INTRODUCTION À L'ENVIRONNEMENT DE PROGRAMMATION STATISTIQUE R

Y. BROSTAUX [1]

RÉSUMÉ

Cette note constitue une introduction au langage et à l'environnement de programmation R dans sa version 1.4.1 pour Windows.

SUMMARY

This paper is an introduction to the R language and programming environment, version 1.4.1 for Windows.

1. INTRODUCTION

Le projet R consiste en une implémentation libre du langage S, développé depuis les années septante dans les laboratoires Bell par John Chambers et son équipe et distribué depuis 1993 sous licence commerciale exclusive par Insightful Corp. Initié dans les années nonante par Robert Gentleman et Ross Ihaka (Université d'Auckland, Nouvelle-Zélande), auxquels sont venus s'ajouter un noyau de chercheurs du monde entier en 1997, il constitue aujourd'hui un langage et un environnement de programmation intégré d'analyse statistique.

L'objectif de ce projet est de fournir un environnement interactif d'analyse de données, doté d'outils graphiques performants et permettant une adaptation aisée aux besoins des utilisateurs, depuis l'exécution de tâches routinières jusqu'au développement d'applications entières.

[1] Assistant à la Faculté universitaire des Sciences agronomiques de Gembloux

Le choix s'est donc porté sur un *langage fonctionnel orienté-objet*, structure alliant la facilité d'utilisation à la souplesse et la puissance de la programmation.

De plus, l'adoption d'une licence libre de type GNU/GPL (*General Public License*) a favorisé son développement et permis son port vers de nombreux systèmes informatiques (Unix, Linux, Macintosh, Windows, etc.). Projet dynamique, `R` est en constante évolution et bénéficie de fréquentes mises à jour, disponibles gratuitement sur le site du CRAN (Comprehensive `R` Archive Network, `http://cran.r-project.org/`).

Avant tout destiné aux scientifiques, il est aujourd'hui largement diffusé dans la communauté académique et sert de support à de nombreuses recherches et publications.

Cette note introduit les principes de base qui sous-tendent ce langage dans sa version 1.4.1 (30/01/2002), afin de faciliter sa prise en main et son apprentissage plus approfondi au départ des sources externes disponibles. Après cette introduction (paragraphe 1), on décrira de manière générale l'environnement et les principes fondamentaux du langage `R` (paragraphe 2), ainsi que l'accès aux sources de données externes (paragraphe 3). On illustrera ensuite son utilisation par quelques exemples (paragraphe 4) et on terminera par la mention de sources d'informations complémentaires concernant le projet `R` (paragraphe 5).

2. PRÉSENTATION GÉNÉRALE

On présentera tour à tour l'interface Windows de l'environnement `R` (paragraphe 2.1), les principales structures de données disponibles (paragraphe 2.2) et les principes de base de l'utilisation et de la conception des fonctions (paragraphe 2.3). Ces différentes notions seront illustrées par de courts exemples.

2.1. Interface

L'application `Rgui.exe` forme une interface utilisateur simple pour l'environnement `R`. Elle est structurée autour d'une barre de menu et de diverses fenêtres.

Les menus sont peu développés. Leur objectif est de fournir un raccourci vers certaines commandes parmi les plus utilisées, mais leur usage n'est pas exclusif, ces mêmes commandes pouvant être pour la plupart exécutées depuis la console.

Le menu File contient les outils nécessaires à la gestion de son espace de travail, tels que la sélection du répertoire par défaut, le chargement de fichiers sources externes, la sauvegarde et le chargement d'historiques de commandes, etc. Il est recommandé d'utiliser un répertoire différent pour chaque projet d'analyse, ce afin d'éviter la confusion dans les données et les programmes au fur et à mesure de l'accumulation des projets successifs.

Le menu Edit contient les habituelles commandes de copier-coller, ainsi que la boîte de dialogue autorisant la personnalisation de l'apparence de l'interface, tandis que le menu Misc traite de la gestion des objets en mémoire et permet d'arrêter une procédure en cours de traitement.

Le menu Packages automatise la gestion et le suivi des librairies de fonctions, permettant leur installation et leur mise à jour de manière transparente au départ du site du CRAN ou de toute autre source locale.

Enfin, les menus Windows et Help assument des fonctions similaires à celles qu'ils occupent dans les autres applications Windows, à savoir la définition spatiale des fenêtres et l'accès à l'aide en ligne et aux manuels de références de R.

Parmi les fenêtres, on distingue la console, fenêtre principale où sont réalisées par défaut les entrées de commandes et les sorties de résultats en mode texte. A celle-ci peuvent s'ajouter un certain nombre de fenêtres facultatives, telles que les fenêtres graphiques et les fenêtres d'informations (historique des commandes, aide, visualisation de fichier, etc.), toutes appelées par des commandes spécifiques via la console.

2.2. Structures de données

2.2.1. Vecteurs

La structure élémentaire de données dans R est le **vecteur** (*vector*), entité unique formée d'une simple collection ordonnée d'éléments de même nature ou *mode*. Suivant leur contenu, les vecteurs peuvent être numériques (*numerical vectors*), logiques (*logical vectors*) ou alphanumériques (*character vectors*).

La constitution manuelle d'un vecteur est réalisée grâce à la fonction `c(élément1, élément2, …)`. Les nombres décimaux doivent être encodés avec un point décimal, les chaînes de caractères entourées de guillemets doubles " ", et les valeurs logiques codées par les valeurs TRUE et FALSE ou leurs abréviations T et F respectivement. Les données manquantes sont codées par défaut par la chaîne NA.

