Travaux pratiques en langage R

Pierre Legendre
Département de sciences biologiques
Université de Montréal

Octobre, décembre 2004 ; mai, juillet, octobre 2005
janvier, mai, juillet 2006 ; janvier, avril, septembre 2007
avril, août, octobre 2008 ; février à mai 2009 ; septembre 2010 ;
janvier 2011 ; juillet, octobre 2012 ; avril, juin, octobre 2016 ;
janvier, mars, novembre 2017 ; janvier, juin, novembre 2019 ;

0. Bibliothèques R

Installer les bibliothèques R suivantes. Elles seront utilisées dans ces travaux pratiques. —

• Installer les bibliothèques disponibles sur le site CRAN :

install.packages(c("ade4", "adegraphics", "adespatial", "ape", "cclust", "cluster", "FD", "geoR", "labdsv", "mapdata", "maps", "rgl", "spdep", "vegan"), dependencies=TRUE)

• Installer mvpart depuis Github. D'abord, installez la bibliothèque "devtools" si elle n'est pas déjà installée sur votre ordinateur : install.packages("devtools"). Ensuite:

library(devtools)

install_github("cran/mvpart", force=TRUE)

• Téléchargez 'rdaTest' de l'adresse http:/numericalecology.com. Installez cette bibliothèque en suivant les instructions de la section 3 du document "Introduction_R.pdf".

• D'autres fonctions, disponibles sur http:/numericalecology.com, seront utilisées dans le cours.

1. Calcul des statistiques de base en langage R: exemple Merle

```
# Il faut d'abord indiquer à R quel est votre dossier de travail.
# Windows: menu File ⇒ Change dir...   Mac OSX: Divers ⇒ Changer de Répertoire de Travail

# Importer le fichier 'Merle' dans l'objet de type 'data frame' 'merle' :
  merle <- read.table("Merle.txt")          # ou merle = read.table("Merle.txt")
# ou, mieux encore :
  merle = read.table( file.choose() )       # cette commande ouvre une boîte de dialogue

# Vérification pour savoir si les données ont été lues correctement:
  merle

# Saisie des valeurs de la première colonne: la longueur de l'aile.
# La longueur est recopiée dans l'objet 'aile':
  aile = merle[,1]
  aile                                      # Vérification du contenu de l'objet 'aile'
  is.vector(aile)
  is.matrix(aile)

# Transformation du vecteur 'aile' en un objet de type 'matrix' en cas de besoin:
  aile.mat = as.matrix(aile)
  aile.mat                                  # Vérification du contenu de la matrice 'aile.mat'

  is.vector(aile.mat)
  is.matrix(aile.mat)

# 'aile' et 'aile.mat' sont-ils des objets R du même type ?
  class(aile)
  class(aile.mat)
```

```
# Calcul de la moyenne des longueurs d'aile (commande 'mean'):
  moyenne = mean(aile)                    # ou : moyenne = mean(aile.mat)
  moyenne                                 # Vérification de la valeur de la moyenne

# Calcul de la médiane des longueurs d'aile (commande 'median'):
  mediane = median(aile)                  # ou : mediane = median(aile.mat)
  mediane                                 # Vérification de la valeur de la médiane

# Calcul de la variance des longueurs d'aile (commande 'var'):
  variance = var(aile)                    # ou : variance2 = var(aile.mat)

# Vérification de la variance:
  variance                                # ou : variance2
  is.vector(variance)                     # ou : is.vector(variance2)
  is.matrix(variance)                     # ou : is.matrix(variance2)

# Calcul de la taille de l'échantillon 'n':
  n = length(aile)
  n                                       # Vérification de la valeur de 'n'
  n1 = nrow(aile.mat)          # ou : n1 = dim(merle)[1]     # ou : n1 = dim(aile.mat)[1]
  n1                                      # Vérification de la valeur de 'n1'

# Calcul du coefficient d'asymétrie A3.
#
# Calcul préparatoire: estimation non biaisée du moment d'ordre 3, k3:
  k3 = (n*sum((aile.mat-mean(aile.mat))^3))/((n-1)*(n-2))
  k3                                      # Vérification de k3
#
# Calcul du coefficient d'asymétrie, noté A3:
  A3 = k3/((sqrt(variance))^3)
  A3                                      # Vérification de A3

# Calcul du coefficient d'aplatissement A4.
# Calcul préparatoire: estimation non biaisée du moment d'ordre 4, k4, et vérification:
  k4 = (n*(n+1)*(sum((aile.mat-mean(aile.mat))^4))-3*(n-1)*((sum((aile.mat-mean(aile.mat)
)^2))^2))/((n-1)*(n-2)*(n-3))
  k4
#
# Calcul du coefficient d'aplatissement, noté A4, et vérification:
  A4 = k4/((sqrt(variance))^4)
  A4

# Calcul de l'étendue de la variation (ou plage de variation):
  etendue = max(aile)-min(aile)                    # ou : plage = range(aile)
  etendue                                          # plage

# Calcul de l'écart type, noté sx:
  sx = sd(aile)
  sx

# Tracer un histogramme

# La façon la plus simple d'obtenir un histogramme est d'utiliser la fonction 'hist' avec toutes ses
valeurs par défaut :
  hist(merle[,1])
```

L'histogramme apparaît dans la fenêtre graphique. On peut le sauvegarder en cliquant sur « Sauver comme... » dans le menu « Fichier ».

On peut spécifier les détails de présentation de l'histogramme.
Fermer la fenêtre praphique créée par la commande précédente et reprendre.

```
par(mai = c(1.5, 0.75, 0.5, 0.5))          # Modifie les marges du graphique. Voir   ?par

hist(merle[,1], breaks = "Sturge", freq = TRUE, right =FALSE, main = NULL, xlab = NULL, ylab =
  NULL, axes = TRUE)
```

Étudiez ces spécifications dans le fichier d'aide de 'hist'
```
?hist
```

Identifiez les axes et ajoutez un titre

```
mtext(text="Frequence", side=2, line=3, cex=1, font=1)
mtext(text="Longueur de l'aile (mm)", side=1, line=2, cex=1, font=1)
mtext(text="Histogramme de la longueur de l'aile", side=1, line=4, cex=1.5, font=1)
```

==========

Reprendre l'exercice pour la variable Masse(kg), Hauteur(cm) ou Longueur(cm) du fichier 'Ours.txt'. Attention à la lecture de ce fichier de données !

Essayez les commandes suivantes pour les données d'ours :
```
summary(nom-du-data-frame)
plot(nom-du-data-frame)
```

 # ========

Attribution de valeurs aux paramètres d'une fonction R

Les fonctions R peuvent avoir plusieurs paramètres (voir par exemple ?read.table ou encore ?hist) et ces paramètres ont souvent des valeurs par défaut. Par exemple, la fonction

```
add3 <- function(a = 0, b = 10, c = -5)   a+b+c
```

possède trois paramètres, a, b et c, qui représentent des nombres. On a attribué des valeurs par défaut à ces paramètres : $a = 0$, $b = 10$, $c = -5$. La fonction additionne les trois nombres. Exemple :

```
add3()
```

donne comme résultat la somme des trois valeurs par défaut.

On peut remplacer les valeurs par défaut par des valeurs fournies par l'utilisateur. Exemple de remplacement selon la position des paramètres :

```
add3(2, 4, 6)                    # L'ordre détermine l'attribution des valeurs aux paramètres
```

Même résultat si on fournit des valeurs aux paramètres *a*, *b* et *c* de façon explicite :

```
add3(a=2, b=4, c=6)              # ou encore : add3(b=4, c=6, a=2)
```

Pouvez-vous anticiper le résultat des appels suivants à la fonction ?

