Timothée Poisot

Introduction à R

Par la pratique

UQAR — Mai 2012

Table des matières

Séance

1

Introduction

L'objectif de ce cours est de fournir une introduction générale à R [1], un langage qui est en train de devenir un standard en analyse de données et en calcul scientifique. R est souvent présenté dans l'optique d'une utilisation pour les statistiques. Si c'est effectivement une des capacités les plus souvent utilisées de ce langage, ce cours n'abordera pas ce domaine. L'objectif des 6 séances est plutôt de fournir, au travers d'exemples et de mises en application, un apercu suffisament vaste de R pour le rendre utilisable dans la plus grande majorité des situations : nous aborderons donc les bases du langage, en commençant par les types d'objets utilisés par R, puis consacrerons une partie importante du temps a lire, écrire, et manipuler des jeux de données. Les dernières séances sont consacrées à la programmation et aux graphiques, qui permettront d'aller plus loin par la suite.

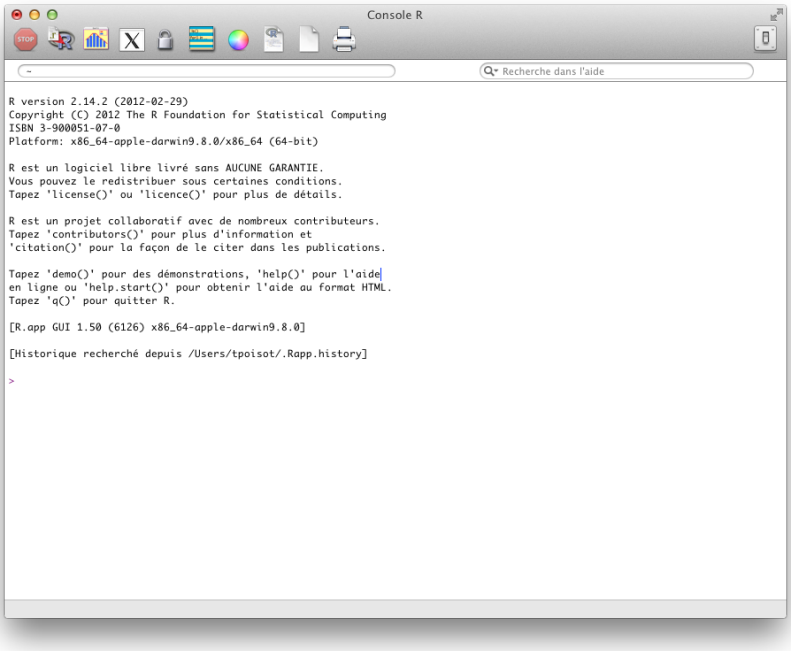
Si R a acquis une très grande popularité dans le milieu de la recherche, c'est parce qu'il permet très facilement de rajouter des fonctionnalités. *via* le système de *packages*. En écologie et biologie évolutive, les principaux sont *vegan* (analyse de structure des communautés), *simecol* (simulation de systèmes dynamiques), *ape* (analyses phylogénétiques), *bipartite* (analyse des réseaux de pollinisation), et *emdbook* (modèles statistiques). Chacun de ces *packages* est accompagné d'un livre ou d'une série de publications, et couvrent une variété énorme d'analyses. Depuis plus récemment, le projet *ROpenSci* [1] propose des *packages* permettant d'interagir avec les principales bases de données en écologie et évolution, et notamment de réaliser des analyses bibliométriques. Il est aussi de plus en plus fréquent de voir des articles dans lesquels les auteurs ont écrit un *package* R qui permet de reproduire leurs analyses, ou au moins mis en *supplementary material* des nouvelles fonctions utilisées dans leur travail.

Le cours s'étend sur 3 jours, et se compose de 6 séances couvrant chacune une demie journée. L'objectif est qu'à l'issue des 6 séances, vous soyez capables de comprendre le fonctionnement de R (séance 1), de lire, et écrire des données (séance 2), et de les manipuler (séance 3). La séance 4 comporte une introduction à la programmation, qui vous permettra de réaliser des choses plus poussées. La séance 5 est consacrée aux graphiques dans R, et couvre les outils de base. La séance 6 est prévue pour répondre a des questions plus générales. Chaque séance se déroule en deux temps. Premièrement, une introduction assez générale des concepts qui seront utilisés, avec des exemples de commandes et leurs résultats. Deuxièmement, une série de mises en application, qui consistent en un ou plusieurs petits problèmes généraux, mettant en application les notions acquises jusque la.

1. http://ropensci.org/

1.1 Environnement de travail

La première étape est de mettre en place notre environnement de travail. Si ce n'est pas déjà fait, commencez par installer R, disponible à `http://www.r-project.org/`. Au lancement de R s'affiche la fenêtre suivante :

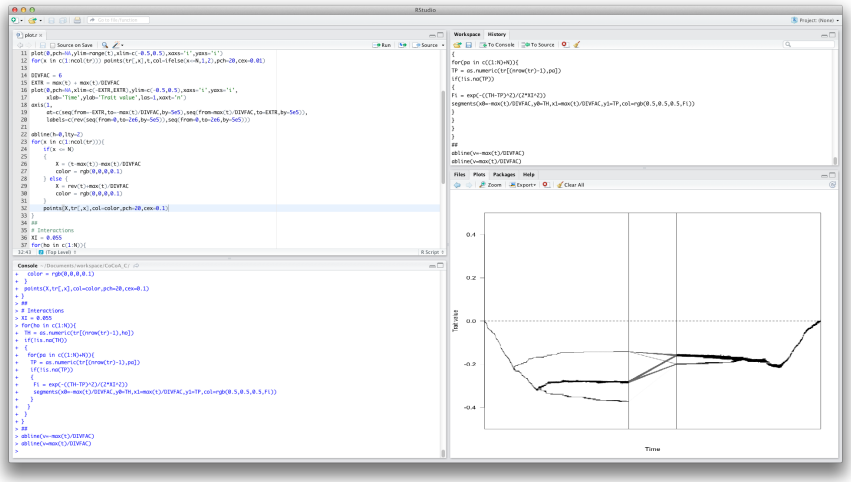


R n'est donc pas un logiciel comme *JMP*, *Statistica* ou *Excel*. Sa philosophie est entièrement différente : plutôt que d'utiliser des boutons et des menus, on communique avec R via des commandes, qu'on entre soit directement dans la console, soit dans des fichiers `.r`. R est accessible depuis de nombreux autres programmes, incluant *Komodo Edit* [2], *Eclipse* [3], et *NotePad ++* [4].

Dans le cadre de ce cours, nous utiliserons *RStudio*. *RStudio* est un environnement intégré de travail pour R grauit, multi-plateforme, disponible en ligne à `http://rstudio.org/`. Ce programme permet de réaliser, dans le même environnement, la majorité des tâches qu'on peut vouloir effectuer dans R.

Au lacenemt de *RStudio*, une fenêtre ressemblant à celle ci-dessous s'affice :

2. `http://www.sciviews.org/SciViews-K/`
3. `http://www.walware.de/goto/statet`
4. `http://sourceforge.net/projects/npptor/`



Les différentes sections permettent d'avoir accès simultanément à la console R (en bas à gauche), à la fenêtre des scripts (en haut à gauche), à l'historique des dernières commandes, et aux graphiques. Au cours de la première séance, en même temps que nous découvrirons les bases du langage R, nous survolerons les fonctionnalités de *RStudio*.

1.2 Types de données

Cette section présente les types de données comprises par R. Les langages de programmation stockent en général des variables de différents types d'une manière différente. Dans les langages de bas niveau, comme C, ces différences peuvent être relativement restrictives, pour, par exemple, convertir un entier en flottant. R est un langage dit faiblement typé (dans la réalité, R est surtout *très mal* typé, mais c'est un autre problème...), en ce que les valeurs peuvent facilement changer de type.

