condition énoncée par un test logique. Les variantes `which.min( )` et `which.max( )` permettent d'identifier les valeurs minimales et maximales, respectivement.

```
> which(Nbre.pistes > 11) #retourne les indices
[1] 2 3 7
> Nbre.pistes[which(Nbre.pistes > 11)] #retourne les valeurs
[1] 14 14 20
> which(Albums == "Folklore") #retourne les indices
[1] 6
> which.max(Nbre.pistes) #indice de l'observation maximale
[1] 7
> which.min(Nbre.pistes) #indice de l'observation minimale
[1] 1
```

La fonction `ifelse( )` permet de faire un test conditionnel sur un vecteur. Par exemple, pour créer une variable binaire à partir de la variable `Nbre.pistes` (c.-à-d., `< = 10` ou `> 10`), on procéderait ainsi :

```
> Bin10 <- ifelse(Nbre.pistes <= 10, 0, 1)
> Bin10
[1] 0 1 1 1 1 0 1
```

On procède de la même façon si le vecteur contient des caractères :

```
> y20.caracteres
 [1] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
 [8] "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux"
[15] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
[22] "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux"
[29] "neuf"  "vieux" "neuf"  "vieux" "neuf"  "vieux" "neuf"
[36] "vieux" "neuf"  "vieux" "neuf"  "vieux"
> bin.y20 <- ifelse(y20.caracteres == "neuf", "nouveau", "ancien")
> bin.y20[1:5]
[1] "nouveau" "ancien"  "nouveau" "ancien"  "nouveau"
```

La fonction `ifelse( )` est aussi pratique pour créer des classes (i.e., une variable catégorique) en imbriquant plusieurs tests sur le vecteur :

```
> Classe.pistes <- ifelse(Nbre.pistes < 10, "< 10",
                          ifelse(Nbre.pistes >= 10 & Nbre.pistes < 15,
                                 "10 - 15", "> 15"))
```

Le code ci-dessus est réparti sur plusieurs lignes pour en améliorer la lisibilité, mais nous aurions obtenu exactement le même résultat si tout le code s'était trouvé sur une même ligne. Lorsqu'on saisit une commande sur plusieurs lignes, `R` attend la parenthèse de fermeture avant de réaliser l'opération. On débute en déterminant si `Nbre.pistes < 10`, et on attribue la valeur `"< 10"` le cas échéant. Sinon, on teste si `Nbre.pistes >= 10` mais `< 15`, et, si c'est le cas, on donne la valeur de `"10-15"` au vecteur. Si ce teste est faux, on attribue une valeur de `"> 15"` au vecteur.

On peut utiliser plusieurs des opérateurs logiques pour faire des tests de conditions pour décider de l'exécution d'autres opérations. Par exemple, si le test est vrai, on effectue une opération particulière, et, si le test est faux, on effectue une opération différente. L'énoncé `if`

appelle un test de condition spécifié entre parenthèses. L'opération à exécuter le cas échéant sera indiquée entre `{ }`.

Dans l'exemple ci-dessous, on teste si l'objet contient une valeur numérique ou une chaîne de caractères. Si la valeur est une chaîne de caractères, on imprime à l'écran `"des caractères"` à l'aide de la fonction `paste( )`. Pour une valeur numérique, le test imprimera `"pas des caractères"` à l'écran.

```
> valeur <- 200 #une valeur numérique
> if(is.character(valeur)) {paste("des caractères")
  } else {paste("pas des caractères")}
[1] "pas des caractères"
> valeur <- "du texte" #une chaîne de caractères
> if(is.character(valeur)) {paste("des caractères")
  } else {paste("pas des caractères")}
[1] "des caractères"
```

On remarque que, si le test est faux, l'opération à effectuer doit être spécifiée après l'énoncé `else` et doit être insérée entre `{ }`. Il est possible d'imbriquer les tests de conditions. Pour les vecteurs plus haut, nous avons utilisé `&` et `|` et le comportement était similaire à des opérateurs mathématiques : le calcul était exécuté pour chaque paire d'éléments des deux vecteurs comparés. Les homologues `&&` et `||` ont un usage légèrement différent. En effet, lorsqu'on utilise `&&`, le test à droite s'effectue sur le premier élément de l'objet seulement si le test sur le premier élément de gauche est vrai. Avec `||`, le test à droite de l'opérateur sera évalué sur le premier élément seulement si le test de gauche sur le premier élément est faux.

```
> des.valeurs <- 1:10
> des.valeurs
 [1]  1  2  3  4  5  6  7  8  9 10
> des.valeurs > 0
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> des.valeurs < 0
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE
> des.valeurs > 0 & des.valeurs < 0
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE
> des.valeurs > 0 | des.valeurs < 0
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> des.valeurs > 0 && des.valeurs < 0
[1] FALSE
> des.valeurs > 0 || des.valeurs < 0
[1] TRUE
```

Nous revisiterons les tests logiques à la section *Créer des sous-ensembles* pour créer des sous-ensembles de jeux de données.