Exemple:

```
> m <- c(5, 7.8, 6.1, 2.21)
> m
[1] 5.00 7.80 6.10 2.21
```

Comme on peut le constater sur cet exemple qui assigne un vecteur de quatre valeurs à un objet m et affiche ensuite son contenu, l'opérateur d'assignation est symbolisé par les deux caractères '<-' (ou alternativement par le caractère unique '_', mais son usage est déconseillé pour garantir une meilleure lisibilité du code). L'invite de commande R est quant à elle représentée par le caractère '>' et le chiffre 1 entre crochets débutant l'affichage du résultat est le numéro d'index du premier élément de la ligne.

Les vecteurs peuvent être utilisés tels quels dans des expressions arithmétiques classiques. Les opérations sont alors effectuées élément par élément. Les différents vecteurs intervenant dans l'expression ne doivent pas nécessairement présenter la même longueur. Si ce n'est pas le cas, le résultat est un vecteur possédant une longueur identique à celle du plus long vecteur de l'expression. Les valeurs des vecteurs plus courts sont alors recyclées, éventuellement partiellement, de manière à atteindre cette longueur.

Exemple:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(0.5, -0.5)
> z <- x*y
> z
[1]  0.5 -1.0  1.5 -2.0  2.5
```

De la même manière, les vecteurs peuvent être utilisés dans des expressions logiques, renvoyant alors un vecteur logique correspondant au résultat de l'expression appliquée sur chaque élément du vecteur de départ.

Exemple:

```
> x <- c(1, 2, 3, 4, 5)
> x>3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

L'extraction d'éléments d'un vecteur est rendue possible par l'utilisation d'un vecteur d'index, placé entre crochets à la suite du premier. R possède une grande souplesse concernant cette indexation, ce qui rend les opérations d'extraction et de modification des éléments de vecteurs très aisées. Ces vecteurs d'index peuvent être de quatre types distincts :

- **vecteur logique** de même longueur que le vecteur indexé

Chaque élément correspondant à un élément TRUE du vecteur logique est sélectionné. Le résultat a pour longueur le nombre d'éléments TRUE du vecteur logique.

Exemple:

```
> x <- c(1, 2, 3, 4, 5)
> x[x>3]
[1] 4 5
```

- **vecteur d'entier positifs** de longueur quelconque

Chaque élément dont la position correspond à la valeur mentionnée dans le vecteur d'index est sélectionnée. Le résultat a la longueur du vecteur d'index.

Exemple:

```
> z <- c(0.5, -1.0,  1.5, -2.0,  2.5)
> z[c(2, 3)]
[1] -1.0  1.5
```

- **vecteur d'entier négatifs**

Chaque élément dont la position correspond à la valeur absolue du nombre contenu dans le vecteur d'index est exclus de la sélection.

Exemple:

```
> z <- c(0.5, -1.0,  1.5, -2.0,  2.5)
> z[-c(2, 3)]
[1]  0.5 -2.0  2.5
```

- **vecteur de chaînes de caractères**

Chaque élément de vecteur peut être préalablement nommé explicitement par la fonction `names()`. Les éléments dont le nom correspond aux chaînes de caractères du vecteur d'index sont sélectionnés, accompagnés de leur nom.

Exemple:

```
> z <- c(0.5, -1.0,  1.5, -2.0,  2.5)
> names(z) <- c("un", "deux", "trois", "quatre", "cinq")
> z[c("deux", "trois")]
deux trois
-1.0   1.5
```

2.2.2. Listes et tableaux de données

Au côté des vecteurs coexiste un second type d'objets très utilisé dans le langage R, la **liste** (*list*). Une liste est une collection ordonnée d'objets, non nécessairement du même mode. Les éléments des listes peuvent donc être n'importe quel objet défini

dans R. Cette propriété est notamment utilisée par certaines fonctions pour renvoyer des résultats complexes sous forme d'un seul objet liste.

La constitution d'une liste passe par la fonction `list(`*`nom1=élément1, nom2=élément2, …`*`)`, l'utilisation des noms étant facultative. Chaque élément de la liste peut être accédé par son index numérique entre double crochets [[…]], ou son nom précédé du signe $.

Exemple:

```
> liste1 <- list(coeff=y, prod=z, nb=c("un", "deux"))

> # extraction du premier élément de la liste par index
> liste1[[1]]
[1] -0.5  0.5

> # extraction du premier élément de la liste par nom
> liste1$coeff
[1] -0.5  0.5

> # extraction d'un sous-élément de coeff par index
> liste1[[1]][1]
[1] -0.5
```

L'indexation par les simples crochets est également possible, mais sélectionne alors une sous-liste, éventuellement de longueur unitaire, et non plus l'objet élémentaire lui-même.

Exemple:

```
> mode(liste1[[1]])
[1] "numeric"
> mode(liste1[1])
[1] "list"
```

Il est possible de concaténer plusieurs listes en une seule au moyen de la fonction `c(`*`liste1, liste2, …`*`)`, comme pour les vecteurs. En effet, l'utilisation de la fonction *`list(liste1, liste2, …)`* conduirait à l'obtention d'une liste de listes.

Les **tableaux de données** (*data.frame*) constitue une classe particulière de listes, consacrée spécifiquement au stockage des données destinées à l'analyse. Chaque composant (le plus souvent un vecteur) de la liste forme alors l'équivalent d'une colonne ou variable, tandis que les différents éléments de ces composants correspondent aux lignes ou individus. A cette classe d'objet sont associées une série de méthodes spécifiques dérivées des fonctions de base et destinées à faciliter la visualisation et l'analyse de leur contenu (`plot()`, `summary()`, …)

L'obtention d'un tableau de données peut être réalisée par rassemblement de ses composants individuels via la fonction `data.frame(`*`nom1=élément1,`*

nom2=élément2, …), homologue de `list()`, par conversion d'un objet existant par `as.data.frame()` ou encore directement au départ de la lecture des données dans un fichier externe.