Il existe deux niveaux d'organisation pour représenter des valeurs dans un format compris par R : les types de données, et les collections de données. Ce vocabulaire n'est certainement pas le plus précis, mais établit au moins une différence entre les valeurs (les données) qu'on veut regrouper dans des structures plus vastes, et ces structures en elles-même (les collections).

Types de valeurs

Dans R, stocker n'importe quelle valeur, ou collection de valeurs, dans un objet, se fait de la manière suivante :

```
objet <- valeur
```

On peut aussi utiliser indifféremment les opérateurs <- ou =. Ma préférence va au dernier, puisque dans une lecture rapide `a <- 2` ($a = 2$) ressemble beaucoup (trop) à `a < -2` ($a < -2$). Cependant, les recommandations sur la présentation de la syntaxe de R vont dans le sens de l'utilisation de <- (entouré de deux espaces), et c'est cette notation qui sera utilisée ici.

Numériques

R est extrèmement performant dans le stockage de nombres, ce qui ne devrait pas surprendre de la part d'un langage dévellopé pour traiter des problèmes statistiques. Les exemples suivants montrent les différentes manières de déclarer un object contenant un nombre :

```
a <- 0.1
b <- 1
c <- 1e-06
a
```

```
## [1] 0.1
```

```
b
```

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

```
c
```

```
## [1] 1e-06
```

Pour forcer R a afficher une valeur, on peut utiliser la commande `print`. Par exemple,

```
print(a)
```

```
## [1] 0.1
```

affichera toujours la valeur contenue dans `a`. Cette commande est importante si vous utilisez des fichiers R *via* la commande `source` (*cf.* séance suivante) ; dans ce cas, `print` est le seul moyen de forcer l'affichage d'une valeur.

Chaînes

Les chaînes permettent de stocker des expressions textuelles. Par exemple,

```
texte <- "Hello, world!"
texte
```

```
## [1] "Hello, world!"
```

R permet de manipuler les chaînes, même s'il n'est pas le choix le plus recommandé dans ce domaine (PERL ou Python font un bien meilleur travail). On peut mentionner quelques commandes simples :

Pour coller plusieurs chaînes entre elles,

```
paste("Hello", "World!")
```

```
## [1] "Hello World!"
```

```
paste("Hello", "World!", sep = ", ")
```

```
## [1] "Hello, World!"
```

On peut aussi couper des chaînes sur un motif donnée, par exemple, pour connaître l'extension d'un fichier :

```
filename <- "mon_fichier.dat"
strsplit(filename, "\\.")
```

```
## [[1]]
## [1] "mon_fichier" "dat"
##
```

```
strsplit(filename, "_")
```

```
## [[1]]
## [1] "mon"         "fichier.dat"
##
```

Le fait d'écrire
. et pas simplement . vient du fait que R peut utiliser des expressions régulières pour la découpe des chaînes, par exemple couper au premier chiffre, et que . a dans ce contexte un sens différent du caractètre ..

Booléens

Conversions

Collections de valeurs

Vecteurs et vectorisation

Le vecteur est, avec la matrice, l'objet le plus important de R. R est un langage dit *vectorisé*, c'est-à-dire qui peut traîter plusieurs valeurs regroupées dans un objet unique. Si on utilise une commande très simple, comme

```
2
```

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

on remarque que la sortie est `[1] 2`. L'indicateur `[1]` indique que la valeur retournée est le premier élément d'un vecteur. La puissance de la notation vectorielle est qu'on peut accéder à une partie du vecteur, avec un *indice*. Si on prend l'exemple suivant,

```
a <- 2
a[1]
```

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

```
a[2]
```

```
## [1] NA
```

, accéder à la position `1` *via* l'*indice* `[1]`, on récupère la première valeur du vecteur `a`. Voilà une des particularités de R : tout objet est un vecteur ! Accéder à la position `[2]` retourne `NA`, parce que le vecteur `a` ne possède pas de deuxième position.

On peut créer des vecteurs dans R en utilisant la commande `c`.

```
vecteur <- c(1, 2, 3, 4, 5)
```

R propose différents raccourcis pour créer rapidement des vecteurs. Par exemple, examinez le comportement des commandes suivantes :

```
seq(from = 0, to = 5, by = 1)
```

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

```
seq(from = 0, to = 10, length = 3)
```

```
## [1]  0  5 10
```

```
c(0:5)
```

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

L'avantage de la vectorisation est que R va automatiser une grande partie des opérations sur les vecteurs. Par exemple, examinez l'effet des commandes suivantes :

```
a <- c(1:10)
a/2
```

```
##  [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
log(a, 10)
```

```
##  [1] 0.0000 0.3010 0.4771 0.6021 0.6990 0.7782 0.8451 0.9031 0.9542 1.0000
```

```
a * a
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Avec les vecteurs vient le concept important de *recyclage*. Le recyclage consiste a répéter un vecteur autant de fois que nécessaire pour le rendre compatible avec un autre vecteur dans le cadre d'une opération. Par exemple, les commandes

```
c(1, 2, 3, 4, 5) + c(1, 2)
```

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

et

```
c(1, 2, 3, 4, 5) + c(1, 2, 1, 2, 1)
```

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

sont équivalentes. Le vecteur `c(1,2)` du premier exemple est *recyclé* pour atteindre la longueur du premier vecteur.

La taille d'un vecteur est obtenue en utilisant la fonction `length` :

```
length(1)
```

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

```
length(c(1:5))
```

```
## [1] 5
```

todo Créer des vecteurs null

Matrices

Le type matrice est central dans le fonctionnement de R. Une fois le principe des vecteurs compris, le fonctionnement des matrices est assez intuitif. Une matrice est en fait un vecteur a deux dimensions. Si un vecteur est une ligne, une matrice est une série de lignes et de colonnes. Dans R, on peut créer une matrice avec la commande `matrix`, de sorte que

```
test_mat <- matrix(0, ncol = 2, nrow = 4)
```

renvoie une matrice pleine de 0, avec 2 colonnes (`ncol`) et 4 lignes (`nrow`).

On peut connaître les dimensions d'une matrice de différentes manières :

```
dim(test_mat)
```

```
## [1] 4 2
```

```
nrow(test_mat)
```

```
## [1] 4
```

```
ncol(test_mat)
```

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

Accéder à une position particulière d'une matrice se fait en deux temps. D'abord, par le numéro de la ligne, ensuite par le numéro de la colonne. Par exemple, on peut fixer l'élément sur la première ligne, deuxième colonne de `test_mat` à 2, avec

```
test_mat[1, 2] <- 2
test_mat
```

```
##      [,1] [,2]
## [1,]    0    2
## [2,]    0    0
## [3,]    0    0
## [4,]    0    0
```

Dans certaines situations, on peut souhaiter avoir accès à une ligne ou une colonne en particulier. R permet donc de ne spécifier qu'un numéro de ligne, ou un numéro de colonne :

```
test_mat[, 2]
```

```
## [1] 2 0 0 0
```

```
test_mat[1, ]
```

```
## [1] 0 2
```

Une matrice peut aussi posséder des noms de lignes et de colonnes :

```
colnames(test_mat) <- c("a", "b")
rownames(test_mat) <- c("A", "B", "C", "D")
test_mat
```

```
##   a b
## A 0 2
## B 0 0
## C 0 0
## D 0 0
```

```
colnames(test_mat)
```

```
## [1] "a" "b"
```

Cela permet aussi d'accéder plus facilement a certaines positions de la matrice :

```
test_mat["A", ]
```

```
## a b
## 0 2
```

```
test_mat["A", "b"]
```

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

Listes

Un des derniers types d'objets qu'il faut connaître est les listes. Une liste, dans R, est une manière de stocker de l'information venant de source diverses, pour y accéder facilement. On verra dans la séance 3 que les listes permettent aussi de traiter très rapidement plusieurs jeux de données à la suite.

Pour créer une liste, on peut utiliser différentes méthodes :

```
n_list <- list(a = 1, b = 2, c = 3)
n_list
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
##
```

```
u_list <- list(1, 2, 3)
u_list
```

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

Les listes, comme `n_list`, peuvent être nommées :

```
names(n_list)
```

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

On peut accéder aux éléments des listes de différentes manières. Si la liste est nommée, la notation `liste$nom` est possible ; dans tous les autres cas, `liste[[indice]]` fonctionne.