1. Attribution selon la position. Quelle valeur prendra *c* ?
```
add3(2, 4)
```

2. Attribution explicite d'une valeur à *b*. Quelle valeur prendront *a* et *c* ?
```
add3(b=7)
```

========

Examinez le fichier de sortie produit par rda() ou cca() de {vegan}

Exemple: créer un objet de sortie de la fonction **rda()**

rda.object.name <- rda(un.fichier.de.donnees)

Obtenir la liste des éléments de premier niveau de cet objet de sortie qui est plutôt complexe

attributes(rda.object.name)

Vous pouvez aussi examiner le contenu de ces éléments séparément. Exemples:

rda.object.name$call
rda.object.name$CA
rda.object.name$tot.chi

La structure des objets de sortie de **rda()** et **cca()** est décrite dans un fichier de documentation :

?cca.object

Vous pouvez aussi examiner *la structure détaillée* d'un objet de sortie particulier de rda ou cca ; la liste est longue

str(rda.object.name)

Dans l'exercice d'ACP des pages 7-8, nous utiliserons la fonction **summary.cca()** de vegan pour obtenir les résultats de l'ACP

summary(rda.out, scaling=1)

Notez que l'appel de la fonction est **summary()** et non **summary.cca()**. Cette fonction reconnaît la classe de l'objet rda.out; cette classe est "rda" "cca".

Si la fonction répond "Total inertia: 0" suivi par un message d'erreur, cela pourrait être dû au fait que {ade4} a été chargé après {vegan} et la fonction **cca()** de {ade4} masque la fonction **cca()** de {vegan}. Si vous avez chargé {adespatial} après {vegan}, {adespatial} a chargé {ade4} et vous faites face au même problème. Pour résoudre la difficulté et obtenir les résultats de l'ACP ou de la RDA à l'aide de **summary.cca()**, il est nécessaire de détacher complètement {vegan} et de le recharger:

unloadNamespace("vegan")
library("vegan")

Après cette opération, la fonction **cca()** de {vegan} masque la fonction **cca()** de {ade4} et **summary.cca()** devrait faire son travail normalement.

2. Méthodes d'ordination simple : analyse en composantes principales, analyse factorielle des correspondances, analyse en coordonnées principales, nMDS

2.1. Analyse en composantes principales (ACP)
2.1.1. Analyse en composantes principales (ACP) par algèbre matricielle

Créer la matrice de données :

```
Y.mat = matrix(c(2,3,5,7,9,1,4,0,6,2), 5, 2)
class(Y.mat)                          # Vérifier la classe de l'objet Y.mat
```

Centrer la matrice Y.mat par colonne à l'aide de la fonction 'scale':

```
Y.cent = scale(Y.mat, center=TRUE, scale=FALSE)      # Données centrées mais non réduites.
```

Calculer la matrice de covariance:

```
Y.cov = cov(Y.cent)
```

Si les données *ne sont pas centrées-réduites*, la commande suivante est équivalente:

```
Y.cov = cov(Y.mat)          # mais ce n'est pas le cas si scale=TRUE.
```

.

Calculer les valeurs propres et les vecteurs propres:

```
Y.eig = eigen(Y.cov)        # Comparez avec le résultat de    Ycov.svd = svd(Y.cov)
                            # Comparez avec le résultat de    Y.svd = svd(Y.cent)
```

Vérifier les valeurs propres et les vecteurs propres:

```
Y.eig$values
Y.eig$vectors
```

Transférer les vecteurs propres dans U (représentation des variables en cadrage de type 1) :

```
U = Y.eig$vectors
```

Calculer la matrice F (représentation des objets en cadrage de type 1) : F = Ycentré * U

```
F = Y.cent %*% U
F                                     # Vérifier le contenu de la matrice F
```

Calcul de la matrice U2 pour la représentation des variables en cadrage de type 2 :

```
U2 = U %*% diag(Y.eig$values^(0.5))
```

Calcul de la matrice G pour la représentation des objets en cadrage de type 2 :

```
G = F %*% diag(Y.eig$values^(-0.5))
```

Cadrage de type 1 —
Diagramme de dispersion des 2 premières colonnes de F produit par la fonction générale plot() :

```
plot(F[,1], F[,2], xlim=c(-4,4), ylim=c(-3,3), asp=1, xlab="Axe 1", ylab="Axe 2")
```

Notes: (-4,4) = bornes de l'abscisse, (-3,3) = bornes de l'ordonnée.
asp=1 : le rapport des dimensions abscisse/ordonnée est fixé à 1
pour obtenir un graphique qui représente correctement les distances entre les objets.

Voici une autre manière de faire. La fonction "range" permet de connaître la plage de variation des valeurs sur les axes 1 et 2. On l'applique aux colonnes de F par la commande "apply" :

```
F.range = apply(F, 2, range)
F.range
```

Créer les vecteurs "xlim" et "ylim" qui fourniront les valeurs limites des axes du graphique :

```
xlim = c(F.range[1,1], F.range[2,1])
ylim = c(F.range[1,2], F.range[2,2])

plot(F[,1], F[,2], xlim=xlim, ylim=ylim, asp=1, xlab="Axe 1", ylab="Axe 2")
```

Ajouter au diagramme des flèches représentant les 2 premières colonnes de la matrice U:
```
arrows(x0=0,y0=0,U[,1]*3,U[,2]*3)
```
Note: pour cet exemple, les coordonnées des variables (tirées de U) sont multipliées par 3.

Exercice : tracer le diagramme avec cadrage de type 2
Utiliser maintenant la fonction biplot() de {stat}, écrite pour l'ACP, avec les matrices G et U2

```
biplot(G, U2)
```

========

2.1.2. Analyse en composantes principales (ACP) des données de l'exemple à l'aide de 'vegan'

Chargez d'abord la bibliothèque 'vegan'
```
library(vegan)
```

Pour cet exercice, les données se trouvent dans l'objet Y.mat créé au début de la section 2.1.1.
```
rda.out = rda(Y.mat)                    # Option par défaut : scale=FALSE
```

L'ACP est calculée par la fonction rda() de l'analyse canonique de redondance.
Cette fonction produit une ACP lorsqu'on lui fournit un seul tableau de données.
Avec l'argument "scale=FALSE", les données seront centrées par colonnes mais pas réduites.

Examinez l'objet produit par la fonction rda(). Prenez connaissance des notes de l'encadré de la page 5, « Examinez le fichier de sortie produit par rda() ou cca() de {vegan} ».

Examinez les résultats de l'ACP –
```
eigenvals(rda.out)                      # Valeurs propres
```

Scaling = 1: préserve les distances euclidiennes entre les objets
Scaling = 2: préserve les corrélations entre les descripteurs
ACP avec cadrage de type 1:
```
summary(rda.out, scaling=1)             # Position des espèces et des sites (modifiés par vegan)
biplot(rda.out, scaling=1)              # Graphique produit par la fonction biplot.rda() de vegan
```

ACP avec cadrage de type 2:
```
summary(rda.out, scaling=2)             # position des espèces et des sites (modifiés par vegan)
biplot(rda.out, scaling=2)              # Graphique produit par la fonction biplot.rda() de vegan
```

Attention : la fonction **summary.cca()** de vegan transforme les résultats avant de les afficher. En cadrage=1, les coordonnées des espèces (U) sont multipliées par la « General scaling constant » et les coordonnées des sites (F) sont divisées par cette constante. En cadrage=2, la transformation est plus complexe ; voir le document « decision-vegan.pdf ». Les coordonnées des espèces et des sites sont des transformations linéaires des coordonnées calculées dans les matrices U2 et G de l'exercice.

==========

2.1.3. Analyse en composantes principales (ACP) des données d'araignées à l'aide de 'vegan'

ACP des araignées des Pays-Bas[1] (fichier "Spiders_28x12_spe.txt"). Puis, calcul de la l'inertie cumulée des espèces et des sites sur les axes de l'ACP par la fonction goodness().

```
spiders.spe <- read.table( file.choose( ) )
```

• Abondance d'espèces : il est nécessaire de transformer les données avant l'ACP, car l'ACP préserve la distance euclidienne entre les objets et cette distance est inappropriée pour l'analyse des données de communautés. Voir la section "**4. Transformations**" plus bas.

```
spiders.hel <- decostand(spiders.spe, "hellinger")      # Transformation de Hellinger
```

La fonction rda() de vegan calcule une ACP lorsqu'on lui fournit un seul fichier de données.

```
res <- rda(spiders.hel)
```

Examinez la structure de l'objet généré par cette fonction : str(res). Voir page 5 de ce document.