8 MATRICES ET CALCULS MATRICIELS

8.1 Créer une matrice

La création et la manipulation de matrices sont simples dans `R`. La matrice est un autre type d'objet. Elle contient des éléments qui sont soit numériques ou des caractères, mais pas un mélange des deux. Par exemple, si la matrice contient au moins un seul caractère, tous les éléments seront convertis en caractères. Créons une matrice simple à l'aide de `matrix( )` constituée d'une séquence de 1 à 9 avec un pas de 1 :

```
> mat <- matrix(data = 1:12, nrow = 3, ncol = 4)
> mat
```

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

Les arguments `nrow` et `ncol` indiquent les dimensions de la matrice que nous désirons obtenir (une matrice de 3 rangées et de 4 colonnes). On remarque que les éléments sont insérés dans la matrice par colonnes : la première colonne est remplie, ensuite la deuxième et la troisième. L'argument `byrow` permet de changer la manière d'insérer les éléments dans la matrice.

```
> mat<-matrix(data = 1:12, nrow = 3, ncol = 4, byrow = TRUE)
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

On obtient une nouvelle matrice dans laquelle les valeurs ont été insérées une rangée à la fois. Par défaut, l'argument `byrow` prend la valeur `FALSE`.

8.2 Sélectionner des valeurs

Tout comme pour le vecteur, on utilise `[ ]` pour extraire des éléments d'une matrice. Toutefois, on devra spécifier les deux dimensions `[rangée, colonne]`. Par exemple,

```
> mat[1, 3]
[1] 3
> mat[1, ]
[1] 1 2 3 4
> mat[, 2]
```

```
[1]  2  6 10
> mat[1:2, 1]
[1] 1 5
> mat[c(1, 3), 3]
[1]  3 11
```

À noter que si on spécifie seulement l'indice de la rangée, toute la rangée sera extraite. De la même façon, si seul l'indice de la colonne est spécifié, toute la colonne sera extraite.

8.3 Connaître les caractéristiques d'une matrice

On peut vérifier les caractéristiques de la matrice à l'aide de la fonction `class` ou de tests logiques.

```
> class(mat)
[1] "matrix"
> is.numeric(mat)
[1] TRUE
> is.vector(mat)
[1] FALSE
```

La fonction `length( )` permet de déterminer le nombre d'éléments de la matrice. On peut utiliser `dim( )`, `nrow( )` ou `ncol( )` pour connaître les dimension de la matrice (rangée, colonne).

```
> length(mat)
[1] 12
> dim(mat)
[1] 3 4
```

```
> nrow(mat)
[1] 3
> ncol(mat)
[1] 4
```

Il est possible d'ajouter des étiquettes aux colonnes et aux rangées à l'aide de `colnames( )` et `rownames( )`, respectivement.

```
> colnames(mat)
NULL
> colnames(mat) <- c("C1", "C2", "C3", "C4")
> colnames(mat)
[1] "C1" "C2" "C3" "C4"
> rownames(mat) <- c("R1", "R2", "R3")
> mat
   C1 C2 C3 C4
R1  1  2  3  4
R2  5  6  7  8
R3  9 10 11 12
```

8.4 Opérations et manipulation de matrices

Plusieurs fonctions agissent sur les matrices. Par exemple, `diag( )` permet d'extraire la diagonale de la matrice.

```
> diag(mat)
[1]  1  6 11
```

On peut facilement transposer une matrice.

```
> t(mat)
   R1 R2 R3
C1  1  5  9
C2  2  6 10
C3  3  7 11
C4  4  8 12
```

Afin d'obtenir la somme ou la moyenne des colonnes d'une matrice, on peut utiliser les fonctions `colSums( )` et `colMeans( )`, respectivement. Il existe des fonctions similaires pour les rangées d'une matrice.

```
> colSums(mat)
C1 C2 C3 C4
15 18 21 24
> colMeans(mat)
C1 C2 C3 C4
 5  6  7  8
> rowSums(mat)
R1 R2 R3
10 26 42
> rowMeans(mat)
  R1   R2   R3
 2.5  6.5 10.5
```

Pour multiplier des matrices, il faut mettre l'opérateur de multiplication entre des signes de pourcentage :

```
> vecteur <- c(2, 4, 5)
> vecteur%*%mat
```

```
     C1 C2 C3  C4
[1,] 67 78 89 100
```

9 JEUX DE DONNÉES

Un des types d'objet les plus utilisés dans `R` est le jeu de données ou `data.frame`. L'objet `data.frame` a deux dimensions et peut être constitué de colonnes de différents types. On peut retrouver des colonnes de valeurs numériques, de valeurs logiques, d'entiers, de caractères ou de facteurs (`factor`).

9.1 Créer un jeu de données

La fonction `data.frame( )` permet de créer un jeu de données. On peut combiner une série de vecteurs en un `data.frame`.