2.2.3. Autres objets de données

Au départ de ces classes élémentaires d'objets sont définies un grand nombre d'autres classes plus spécifiques, chacune caractérisée par des attributs et des méthodes propres.

Ainsi les matrices (*arrays*) sont constituées par un vecteur auquel on a adjoint un attribut de dimensions, les facteurs (*factors*) sont formés par un vecteur numérique muni d'étiquettes de description, etc. De plus, la plupart des librairies de fonctions possèdent leurs propres classes d'objets qui viennent s'ajouter aux classes de bases. Il convient alors de se reporter à la description qui accompagne ces librairies pour une information plus détaillée.

Cette structuration hiérarchique des types de données, typique des langages orientés objets, assure une grande souplesse de programmation permettant d'obtenir, au départ d'un nombre restreint de fonctions, une grande variété de comportements adaptés au contexte de l'exécution.

2.3. Fonctions

Les fonctions forment l'unité de base de la programmation sous R. Elles sont stockées en mémoire sous forme d'objets élémentaires de mode *function* et peuvent contenir une suite d'instructions R, y compris des appels à d'autres fonctions.

C'est sur ce principe pyramidal de fonctions construites sur base d'autres fonctions que repose une grande part de la puissance et de la concision du langage. La grande majorité des fonctions internes sont construites sur ce mode directement en langage R, la base de cette pyramide étant formée d'un petit nombre de fonctions basiques externes.

2.3.1. Syntaxe

La syntaxe d'appel d'une fonction est simple et intuitive : `nom_fonction(`*`argument1 = valeur1, argument2 = valeur2, …`*`)`. Les arguments peuvent être spécifiés sous forme d'une liste nommée, la reconnaissance s'effectuant sur base du nom de l'argument, d'une liste ordonnée, basée alors sur la position de l'argument dans la liste (l'emplacement des arguments manquants étant malgré tout réservé au moyen des virgules de séparation), ou d'un mélange des deux.

Exemple:

```
> plot(x=hauteur, y=poids, type="p")
```

Cet appel de `plot()`, fonction graphique présentant ici trois arguments *x*, *y* et *type*, est équivalent à

```
> plot(hauteur, poids, , ,"p")
```

les arguments *x*, *y* et *type* intervenant respectivement en première, deuxième et cinquième position dans la définition de la fonction, ou encore à

```
> plot(hauteur, poids, type="p")
```

les deux premiers arguments étant à leur position nominale et l'argument *type* nommé.

En outre la plupart des fonctions sont définies avec une liste d'arguments par défaut, consultable via l'aide associée. Les arguments possédant une valeur par défaut peuvent être omis lors de l'appel de la fonction et sont alors remplacés par cette valeur lors de l'interprétation du code. Cette propriété permet la définition de fonctions présentant un nombre important d'arguments et donc une grande souplesse d'exécution, tout en conservant la facilité d'emploi et la concision du code.

Etant donné le nombre important de fonctions disponibles sous l'environnement R et dans les librairies associées, un effort particulier a été consenti pour leur documentation. Outre une aide concise concernant la syntaxe, les arguments et les principes sous-jacents à l'utilisation de chaque fonction, accessible par la commande `help(`*nom_fonction*`)`, l'utilisateur peut bénéficier d'une démonstration par l'exemple de l'emploi des différentes fonctions, grâce à `example(`*nom_fonction*`)`.

2.3.2. Librairies externes

Le langage R est structuré en libraires de fonctions (*packages* ou *libraries*). Les fonctions élémentaires sont regroupées dans la librairie `base`, chargée en mémoire au démarrage. Une sélection d'autres librairies, recommandées par le groupe de coordination du projet R, est également installée par défaut en même temps que le programme principal mais ces dernières doivent être chargées manuellement pour être disponibles au départ de la console.

En effet, afin de pouvoir utiliser une fonction externe appartenant à une librairie, il est nécessaire de charger cette librairie au préalable dans l'espace de travail courant. Cette opération peut être réalisée via la menu Packages ou la commande `library(`*nom_librairie*`)`.

Grâce à la licence GNU, une communauté importante d'utilisateurs a pu contribuer à l'extension des fonctionnalités de R. Un large choix de librairies additionnelles est ainsi disponible sur le site du CRAN, couvrant des domaines très variés de l'analyse statistique au sens large (analyse multivariée, géostatistique, séries

chronologiques, modèles non linéaires, etc.). Leur installation est facilitée par les commandes du menu Packages, de même que leur mise à jour automatique.

A ces librairies officielles peuvent encore s'ajouter celles créées par l'utilisateur lui-même, étant donné la possibilité de concevoir aisément ses propres fonctions R.

Enfin, signalons que des interfaces vers des procédures externes d'autres langages courants, notamment vers le C et le Fortran sont également disponibles.

2.3.3. Définition de nouvelles fonctions

Il est possible de définir des fonctions personnalisées soit directement au départ de la console, soit via un éditeur de texte externe (par défaut, le Bloc-notes) grâce à la commande `fix(nom_fonction)`. La seconde possibilité présente de permettre la correction du code en cours d'édition, tandis que la première s'effectue ligne par ligne, sans retour en arrière possible.

D'une manière générale, la définition d'une fonction passe par l'expression suivante :

```
nom_fonction <- function(arg1[=expr1], arg2[=expr2], …){
      bloc d'instructions
}
```

Les accolades signalent à l'interpréteur de commande le début et la fin du code source de la fonction ainsi définie, tandis que les crochets ne font pas partie de l'expression mais indiquent le caractère facultatif des valeurs par défaut des arguments.