La fonction goodness(x, display="species") calcule le R^2 cumulatif de la représentation de chaque **espèce** sur l'axe 1, puis sur les axes 1+2, 1+2+3, etc. Les R^2 de la colonne PC2, par exemple, indiquent à quel point les espèces sont bien représentées dans le plan des deux premiers axes. Dans la colonne PC12 de cet exemple, toutes les espèces sont parfaitement représentées et ont un R^2 de 1.

```
spi.good.sp <- goodness(res, display="species", model="CA")
```

L'argument model="CA" indique que l'analyse est une ordination simple, une ACP. On peut obtenir ces mêmes statistiques de la fonction goodness() pour une ordination canonique avec l'argument model="CCA" ; la dernière colonne du fichier de sortie contient alors des valeurs inférieures ou égales à 1. Voir le manuel (*Numerical ecology*, 2012) p. 442.

Comparer ces résultats au graphique d'ordination ACP

```
biplot(res)
```

La fonction goodness(x, display="sites") calcule la proportion de la norme de chaque **vecteur-site** qui est représentée en 1, 1+2, 1+2+3, etc. dimensions. Voir le manuel p. 443.

```
spi.good.si <- goodness(res, display="sites", model="CA")
```

==========

2.1.4. Analyse en composantes principales à l'aide de la fonction 'PCA.newr'

Répétez l'exercice sur les araignées en utilisant la fonction 'PCA.newr' qui se trouve parmi les fonctions du livre *Numerical ecology with R*. Le code R de cette fonction est pour l'essentiel le code qui se trouve à la section 2.1.1. Des options supplémentaires ont été ajoutées à la fonction. Vous pouvez examiner le code en ouvrant la fonction à l'aide d'un éditeur de texte.

Il faut d'abord charger la fonction dans la console R:
Clients Windows : allez dans le menu **Fichiers => Sourcer du code R...**
Clients MacOS X : allez dans le menu **Fichiers => Sourcer Fichier...**

```
pca.out = PCA.newr(spiders.hel)        # Valeur par défault: variables non réduites
biplot(pca.out)                        # La valeur par défaut est le cadrage de type 1
```

Comparez les résultats : (a) ceux de la fonction rda, soit les valeurs propres, les vecteurs propres et les composantes principales (eigenvals(rda.out), scores(rda.out, scaling=1)) et (b) ceux de la fonction PCA.newr (pca.out$eigenvalues, pca.out$U, pca.out$F). Pourquoi y a-t-il des differences? — Voir le document "decision-vegan.pdf", p. 4, Table 1, dans le dossier vegan.

---

[1] Aart, P.J.M. van der & N. Smeenk-Enserink. 1975. Neth. J. Zool. 25: 1-45.

```
# 2.2. Analyse factorielle des correspondances (AFC)
# 2.2.1. Analyse factorielle des correspondances (AFC) par algèbre matricielle

# Lire le tableau de données qui sera soumis à l'analyse des correspondances.
# Choisissez 'Table_9.11.txt' (petit exemple numérique tiré du chapitre 9 du manuel)
# ou 'Spiders_28x12_spe.txt' (tableau de données réelles, plus grand).
Y = read.table( file.choose() )

# Calculer les paramètres de base du tableau Y; sauvegarder les noms des lignes et des colonness
n = nrow(Y)
p = ncol(Y)
n.eigval = min((n-1),(p-1))
site.names = rownames(Y)
sp.names = colnames(Y)

# Construire la matrice Qbar (contributions au khi-carré)
fi. = matrix(apply(Y,1,sum),n,1)
f.j = matrix(apply(Y,2,sum),1,p)
f.  = sum(fi.)
pi. = as.vector(fi./f.)
p.j = as.vector(f.j/f.)
E = (fi. %*% f.j)/f.
Qbar = (Y - E) * E^(-0.5) / sqrt(f.)

# Analyser Qbar par 'svd'
svd.res = svd(Qbar)
eigenvalues = svd.res$d[1:n.eigval]^2
U = svd.res$v[,1:n.eigval]
Uhat = svd.res$u[,1:n.eigval]

# On peut également analyser Qbar par 'eigen'
Qbar = as.matrix(Qbar)
QprQ.eig = eigen( t(Qbar) %*% Qbar )
eigenvalues = QprQ.eig$values[1:n.eigval]
U = QprQ.eig$vectors[,1:n.eigval]
Uhat = Qbar %*% U %*% diag(eigenvalues^(-0.5))

# Construire les matrices V, Vhat, F et Fhat qui serviront à tracer les graphiques
V = diag(p.j^(-0.5)) %*% U
Vhat = diag(pi.^(-0.5)) %*% Uhat
F = Vhat %*% diag(eigenvalues^(0.5))
Fhat = V %*% diag(eigenvalues^(0.5))

# Trouver les bornes des axes en vue des graphiques
V.range = apply(V[,1:2],2,range)
Vhat.range = apply(Vhat[,1:2],2,range)
F.range = apply(F[,1:2],2,range)
Fhat.range = apply(Fhat[,1:2],2,range)

par(mfrow=c(1,2))              # Créer une fenêtre graphique pour deux diagrammes

# Diagramme de double projection, cadrage 1 : utiliser F pour les sites et V pour les espèces
# Les sites sont au centroïde (barycentre) des espèces
# Cette projection préserve la distance du khi-carré entre les sites

xmin = min(V.range[1,1], F.range[1,1]) - 0.5
```

```
xmax = max(V.range[2,1], F.range[2,1]) + 0.5
ymin = min(V.range[1,2], F.range[1,2]) - 0.5
ymax = max(V.range[2,2], F.range[2,2]) + 0.5

plot(F[,1:2], asp=1, pch=20, cex=2, xlim=c(xmin,xmax), ylim=c(ymin,ymax), xlab="AFC axe 1", ylab="AFC axe 2")
text(F[,1:2], labels=site.names, pos=4, offset=0.5)
points(V[,1:2], pch=22, cex=2)
text(V[,1:2], labels=sp.names, pos=4, offset=0.5)
title(main = c("AFC, diagramme de double projection","cadrage de type 1"), family="serif")

# Diagramme de double projection, cadrage 2 : utiliser Vhat pour les sites et Fhat pour les espèces
# Les espèces sont au centroïde (barycentre) des sites
# Cette projection préserve la distance du khi-carré entre les espèces

xmin = min(Vhat.range[1,1], Fhat.range[1,1]) - 0.5
xmax = max(Vhat.range[2,1], Fhat.range[2,1]) + 0.5
ymin = min(Vhat.range[1,2], Fhat.range[1,2]) - 0.5
ymax = max(Vhat.range[2,2], Fhat.range[2,2]) + 0.5

plot(Vhat[,1:2], asp=1, pch=20, cex=2, xlim=c(xmin,xmax), ylim=c(ymin,ymax), xlab="AFC axe 1", ylab="AFC axe 2")
text(Vhat[,1:2], labels=site.names, pos=4, offset=0.5)
points(Fhat[,1:2], pch=22, cex=2)
text(Fhat[,1:2], labels=sp.names, pos=4, offset=0.5)
title(main = c("AFC, diagramme de double projection","cadrage de type 2"), family="serif")
```

2.2.2 Analyse des correspondances (AFC) à l'aide de la bibliothèque 'vegan'

Scaling = 1: préserve les distances du khi-carré entre les objets.
Utilise les matrices F et V, comme dans Canoco.
Scaling = 2: préserve les distances du khi-carré entre les espèces.
Utilise les matrices V-chapeau et F-chapeau, comme dans Canoco.

Exemple 1 : AFC des données d'araignées. Lire le fichier de données 'Spiders_28x12_spe.txt'

```
library(vegan)
spiders=read.table( file.choose() )
spiders.ca=cca(spiders)
summary(spiders.ca, scaling=1)
plot(spiders.ca, scaling=1)
```

Note : l'AFC ne peut pas être calculée si le fichier contient des lignes ou colonnes qui somment à 0. Cela est dû à la transformation en matrice Q-bar qui est la première étape de l'AFC.
S'il y a des lignes ou des colonnes qui ont une somme de 0, on peut utiliser la fonction 'c' (« combine ») pour sélectionner pour l'analyse les lignes ou les colonnes qui ont une somme supérieure à 0.