```
> Temps <- c(1.2, 3.4, 2.1, 5.5) #vecteur de temps
> Masse <- c(2.5, 4.2, 5.6, 3.4) #vecteur de masse
> Sexe <- c("mâle", "femelle", "mâle", "femelle") #vecteur de caractères
> jeu <- data.frame(Temps, Masse, Sexe) #création du data.frame
> jeu
  Temps Masse    Sexe
1   1.2   2.5    mâle
2   3.4   4.2 femelle
3   2.1   5.6    mâle
4   5.5   3.4 femelle
```

On peut bien sûr créer un jeu de données en une seule étape :

```
> autos <- data.frame(Vitesse = c(25, 40, 70, 100),
                      Type = c("auto", "camion", "auto", "camion"))
> autos
  Vitesse   Type
1      25   auto
2      40 camion
3      70   auto
4     100 camion
```

9.2 Connaître les caractéristiques de jeu de données

Plusieurs des fonctions que nous avons déjà présentées permettent de décrire un jeu de données. Notons les fonctions `class( )`, `length( )`, `dim( )`, `nrow( )`, `ncol( )`, `names( )`, `rownames( )`, `colnames( )`. Nous pouvons aussi utiliser la fonction `str( )` pour connaître la structure du jeu de données, incluant l'identification de chacune des variables, son type, ainsi que les premières observations.

```
> str(jeu)
'data.frame':        4 obs. of  3 variables:
 $ Temps: num  1.2 3.4 2.1 5.5
 $ Masse: num  2.5 4.2 5.6 3.4
 $ Sexe : Factor w/ 2 levels "femelle","mâle": 2 1 2 1
```

Remarquons que la variable `Sexe` a été créée comme vecteur de caractères et est maintenant reconnue comme facteur (`factor`) une fois ajoutée à l'objet de type `data.frame`. Plusieurs fonctions sont disponibles pour transformer un objet d'un type à un autre. Mentionnons ici `as.factor( )` qui permet de convertir un vecteur en variable catégorique. La fonction `summary( )`, quant à elle, nous informe sur les statistiques descriptives des variables du jeu de données.

```
> summary(jeu)
     Temps           Masse           Sexe
 Min.   :1.20   Min.   :2.50   femelle:2
 1st Qu.:1.88   1st Qu.:3.17   mâle   :2
 Median :2.75   Median :3.80
 Mean   :3.05   Mean   :3.92
 3rd Qu.:3.92   3rd Qu.:4.55
 Max.   :5.50   Max.   :5.60
```

D'autres fonctions, comme `head( )` ou `tail( )`, permettent d'afficher les six premières ou six dernière lignes du jeu de données. Utilisons ces fonctions sur un jeu de données déjà dans `R`, lequel traite de données de diamètre, de hauteur et du volume de cerisiers. Afin de déterminer les jeux de données disponibles dans `R`, il suffit d'utiliser la commande `data( )` sans valeurs entre les parenthèses. Afin de charger le jeu de données de notre choix, il suffit d'exécuter la commande `data( )`, cette fois-ci en spécifiant le nom du jeu de données entre parenthèses.

```
> data(trees) #jeu de données déja dans R
> str(trees)
'data.frame':        31 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
> head(trees) #comparer à trees[1:6, ]
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
```

```
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
> tail(trees) #comparer à trees[(nrow(trees)-5):nrow(trees), ]
   Girth Height Volume
26  17.3     81   55.4
27  17.5     82   55.7
28  17.9     80   58.3
29  18.0     80   51.5
30  18.0     80   51.0
31  20.6     87   77.0
```

9.3 Accéder aux variables d'un jeu de données

On remarque rapidement qu'il n'est pas possible d'aller chercher une variable d'un jeu de données simplement en tapant son nom. Par exemple, la variable `Vitesse` se trouve dans le jeu de données `autos` créé plus haut.

```
> autos
  Vitesse   Type
1      25   auto
2      40 camion
3      70   auto
4     100 camion
> names(autos)
[1] "Vitesse" "Type"
```

Pourtant, on obtient un message d'erreur si on spécifie `Vitesse`. L'objet `autos` est un jeu de données et les variables en sont des éléments. Il existe différentes stratégies pour extraire

des éléments d'un jeu de données. La première, et celle qui est la plus polyvalente, consiste à utiliser l'opérateur `$`.

```
> Vitesse

Erreur : objet 'Vitesse' introuvable

> autos$Vitesse
[1]  25  40  70 100
```

Nous spécifions le jeu de données suivi de l'opérateur `$`, lui-même suivi du nom de la variable d'intérêt : `Vitesse` est un élément de `autos`. Nous utiliserons la même stratégie pour extraire certains éléments d'objets plus complexes, tels que la sortie (*output*) d'une analyse statistique.

Comme seconde stratégie, on peut utiliser `[ , ]` pour extraire les éléments du jeu de données, puisqu'il a deux dimensions comme la matrice.

```
> autos[, 1]
[1]  25  40  70 100
```

Une variante de l'utilisation de `[ , ]` consiste à spécifier le nom de la variable.