```
n_list$a
```

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

```
n_list[[1]]
```

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

Les listes peuvent être utilisées dans le contexte de la structure `with`, qui permet d'accéder facilement aux différents éléments. Pour simplifier, `with` permet d'éviter d'écrire `liste$nom` pour n'écrire que `nom` ; les deux commandes ci-dessous sont donc équivalentes :

```
n_list$a + n_list$b - n_list$c
```

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

```
with(n_list, {
    a + b - c
})
```

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

Dans certains cas, il peut être intéréssant d'applatir une liste en un vecteur. R propose la fonction `unlist` pour effectuer cette opération :

```
t_list <- list(1, 2, 3, 4, 5, 6)
t_list
```

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

```r
unlist(t_list)
```

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

Data frames

Vu plus en détail dans les autres séances.

Séance

2

Lecture et écriture des données

R propose plusieurs manières de lire des données, depuis des fichiers textes ou des tableaux Excel. L'objectif de cette séance est de lire, et mettre en forme des données. Nous aborderons aussi les moyens de sauvegarder ces données sur le disque. Pour la durée de la séance, on suppose que l'ensemble des données qu'on veut lire sont stockées dans le répertoire `./data/`.

2.1 Lecture de données

La méthode la plus simple de stocker des données, et la seule que l'on devrait recommander si on veut s'assurer de pouvoir lire les données partout, en tout temps, est d'utiliser des fichiers texte. À la différence d'un fichier produit par *Excel* ou *OpenOffice Calc*, un fichier en texte brut ne contient pas d'autre informations que ce qu'on y a entré. Il est possible de lire dans n'importe quel programme, et son format ne changera *jamais* – sans mentionner que sa lecture ne coûte rien...

Depuis des fichiers textes

Dans cet exemple, on utilisera les données prises par Poisot et Desdevises [2] sur 147 parasites du genre *Lamellodiscus*, parasites de poissons communs en Méditerranée. Ces données correspondent aux relevés morphométriques effectuées sur les parties solides de l'appareil d'attachement. Les données sont classées selon l'espèce du parasite (`sppar`), et l'espéce de l'hôte sur lequel le parasite a été isolé (`sphote`).

```
morpho <- read.table("data/lamellodiscus.txt", h = TRUE, sep = "\t")
head(morpho)
```

```
##   sphote sppar      para    a    b    c    d    f    g   aa   bb   cc   lm
## 1   Divu  eleg elegDivu1 2.06 1.93   NA   NA   NA   NA 1.93 1.81 1.16 5.43
## 2   Divu  eleg elegDivu2 1.93 1.82 1.51 0.41 0.57 0.20 1.77 1.67 1.20 2.35
## 3   Divu  eleg elegDivu2 1.67 1.56 1.20 0.31 0.36 0.26 1.56 1.51 0.94 1.88
## 4   Disa  eleg elegDisa1 1.46 1.41 1.04 0.47 0.67 0.31 1.25 1.20 0.83 1.46
## 5   Disa  eleg elegDisa1 1.30 1.25 0.94 0.36 0.52 0.31 1.14 1.09 0.78   NA
## 6   Disa  eleg elegDisa1 1.41 1.35 0.99 0.36 0.57 0.36 1.25 1.20 0.83 2.08
```

```
##     li
## 1 2.66
## 2 2.66
## 3 1.77
## 4 2.19
## 5 2.03
## 6 2.55
```

2.2 Écriture de données

2.3 Bases de données

Séance

3

Opérations sur les tables de données

3.1 Travail sur les lignes et colonnes

Dans une grande variété de situations, il peut être avantageux de répéter une opération sur toutes les lignes, ou toutes les colonnes. R propose une fonction pour automatiser ce traitement, *via* la fonction `apply`.

```
dat <- matrix(rnorm(100), nrow = 10)
apply(dat, 1, mean)
```

```
##  [1] -0.01217 -0.18201  0.23044  0.18467  0.22523  0.55003  0.24744
##  [8] -0.23401  0.10894 -0.17638
```

```
apply(dat, 2, var)
```

```
##  [1] 0.2502 1.2307 0.3229 1.5699 1.0013 0.4071 1.3755 0.8582 1.2821 0.4935
```

3.2 Traitement des données

3.3 Division et traitement par niveau

En utilisant différentes fonctions, on peut traiter facilement un jeu de données par «niveaux» d'un facteur (p.ex. traitement expérimental). En rechargeant les données *Lamellodiscus*, on peut par exemple chercher à connaître la moyenne et la variance de la taille de chaque pièce sclérifiée.

```
morpho <- read.table("data/lamellodiscus.txt", h = TRUE, sep = "\t")
```

L'étape suivante est de diviser les données, en utilisant la fonction `split`. Cette fonction prend une `data.frame`, la divise selon les valeurs de la colonne (ou combinaison de colonnes) choisie, et renvoie les sous-tableaux sous forme de liste.

```
morpho_split <- split(morpho, morpho$sppar)
names(morpho_split)
```

```
##  [1] "conf" "dipl" "eleg" "erge" "falc" "frat" "furc" "igno" "kech" "morm"
## [11] "neif" "ther" "tome"
```

```
morpho_split$conf
```

```
##     sphote sppar       para     a     b      c      d      f      g     aa
## 153   Sasa  conf Sasa1conf1 1.300 1.250 0.5500 0.3000 0.3500 0.2000 1.2000
## 188   Disa  conf Disa9conf1 1.204 1.244 0.7498 0.7443 0.6972 0.3702 0.9759
##         bb     cc    lm    li
## 153 1.1000 0.6500 1.300 1.300
## 188 0.9007 0.6713 1.173 1.144
```

On obtient 13 tableaux de données, un pour chaque espèce de parasites. On souhaite éliminer ceux qui ont été observés moins de trois fois au total. Ceci implique de parcourir chaque élément de la liste, et de déterminer sa taille. R propose une fonction `lapply`, littéralement `apply` sur une `liste`, pour effectuer cette tâche :

```
n_obs <- unlist(lapply(morpho_split, nrow))
n_obs >= 3
```

```
##  conf  dipl  eleg  erge  falc  frat  furc  igno  kech  morm  neif  ther
## FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
##  tome
##  TRUE
```

Notons que `lapply` retourne une liste. On peut ensuite utiliser les infomations sur le nombre d'observations, `n_obs`, pour choisir quels sous-tableaux garder :

```
morpho_split <- morpho_split[n_obs >= 3]
```

la encore, on remarquera que pour exclure certains éléments d'une liste, on utilise les crochets simples, comme pour un vecteur, et non les crochets doubles. On vérifie maintenant qu'il ne reste plus que des espèces avec plus de 3 observations :

```
unlist(lapply(morpho_split, nrow))
```

```
## eleg erge falc frat furc igno kech neif tome
##   59   19    9    6    7   43   30    6    3
```

On veut maintenant calculer la moyenne des éléments de chaque sous-tableau, en ne sélectionnant que les colonnes correspondant aux mesures morphométriques. Ces colonnes sont les 4 et suivantes, soit `c(4:ncol(x))` dans le langage de R, si on travaille sur un objet `x`. Une fois ces colonnes extraites, on peut vérifier qu'on obtient bien une matrice,

```
morpho_split$furc[, c(4:ncol(morpho_split$furc))]
```

```
##         a     b     c      d      f      g    aa    bb     cc    lm    li
## 23  2.030 1.930 1.250 0.5200 0.7300 0.3100 1.880 1.770 1.2000 2.350 2.760
## 140 2.100 1.850 1.500 0.6000 0.7000 0.4000 1.800 1.700 1.2000 1.900 2.700
## 141 2.000 1.900 1.300 0.6000 0.7000 0.3000 1.700 1.600 1.2000 1.800 2.800
## 142 1.900 1.850 1.450 0.6000 0.7500 0.4000 1.850 1.700 1.1000 1.950 2.600
## 175 1.905 1.843 1.305 0.6972 0.7975 0.4873 1.745 1.682 1.1482 2.212 2.621
## 176 1.813 1.691 1.240 0.6617 0.7786 0.5374 1.579 1.477 0.9888 1.693 2.187
## 187 2.031 1.939 1.420 0.7219 0.8022 0.4668 1.832 1.762 1.2685 2.007 2.680
```

, dont on peut calculer la moyenne sur chaque colonne par la fonction `apply`.