Il est également possible de créer une fonction personnalisée au départ d'une fonction existante, tout en conservant l'original intact, grâce à

```
nom_fonction2 <- edit(nom_fonction1)
```

Lors de l'exécution, R renvoie par défaut le résultat de la dernière expression évaluée dans la fonction. Il est possible de préciser le résultat renvoyé grâce à la fonction `return(objet1[, objet2, …])`, résultat qui prend la forme d'une liste nommée si plusieurs arguments sont spécifiés.

Les arguments sont passés à la fonction par valeur (*ByValue*) et leur portée ainsi que celle de toute assignation classique à l'intérieur d'une fonction est locale. Lors de l'exécution, une copie des arguments est transmise à la fonction, laissant les originaux intacts. Les valeurs originelles des arguments ne sont donc pas modifiées par les expressions contenues dans la fonction.

Exemple:

```
> x <- 2
> carré <- function(x) {x <- x*x; return(x)}
```

```
> carré(x)
[1] 4
> x
[1] 2
```

Enfin, signalons qu'il est possible voire recommandé d'ajouter des commentaires au code des fonctions, en les faisant précéder du symbole dièse #. La suite de la ligne est alors ignorée lors de l'interprétation de la fonction et peut donc être complétée par un commentaire libre.

3. ACCÈS AUX SOURCES DE DONNÉES EXTERNES

Bien que l'encodage des données directement dans l'environnement R soit possible, dans la majorité des cas les données à analyser proviennent de sources externes sous forme de fichiers. De plus, les objets créés doivent pouvoir être sauvegardés entre les différentes sessions afin de pouvoir reprendre le travail là où on l'avait laissé.

C'est pourquoi divers outils d'accès aux fichiers ont été développés sous R. On distingue ainsi les accès aux fichiers propriétaires de R (paragraphe 3.1), aux fichiers ASCII (paragraphe 3.2), aux fichiers provenant d'autres logiciels d'analyse statistique (paragraphe 3.3), aux bases de données relationnelles (paragraphe 3.4) et aux fichiers binaires (paragraphe 3.5).

3.1. Formats propriétaires R

Les données, fonctions et autres objets créés au cours d'une session R peuvent bien sûr être sauvegardés pour être réutilisés ultérieurement. Les fichiers ainsi construits portent le plus souvent l'extension .R.

Tout d'abord, R propose la sauvegarde intégrale de l'espace de travail via le menu File et lors de la sortie de la session. Cela correspond à l'exécution de la commande `save.image()`, qui crée un instantané de la session de travail sous forme d'un fichier binaire nommé `.Rdata` qui sera chargé automatiquement au prochain démarrage de R dans le répertoire de travail en cours.

Cette commande est en fait un raccourci construit au départ de la fonction générique `save()` qui autorise la sauvegarde de n'importe quelle liste d'objets en mémoire sous un chemin quelconque, aussi bien en format binaire qu'ASCII (utile pour l'échange de données entre différentes plates-formes matérielles). Ces objets peuvent ensuite être rechargés en mémoire grâce à la fonction `load()` qui fait le pendant de la première.

Pour des raisons de compatibilité avec `S`, il existe également une fonction `dump()` qui exporte fonctions et vecteurs vers un fichier en mode texte, pouvant alors être relu et interprétés séquentiellement par la fonction `source()`. Bien que de portée plus limitée, ces fonctions restent très utiles en programmation car les fichiers qu'elles gèrent sont aisément modifiables par l'utilisateur via un éditeur de texte externe.

3.2. Fichiers textes ASCII

Le format d'échange le plus courant en ce qui concerne les données brutes reste le fichier ASCII. La lecture de tels fichiers est prise en charge par la commande élémentaire `scan()`. Les arguments de cette fonction permettent de décrire précisément la structure interne du fichier texte afin de d'interpréter correctement les caractères lus et de transférer le résultat de cette interprétation dans les objets adéquats.

Afin de faciliter la lecture des fichiers de données structurés en colonnes, plusieurs commandes spécifiques ont été développées au départ de la fonction `scan()`. Ces fonctions (`read.table()` et ses dérivés) automatisent la lecture des fichiers de données ASCII standards (CSV, texte délimité, largeur fixe, etc.) et stockent leurs résultats sous forme de tableaux de données prêts à l'analyse. Bien qu'elles soient plus spécifiques que `scan()`, ces fonctions conservent toutefois une grande adaptabilité grâce à l'utilisation de nombreux arguments permettant de préciser le format interne du fichier (présence de titres de colonnes, type de séparateur décimal utilisé, symbole(s) de valeur manquante, etc.).

L'exportation de tableaux de données sous forme de fichiers ASCII standards peut être réalisée par la fonction `write.table()`, complémentaire des précédentes, et présentant les mêmes possibilités en matière de formatage des résultats que celles-ci.

3.3. Logiciels statistiques

Lorsque les données ont été sauvegardées sous un format propriétaire d'un logiciel statistique tiers inaccessible à l'utilisateur, il est nécessaire de disposer d'outils permettant leur transfert vers le système `R` afin de pouvoir les analyser. La libraire `foreign` offre ces outils indispensables pour une sélection des logiciels statistiques les plus courants, à savoir Minitab, S-Plus, SAS, SPSS et Stata.

Ainsi, la fonction `read.tmp()` importe les fichiers 'Minitab Portable Worksheet' sous forme d'une liste, de même que `read.xport()` prend en charge les fichiers SAS Transport (XPORT) et `read.spss()` les données enregistrées au moyen des commandes 'save' et 'export' de SPSS.