```
> autos[, "Vitesse"]
[1]  25  40  70 100
```

9.4 Importer des fichiers de données

Au lieu d'entrer à la main les données dans `R`, nous allons généralement importer les données sauvegardées dans un fichier. Il est facile d'importer des fichiers au format texte brut en `R` en suivant quelques lignes directrices. Il existe une multitude de moyens pour importer des fichiers dans `R`, mais nous n'en présenterons que quelques-uns.

Tableau 3 – Données récoltées lors d'une expérience agricole en format « large ».

Témoin	Engrais A	Engrais B	Engrais C
6.1	6.3	7.1	8.1
5.9	6.2	8.2	8.5
5.8	5.8	7.3	7.6
5.4	6.3	6.9	7.8

Tableau 4 – Données récoltées lors d'une expérience agricole en format "long".

Réponse	Traitement
6.1	Témoin
5.9	Témoin
5.8	Témoin
5.4	Témoin
6.3	EngraisA
6.2	EngraisA
5.8	EngraisA
6.3	EngraisA
...	

Bien que le fichier doive être en format texte, l'extension peut être celle de votre choix. Pour faciliter l'analyse de jeux de données, nous recommandons de structurer vos fichiers en format « long ». Dans ce format, toutes les valeurs d'une même variable se trouvent sous une même colonne. Par exemple, dans une expérience agricole, on pourrait présenter les données dans un tableau en format « large » (tableau 3).

Bien que le format « large » permette de visualiser rapidement les données, il n'est pas celui qui est généralement utilisable par plusieurs fonctions d'analyses statistiques. En effet, le format le plus souvent requis est le format « long ». Le tableau 4 présente le même jeu de données en ce format.

Une fois le fichier en format « long », on peut sauvegarder en texte avec des séparateur tabulations ou espace à partir du chiffrier de notre choix. Afin de rendre l'importation plus agréable et plus facile, il est important de respecter les points suivants :

- réserver la première ligne de chaque colonne pour le nom de la colonne ;

- éviter les espaces dans le nom d'une colonne donnée (p. ex., ne pas écrire hauteur moyenne, mais plutôt hauteur_moyenne, hauteur.moyenne, hauteurmoyenne) ;
- utiliser `NA` pour indiquer les valeurs manquantes à l'intérieur du jeu de données : aucune cellule ne peut être vide ;
- éviter les accents dans les noms des variables ou dans les données.

La fonction principale d'importation de jeux de données est `read.table( )`. Cette fonction comporte un argument `file =`, avec lequel on doit spécifier le chemin complet du fichier. L'argument logique `header` doit prendre la valeur `TRUE`. Pour réussir l'importation, il faut :

- spécifier le bon chemin en se souciant des majuscules et des minuscules ;
- utiliser `/` ou \\ mais pas le \ de MS-Windows ;
- terminer le chemin complet par le nom du fichier et mettre le tout entre `" "` ;
- stocker le jeu de données dans un objet, sinon il ne sera imprimé qu'à l'écran ;
- remplacer les virgules par des points pour indiquer la décimale des valeurs numériques avant d'importer le fichier dans `R` ou spécifier l'argument `dec = ","` de la fonction `read.table( )`. Cette démarche est nécessaire puisque R utilise le point indiquant les décimales.

Bien que `R` reconnaisse généralement sans difficulté les caractères accentués (p. ex., é, à, ê), il peut y avoir un problème lorsque le fichier est créé dans un système d'exploitation et importé dans `R` dans un autre. Dans de tels cas, les caractères accentués ne seront pas reconnus par défaut. Par exemple, si vous travaillez sous Windows et que vous désirez qu'un collègue opérant sous Mac importe un de vos fichiers de données dans `R`, il peut survenir des problèmes d'importation en présence d'accents dans le fichier. Certains arguments de `read.table( )` permettent de modifier ceci (`encoding =`), mais c'est plus simple d'éviter les accents.

Illustrons avec le jeu de données `vers` qui traite de l'abondance de vers de terre dans différents types de parcs publics. Voici trois stratégies équivalentes pour importer un fichier en format texte et qui respecte les consignes mentionnées plus haut :

- Pour la première, on spécifie le chemin complet sur l'ordinateur où se trouve le fichier.

Par exemple, pour importer un jeu de données stocké dans le fichier `vers.txt`, on pourrait envoyer la commande

```
> vers <- read.table(file = "C:/chemin_complet/vers.txt",
                     header = TRUE)
```

- La deuxième variation consiste à spécifier le répertoire où se trouve le fichier à l'aide de la fonction `setwd( )`.

```
> setwd(file = "C:/chemin_complet/")
> vers <- read.table(file = "vers.txt", header = TRUE)
```

- Une troisième stratégie s'avère l'utilisation de la fonction `file.choose( )`. Cette dernière permet de sélectionner le fichier à l'aide d'une fenêtre graphique. Il faut toutefois savoir où se trouve le fichier ...

```
> vers <- read.table(file = file.choose( ), header = TRUE)
```

Si vous ne respectez pas ces consignes, vous risquez de générer un message d'erreur. Vous devrez retracer l'erreur ou alors décortiquer plus en détail les pages d'aide des fonctions d'importation afin de modifier les arguments qui reflètent votre situation. Les fonctions `read.delim( )`, `read.delim2( )`, `read.csv( )`, `read.csv2( )` sont des versions de `read.table( )` avec différentes valeurs par défaut. Il est aussi possible d'importer directement des fichiers en format `MS-Excel` ou `MS-Access` à l'aide de banques de fonctions téléchargeables. Il suffit d'utiliser `RSiteSearch(" ")` avec le sujet de votre requête entre " " pour obtenir plus d'informations.

9.5 Ajouter une variable

Il est facile d'ajouter une variable à un jeu de données. Pour ce faire, on peut utiliser l'opérateur `$` afin d'indiquer à `R` l'objet qui stocke le jeu de données et le nom de la nouvelle variable que l'on veut créer :

```
> head(vers) #affiche les 6 premières observations
> vers$log.Superficie <- log(vers$Superficie) #créer une nouvelle variable
```

```
> head(vers) #inclut maintenant la nouvelle variable
```

La nouvelle variable sera ajoutée dans une colonne à la suite des autres variables du jeu de données.

9.6 Créer des sous-ensembles

Nous pouvons utiliser différentes approches afin de sélectionner des sous-ensembles d'un jeu de données. Nous avons déja vu l'extraction à l'aide de `[ , ]`. Par exemple, dans le jeu de données `trees`, on peut extraire les 15 premières observations des variables `Girth` et `Height`.