```
moy <- function(x) apply(x[, c(4:ncol(x))], 2, mean, na.rm = TRUE)
```

On peut maintenant appliquer cette fonction à nos données divisées en groupes :

```
lapply(morpho_split, moy)
```

```
## $eleg
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.7070 1.6101 1.1589 0.5191 0.5957 0.3475 1.5249 1.4336 0.9794 1.9543
##     li
## 2.2411
##
## $erge
##      a      b      c      d      f      g     aa     bb     cc     lm
## 2.0589 1.9603 1.3527 0.7425 0.9609 0.5216 1.7363 1.6743 0.9852 2.6796
##     li
## 2.5110
##
## $falc
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.4151 1.2760 0.6742 0.5014 0.4512 0.2660 1.1984 1.1344 0.7695 1.4033
##     li
## 1.3521
##
## $frat
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.6333 1.4667 1.0333 0.4833 0.6083 0.2917 1.3333 1.1583 0.9333 1.2000
##     li
```

```
## 1.5000
##
## $furc
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.9684 1.8575 1.3522 0.6287 0.7512 0.4145 1.7694 1.6700 1.1579 1.9873
##     li
## 2.6212
##
## $igno
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.1418 1.0682 0.6863 0.4650 0.5244 0.2613 1.0015 0.9159 0.5487 1.6268
##     li
## 1.4836
##
## $kech
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.8057 1.7346 1.1192 0.5808 0.8611 0.4013 1.5480 1.4304 0.7800 1.8590
##     li
## 1.8269
##
## $neif
##      a      b      c      d      f      g     aa     bb     cc     lm
## 1.0529 0.9435 0.6178 0.4035 0.4394 0.2495 0.8778 0.8019 0.4958 0.7910
##     li
## 1.0076
##
## $tome
##      a      b      c      d      f      g     aa     bb     cc     lm
## 2.2833 2.1833 1.2833 0.8000 1.1500 0.3833 1.8167 1.8333 0.9167 2.8167
##     li
## 3.0833
##
```

On peut aussi convertir facilement cette information en une `data.frame`, que l'on pivote pour avoir les noms des espèces en lignes :

```
t(as.data.frame(lapply(morpho_split, moy)))
```

```
##          a      b      c      d      f      g     aa     bb     cc    lm
## eleg 1.707 1.6101 1.1589 0.5191 0.5957 0.3475 1.5249 1.4336 0.9794 1.954
## erge 2.059 1.9603 1.3527 0.7425 0.9609 0.5216 1.7363 1.6743 0.9852 2.680
## falc 1.415 1.2760 0.6742 0.5014 0.4512 0.2660 1.1984 1.1344 0.7695 1.403
## frat 1.633 1.4667 1.0333 0.4833 0.6083 0.2917 1.3333 1.1583 0.9333 1.200
## furc 1.968 1.8575 1.3522 0.6287 0.7512 0.4145 1.7694 1.6700 1.1579 1.987
## igno 1.142 1.0682 0.6863 0.4650 0.5244 0.2613 1.0015 0.9159 0.5487 1.627
## kech 1.806 1.7346 1.1192 0.5808 0.8611 0.4013 1.5480 1.4304 0.7800 1.859
```

```
## neif 1.053 0.9435 0.6178 0.4035 0.4394 0.2495 0.8778 0.8019 0.4958 0.791
## tome 2.283 2.1833 1.2833 0.8000 1.1500 0.3833 1.8167 1.8333 0.9167 2.817
##         li
## eleg 2.241
## erge 2.511
## falc 1.352
## frat 1.500
## furc 2.621
## igno 1.484
## kech 1.827
## neif 1.008
## tome 3.083
```

Séance

4

Introduction à la programmation

Dans les séances précédentes, nous avons utilisé des fichiers `.R` pour sauvegarder des listes d'instructions. Nous avons aussi chargé et manipulé des jeux de données. Il est souvent nécessaire d'automatiser tout ou partie de ce processus, ce qui implique de faire appel à de la programmation.

L'objectif de cette séance est de se familiariser avec les principaux concepts qu'on utilise pour concevoir un programme. La première partie couvre les bases en algorithmie, c,-à-d. les boucles et les tests. La deuxième partie concerne les fonctions, leur définition dans R, et leur utilisation. À l'issue de cette séance, vous serez en mesure de vous attaquer a pratiquement tous les problèmes nécéssitant de manipuler des données. Tout programme que vous aurez a écrire ne sera qu'une combinaison plus ou moins complexe des éléments abordés dans les séances précédentes.

4.1 Boucles

Les boucles permettent parcourir une liste, ou de répéter une série d'instructions, dans des conditions bien définies ; c'est une des structures de base de l'algorithmique. R propose deux types de boucles, les boucles `for` et les boucles `while`. En français, on peut les résumer par «pour chaque» et «tant que».

Boucles de type *for*

Une boucle `for` permet de répéter un bloc d'instructions un nombre prédéfini de fois, ou d'éxécuter des commandes pour chaque élément d'un tableau de données. La syntaxe de base est la suivante :

```
for (step in c(1:10)) cat(step)
```

```
## 12345678910
```

En clair, pour chaque valeur entre 1 et 10, qu'on nomme `step` (mais qu'on aurait pû nommer n'importe comment), on affiche (`cat`) la valeur de step. On peut bien sûr spécifier plusieurs instructions qui doivent être éxécutées à chaque *itération* (étapes de la boucle) en utilisant les accolades :

```
for (step in c(1:3)) {
    cat(step)
    print(summary(rnorm(100, mean = step)))
}
```

```
## 1   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -1.990   0.260   0.948   0.834   1.510   3.130
## 2   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -0.20    1.43    2.20    2.13    2.79    4.99
## 3   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.969   2.150   2.850   2.950   3.670   6.130
```

Cette commande affiche le numéro de l'itération en cours (`cat(step)`), puis affiche les information de base (`summary`) sur une distribution normale (`rnorm`) centrée sur `step`. Les boucles `for` peuvent contenir des instructions aussi longues que souhaité.

Une autre application des boucles `for` est de parcourir un objet. Par exemple, on peut souhaiter, pour chaque élément d'un objet, afficher sa valeur. R permet de réaliser ce genre d'opérations, avec la syntaxe suivante :

```
vect <- c("a", "b", "c", "d")
for (val in vect) cat(val)
```

```
## abcd
```

Pour chaque élément du vecteur `vect`, que l'on nomme `val` pour pouvoir y accéder pendant les itérations, R va afficher la valeur que l'élément contient.

Les boucles `for` sont donc une façon explicite de faire la même chose que ce qui a été traité dans la séance précédente, avec un inconvénient majeur : elles sont extrèmement demandantes en temps de calcul. Dans la majorité des cas, il est préférable d'avoir recours autant que possible aux fonctions `*apply`, qui sont bien plus optimisées que des boucles.

Boucles de type *while*

Les boucles de type `while`, littéralement *pendant*, permettent de répéter une série d'instructions tant qu'une condition n'a pas été atteinte. Pour cette raison, il faut bien prendre en compte le fait que mal utilisées, ces boucles peuvent ne jamais stopper. Il faut donc faire particulièrement attention à la condition qui est évaluée à chaque itération.

Un exemple simple d'utilisation d'une boucle `while` est le calcul d'une factorielle. On veut calculer $n!$, ce qui se fait simplement en multipliant l'ensemble des $1 \leq k \leq n$.