Exemple 2 : AFC des cafés of Neuchâtel. Fichier de données : "Cafes_10x6_spe.txt". Les données représentent la clientèle de 10 restaurants de la ville de Neuchâtel en Suisse. Les noms des cafés sont réels ; les données de clientèle furent générées par Daniel Borcard pour illustrer les propriétés de l'AFC.

```
library(vegan)
cafes.spe= read.table( file.choose() )
cafes.afc=cca(cafes.spe)
cafes.afc
plot(cafes.afc, scaling=2)
summary(cafes.afc, scaling=2, axes=5)
```

L'analyse des correspondances peut aussi être calculée par la fonction 'CA.newr' disponible sur la page Web numericalecology.com.

==========

2.3. Analyse en coordonnées principales (ACoP)

On obtient cette analyse à l'aide de la fonction **cmdscale** de la bibliothèque **stats**; « cmds » est l'acronyme de « classical multidimensional scaling ».
Des fonctions pour l'ACoP sont également disponibles dans les bibliothèques **ape** (fonction **pcoa**), **ade4** (fonction **dudi.pco**) et **mvpart** (fonction **cmds.diss**).
Le point de départ de l'ACoP est une matrice de distances. Les fonctions de distance sont étudiées à la section 3. Pour illustrer le fonctionnement de l'ACoP, nous utiliserons la distance euclidienne qui est l'option « par défaut » de la fonction **dist** de la bibliothèque **stats**.

```
# Exemple: analyse du fichier "Spiders_28x12_spe.txt"
  spiders = read.table( file.choose() )
# Calculer la matrice de distances euclidiennes
  spiders.D1 = dist(spiders)                    # ou: spiders.D1 = dist(spiders, method="eucl")

# Analyse en coordonnées principales. Conservons k=5 axes. Graphique des 2 premiers axes.
  toto4 = cmdscale(spiders.D1, k=5, eig=TRUE)
  plot(toto4$points[,1], toto4$points[,2], asp=1, xlab="Axe 1", ylab="Axe 2")
```

Note: "asp=1" force les deux axes à utiliser la même échelle. Ainsi, les distances entre les objets sont des projections de leurs distances réelles.

```
 ?cmdscale          # permet de consulter le fichier d'aide de la fonction cmdscale
  summary(toto4)    # permet de connaître les éléments qui composent le fichier toto4
  toto4$points      # contient les coordonnées des objets sur les k=5 dimensions demandées
  toto4$eig         # contient les valeurs propres des différents axes principaux
```

Reprendre le calcul en appliquant la transformation de Hellinger (fonction **decostand** de la bibliothèque **vegan**) aux données d'abondances d'araignées. Transformations : voir la section 4.

```
# Transformation de Hellinger des abondances d'espèces, puis calcul de la distance euclidienne
  library(vegan)
  spiders.hel = decostand(spiders,"hel")        # Transformation de Hellinger
  spiders.DHell = dist(spiders.hel)             # Calcul de la distance de Hellinger

# Analyse en coordonnées principales.
  res = cmdscale(spiders.DHell, 5, eig=TRUE)
  limites = apply(res$points, 2, range)
  xlim = c(limites[1,1], limites[2,1]) + c(-0.2,0)
  ylim = c(limites[1,2], limites[2,2])
  x = res$points[,1]
  y = res$points[,2]
  plot(x, y, xlim=xlim, ylim=ylim, asp=1, xlab="Axe 1", ylab="Axe 2")

# La fonction text permet d'ajouter les noms des objets au graphique.
  noms = rownames(spiders)
  text(x, y, labels=noms, pos=2, cex=1, offset=1)
```

L'ACoP peut aussi être calculée à l'aide de la fonction pcoa() de **ape**. La fonction biplot.pcoa() de cette bibliothèque produit des graphiques d'ordination de meilleure qualité que ceux produits ci-dessus.

2.4. Positionnement (ou cadrage) multidimensionnel non-métrique (nMDS)

nMDS est l'acronyme de « nonmetric multidimensional scaling ». Nous utiliserons d'abord les fonctions **nmds** et **bestnmds** de la bibliothèque **labdsv** ; ce sont des *enveloppes*, écrites par David W. Roberts, pour la fonction **isoMDS** de la bibliothèque **MASS** de Brian D. Ripley.
Le point de départ de la nMDS est une matrice de distances. Les fonctions de distance sont étudiées à la section 3.

```
# Exemple: analyse du fichier "Spiders_28x12_spe.txt"
  spiders = read.table( file.choose() )

# Transformation de Hellinger des abondances d'espèces, puis calcul de la distance euclidienne.
# La matrice « spiders.DHell » contiendra donc des distances de Hellinger.
  library(vegan)
  spiders.hel = decostand(spiders, "hel")
  spiders.DHell = dist(spiders.hel)

# Calcul d'une solution nMDS en k=2 dimensions. Si on ne fournit pas de configuration initiale, les deux premiers axes d'une ACoP forment, par défaut, la configuration initiale.
  library(labdsv)
  toto2 = nmds(spiders.DHell, k=2)
  plot(toto2)

# L'enveloppe bestnmds permet d'examiner le stress des solutions successives obtenues en utilisant une série de configurations aléatoires comme points de départ (paramètre « itr ») :
  toto2 = bestnmds(spiders.DHell, k=2, itr=20)

# La fonction conserve la solution ayant le stress le plus faible. On trace celle-ci.
  plot(toto2)

# ==========

# La bibliothèque vegan contient la fonction metaMDS. Cette enveloppe de isoMDS permet de mettre en œuvre la méthode nMDS en suivant les recommandations de l'écologiste Peter Minchin. La fonction recherche automatiquement une solution stable en utilisant plusieurs départs aléatoires (par défaut, trymax = 20).
# En fin d'analyse, la fonction fait une ACP de la configuration finale ; les objets ont ainsi leur variance maximum le long du premier axe. Les espèces sont représentées dans le graphique à la façon de l'analyse des correspondances.
# La documentation de metaMDS est très détaillée.

# Plutôt que la distance de Bray-Curtis (option par défaut de metaMDS), nous utiliserons la distance de Hellinger obtenue en calculant la distance euclidienne sur les données transformées Hellinger.
# Calcul d'une solution nMDS en k=2 dimensions (valeur par défaut).
  toto3 = metaMDS(spiders.hel, distance="euclidean")
  plot(toto3, type="t")                    # type="t" pour ajouter les espèces au graphique
```

3. Distances

Des fonctions de distance sont disponibles dans au moins quatre bibliothèques du langage R.

3.1. Bibliothèque **stats**, fonction **dist** : 6 distances (information : **?dist**)

euclidean: Usual Euclidean distance between the two vectors (2 norm). **This is the default value.**

maximum: Maximum distance between two components of *x* and *y* (supremum norm).

manhattan: Absolute distance between the two vectors (1 norm).

canberra: *sum(|x_i - y_i| / |x_i + y_i|)*. Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

binary: (aka *asymmetric binary*): The vectors are regarded as binary bits, so non-zero elements are 'on' and zero elements are 'off'. The distance is the *proportion* of bits in which only one is on amongst those in which at least one is on. — *Ce coefficient est (1 – Jaccard)*.

minkowski: The *p* norm, the *p*th root of the sum of the *p*th powers of the differences of the components.

Le choix d'un coefficient se fait en écrivant son nom entre guillemets. Exemple :
dist(donnees, method="binary") *ou* dist(donnees, "binary")
=> En pratique, la fonction dist() est surtout utilisée pour calculer la distance euclidienne.