```
> trees[1:15, c("Girth", "Height")]
   Girth Height
1    8.3     70
2    8.6     65
3    8.8     63
4   10.5     72
5   10.7     81
6   10.8     83
7   11.0     66
8   11.0     75
9   11.1     80
10  11.2     75
11  11.3     79
12  11.4     76
13  11.4     76
14  11.7     69
15  12.0     75
```

Les tests logiques s'avèrent aussi d'excellents outils pour créer des sous-ensembles de jeux de données. Par exemple, on peut créer un sous-ensemble de `trees` pour lequel `Girth` > 12 ou de `vers` pour le type `Vegetation` correspondant à `Prairie`.

```
> trees22 <- trees[trees$Girth >= 12, ]
```

```
> vers.prairie <- vers[vers$Vegetation == "Prairie", ]
```

Il est aussi possible de combiner des tests logiques. Créons, par exemple, un jeu de données à partir de `vers` pour lequel `Vegetation` correspond à `Prairie` et `Pente` > 5 en utilisant l'opérateur `&` (intersection).

```
> versb <- vers[vers$Vegetation == "Prairie" & vers$Pente > 5, ]
```

Comparons le résultat de l'union des deux conditions avec l'opérateur `|`.

```
> versc <- vers[vers$Vegetation == "Prairie" | vers$Pente > 5, ]
```

La fonction `subset( )` utilise les tests logiques pour créer un sous-ensemble d'un jeu de données. Cette fonction est une alternative à la sélection avec `[, ]` combinée à un test logique et donnera exactement le même résultat que dans l'exemple précédent.

```
> vers.les.prairie <- subset(x = vers,
                             subset = vers$Vegetation == "Prairie")
> identical(vers.les.prairie, vers.prairie)
[1] TRUE
> versbb <- subset(x = vers,
                   subset = vers$Vegetation == "Prairie" & vers$Pente > 5)
> identical(versb, versbb)
[1] TRUE
```

9.7 Effectuer un tri

Les tris dans R comportent quelques subtilités : la fonction à utiliser dépend de ce que l'on veut trier. Pour faire le tri en ordre croissant d'une seule variable ou vecteur, on peut utiliser `sort( )`. Cette fonction comporte un argument `decreasing = FALSE` qui peut prendre la valeur `FALSE` pour un tri en ordre décroissant. Comme alternative à l'argument `decreasing = TRUE`, la fonction `rev( )` permet de faire un tri en ordre décroissant.

```
> sort(vers$Superficie) #en ordre croissant
 [1] 0.8 1.5 1.8 1.9 2.1 2.2 2.4 2.8 2.9 2.9 3.1 3.3 3.5 3.6
[15] 3.7 3.8 3.9 4.1 4.4 5.1
> sort(vers$Superficie, decreasing = TRUE) #en ordre décroissant
 [1] 5.1 4.4 4.1 3.9 3.8 3.7 3.6 3.5 3.3 3.1 2.9 2.9 2.8 2.4
[15] 2.2 2.1 1.9 1.8 1.5 0.8
> rev(sort(vers$Superficie)) #en ordre décroissant
 [1] 5.1 4.4 4.1 3.9 3.8 3.7 3.6 3.5 3.3 3.1 2.9 2.9 2.8 2.4
[15] 2.2 2.1 1.9 1.8 1.5 0.8
```

Comme toute autre manipulation dans R, si on veut que le tri prenne effet, il faut assigner ce tri à un objet. Autrement, le résultat s'affichera à l'écran et disparaîtra à tout jamais sans être disponible plus tard dans la session de travail.

Afin de trier un jeu de données, on doit plutôt utiliser `order( )`. Par exemple, pour trier le jeu de données `vers` en ordre croissant selon la variable `Pente`, on procéde ainsi :

```
> ##tri en ordre croissant selon Pente
> vers.ord <- vers[order(vers$Pente), ]
> ##tri en ordre décroissant selon Pente
> vers.ord.dec <- vers[order(vers$Pente, decreasing = TRUE), ]
> ##autre moyen d'obtenir un tri décroissant avec sort( )
> vers.ord.dec2 <- vers[rev(order(vers$Pente)), ]
```

Il est facile d'effectuer un tri pour seulement une partie du jeu de données en spécifiant les colonnes désirées. Une autre option est de faire un tri selon plusieurs variables.