Concernant le logiciel S-Plus, la fonction `read.S()` permet de lire certains objets binaires (notamment les vecteurs, matrices, tableaux de données et listes

contenant ces objets) créés sous les versions 3.x, 4.x ou 2000 sous Windows ou Unix de ce logiciel. La commande `data.restore()` réalise la même opération au départ des versions textes de ces mêmes objets, créées par la commande S-Plus `data.dump()`.

Enfin, les fonctions `read.dta()` et `write.dta()` permettent l'importation et l'exportation des fichiers binaires Stata versions 5.0, 6.0 et 7.0.

3.4. Bases de données relationnelles

La version de base de l'environnement `R` n'est pas adaptée à la gestion de très grosses quantités de données. En effet, tous les objets sont chargés intégralement en mémoire centrale et plusieurs copies de ces objets peuvent être créées lors des procédures de traitement, ce qui peut entraîner une saturation du système dès que la taille totale des jeux de données dépasse une certaine fraction de l'espace mémoire disponible. De plus, il ne permet pas à plusieurs utilisateurs d'accéder aux mêmes données de manière concurrente, c'est-à-dire en intégrant en temps réel les modifications des uns et des autres.

Ce travail de gestion est d'ordinaire le domaine de prédilection des systèmes de gestion de bases de données (SGBD), et plus particulièrement des SGBD relationnels, qu'ils soient commerciaux, tels que Oracle, Microsoft SQL Server, IBM DB/2, etc. ou du domaine libre comme MySQL, PostgreSQL, etc.

Une série de modules faisant office d'interface entre ces SGBD et l'environnement `R` ont donc été développés afin de bénéficier des capacités spécifiques de ces deux domaines d'application : SGBD pour le stockage, la gestion et l'extraction des données brutes, `R` pour leur interprétation analytique et graphique.

On trouve ainsi diverses librairies de fonctions dédiées au pilotage des SGDB, soit propres à un système donné, soit utilisant l'interface générique OBDC (*Open Database Connectivity*) initiée par Microsoft, aujourd'hui disponible sur de nombreux systèmes. Il s'agit de `RPgSQL` pour PostgreSQL, `RMySQL` pour MySQL, `RmSQL` pour MiniSQL, `ROracle` pour Oracle, `RSQLite` pour SQLite et enfin `RODBC` pour les autres SGDB supportant l'ODBC.

Ces outils permettent principalement d'importer et d'exporter des tableaux de données de ou vers les SGBD, d'exécuter des requêtes SQL (conformes à la norme ISO SQL92) sur les serveurs de bases de données et d'importer leurs résultats dans `R` pour traitement et analyse.

3.5. Fichiers binaires

`R` fournit également des outils permettant l'accès aux fichiers binaires, tels que les fichiers images ou sons par exemple, par l'intermédiaire de **connexions**, objets servant de liens entre le système et les fichiers externes.

Une fois créée par la commande générique `open()`, une connexion permet un accès transparent au fichier qu'elle représente. Dans le cas d'une connexion en mode binaire, les fonction `readBin()` et `writeBin()` se chargent des opérations de lecture et d'écriture dans le fichier ainsi ouvert.

4. EXEMPLES D'APPLICATIONS

La description théorique d'un langage ou d'un logiciel ne pouvant jamais remplacer l'expérience pratique de celui-ci, nous proposons ici divers exemples d'applications destinés à être réalisés de manière interactive dans l'environnement `R`. On découvrira ainsi tour à tour l'interface, l'aide en ligne et les commandes graphiques de base (paragraphe 4.1), la lecture de données externes et l'analyse de la variance (paragraphe 4.2), la régression linéaire (paragraphe 4.3), la définition de fonctions et la régression non linéaire (paragraphe 4.4) et enfin la réalisation de graphiques conditionnels aussi appelés treillis (paragraphe 4.5).

Les différents fichiers de données externes nécessaires à ces applications sont disponibles en ligne via la page Publications du site Internet du service de Statistique et Informatique de la FUSAGx (`http://www.fsagx.ac.be/si/`). Ces exemples supposent que ces fichiers ont été copiés dans le répertoire de travail courant de l'environnement R.

Les commandes et les commentaires sont transcrits en caractères gras, les résultats en caractères normaux. Certaines sorties de résultats (signalées par trois points de suspension) ont été coupées afin de ne pas encombrer inutilement les exemples. Les graphiques générés sont repris en annexe. Pour de plus amples explications sur les commandes utilisées, le lecteur se reportera avec fruit à l'aide interne accessible via `help(`*`commande`*`)`.

4.1. Fonctions graphiques de base

```
> # Chargement des données au départ de la librairie interne 'base'
> data(iris)

> # Affichage de l'aide concernant le jeu de données 'iris'
> help(iris)

> # Résumé du tableau de données iris (statistiques descriptives)
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
```

```
 setosa    :50
 versicolor:50
 virginica :50
```

```
> help(summary)
```

```
> # Affichage d'un graphique reprenant les relations deux à deux
> # de l'ensemble des variables du tableau de données 'iris'
> plot(iris)
> help(plot)
```

```
> # Même graphique, mais utilisation d'un symbole différent selon
> # l'espèce concernée via l'argument pch
> plot(iris, pch=(1:3)[iris$Species])
```

```
> # Nuage de points avec en abscisse la variable Sepal_Length,
> # en ordonnée Petal.Length, des symboles différents selon l'espèce
> # un titre général et des titres d'axes personnalisés
> plot(iris$Sepal.Length, iris$Petal.Length, pch=(1:3)[iris$Species],
main="Graphique de base - Iris de Fisher", xlab="Longueur des sépales
(mm)", ylab="Longueur des pétales (mm)")
```

```
> # Ajout d'une légende au graphique
> help(legend)
> legend(7, 2, c("I.setosa", "I.versicolor", "I.virginica"), pch=1:3)
```