3.2. Bibliothèque **vegan**, fonction **vegdist** : 10 dissimilarités (information : **?vegdist**)

euclidean	*d[jk] = sqrt(sum (x[ij]-x[ik])^2)*
manhattan	*d[jk] = sum(abs(x[ij] - x[ik]))*
gower	*d[jk] = (1/M) sum (abs(x[ij]-x[ik])/(max(x[i])-min(x[i])))*; *M* = number of variables
altGower	*d[jk] = (1/NZ) sum(abs(x[ij] - x[ik]))*; *NZ* = n. columns excluding double-zeros
canberra	*d[jk] = (1/NZ) sum ((x[ij]-x[ik])/(x[ij]+x[ik]))*; exclude double-zeros; *NZ* as above
bray	*d[jk] = (sum abs(x[ij]-x[ik])/(sum (x[ij]+x[ik]))* = Bray & Curtis dissim. (par défaut)
kulczynski	*d[jk] 1 - 0.5*((sum min(x[ij],x[ik])/(sum x[ij]) + (sum min(x[ij],x[ik])/(sum x[ik]))*
morisita	*{d[jk] = 1 - 2*sum(x[ij]*x[ik])/((lambda[j]+lambda[k]) * sum(x[ij])*sum(x[ik]))* } where *lambda[j] = sum(x[ij]*(x[ij]-1))/sum(x[ij])*sum(x[ij]-1)*
horn	Like morisita, but *lambda[j] = sum(x[ij]^2)/(sum(x[ij])^2)*
binomial	*d[jk] = sum(x[ij]*log(x[ij]/n[i]) + x[ik]*log(x[ik]/n[i]) - n[i]*log(1/2))/n[i]* where *n[i] = x[ij] + x[ik]*
jaccard	*d[jk] = 1 - (a / (a+b+c))*; binary=TRUE to obtain the Jaccard coeff. for binary data.

"mountford" (coefficient de Mountford), "raup" (coefficient probabiliste de Raup & Crick), "chao" (coefficient de Chao), "cao", et "mahalanobis" peuvent également être calculés. Voir **?vegdist**.

Le choix d'un coefficient se fait en écrivant son nom entre guillemets. Exemple :
vegdist(donnees, method="bray") *ou* vegdist(donnees, "bray")

3.3. Bibliothèque **ade4**, fonction **dist.binary** : 10 distances binaires (information : **?dist.binary**)

=> Les similarités (*S*) qui suivent sont converties en distances par la transformation ***D = sqrt(1 – S)***.

1 = Jaccard index (1901) [S3 coefficient of Gower & Legendre] : *s1 = a / (a+b+c)*

2 = Sokal & Michener index (1958)
[S4 coefficient of Gower & Legendre] : *s2 = (a+d) / (a+b+c+d)*

3 = Sokal & Sneath(1963) [S5 coefficient of Gower & Legendre] : *s3 = a / (a + 2(b + c))*

4 = Rogers & Tanimoto (1960)
[S6 coefficient of Gower & Legendre] : *s4 = (a + d) / (a + 2(b + c) +d)*

5 = Czekanowski (1913) or Sørensen (1948)
[S7 coefficient of Gower & Legendre] : *s5 = 2a / (2a + b + c)*

6 = [S9 index of Gower & Legendre (1986)] : *s6 = (a - (b + c) + d) / (a + b + c + d)*

7 = Ochiai (1957) [S12 coefficient of Gower & Legendre] : *s7 = a / sqrt((a + b)(a + c))*

8 = Sokal & Sneath (1963)
[S13 coefficient of Gower & Legendre] : *s8 = ad / sqrt((a + b)(a + c)(d + b)(d + c))*

9 = Phi of Pearson
[S14 coefficient of Gower & Legendre] : *s9 = (ad - bc) / sqrt((a + b)(a + c)(d + b)(d + c))*

10 = [S2 coefficient of Gower & Legendre] : *s10 = a / (a + b + c + d)*

# Le choix d'un coefficient se fait en écrivant son numéros tiré de la liste ci-dessus. Exemples :
dist.binary(donnees, method=1) *ou* dist.binary(donnees, 1) *ou* dist.binary(donnees, "1")

3.4. Bibliothèque **adespatial**, fonction **dist.ldc()** : 21 dissimilarités (information : **?dist.ldc**)

La fonction permet de calculer 18 dissimilarités quantitatives :

"hellinger", "chord", "chisquare", "profiles", "percentdiff", "ruzicka", "divergence", "canberra",
"whittaker", "wishart", "kulczynski", "ab.jaccard", "ab.sorensen", "ab.ochiai", "ab.simpson",
"euclidean", "manhattan", "modmeanchardiff",
ainsi que 3 dissimilarités binaires : "jaccard", "sorensen", "ochiai".

Ces mesures sont décrites et analysées dans l'article de Legendre & De Cáceres (2013).

Les détails quant à ces 21 indices se trouvent dans la section *Details* du fichier de documentation.

Groupe 1 – Matrice **D** calculée par transformation de **Y** suivie du calcul de la distance Euclidienne

Hellinger D, D[ik] = sqrt(sum((sqrt(y[ij]/y[i+])-sqrt(y[kj]/y[k+]))^2))

chord D, D[ik] = sqrt(sum((y[ij]/sqrt(sum(y[ij]^2))-y[kj]/sqrt(sum(y[kj]^2)))^2))

chi-square D, D[ik] = sqrt(y[++] sum((1/j[+j])(y[ij]/y[i+]-y[kj]/y[k+])^2))

species profiles D, D[ik] = sqrt(sum((y[ij]/y[i+]-y[kj]/y[k+])^2))

Groupe 2 – Autres fonctions *D* appropriées pour l'étude de la diversité bêta. Dans les deux premières fonctions, A = sum(min(y[ij],y[kj])), B = y[i+]-A, C = y[k+]-A.

percentage difference D, D[ik] = (sum(abs(y[ij]-y[k,j])))/(y[i+]+y[k+]) or else, D[ik] = (B+C)/(2A+B+C) =

Ružička D, D[ik] = 1-(sum(min(y[ij],y[kj])/sum(max(y[ij],y[kj])) or else, D[ik] = (B+C)/(A+B+C)

coeff. of divergence D, D[ik] = sqrt((1/pp)sum(((y[ij]-y(kj])/(y[ij]+y(kj]))^2))

Canberra metric D, D[ik] = (1/pp)sum(abs(y[ij]-y(kj])/(y[ij]+y(kj]))

Whittaker D, D[ik] = 0.5*sum(abs(y[ij]/y[i+]-y(kj]/y[k+]))

Wishart D, D[ik] = 1-sum(y[ij]y[kj])/(sum(y[ij]^2)+sum(y[kj]^2)-sum(y[ij]y[kj]))

Kulczynski D, D[ik] = 1-0.5((sum(min(y[ij],y[kj])/y[i+]+sum(min(y[ij],y[kj])/y[k+]))

Groupe 3 – Indices de dissimilarité classiques pour données de présence-absence. Ces indices sont appropriés pour l'étude de la diversité bêta. Les matrices **D** sont présentés sous forme racine carrée, comme dans les indices binaires de la fonction dist.binary() de {ade4}.

Jaccard D, D[ik] = sqrt((b+c)/(a+b+c))

Sørensen D, D[ik] = sqrt((b+c)/(2a+b+c))

Ochiai D, D[ik] = sqrt(1 - a/sqrt((a+b)(a+c)))

Groupe 4 – Indices de Chao et al. (2006) pour données quantitatives d'abondance. Ces fonctions corrigent l'indice pour les espèces non observées à cause d'erreurs d'échantillonnage. Ces indices ne sont pas présentés sous forme racine carée, contrairement aux indices de Jaccard, Sørensen et Ochiai du groupe 3 qui sont transformés par racine carrée.

abundance-based Jaccard D, D[ik] = 1-(UV/(U+V-UV))

abundance-based Sørensen D, D[ik] = 1-(2UV/(U+V))

abundance-based Ochiai D, D[ik] = 1-sqrt(UV)

abundance-based Simpson D, D[ik] = 1-(UV/(UV+min((U-UV),(V-UV))))

Groupe 5 – Indices généraux de dissimilarité qui n'ont pas de borne maximum, ou pas de valeur maximum pour *D*. Ces indices ne sont pas appropriés pour l'étude de la diversité bêta.

Euclidean D, D[ik] = sqrt(sum(y[ij]-y[kj])^2)

Manhattan D, D[ik] = sum(abs(y[ij] - y[ik]))

modified mean character difference, D[ik] = (1/pp) sum(abs(y[ij] - y[ik]))

Les indices sont calculés par une fonction écrite en C pour augmenter la vitesse de calcul, en particulier pour les grands tableaux de données. Ces mêmes indices sont disponibles dans la fonction beta.div() pour le calcul de la diversité bêta et des indices LCBD.