```
> ##tri des variables 1, 2 et 4 selon Pente
> vers.ordb <- vers[order(vers$Pente), c(1,2,4)]
> ##tri du jeu de données selon Pente ET Superficie
> vers.ord2 <- vers[order(vers$Pente, vers$Superficie) , ]
```

9.8 Créer des tableaux récapitulatifs des fréquences

Quelques fonctions sont très pratiques pour réaliser des résumés d'un jeu de données selon quelques variables. Notamment, pour obtenir l'équivalent du « tableau croisé dynamique » de `MS EXCEL` ou du « pilote de données » de `Calc` de `OpenOffice`, la fonction `table( )` s'avère fort utile. Cette fonction résume la fréquence à laquelle chaque valeur d'une variable apparaît dans un jeu de données dans un tableau, lequel est un autre type d'objet en `R`.

```
> table(vers$Vegetation) #la fréquence de chaque type de végétation

   Arable Chaparral   Prairie       Pre    Verger
        3         4         9         3         1

> table(vers$Superficie) #la fréquence de chaque observation numérique

0.8 1.5 1.8 1.9 2.1 2.2 2.4 2.8 2.9 3.1 3.3 3.5 3.6 3.7 3.8
  1   1   1   1   1   1   1   1   2   1   1   1   1   1   1
3.9 4.1 4.4 5.1
  1   1   1   1

> class(table(vers$Superficie))

[1] "table"
```

On peut également construire un tableau à deux dimensions ou de dimensions supérieures selon des variables généralement catégoriques. L'argument `deparse.level` permet d'ajouter le nom des variables aux rangées ou aux colonnes.

```
> ##tableau à deux dimensions
> table(vers$Vegetation, vers$Pente)
           0 1 2 3 4 5 6 8 10 11
  Arable   1 0 2 0 0 0 0 0  0  0
  Chaparral 1 0 0 0 0 0 0 1  2  0
  Prairie  0 2 2 2 1 0 1 0  0  1
  Pre      2 0 0 0 0 1 0 0  0  0
  Verger   1 0 0 0 0 0 0 0  0  0
> ##tableau à deux dimensions avec les noms
> table(vers$Vegetation, vers$Pente, deparse.level = 2)
              vers$Pente
vers$Vegetation 0 1 2 3 4 5 6 8 10 11
      Arable    1 0 2 0 0 0 0 0  0  0
      Chaparral 1 0 0 0 0 0 0 1  2  0
      Prairie   0 2 2 2 1 0 1 0  0  1
      Pre       2 0 0 0 0 1 0 0  0  0
      Verger    1 0 0 0 0 0 0 0  0  0
> ##tableau à trois dimensions
> table(vers$Vegetation, vers$Pente, vers$Humide)
, ,  = FALSE

           0 1 2 3 4 5 6 8 10 11
  Arable   1 0 2 0 0 0 0 0  0  0
  Chaparral 1 0 0 0 0 0 0 0  1  0
  Prairie  0 2 2 2 0 0 1 0  0  1
  Pre      0 0 0 0 0 0 0 0  0  0
```

```
  Verger    1 0 0 0 0 0 0 0  0  0

, ,  = TRUE

           0 1 2 3 4 5 6 8 10 11
  Arable    0 0 0 0 0 0 0 0  0  0
  Chaparral 0 0 0 0 0 0 0 1  1  0
  Prairie   0 0 0 0 1 0 0 0  0  0
  Pre       2 0 0 0 0 1 0 0  0  0
  Verger    0 0 0 0 0 0 0 0  0  0
```

La dernière commande présentée ci-dessus crée une série de tableaux à deux dimensions pour chaque valeur de la variable `Humide` (i.e., `TRUE`, `FALSE`).

La fonction `xtabs( )`, quant à elle, offre la possibilité de résumer une variable de fréquence qui apparaît dans un jeu de données selon une série de variables en utilisant une formule du genre `freq` $\sim$ `var1` + `var2` où la variable de fréquence apparaît à la gauche de l'équation et les variables de classification à la droite.

```
> ##on crée un jeu de données
> jeu.freq <- data.frame(Freq = c(0, 1, 0, 10, 2, 3, 5, 0),
                         Trait = rep(c("A", "B"), 4),
                         Sexe = rep(c("F", "M"), 4))
> jeu.freq
  Freq Trait Sexe
1    0     A    F
2    1     B    M
3    0     A    F
4   10     B    M
```

```
5    2     A    F
6    3     B    M
7    5     A    F
8    0     B    M
> xtabs(Freq ~ Trait + Sexe, data = jeu.freq)
     Sexe
Trait  F  M
    A  7  0
    B  0 14
```

Cette fonction possède un argument `data` qui permet de spécifier où se trouvent les variables pour lesquelles on désire un tableau. La plupart des fonctions de modélisation, comme `lm( )` pour effectuer des modèles de régression, comportent ce même argument pour faciliter la syntaxe. Si aucune variable de fréquence n'apparaît dans le jeu de données, `xtabs( )` aura le même effet que `table( )`.