```
n <- 5
k <- n
while (k > 1) {
    k <- k - 1
    n <- n * k
}
```

```
print(n)
```

```
## [1] 120
```

On remarquera que dans la parenthèse après `while` se trouve un test logique ; les tests sont abordés dans la partie suivante.

Sortir d'une boucle et sauter des étapes

Lors de l'éxécution d'une boucle, on peut ne pas vouloir éxécuter toutes les instructions. R possède des structures de contrôle pour effectuer ce type d'opérations. Les principales qu'il faut connaître sont `break` et `next`. L'instruction `break` permet de stopper l'éxécution de la boucle, c'est-à-dire de sortir de la boucle comme si la condition de sortie était remplie, ou le nombre maximal d'itérations atteint. La boucle suivante va afficher les valeurs de `i` tant qu'elles sont inférieures à 3, et afficher `bye!` puis stopper la boucle sinon.

```
for (i in c(1:5)) {
    if (i < 3) {
        cat(i)
    } else {
        cat(" bye!")
        break
    }
}
```

```
## 12 bye!
```

Une autre structure de contrôle intéressant est `next`, qui permet de sauter une itération si la valeur de l'itérateur ne nous plaît pas. Cette structure est particulièrement utile quand on réalise un nombre important d'opérations à chaque itération, et qu'on ne veut pas perdre de temps a traiter des valeurs qui ne nous intéressent pas. On peut utiliser `next` pour avoir, par exemple, une liste de tous les nombres pairs entre 1 et 10 :

```
is.even <- function(x) x%%2 == 0
for (i in c(1:10)) {
    if (is.even(i)) {
        cat(i)
        cat(" is even\n")
    } else {
        next
    }
}
```

```
## 2 is even
## 4 is even
## 6 is even
```

```
## 8 is even
## 10 is even
```

4.2 Tests

Cette partie est consacrée à l'utilisation des tests. On a vu dans les séances précédentes l'existence de variables de type booléen, qui prennent les valeurs `TRUE` ou `FALSE`. Des variables de ce type sont utilisées dans le cadre d'expressions conditionelles, c'est-à-dire quand on souhaite effectuer différentes instructions en fonction de la valeur d'une condition.

Expressions conditionnelles

La structure de base d'une expression conditionnelle est la suivante :

```
if (condition) {
    instruction(1)
} else {
    instruction(2)
}
```

Une notation comme

```
if (condition) instruction
```

est aussi acceptable, de même que

```
val <- ifelse(condition, valeur_if, valeur_else)
```

Cette dernière notation permet de gagner du temps quand on veut que la valeur d'une variable dépende d'une condition. Par exemple, $(n \geq 2) \vee (n = 0)$ s'écrit `(n <= 2) or (n == 0)`, ou encore `(n <= 2) | (n == 0)`. Les différents opérateurs logiques sont regroupés dans le tableau 4.1.

L'argument `condition` prend la forme d'un test logique, qui peut être d'une complexité aussi grande que l'on veut. Pour certains des opérateurs, il existe deux variantes (`|` et `||`, `&` et `&&`). On peut comprendre pourquoi avec les exemples suivants :

```
a <- c(1, 2, 3, 4, 5, 6)
b <- c(1, 2, 3, 4, 4, 6)
d <- c(3, 2, 3, 4, 5, 4)
(a == b)
```

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

```
(a == d)
```

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

Opérateur	Signification	Version 1	Version 2
$\vee$	ou (au moins une des deux)	`or`	`|, ||`
$\wedge$	et (les deux)	`and`	`&, &&`
$\neg$	non	`not`	`!`
$\oplus$	ou conditionnel (seulement une des deux)	`xor(a, b)`	
$\in$	est compris dans	`%in%`	
$=$	égalité		`==`
$\leq$	inférieur ou égal		`<=`
$\geq$	supérieur ou égal		`>=`
$>$	supérieur		`>`
$<$	inférieur		`<`

Table 4.1: Différents opérateurs logiques disponibbles dans R.

```
(b == d)
```

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

```
(a == d) & (a == b)
```

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

```
(a == d) && (a == b)
```

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

```
(a == d) | (a == b)
```

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

```
(a == d) || (a == b)
```

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

```
all.equal(a, b)
```

```
## [1] "Mean relative difference: 0.2"
```

L'opérateur répété une seule fois travaille par élément : les éléments des vecteurs sont comparés deux à deux (et on applique du recyclage). L'opérateur répété deux fois ne prend en compte que les premières valeurs des vecteurs (un message d'avis sera émis si le vecteur a une taille supérieure à 1).

Manipulation des valeurs booléennes

Comme tous les autres types de données de R, les valeurs booléennes peuvent être manipulées dans des additions.

```
TRUE + TRUE
```

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

```
FALSE + TRUE
```

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

```
TRUE * TRUE
```

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

Cette possibilité de travailler sur les valeurs booléennes comme si `TRUE = 1` et `FALSE = 0` permet des raccourcis intéréssants. Par exemple, on peut facilement connaître le nombre d'éléments d'un vecteur qui répondent à une condition particulière, ici $x \in (-0.5; 1]$:

```
a <- rnorm(100)
sum(a[(a < 1) & (a >= 0.5)])
```

```
## [1] 14.26
```

```
sum((a < 1) & (a >= 0.5))
```

```
## [1] 19
```

Dans l'exemple précédemt, la première ligne fait la somme de `a` aux indices satisfaisant la condition fixée. La deuxième ligne fait la somme de ces indices, c.-à-d. d'un vecteur de booléens.

4.3 Fonctions

Généralités

L'utilisation des fonctions va permettre de gagner du temps dans la programmation. Comprendre le principe des fonctions dépasse de beaucoup le cadre de R, et mérite qu'on s'y arrête. Qu'est-ce qu'une fonction ? Une série d'instructions qui vont, à partir d'arguments, renvoyer un résultat. En quoi est-ce différent des scripts que nous avons utilisé jusqu'ici ? Écrire une fonction revient en quelque sorte a 'expliquer' le code une fois, et R se charge ensuite de redonner la bonne valeur aux arguments.

Le parallèle le plus évident est celui des fonctions mathématiques : si $f(x) = x^2$, on peut calculer $f(x)$ pour tout x, parce qu'on sait quoi faire. Ça devient donc très avantageux si on doit calculer $f(x)$ un grand nombre de fois. En écrivant uniquement des scripts, pour calculer la valeur de beaucoup de x^2, on aurait écrit :

```
1^2
2^2
3^2
4^2
5^2
```

ou encore

```
for (i in c(1:5)) i^2
```

Si il faut revenir sur ce code plus tard, et transformer tous les `^2` en `^3`, la première solution implique de tout corriger manuellement. En utilisant une fonction, la logique est différente :

```
f <- function(x) x^2
f(2)
```

```
## [1] 4
```

Peut importe le nombre de fois ou on devra effectuer l'opération contenue dans `f`, si on veut la modifier, elle sera toujours stockée au même endroit. Dans la pratique, la majorité des instructions utilisées jusau'à présent sont des fonctions. Par exemple, `print`, `read.table`, et `apply` sont des fonctions rendues disponibles par R. Écrire une instruction comme `read.table('file.txt', h=TRUE)`, c'est faire appel à une fonction, en lui spécificiant certains *arguments*. L'objectif des parties suivantes est de comprendre comment les fonctions fonctionnent (!), afin de pouvoir en écrire de nouvelles.

Les fonctions existent comme un espace «à part» dans R : ce qui se déroule dans une fonction, reste dans une fonction. Prennons le cas du code suivant :