4.2. Analyse de la variance

```
> # Lecture du tableau de données à partir d'un fichier ASCII
> fumures.df <- read.table("fumures.dat", header=TRUE)

> # Définition des colonnes 'fumures' et 'blocs' en tant que facteurs
> fumures.df$fumures<- as.factor(fumures.df$fumures)
> fumures.df$blocs<- as.factor(fumures.df$blocs)

> # Affichage du tableau de données
> fumures.df
   fumures blocs    P2O5
1        1     1  8.0216
2        1     2  9.0205
3        1     3  7.9772
...

> # chargement des variables du tableau 'fumures.df' en mémoire
> # afin d'éviter sa répétition lors des appels multiples à venir
> # exemple: fumures.df$P2O5 devient simplement P2O5
> attach(fumures.df)

> # calcul des statistiques descriptives par type de fumures
> by(P2O5, fumures, summary)
INDICES: 0
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.448   4.462   5.215   4.984   5.737   7.058
------------------------------------------------------------
INDICES: 1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  7.977   8.011   8.521   8.527   9.038   9.090
------------------------------------------------------------
INDICES: 2
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  6.938   7.674   8.235   8.169   8.731   9.268
------------------------------------------------------------
...

> # Graphe de type boxplot des taux de phosphate par type de fumures
> boxplot(P2O5 ~ fumures, data=fumures.df)

> # Analyse de la variance
> # Analyse par aov() et stockage du résultat dans 'fumures.aov'
> fumures.aov <- aov(P2O5 ~ fumures + blocs, data=fumures.df)
```

```
> # Affichage du tableau d'analyse de la variance
> summary(fumures.aov)
            Df  Sum Sq Mean Sq F value  Pr(>F)
fumures      7 112.880  16.126  3.3434 0.01487 *
blocs        3  28.940   9.647  2.0001 0.14485
Residuals   21 101.285   4.823
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> # Structure des objets résultats de l'analyse
> str(fumures.aov)
List of 13
 $ coefficients : Named num [1:11] 4.59 3.54 3.18 5.43 3.96 ...
  ..- attr(*, "names")= chr [1:11] "(Intercept)" "fumures1" "fumures2"
"fumures3" ...
 $ residuals    : Named num [1:32] -0.1149  1.0945  0.0965 -1.0760 -
0.8405 ...
  ..- attr(*, "names")= chr [1:32] "1" "2" "3" "4" ...
...
> str(summary(fumures.aov))
List of 1
 $ :Classes anova  and 'data.frame':    3 obs. of  5 variables:
  ..$ Df     : num [1:3] 7 3 21
  ..$ Sum Sq : num [1:3] 112.9  28.9 101.3
  ..$ Mean Sq: num [1:3] 16.13  9.65  4.82
  ..$ F value: num [1:3] 3.34 2.00   NA
  ..$ Pr(>F) : num [1:3] 0.0149 0.1448     NA
 - attr(*, "class")= chr [1:2] "summary.aov" "listof"

> # graphiques diagnostics de l'analyse de la variance
> plot(fumures.aov)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:

> # Mise à jour du modèle d'analyse
> fumures.aov1 <- update(fumures.aov, . ~ . - blocs)
> summary(fumures.aov1)
            Df  Sum Sq Mean Sq F value  Pr(>F)
fumures      7 112.880  16.126  2.9719 0.02157 *
Residuals   24 130.225   5.426
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.3. Régression linéaire

```
> # Lecture et affichage du tableau de données 'insect'
> insect <- read.table("insect.dat", header=TRUE)
> insect
```

```
   sexe  pop  pdsO dvtON  pdsN dvtNA  pdsA dvtOA
1     F pop1 0.068  30.4  4.71   7.2  3.92  37.6
2     M pop1 0.072  31.5  4.53   7.3  3.76  38.3
3     F pop2 0.094  40.7  3.19   6.2  2.74  46.0
4     M pop2 0.097  40.9  2.67   5.9  2.27  46.4
...
```

```
> # Matrice graphique par paires
> plot(insect)

> # Graphique simple log/log
> # couleur variable selon le sexe grâce à l'argument col
> # symbole variable selon la population grâce à l'argument pch
> # pop étant un facteur, il est converti en nombre par as.numeric()
> attach(insect)
> plot(log(pdsO), log(dvtOA), col=c("red", "blue")[sexe],
pch=as.numeric(pop))

> # relation linéaire log(dvtOA) = a + b * log(pdsO)
> # simple, a distincts ou a et b distincts selon le sexe
> # ajout d'une variable dans le modèle par l'opérateur '+'
> # interaction entre facteurs codée par l'opérateur ':'
> insect.lm <- lm(log(dvtOA)~log(pdsO))
> insect.lm1 <- lm(log(dvtOA)~sexe+log(pdsO))
> insect.lm2 <- lm(log(dvtOA)~sexe+log(pdsO)+sexe:log(pdsO))

> # Comparaison des trois modèles via une table ANOVA
> anova(insect.lm, insect.lm1, insect.lm2)
Analysis of Variance Table

Model 1: log(dvtOA) ~ log(pdsO)
Model 2: log(dvtOA) ~ sexe + log(pdsO)
Model 3: log(dvtOA) ~ sexe + log(pdsO) + sexe:log(pdsO)
  Res.Df      RSS Df Sum of Sq      F Pr(>F)
1     16 0.047566
2     15 0.047484  1  0.000082 0.0241 0.8787
3     14 0.047360  1  0.000124 0.0367 0.8508