```
> table(jeu.freq$Trait)
A B
4 4
> xtabs(~ Trait, data = jeu.freq)
Trait
A B
4 4
> table(jeu.freq$Trait, jeu.freq$Sexe)
    F M
  A 4 0
  B 0 4
```

```
> #comparer avec l'exemple plus haut avec Freq ~ Trait + Sexe
> xtabs(~ Trait + Sexe, data = jeu.freq)
     Sexe
Trait F M
    A 4 0
    B 0 4
```

9.9 Modifier la structure d'un jeu de données

On peut convertir un jeu de données du format « long » au format « large ». En présence de mesures répétées, la fonction `reshape( )` permet de passer d'un format à l'autre. Considérons, par exemple, un jeu de données qui traite de la concentration en indométhacine (un anti-inflammatoire) dans le plasma sanguin de patients. Le jeu de données original est en format « long ». On peut le convertir en format « large » comme suit :

```
> head(Indometh, 15) #15 premières observations - format long
   Subject time conc
1        1 0.25 1.50
2        1 0.50 0.94
3        1 0.75 0.78
4        1 1.00 0.48
5        1 1.25 0.37
6        1 2.00 0.19
7        1 3.00 0.12
8        1 4.00 0.11
9        1 5.00 0.08
10       1 6.00 0.07
11       1 8.00 0.05
```

```
12       2 0.25 2.03
13       2 0.50 1.63
14       2 0.75 0.71
15       2 1.00 0.70
> large <- reshape(data = Indometh, idvar = "Subject",
                   timevar = "time", direction = "wide") #mettre en format large
> large
   Subject conc.0.25 conc.0.5 conc.0.75 conc.1 conc.1.25
1        1      1.50     0.94      0.78   0.48      0.37
12       2      2.03     1.63      0.71   0.70      0.64
23       3      2.72     1.49      1.16   0.80      0.80
34       4      1.85     1.39      1.02   0.89      0.59
45       5      2.05     1.04      0.81   0.39      0.30
56       6      2.31     1.44      1.03   0.84      0.64
   conc.2 conc.3 conc.4 conc.5 conc.6 conc.8
1    0.19   0.12   0.11   0.08   0.07   0.05
12   0.36   0.32   0.20   0.25   0.12   0.08
23   0.39   0.22   0.12   0.11   0.08   0.08
34   0.40   0.16   0.11   0.10   0.07   0.07
45   0.23   0.13   0.11   0.08   0.10   0.06
56   0.42   0.24   0.17   0.13   0.10   0.09
```

Il est possible de passer du format « large » au format « long » avec la même fonction :

```
> long <- reshape(data = large, idvar = "Subject",
                  timevar = "Time", v.names = "Concentration",
                  varying = 2:12,
                  direction = "long") #mettre en format long
> long.ord <- long[order(long$Subject), ] #tri selon Subject
```

Dans certains cas, les valeurs d'une variable sont entrées dans différentes colonnes selon les traitements, comme dans le tableau 3. La plupart des analyses en `R` ne pouvant utiliser ce format, il faut convertir le jeu de données en format long afin d'insérer les valeurs de la variable réponse dans une seule colonne. Les fonctions `stack( )` et `unstack( )` permettent de telles manipulations.

```
> fert <- data.frame(Temoin = c(6.1, 6.3, 7.1, 8.1),
                     EngraisA = c(5.9, 6.2, 8.2, 8.5),
                     EngraisB = c(5.8, 5.8, 7.3, 7.6),
                     EngraisC = c(5.4, 6.3, 6.9, 7.8))
> fert #format large
  Temoin EngraisA EngraisB EngraisC
1    6.1      5.9      5.8      5.4
2    6.3      6.2      5.8      6.3
3    7.1      8.2      7.3      6.9
4    8.1      8.5      7.6      7.8
> long <- stack(fert) #format long - 1 colonne par variable
> long
   values      ind
1     6.1   Temoin
2     6.3   Temoin
3     7.1   Temoin
4     8.1   Temoin
5     5.9 EngraisA
6     6.2 EngraisA
7     8.2 EngraisA
8     8.5 EngraisA
9     5.8 EngraisB
```

```
10    5.8 EngraisB
11    7.3 EngraisB
12    7.6 EngraisB
13    5.4 EngraisC
14    6.3 EngraisC
15    6.9 EngraisC
16    7.8 EngraisC
> unstack(long) #remettre en format large
  EngraisA EngraisB EngraisC Temoin
1      5.9      5.8      5.4    6.1
2      6.2      5.8      6.3    6.3
3      8.2      7.3      6.9    7.1
4      8.5      7.6      7.8    8.1
```

9.10 Exporter un jeu de données

Alors que `read.table( )` permet d'importer un jeu de données à partir d'un fichier, `write.table( )` permet de sauvegarder un jeu de données (un objet `R`) dans un fichier. Tout comme son homologue `read.table( )`, elle comporte des arguments pour spécifier le chemin et le nom du fichier, le type de séparateur, l'option d'inclure ou non les en-têtes des colonnes ou les étiquettes des rangées. Illustrons avec le jeu de données `fert` créé plus haut.

```
> ##exporter à un fichier avec séparateur espace
> write.table(x = fert, file = "/chemin_sur_ordi/fert.txt",
              row.names = FALSE, col.names = TRUE, sep = " ")
```

L'argument `sep` spécifie le type de séparateur entre les valeurs du jeu de données du fichier. Les plus communes sont `sep = " "` pour un espace, `sep = ","` pour la virgule, `sep = "\t"` pour la tabulation, `sep = ";"` pour le point-virgule. Lors de l'exportation avec

write.table( ), R met le nom des en-têtes de colonnes, les étiquettes des rangées ainsi que les valeurs de variables catégoriques ou de caractères entre " ". L'argument quote prend la valeur TRUE par défaut (garde les " " dans le fichier). L'importation d'un fichier avec " " dans R se fait sans difficulté, mais peut causer des problèmes dans d'autres logiciels. On peut simplement spécifier quote = FALSE afin de remédier à la situation.

Une autre option possible est de stocker un objet avec la fonction save( ). Alors que write.table( ) permet d'exporter des fichiers de données (un objet à 2 dimensions), la fonction save( ) sauvegarde des objets beaucoup plus complexes tout en gardant leur structure intacte.