3.5. Bibliothèque **cluster**, fonction ***daisy()*** : 3 dissimilarités quantitatives (information : **?daisy**)

On peut calculer des dissimilarité entre objets pour des variables environnementales à l'aide des coefficients de la fonction 'daisy' qui manipule correctement les absences d'information (valeur NA). Les coefficients disponibles sont : la distance euclidienne, la distance de Manhattan et la dissimilarité de Gower. Le choix d'un coefficient se fait en écrivant son nom entre guillemets.

Dans le calcul du coefficient de Gower, ***daisy()*** reconnaît et traite différemment d'une part les variables quantitatives, semi-quantitatives (« ordered factor ») et binaires, et d'autre part les variables qualitatives multiclasses (« factor »). L'absence d'information (code NA) est permise. $D = 1 - S$. Exemple :

```
data(dune.env)        # disponible dans la bibliothèque 'vegan'
?dune                 # Information sur ces données
```

La première variable est quantitative. Les variables #2, 4 et 5 sont des « ordered factors » qui seront traitées comme des variables quantitatives. La variable #3 est de type « factor ».

```
mat.gower = daisy(dune.env, "gower")
```

La méthode 'gower' de la fonction ***vegdist()*** de 'vegan' ne traite pas les absences d'information ni les facteurs ; elle ne traite que les variables quantitatives et les variables binaires. Exemple :

```
mat.gower.2 = vegdist(dune.env, "gower")
```

Exemple de données avec absence d'information, NA (*Numerical ecology* 2012, p. 280)

```
ex.p260 = matrix(c(2,1,1,2,3,6,NA,3,1,2,1,5,2,2,3,4,2,5,2,2,4,6,5,10),3,8)
ex.p260
res.gower = daisy(ex.p260, "gower")
```

La fonction ***vegdist()*** de 'vegan' ne peut pas traiter la valeur absente (NA) de la variable #3

```
res.gower.2 = vegdist(ex.p260, "gower")
res.gower.2 = vegdist(ex.p260[,-3], "gower")
```

3.6. Bibliothèque **FD**, fonction ***gowdis()***: dissimilarité de Gower (information : **?gowdis**)

C'est la fonction la plus complète pour calculer l'indice de Gower. Les variables peuvent appartenir aux types suivants : quantitatives, ordonnées, facteurs et binaires. On peut spécifier des poids (w) différents pour les variables. L'absence d'information (code NA) est permise. $D = 1 - S$. Faites tourner dans ***gowdis()*** et ***daisy()*** le fichier 'dummy$trait' utilisé dans les exemples de gowdis.

==========

Quelques trucs

Si vous importez une matrice carrée de **similarités** calculée à l'extérieur de R, vous pouvez la transformer en une matrice de dissimilarité par la commande :

```
mat.D = 1-mat.S
```

# ou, ce qui est préférable dans bien des cas pour éviter l'apparition de valeurs propres négatives en analyse en coordonnées principales (ACoP) :
mat.D = sqrt(1-mat.S)

# Avant de l'utiliser, vous devrez transformer cet objet, qui est de type 'data.frame', en un objet de type 'dist' à l'aide de la commande ***as.dist()***. Cela est exigé par ***hclust()***, ***cmdscale()*** et les fonctions de nMDS.
mat.DD = as.dist(mat.D)

# Une fonction de 'ade4' vous permet de savoir si une matrice de dissimilarité 'mat.D' est euclidienne, c'est-à-dire si elle produira analyse en coordonnées principales (ACoP) sans valeurs propres négatives. Voir *Numerical ecology* (2012), p. 296-298 et 500-506.
is.euclid(mat.D)

Exemples
is.euclid(mat.gower) # Matrice 'mat.gower' calculée par ***daisy()*** ci-dessus
is.euclid(sqrt(mat.gower)) # Matrice 'mat.gower' calculée par ***daisy()*** ci-dessus

4. Transformations

Transformation des données. En particulier, on peut vouloir transformation des tableaux d'abondances d'espèces avant une analyse linéaire qui préserve la distance euclidienne (ACP, RDA, partitionnement *K*-means) ou avant le calcul de la distance euclidienne en vue d'une ACoP

Charger d'abord la bibliothèque **vegan** de Yari Oksanen.

```
library(vegan)
```

La fonction de **vegan** qui réalise les transformations est **decostand**. Info : **?decostand**

Transformations pour tableaux de présence-absence ou d'abondance d'espèces :

- **total**: divide by margin total (default MARGIN = 1). *MARGIN = 1 signifie «par ligne»*.
- **normalize**: make margin sum of squares equal to one (default MARGIN = 1).
- **chi.square**: divide by row sums and square root of column sums, and adjust for square root of matrix total (Legendre & Gallagher 2001). When used with Euclidean distance, the matrix should be similar to the Chi-square distance used in correspondence analysis. However, the results from **cmdscale** would still differ, since CA is a weighted ordination method (default MARGIN = 1).
- **hellinger**: square root of method = "total" (Legendre & Gallagher 2001).
- **log**: logarithmic transformation suggested by Anderson et al. (2006). This is **not** log(y+1).
- **pa**: scale into presence/absence scale (0/1).

Autres transformations pour variables physiques :

- **max**: divide by margin maximum (default MARGIN = 2). *MARGIN = 2 signifie «par colonne»*.
- **freq**: divide by margin maximum and multiply by number of non-zero items, so that the average of non-zero entries is one (Oksanen 1983; default MARGIN = 2).
- **range**: standardize values into range 0 ... 1 (default MARGIN = 2). If all values are constant, they will be transformed to 0.
- **standardize**: scale into zero mean and unit variance (default MARGIN = 2).
- **pa**: scale into presence/absence scale (0/1).

Exemples d'utilisation pour des tableaux d'abondances d'espèces :

```
spiders.transf = decostand(spiders,"total")    # transformation en profils d'abondances relatives
spiders.transf = decostand(spiders,"norm")     # transformation de corde
spiders.transf = decostand(spiders,"chi.sq")   # transformation de khi-carré
spiders.transf = decostand(spiders,"hel")      # transformation de Hellinger
spiders.transf = decostand(spiders,"pa")       # transformation en présences-absences
```

Exemples d'utilisation pour des variables physiques :

```
data.transf = decostand(data,"range")          # cadrage des variables dans l'intervalle [0, 1]
data.transf = decostand(data,"stand")          # centrage et réduction des variables
```

5. Régression multiple

```
# Lire les données du fichier 'Ours.txt' pour produire le 'data frame' Ours :
  Ours = read.table(file.choose(), header=TRUE, row.names=1)
  Ours

# La dernière colonne du tableau contient un  facteur (sexe) représenté par des caractères (lettres).
# Créer un fichier de type ‘matrix’ ne contenant que la portion numérique du 'data frame' :
  Ours.mat = as.matrix(Ours[,1:4])

# Un modèle de régression (fonction ‘lm’) peut être écrit pour le ‘data.frame’ Ours, en utilisant
# la notation ~ pour indiquer les colonnes à utiliser dans le modèle (variable réponse, variables
# explicatives). 'lm' est l’abréviation de 'linear model' alors que  ~ signifie ‘fonction de’.
  toto = lm(Ours[,2] ~ Ours[,1] + Ours[,3] + Ours[,4])

# Les coefficients de régression se trouvent dans l'objet 'toto'.
  toto

# On obtient les coefficients de régression et les tests de signification par la commande 'summary' :
  summary(toto)

# Dans le modèle, on peut utiliser directement les noms des variables du 'data frame' Ours.
# Il est utile de prévoir des noms courts pour les variables :
  toto = lm(Masse.kg ~ Age.estime + Hauteur.cm + Longueur.cm, data=Ours)
  summary(toto)

# On peut aussi écrire le modèle en utilisant le fichier ‘Ours.mat' qui est de type ‘matrix’ :
  toto = lm(Ours.mat[,2] ~ Ours.mat[,1] + Ours.mat[,3] + Ours.mat[,4])
  summary(toto)

# Pour le fichier de type ‘matrix’, la commande peut également s'écrire comme suit :
  toto = lm(Ours.mat[,2] ~ Ours.mat[, c(1,3,4)])
  summary(toto)

# La 5e variable du fichier Ours est une variable qualitative à deux classes, M et F.
# Si on transforme cette variable en un facteur
  Ours[, 5] = as.factor(Ours[, 5])

# on peut l’utiliser directement dans la régression qui devient une analyse de covariance :
  toto = lm(Ours[,2] ~ Ours[,1] + Ours[,3] + Ours[,4] + Ours[,5])
  summary(toto)

# On peut obtenir les valeurs ajustées au modèle de régression:
  Ajuste = predict(toto)        # ou encore     Ajuste = fitted.values(toto)
  Ajuste

# On peut également obtenir les résidus du modèle :
  Resid = residuals(toto)
  Resid
```

```
# Pour l'analyse des résidus, quatre graphiques sont produits en séquence par la commande :
  plot(toto)
# Ces graphiques sont décrits dans le fichier d'aide
  ?plot.lm
# Les deux graphiques les plus intéressants tracent les résidus en fonction des valeurs prédites :
  plot(toto, which=c(1,3))
```