> # Résumé statistique
> summary(insect.lm)

Call:
lm(formula = log(dvtOA) ~ log(pdsO))

Residuals:
      Min        1Q    Median        3Q       Max
-0.085114 -0.032988  0.009703  0.036384  0.094208

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.13550    0.12893   39.83  < 2e-16 ***
log(pdsO)    0.57313    0.04864   11.78 2.68e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.05452 on 16 degrees of freedom
Multiple R-Squared: 0.8967,     Adjusted R-squared: 0.8902
F-statistic: 138.8 on 1 and 16 DF,  p-value: 2.681e-009

> # représentation graphique de la droite de régression
```

```
> abline(insect.lm, col="black")
```

4.4. Régression non linéaire

```
> # Lecture du tableau de données
> ecaille <- read.table("ecaille.dat")

> # spécification des noms des variables lues
> colnames(ecaille) <- c("age", "long")

> # Résumé statistique et aperçu graphique de la relation
> summary(ecaille)
      age             long
 Min.   :0.500   Min.   :127.0
 1st Qu.:0.500   1st Qu.:159.0
 Median :1.500   Median :215.0
 Mean   :1.803   Mean   :208.8
 3rd Qu.:2.500   3rd Qu.:257.0
 Max.   :4.500   Max.   :311.0
> plot(ecaille)

> # Définition du modèle non linéaire (Mitscherlich)
> # via une fonction personnalisée à deux arguments
> # x: valeur de l'abscisse à laquelle le modèle doit être estimé
> # parm: vecteur reprenant les trois paramètres M, a et b du modèle
> mitsch <- function(x, parm) {
+ M <- parm[1]
+ a <- parm[2]
+ b <- parm[3]
+ u <- (x - a)/b
+ y <- M * (1 - exp(-u))
+ }
> attach(ecaille)

> # Chargement de la libraire de fonction nls (non-linear least-square)
> library(nls)

> # Estimation des paramètres du modèle (algorithme de Gauss-Newton)
> ecaille.nls <- nls(long ~ mitsch(age, c(M, a, b)), start=c(M=300, a=-
0.7, b=2))
> summary(ecaille.nls)

Formula: long ~ mitsch(age, c(M, a, b))

Parameters:
  Estimate Std. Error t value Pr(>|t|)
M 298.2295    13.1594  22.663  < 2e-16 ***
a  -0.7010     0.1562  -4.489 2.21e-05 ***
b   1.8132     0.3071   5.904 6.89e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 18.38 on 86 degrees of freedom

Correlation of Parameter Estimates:
        M     a
```

```
a -0.8111
b  0.9492 -0.94
```

```
> # Représentation graphique du modèle sur les données brutes
> # Génération systématique de 101 valeurs entre 0.5 et 4.5
> x <- (0:100)/25+0.5

> # Tracé des valeurs estimées par le modèle non-linéaire
> # au départ des 100 valeurs de x générées
> # La fonction m$getAllPars(), propre au modèle nls, renvoie
> # les valeurs estimées des paramètres de celui-ci
> lines(x, mitsch(x, ecaille.nls$m$getAllPars()), col="red")
```

4.5. Graphiques en treillis (graphes conditionnels)

```
> # Lecture du tableau de données et résumé statistique
> antig.df <- read.table("antigene.dat", header=TRUE)
> summary(antig.df)
    age         antigene
 20-39:  9   Min.   : 0.000
 40-49: 45   1st Qu.: 0.500
 50-59:116   Median : 1.000
 60-69:156   Mean   : 2.000
 70+  :174   3rd Qu.: 2.125
             Max.   :75.800
> attach(antig.df)

> # Chargement de la libraire de fonction lattice
> library(lattice)