```
a <- 2
b <- 3
f <- function(x) {
    y <- x + a
    z <- y
    return(z)
}
f(b)
```

```
## [1] 5
```

La fonction `f` peut aller chercher la valeur de `a` dans l'environnement global, mais tout ce qui est défini au sein de `f` est innaccessible. D'ailleurs, tout ce qui est créé dans la fonction est détruit – retiré de la mémoire – une fois que la dernière instruction est éxécutée. Voyez la section sur la fonction `return` pour plus de détails. Cette notion de quel objet est accessible est extrèmement importante à maîtriser.

Dans R, les objets existent dans deux «mondes», l'environnement global, et l'intérieur de chaque fonction. L'intérieur de chaque fonction peut avoir accès aux objets et aux variables de l'environnement global, même si cette pratique est à éviter pour différentes raisons (une variable globale peut

être modifiée entre deux appels à la fonction, notamment). En revanche, l'environnement global n'a pas accés aux objets crées ou modifiés à l'intérieur d'une fonction. La communication avec les fonctions se fait ...

Le fait qu'on puisse passer des objets d'un environnement à l'autre peut entraîner un comportement assez surprenant. Dans l'exemple suivant :

```
a <- 2
print(a)

## [1] 2

f <- function(a) {
    a <- a + 1
    return(a)
}
print(f(a))

## [1] 3

print(a)

## [1] 2
```

l'appel à la fonction `f` devrait modifier `a`, puisque la seule instruction de cette fonction est `a = a+1`. Or, quand on appelle cette fonction, puis qu'on affiche la valeur de `a`, elle n'a pas changé. Ce comportement vient du fait que l'objet `a` qui existe dans la fonction n'est pas celui qui existe dans l'environnement global. Par conséquent, le `a` de la fonction peut être modifiè, sans que cela n'affecte le `a` de l'environnement de travail.

Cet example illustre aussi pourquoi le nom des variables est important. On sait que R est capable d'aller chercher, depuis une fonction, des variables de l'environnement global. Quelle version de `a` faut-il aller chercher ? Pour éviter les erreurs liées au fait que plusieurs variables aient le même nom, on essaie de donner un nom unique à toutes les variables. C'est sans doute plus long à écrire – même si cet argument n'est pas valable avec un éditeur qui auto-complète le code –, mais ça évite surtout les erreurs à l'éxécution.

Déclarer une fonction

Comme illustré dans les exemples précédents, la déclaration d'une fonction se fait par

```
nom_de_la_fonction <- function(argument1, argument2) {
    instructions
    return(sortie)
}
```

Les éléments les plus importants sont les arguments et l'instruction `return`.

La commande *return*

La commande `return` est en général la dernière ligne d'une fonction : ce qui se passe après est ignoré. Cette commande va renvoyer ce qui se trouve entre ses parenthèse dans l'environnement global de R : c'est un des points de communication entre l'environnement global et l'environnement de la fonction. La commande `return` ne peut prendre qu'un seul argument, il faut donc regrouper les variables sous forme de liste ou autres si vous en avez plusieurs.

Prennons un exemple simple :

```
somme_1 <- function(a, b) {
    S <- a + b
}
somme_2 <- function(a, b) {
    S <- a + b
    return(S)
}

a <- somme_1(2, 3)
a

## [1] 5

b <- somme_2(2, 3)
b

## [1] 5
```

Quelques précisions. On peut tout a fait nommer des variables `a` et `b` dans l'environnement global, même si les fonctions `somme_1` et `somme_2` utilisent des variables avec ces noms, grâce au fait que le *scope* de ces variables n'est pas le même. Il est toutefois déconseillé de le faire. Dans l'exemple précédent, la variable `S` n'existe pas, à aucun moment, dans l'environnement global de R. Le seul moyen de récupérer ce qui a été renvoyé par les fonctions est de les assigner à une variable de l'environnement global.

Lorsque que la fonction ne se compose que d'une seule ligne, le résultat de cette dernière ligne est renvoyé et il n'y a pas besoin de spécifier `return` :

```
multipl <- function(a, b) a * b
multipl(2, 3)

## [1] 6
```

Arguments

Les arguments sont des variables que l'on peut échanger entre R et l'intérieur d'une fonction. Dans l'exemple précédent, pour faire une somme, il faut additionner deux nombres : $f(a,b) = a+b$ s'écrit `f = function(a,b) a + b`. Les éléments entre parenthèse sont les arguments de la fonction.

Typiquement, un argument est noté par `nom = valeur`, où `nom` est l'identifiant de cet argument *dans la fonction*, et `valeur` est la valeur par défaut.

Les exemples suivants permettent de comprendre le fonctionnement des arguments et des valeurs par défaut.

```
somme <- function(a = 1, b = 1) a + b
somme()
```

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

```
somme(a = 2)
```

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

```
somme(b = 3)
```

```
## [1] 4
```

```
somme(1, 2)
```

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

Il existe un argument particulièrement intéressant dans R : `...`. Cet argument contient tout objet passé à une fonction, pour lequel il n'y pas pas de nom d'argument correspondant. Par exemple :

```
print_message <- function(msg = "Hello", ...) {
    print(msg)
    print(...)
}
print_message(msg = "Hello", "world!")
```

```
## [1] "Hello"
## [1] "world!"
```

L'utilisation de cet argument *catch all* demande un peu de réflexion et beaucoup de pratique, mais s'avère particulièrement utile quand le format exact des arguments qui seront donnés a une fonction n'est pas connu d'avance.

4.4 Mise en application

Test par permutation

Lorsque les données devient de la normalité, on peut préférer réaliser un test paramétrétique avec des permutations plutôt qu'un test non paramétrique. La plupart des programmes de statistique n'offrent pas cette possibilité qui demande pourtant très peu d'efforts pour être implémentée dans R. Dans cette mise en application, on veut effectuer un test t, pour comparer deux distributions, disponibles dans un fichier `s4-data.txt`.

Le principe d'un test par permutations est simple. La première étape est d'effectuer le test sur l'échantillon non permuté, pour obtenir la valeur de la statistique (T). Dans le cas du test t, R propose la fonction `t.test`, et un rapide survol de `?t.test` vous donnera les arguments nécéssaires et la manière de récupérer la statistique. Commencent ensuite les permutations a proprement parler. Pour un nombre n d'itérations choisies (en général 9999), on mélange l'ensemble des valeurs des deux distributions. On reconstruit ensuite, en tirant au hasard dans le *pool* de valeurs ainsi formées, deux distributions de taille égale. Cette étape peut, par example, prendre la forme d'une fonction `resample`, qui prendrait une `data.frame` avec deux colonnes (la valeur, et le groupe d'origine) en argument. R propose la fonction `sample`, qui permettra de mélanger la colonne correspondant au groupe (ce qui recréera automatiquement les deux distributions – économisons nous !). Une fois les deux distributions reconstruites, on calcule la nouvelle statistique T'. Si la valeur de T' est inférieure ou égale à la valeur de T, on incrémente une variable N de 1. Sinon, la valeur de N reste la même.

Le calcul de la *p-value* se fait de la manière suivante :

$$p = \frac{N}{n+1} \tag{4.1}$$

À partir de ces informations, et des informations données dans l'introduction de cette séance, vous devez être en mesure de programmer sans difficultés une fonction `t.test.permut`, qui permet de réaliser un test t par permutations. En bonus, vous pouvez ajouter des arguments qui permettent de contrôler le nombre de permutations qui doivent être réalisées.

4.5 Solution des mises en application

Test par permutation

On commence par écrire une fonction `resample`, qui mélange l'attribution des valeurs à un des deux groupes.