Dans le cas d'une **régression linéaire simple** (une seule variable explicative), on peut produire comme suit un graphique des points montrant la droite de régression :

```
  toto = lm(Masse.kg ~ Age.estime, data = Ours)
  summary(toto)
  a = summary(toto)$coefficients[1,1]                          # Ordonnée à l'origine
  b = summary(toto)$coefficients[2,1]                          # Pente
  plot(Ours[,1], Ours[,2], xlab="Age", ylab="Masse")           # Tracer les points
  abline(a, b, col="red")                                      # Tracer la droite de régression
```

Tests statistiques spéciaux –

```
# On peut obtenir un tableau de tests additionnels par la commande :
  anova(toto)
```

Attention: ce tableau est d'emploi rare et difficile à interpréter. On y trouve le test de **l'effet additionnel** de chaque variable dans l'ordre de leur inclusion dans le modèle.

```
# On peut comparer deux modèles emboîtés :
  modele1 = lm(Ours[,2] ~ Ours[,1] + Ours[,3] + Ours[,4] + Ours[,5])
  modele2 = lm(Ours[,2] ~ Ours[,1] + Ours[,3])
  anova(modele1, modele2)

# Cette fonction peut être utilisée pour tester l'effet partiel d'une seule variable :
  modele1 = lm(Ours[,2] ~ Ours[,1] + Ours[,3] + Ours[,4] + Ours[,5])
  modele3 = lm(Ours[,2] ~ Ours[,1] + Ours[,3] + Ours[,4])
  anova(modele1, modele3)

# Comparer ce résultat avec celui du test du coefficient de régression partielle
# calculé plus haut pour la variable 5. Rappelez-vous que pour une seule variable, t = sqrt(F)
  summary(modele1)

# ==========
```

La fonction 'lm' peut être utilisée pour la régression, l'analyse de variance à un critère de classification et l'analyse de covariance. La fonction 'aov' est cependant supérieure pour ces analyses plus complexes.

La fonction 'aov' est une enveloppe de la fonction 'lm'. Elle permet le calcul d'analyses de variance pour des plans à un seul ou à plusieurs facteurs croisés, dans les situations équilibrées ou non, de même que l'analyse de covariance. Les facteurs doivent être déclarés comme tels par la fonction 'as.factor'. La fonction 'summary.aov' fournit les résultats détaillés de ces analyses.

On utilise la fonction 'manova' pour l'analyse de variance multivariable. Les résultats détaillés de la manova sont fournis par 'summary.manova'.

6. Analyses canoniques (RDA et CCA)

6.1. Analyse canonique de redondance (RDA)
6.1.1. Analyse canonique de redondance (RDA) par algèbre matricielle

Lire les données du fichier "Table_11-3.txt" et les transférer dans la matrice Table11.3.mat,
puis dans les matrices Y.mat et X.mat:

```
Table11.3 = read.table( file.choose() )
Table11.3.mat = as.matrix(Table11.3)
Y.mat = Table11.3.mat[,1:6]                # Vérifiez le contenu de Y.mat
X.mat = Table11.3.mat[,7:10]               # Vérifiez le contenu de X.mat
```

Centrer la matrice Y, centrer et réduire la matrice X.
Note: le paramètre 2 indique d'appliquer la fonction 'scale' par colonne

```
Y = scale(Y.mat, center=TRUE, scale=FALSE) # ou : Y = apply(Y.mat,2,scale,center=T,scale=F)
X = scale(X.mat, center=TRUE, scale=TRUE)  # ou : X = apply(X.mat,2,scale,center=T,scale=T)
```

Notez que la 4ème colonne de X est colinéaire aux colonnes 2 et 3.
Créer la matrice X3 contenant les 3 premières colonnes de X :

```
X3 = X[,1:3]
```

Première partie de la RDA : régression multivariable

Créer X'X et calculer son inverse, inv[X'X] :

```
invX3 = solve( t(X3) %*% X3 )
```

Calculer le projecteur (i.e. la matrice de projection), proj(nxn) = X inv[X'X] X' :

```
projX3 = X3 %*% invX3 %*% t(X3)
```

Calculer la matrice des valeurs ajustées des régressions, Yhat = projX %*% Y:

```
Yhat3 = projX3 %*% Y                       # Vérifiez le contenu de Yhat3
```

Les valeurs ajustées Yhat sont les mêmes si les colonnes de X.mat sont simplement centrées

```
X4 = scale(X.mat[,1:3], center=TRUE, scale=FALSE)
Yhat4 = X4 %*% solve(t(X4)%*%X4) %*% t(X4) %*% Y
```

```
# Preuve : la somme des différences entre les matrices Yhat3 et Yhat4 est zéro
( diff = sum( abs(Yhat3 - Yhat4) ) )
```

Pourrait-on calculer Yhat à partir de la matrice X (qui contient 4 colonnes) plutôt que X3?

```
library(MASS)
invX = ginv( t(X) %*% X )      # Est-ce que "invX = solve(t(X) %*% X)" fonctionnerait aussi?
projX = X %*% invX %*% t(X)
Yhat5 = projX %*% Y                        # Vérifiez le contenu de Yhat5
```

Les valeurs ajustées dans Yhat5 sont-elles les mêmes que dans Yhat3 ?

```
( diff = sum( abs(Yhat3 - Yhat5) ) )
```