> # Histogramme et densité estimée de la distribution
> # du taux d'antigène conditionnellement à la classe d'âge
> # L'opérateur conditionnel est représenté par le symbole '|'
> histogram(~antigene|age, type="percent")
> histogram(~log(antigene+1)|age, type="percent")
> densityplot(~log(antigene+1)|age, col="black")
```

5. SOURCES D'INFORMATIONS COMPLÉMENTAIRES

La majeure partie de l'information sur le projet R est disponible au départ du site WWW du projet (http://www.r-project.org/) ou du site du CRAN (http://cran.r-project.org/). Les adresses mentionnées ici sont celles valides à la date de la publication de cette note.

On y trouvera notamment les manuels en format pdf (installés sur demande en même temps que la version Windows de R),

http://cran.r-project.org/manuals.html

une Foire Aux Questions (FAQ),

`http://cran.r-project.org/doc/FAQ/`

trois mailing-lists (r-announce, r-devel et r-help),

`http://www.r-project.org/mail.html`

et un journal, R News,

`http://cran.r-project.org/doc/Rnews/`

En outre, un outil de recherche combinée, permettant de parcourir l'ensemble de la documentation R, l'aide des fonctions et les archives de la mailing-list r-help est disponible à l'adresse :

`http://finzi.psych.upenn.edu/search.html`

6. BIBLIOGRAPHIE

IHAKA R. ET GENTLEMAN R. [1996]. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3), 299-314.

R CORE TEAM. [2001]. What is R ? *R News*, 1(1), 2-3.

R CORE TEAM. [2002]. *An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics (version 1.4.1)*. 105 p.

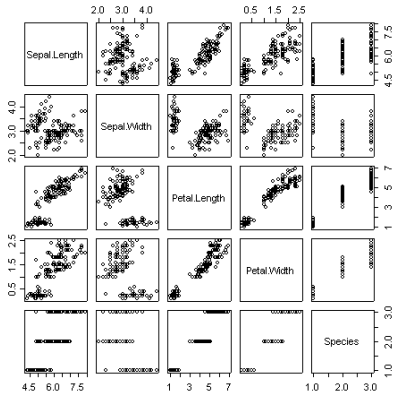
R CORE TEAM. [2002]. *R Data Import/Export (version 1.4.1)*. 35 p.

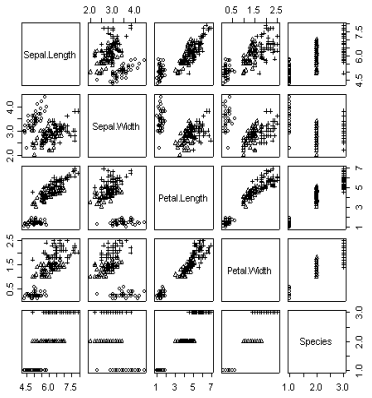
R CORE TEAM. [2002]. *The R Reference Index (version 1.4.1)*. 1050 p.

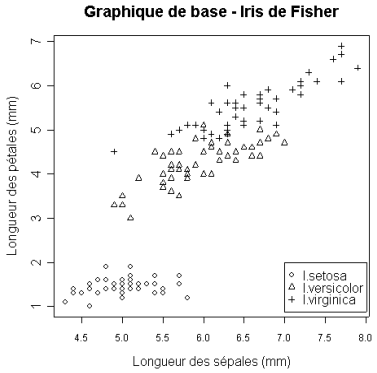
VENABLES W.N. ET RIPLEY B.D. [1997]. *Modern Applied Statistics with S-Plus*. 2nd Edition, New York, Springer, 548 p.

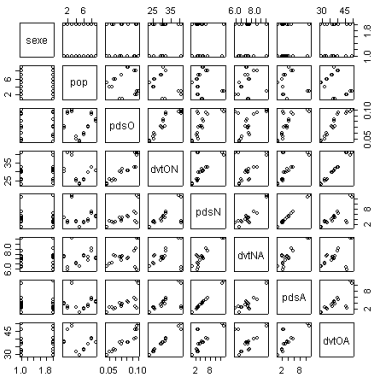
ANNEXE

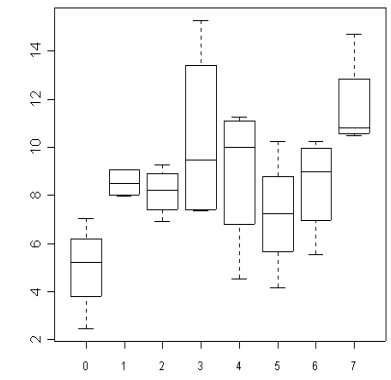
Graphiques générés au cours des exemples

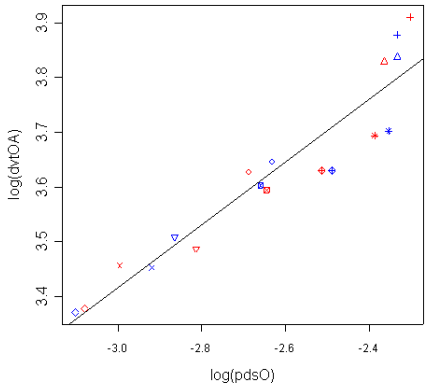


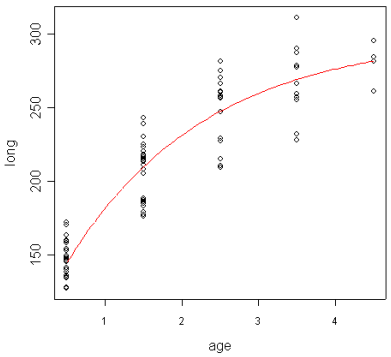


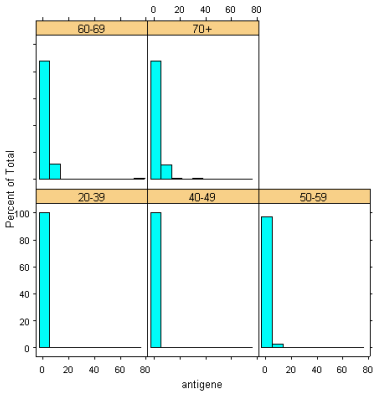


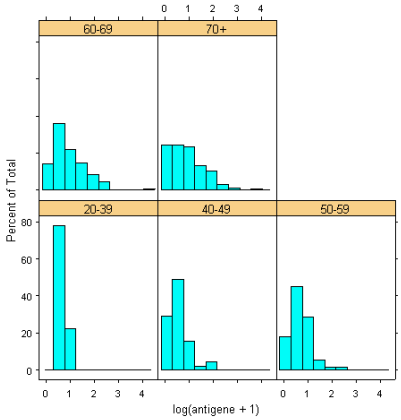


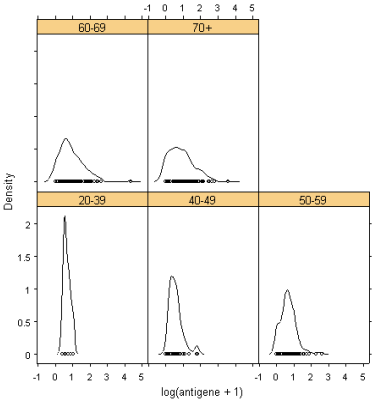



log(dvtOA)
log(pdsO)


long
age


60-69
70+
20-39
40-49
50-59
Percent of Total
antigene


60-69
70+
20-39
40-49
50-59
Percent of Total
log(antigene + 1)


60-69
70+
20-39
40-49
50-59
Density
log(antigene + 1)