```
resample <- function(df) {
    df$group <- sample(df$group)
    return(df)
}
```

Puis on écrit la fonction `t.test.permut`, qui effectue les permutations à proprement parler

```
t.test.permut <- function(df, n = 9999) {
    baseStat <- t.test(value ~ group, df)$statistic
    N <- 0
    for (repl in c(1:n)) if (t.test(value ~ group, resample(df))$statistic <=
        baseStat)
        N <- N + 1
    return(N/(n + 1))
}
```

On peut générer un jeu de données de test :

```
value <- c(rnorm(100, mean = 0), rnorm(100, mean = 1))
group <- c(rep("a", 100), rep("b", 100))
test_df <- as.data.frame(cbind(value = value, group = group))
test_df$value <- as.numeric(as.vector(test_df$value))
```

puis vérifier que la *p-value* est inférieure à 0.05 :

```
print(t.test.permut(test_df, n = 9))
```

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

Pour éviter les problèmes de lenteur des boucles `for`, on peut écrire la même fonction basée sur la fonction `replicate` :

```
t.test.permut <- function(df, n = 9999) {
    baseStat <- t.test(value ~ group, df)$statistic
    return(sum(replicate(n, t.test(value ~ group, resample(df))$statistic <=
        baseStat))/(n + 1))
}
```

La différence sur 10000 réplicats est assez minime (3 secondes environ). Mais dans un contexte ou il faut analyser plusieurs fois des jeux de données, ces petits écarts finissent par faire une différence importante.

Séance

5

Graphiques

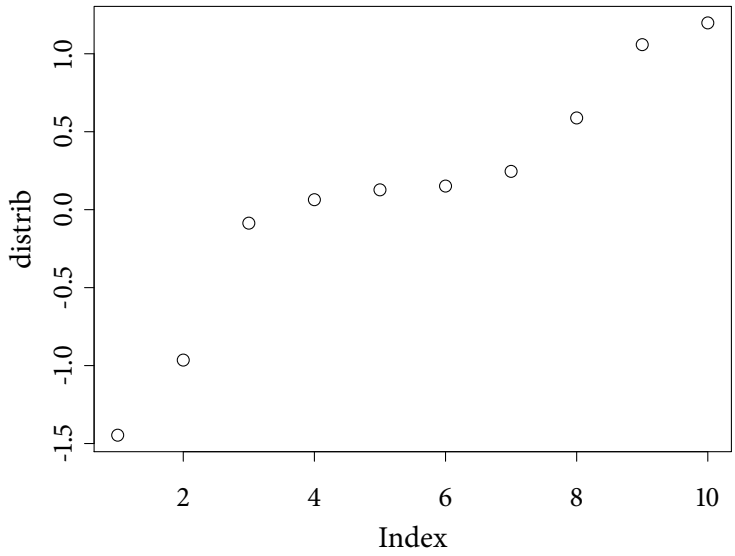
R propose un moteur grahique d'une très grande puissance, qui possède en plus une flexibilité importante. Il est possible, avec un effort minimal, de produire à peu près tous les types de visualisations possibles directement dans R. L'objectif de cette séance est de vous familiariser avec les commandes de base disponibles dans R par défaut. Les personnes à la recherche de solutions toutes faites pour visualiser des données complexes peuvent aller voir la documentation des *packages* `ggplot2` (intuitif à utiliser, assez lent) ou `lattice` (utilisation complexe, assez rapide).

5.1 Principaux types de visualisations

Nuages de points

La manière la plus simple de représenter un objet dans R est d'utiliser la fonction `plot`. Par exemple,

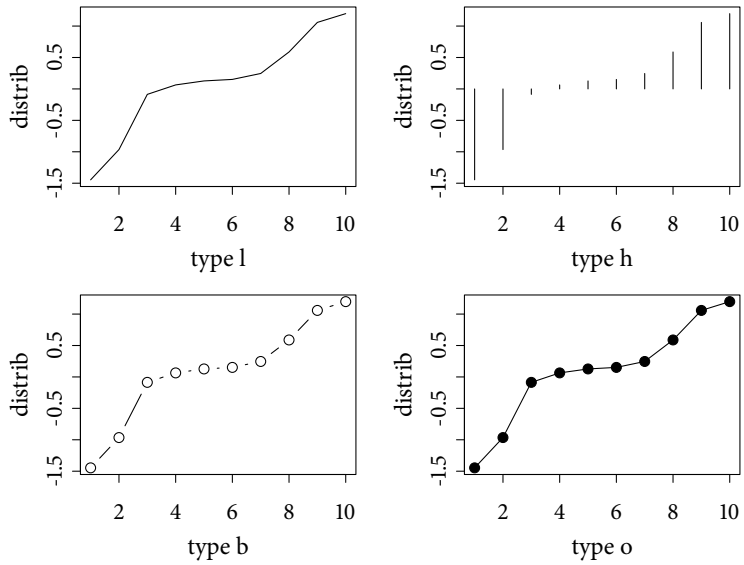
```
distrib <- sort(rnorm(10))
plot(distrib)
```



R va prendre en charge le calcul d'une grande partie des paramètres nécéssaires à la visualisation, comme par exemple les limites des différents axes, et l'espacement entre les valeurs sur les axes. Il est possible de manuellement spécifier l'ensemble de ces paramètres. Par exemple, on peut vouloir changer les étiquettes des axes x et y par quelque chose de plus explicite.

R permet aussi de choisir comment visualiser ces données, en changeant la valeur de l'argument `type` :

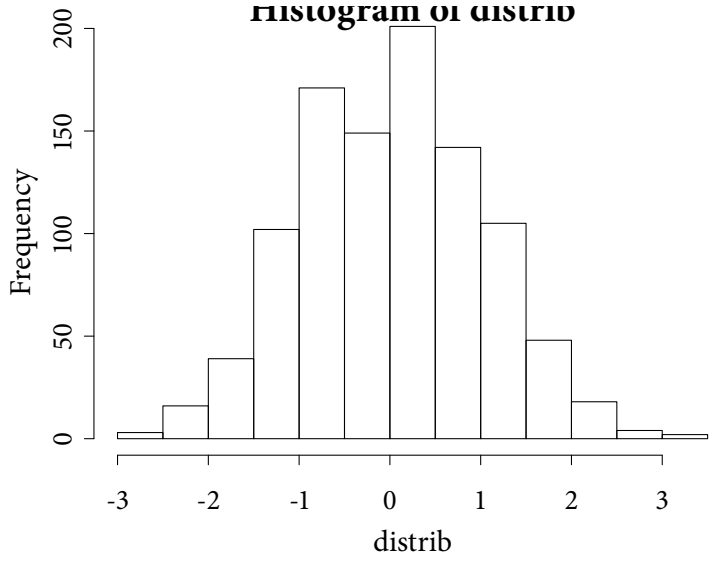
```
par(mfcol = c(2, 2))
plot(distrib, type = "l", xlab = "type l")
plot(distrib, type = "b", xlab = "type b")
plot(distrib, type = "h", xlab = "type h")
plot(distrib, type = "o", pch = 19, xlab = "type o")
```



Histogrammes

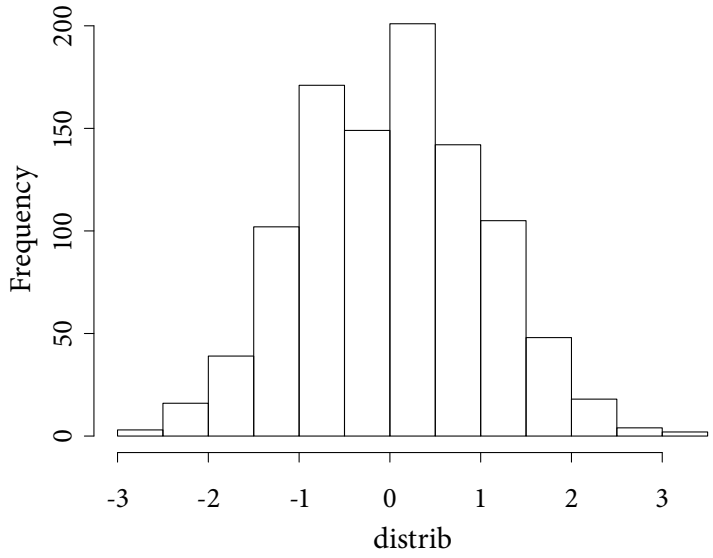
R offre la possibilité de représenter facilement des distributions, *via* des commandes particulières. La plus simple d'utilisation est `hist`, qui permet de représenter un histograme.