```
> ##on crée un tableau
> tableau <- table(vers$Vegetation, vers$Pente, vers$Humide)
> tableau
, ,  = FALSE


           0 1 2 3 4 5 6 8 10 11
  Arable   1 0 2 0 0 0 0 0  0  0
  Chaparral 1 0 0 0 0 0 0 0  1  0
  Prairie  0 2 2 2 0 0 1 0  0  1
  Pre      0 0 0 0 0 0 0 0  0  0
  Verger   1 0 0 0 0 0 0 0  0  0

, ,  = TRUE


           0 1 2 3 4 5 6 8 10 11
  Arable   0 0 0 0 0 0 0 0  0  0
```

```
  Chaparral 0 0 0 0 0 0 0 1  1  0
  Prairie   0 0 0 0 1 0 0 0  0  0
  Pre       2 0 0 0 0 1 0 0  0  0
  Verger    0 0 0 0 0 0 0 0  0  0
> save(tableau, file = "un.tableau.rdata")
```

Par exemple, après avoir stocké le résultat d'une analyse dans un objet, il est possible de sauvegarder les résultats dans un fichier. Ce fichier peut-être importé dans `R` à un moment ultérieur en invoquant la fonction `load( )`. Chacune de ces deux fonctions nécessite le chemin menant au fichier.

```
> ##dans une session ultérieure, on importe l'objet directement
> load(file = "un.tableau.rdata")
> tableau
, ,  = FALSE


            0 1 2 3 4 5 6 8 10 11
  Arable    1 0 2 0 0 0 0 0  0  0
  Chaparral 1 0 0 0 0 0 0 0  1  0
  Prairie   0 2 2 2 0 0 1 0  0  1
  Pre       0 0 0 0 0 0 0 0  0  0
  Verger    1 0 0 0 0 0 0 0  0  0


, ,  = TRUE


            0 1 2 3 4 5 6 8 10 11
  Arable    0 0 0 0 0 0 0 0  0  0
```

```
Chaparral 0 0 0 0 0 0 0 1  1  0
Prairie   0 0 0 0 1 0 0 0  0  0
Pre       2 0 0 0 0 1 0 0  0  0
Verger    0 0 0 0 0 0 0 0  0  0
```

Nous venons de faire un survol des fonctions de base de R. Nous continuerons à explorer d'autres fonctionnalités de R lors des leçons subséquentes.

Index

`*`, 14

`+`, 14

`-`, 14

`/`, 14

`<-`, 8

`[ ]`, 10

`&&`, 20

`c`, 9, 11, 12

`length`, 14

`mean`, 5, 6

`rep`, 11

`seq`, 12, 13

aide

- `?`, 7
- `RSiteSearch`, 7
- `help.search`, 6
- `help.start`, 6

création de vecteur, 8

créer une variable, 33

création de série, 12

détermination du nombre d'observations, 14

exportation, 44

- `save`, 45
- `write.table`, 44

fonctions mathématiques, 16

fonctions trigonométriques, 16

importation de fichiers de données, 30

- `read.table`, 32
- accents, 32
- données manquantes, 32
- format, 31
- séparateur
  - tabulation, 33
  - virgule, 33
- type de fichier, 31

jeux de données, 26

- `dim`, 27
- `ncol`, 27
- `nrow`, 27
- accéder aux variables, 30
- ajouter une variable, 33
- création, 26
  - `data.frame`, 26
- dans `R`
  - `data`, 28
- exportation, 44
  - `save`, 45
  - `write.table`, 44
- restructuration, 41

sous-ensembles, 34–35

structure

`colnames`, 27

`head`, 28

`names`, 27

`rownames`, 27

`str`, 27

`summary`, 27

`tail`, 28

tri, 36

langage, 5

matrices, 21–25

calculs matriciels, 24–25

sélection d'éléments, 22

structure, 23–24

modification de jeux de données, 41

objet, 5

différents types, 5

opérateur d'assignation, 8

opérateurs mathématiques, 14

répertoire de travail, 33

répétition de valeurs, 11

sélection

à partir d'un jeu de données, 30

à partir d'un vecteur, 10

sélection d'éléments

à partir d'un jeu de données, 34

`[ , ]`, 34

subset, 35

tests logiques, 35

à partir d'un vecteur, 10

`[ ]`, 11

matrice, 22

syntaxe, 5

tableaux de fréquences

`table`, 37

`xtabs`, 39

tests logiques, 15

`!=`, 15

`<=`, 15

`<`, 15

`==`, 15

`>=`, 15

`>`, 15

`&`, 15

`all.equal`, 17

`all`, 17

`any`, 17

`else`, 20

`identical`, 17

`ifelse`, 18

`if`, 19

`which`, 18

trier

les observations d'un jeu de données, 36

les observations d'une seule variable, 36

vecteurs, 8