```
distrib <- rnorm(1000)
hist(distrib)
```



Par défaut, R attribue un titre à ces graphiques ; on peut supprimer ce titre en mettant l'argument `main` à une valeur nulle.

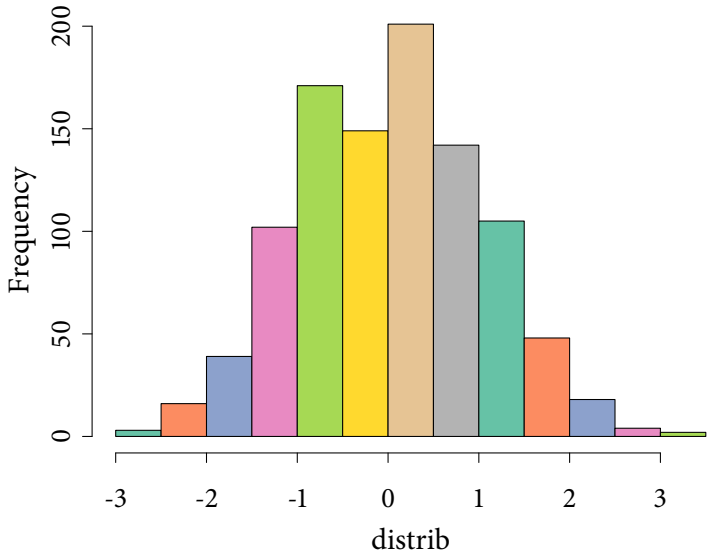
```
hist(distrib, main = "")
```



On peut choisir la couleur des barres, *via* l'argument `col`. Comme dans un grand nombre de situations dans lesquelles on fait appel a des vecteurs, R utilise le recyclage : si le nombre de barres

de l'histogramme est plus grand que le nombre de couleurs fournies, les couleurs des barres sont alternées.

```
hist(distrib, main = "", col = c(1:8))
```



Boxplots

R permet de créer facilement des *boxplots*, qui permettent de visualiser certains propriétés d'une distribution statistique. Il existe deux manières de créer des *boxplots*, dans R, *via* une formule et *via* une liste. *Via* une formule, on spécifie les éléments a visualiser selon la syntaxe `reponse~traitement, data` :

```
boxplot(a ~ sppar, morpho_split, las = 2)
```

```
## Error: arguments imply differing number of rows: 59, 19, 9, 6, 7, 43, 30, 3
```

On notera que l'argument `las` (`?par`) permet de choisir comment les étiquettes des axes sont affichées. Dans le cas des *boxplots*, avoir les étiquettes perpendiculaires permet qu'elles soient toutes affichées. La deuxième façon de spécifier les données est de les passer sous forme de liste. Dans ce cas, on peut aboutir au même résultat que la figure précédente avec :

```
boxplot(split(morpho_split$a, morpho_split$sppar), las = 2)
```

```
## Error: le premier argument doit être un vecteur
```

Diagrammes en barres

5.2 Ajout d'éléments sur un graphique

Autres séries de données

Légendes et axes

Annotations

5.3 Enregistrement des figures

R permet non seulement d'afficher les graphiques dans une fenêtre à part, mais aussi de les enregistrer dans différents formats. La structure générale du code permettant d'enregistrer une figure est toujours la même :

```
open_device(file = "file.extension")
plot(my_data)
dev.off()
```

La commande `open_device` peut prendre plusieurs formes selon le type de fichier désiré en sortie. Les plus communes sont sans doute `pdf`, `png`, et `tiff`. Reportez vous à l'aide de chacune de ces fonctions pour comprendre les arguments.

La commande `dev.off()` est extrèmement importante : elle permet de fermer le périphérique graphique actif. Sans cette commande, le fichier n'est pas fini d'écrire, et il ne pourra pas être lu à la fermeture de R. Dans RStudio, quel est l'effet de la commande `dev.off()` après qu'un graphique ait été affiché ?

5.4 Mise en application

Diagramme en barres

Dans cette mise en application, on veut créer une visualisation qui met en avant les valeurs extrèmes d'une distribution, en utilisant un diagramme en barre. Spécifiquement, on souhaite que les barres correspondant à des valeurs plus petites, ou plus grandes, que des valeurs fixées, soient colorées différement. En vous aidant de ce qui a été vu jusqu'ici, et de l'aide de la fonction `barplot`, produisez cette visualisation.

5.5 Solution des mises en application

Diagramme en barres

On souhaite représenter une diagramme en barres, et colorer les barres qui sont au dessous ou au dessus de valeurs données. On commence par choisir la série de données qui nous intéresse – on peut la trier en ordre croissantm, pour faciliter la lecture du graphique :

```
test_data <- c(1, 4, 3, 12, -2, -6, -1, 1)
test_data <- sort(test_data)
```

Pour colorer chaque barre de manière indépendante, on va créer un vecteur `colors`, qui contiendra une valeur par barre :

```
colors <- rep(2, length(test_data))
```

Chaque barre sera donc de la couleur `2`, qui dépend de la `palette` actuelle. Pour changer la couleur de chaque barre, il y a deux approches. La première consiste a écrire une boucle :

```
for (co_idx in c(1:length(test_data))) {
    if (test_data[co_idx] > 10)
        colors[co_idx] <- 3
    if (test_data[co_idx] < -5)
        colors[co_idx] <- 1
}
```

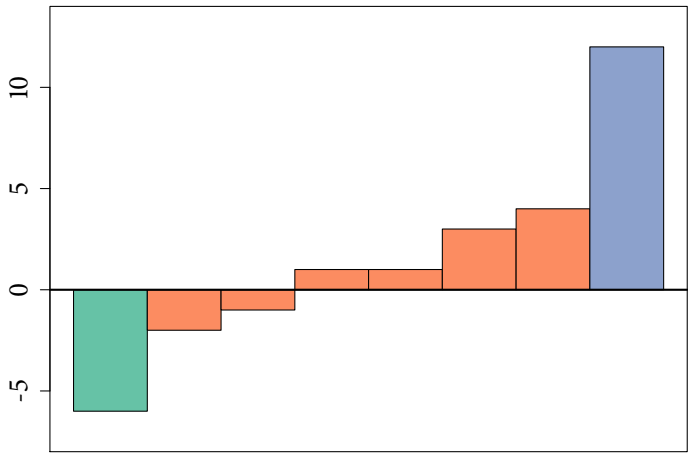
Une méthode plus élégante tire parti de la vectorisation :

```
colors[test_data > 10] <- 3
colors[test_data < -5] <- 1
colors
```

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

On peut ensuite afficher le graphique avec les barres de chaque couleur :

```
barplot(test_data, col = colors, ylim = range(test_data) + c(-2,
    2), space = 0)
abline(h = 0, lwd = 2)
box()
```


10
5
0
-5

Introduction au calcul parallèle

L'objectif de ce chapitre est de fournir des «recettes» sur le calcul parallèle, en utilisant les librairies `snow` et `snowfall`.

Bibliographie

[1] R Development Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2008.

[2] T. Poisot et Y. Desdevises. "Putative speciation events in Lamellodiscus (Monogenea : Diplectanidae) assessed by a morphometric approach". Dans : *Biological Journal of the Linnean Society* 99.3 (fév. 2010), p. 559–569.