Pourquoi ginv() (inversion généralisée) fonctionne-t-elle malgré la colinéarité de la matrice X?
Consultez la documentation de la fonction ginv() de la bibliothèque MASS.

```
### Seconde partie de la RDA : ACP de la matrice des valeurs ajustées Yhat

# Calculer la matrice de covariance des valeurs ajustées :

  Yhat3.cov = cov(Yhat3)

# Calculer les valeurs propres et les vecteurs propres :

  Yhat3.eig = eigen(Yhat3.cov)

# Examiner les valeurs propres et les vecteurs propres:

  Yhat3.eig$values
  Yhat3.eig$vectors

# Calculer les valeurs propres comme fractions de la variance totale de Y.
# On calcule d'abord la somme des variances de Y.
# On peut le faire en sommant les termes diagonaux de la matrice de covariance de Y,
# ou encore en sommant les valeurs de Y mises au carré par produit Hadamard ; Y est déjà centrée.

  trace = sum(diag(cov(Y)))        # ou :   trace = (sum(Y*Y)) / (n-1)
  trace                            # ou :   trace = (sum(Y^2)) / (n-1)

  EigvalPercent = Yhat3.eig$values[1:3] / trace
  EigvalPercent

# Cadrage de type 1 : les diagrammes utiliseront les matrices F.mat et U ou les matrices Z.mat et U.
# Les équations sont expliquées dans Numerical ecology (2012), p. 639.
# Transférer les 3 premiers vecteurs propres dans U ('species' dans la liste de sortie de rda() ) :

  U = Yhat3.eig$vectors[,1:3]

# Calculer la matrice F.mat:   F.mat = Y * U ('sites' dans l'objet de sortie de rda() ) :

  F.mat = Y %*% U
  F.mat

# Calculer la matrice Z.mat:   Z.mat = Yhat * U ('constraints' dans l'objet de sortie de rda() )

  Z.mat = Yhat3 %*% U
  Z.mat
```

```
# Diagramme des deux premières colonnes de F.mat, avec l'axe 2 inversé;
# ajouter au diagramme les deux premières colonnes de la matrice U*10:

  plot(F.mat[,1], -F.mat[,2], xlim=c(-16,16), ylim=c(-10,10), asp=1, xlab="Axe 1", ylab="Axe 2")
  arrows(x0=0 ,y0=0 ,U[,1]*10 ,-U[,2]*10, code=0)

# Diagramme des deux premières colonnes de Z.mat (avec l'axe 2 inversé) et de U:

  plot(Z.mat[,1], -Z.mat[,2], xlim=c(-16,16), ylim=c(-10,10), asp=1, xlab="Axe 1", ylab="Axe 2")
  arrows(x0=0, y0=0, U[,1]*10, -U[,2]*10, code=0)

# Notes: bornes de l'abscisse = (-16,16), bornes de l'ordonnée = (-10,10).
# asp=1: le rapport des dimensions abscisse/ordonnée est fixé à 1.
# Arrows, code=0: flèches sans pointes; code=2: pointe à l'extrémité.

# Représentation des variables explicatives dans le diagramme.
# Calculer d'abord la matrice des corrélations entre X et Z.mat:

  corXZ=cor(X, Z.mat)
  corXZ

# Former la matrice diagonale 'D' des poids, sqrt(lambda(k)/trace):

  D = diag(sqrt(Yhat3.eig$values[1:3]/trace))

# Calculer la matrice de position de variables explicatives dans le graphique ('biplot' dans l'objet de
# sortie de rda() ; tracer les variables explicatives dans le diagramme:

  posX = corXZ %*% D
  arrows(x0=0, y0=0, posX[,1]*10, -posX[,2]*10, code=2)

# Note : pour le cadrage de type 1, il serait incorrect de tracer les variables explicatives en utilisant
# les valeurs de la matrice des corrélations 'corXZ' : les variables de classes de substrat ne
# pointeraient pas vers le centroïde des groupes de sites. Vous pouvez le vérifier par la commande :

  arrows(x0=0, y0=0, corXZ[,1]*10, -corXZ[,2]*10, code=2, col="red")

# ==========

# Exercice optionnel: écrivez les équations pour tracer un diagramme obéissant au cadrage de type 2.
Les équations sont décrites dans *Numerical ecology* (2012), p. 640.
```

6.1.2. Analyse canonique de redondance (RDA) à l'aide de 'vegan'

Les données se trouvent dans un fichier texte "Table_11-3.txt"
Lire les données et les transférer dans la matrice Table11.3.mat,
puis dans les matrices Y.mat (variables réponse, colonnes 1 à 6 de Table11.3.mat) et X.mat
(variables explicatives, colonnes 7 à 10):

```
Table11.3 = read.table( file.choose() )
Y.mat=Table11.3[,1:6]
X.mat=Table11.3[,7:10]
```

Y.mat

	Spec1	Spec2	Spec3	Spec4	Spec5	Spec6
Site1	1	0	0	0	0	0
Site2	0	0	0	0	0	0
Site3	0	1	0	0	0	0
Site4	11	4	0	0	8	1
Site5	11	5	17	7	0	0
Site6	9	6	0	0	6	2
Site7	9	7	13	10	0	0
Site8	7	8	0	0	4	3
Site9	7	9	10	13	0	0
Site10	5	10	0	0	2	4

X.mat

	Depth	Coral	Sand	Other
Site1	1	0	1	0
Site2	2	0	1	0
Site3	3	0	1	0
Site4	4	0	0	1
Site5	5	1	0	0
Site6	6	0	0	1
Site7	7	1	0	0
Site8	8	0	0	1
Site9	9	1	0	0
Site10	10	0	0	1

```
# Activer la bibliothèque 'vegan'
  library(vegan)
# Analyse canonique de redondance (ACR ou RDA, fonction 'rda')
  rda.out = rda(Y.mat, X.mat)          # Ordre des matrices : matrice réponse, matrice explicative
  rda.out
Call:
rda(X = Y.mat, Y = X.mat)

              Inertia Rank
Total         112.889
Constrained   108.341    3
Unconstrained   4.548    4
Inertia is variance
Some constraints were aliased because they were collinear (redundant)
           # La variable binaire "Other", qui était colinéaire, a été éliminée
Eigenvalues for constrained axes:
  RDA1   RDA2   RDA3
74.523 24.942  8.876

Eigenvalues for unconstrained axes:
     PC1      PC2      PC3      PC4
4.188785 0.313863 0.037037 0.008463
```

Examinez l'objet produit par la fonction rda(). Prenez connaissance des notes de l'encadré de la page 5, « Examinez le fichier de sortie produit par rda() ou cca() de {vegan} ».

Examinez les résultats de la RDA

summary(rda.out, scaling=1)
summary(rda.out, scaling=2)

Graphique :

plot(rda.out, scaling=2, display=c("sp","sites","bp"))

Calcul du R^2 et du R^2 ajusté :

RsquareAdj(rda.out)

Test de signification de la relation canonique :

anova(rda.out)

```
Permutation test for rda under reduced model

Model: rda(X = Y.mat, Y = X.mat)
         Df     Var      F  N.Perm Pr(>F)
Model     3 108.341 47.642 199.000  0.005 **
Residual  6   4.548
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
```

Information sur la fonction 'anova' :
?anova.cca

On peut imposer un nombre de permutations prédéterminé. Par exemple, pour obtenir 9999
permutations aléatoires, on écrit la commande comme suit :

anova(rda.out, permutations = how(nperm=9999))

# Il y a d'autres façons de spécifier le modèle de RDA :
rda.out = rda(Table11.3[,1:6] ~ Depth + Coral + Sand + Other, data=Table11.3)

# ou encore
rda.out = rda(Y.mat ~ . , data=X.mat)

Le caractère ~ signifie « fonction de », comme dans les modèles de régression.
La déclaration 'data=' spécifie dans quel fichier se trouvent les variables qui suivent le tilde (~). Ce fichier doit posséder la classe 'data.frame'.
Le point (.) qui suit le tilde (~) indique d'inclure toutes les variables du fichier « data » comme variables explicatives dans le modèle.
Il faut utiliser la notation fonctionnelle pour obtenir les tests de signification by="axis", by="terms" ou by="margin" (page suivante).

==========

```
# 6.1.3. Test des valeurs propres en analyse canonique

  Table11.3 = read.table( file.choose() )        # Lire le fichier "Table_11-3.txt"
  Table.spe = Table11.3[,1:6]
  Table.env = Table11.3[,7:9]

# 1. Test automatique des valeurs propres canoniques individuelles : méthode 'marginal' de vegan

  toto = rda(Table.spe ~ Depth + Coral + Sand, data=Table.env)
  anova(toto, by="axis")

# 2. Comment ce test fonctionne-t-il ? Test manuel des valeurs propres canoniques individuelles
# Examinez d'abord la structure de l'objet produit par la fonction rda()
  str(toto)

# La position des objets dans l'espace canonique se trouve dans la matrice 'toto$CCA$u'
  toto$CCA$u

# Tester les trois axes, en succession, en présence des axes précédents (matrice des covariables).
# C'est la méthode 'forward' de Cajo J. F. ter Braak.

  rda.axe1 = rda(Table.spe, toto$CCA$u[,1])
  anova(rda.axe1)

  rda.axe2 = rda(Table.spe, toto$CCA$u[,2], toto$CCA$u[,1])
  anova(rda.axe2)

  rda.axe3 = rda(Table.spe, toto$CCA$u[,3], toto$CCA$u[,1:2])
  anova(rda.axe3)

# 6.1.4. Tests de signification by="terms" et by="margin"

# Test de signification de la contribution additionnelle de chaque variable explicative, de façon
séquentielle, en suivant l'ordre de leur inclusion dans le modèle. Comparez les résultats suivants :

  toto1 = rda(Table.spe ~ Depth+Coral+Sand, data=Table.env)
  anova(toto1, by="terms")

  toto2 = rda(Table.spe ~ Sand+Coral+Depth, data=Table.env)
  anova(toto2, by="terms")

# Test de la contribution additionnelle (marginale) de chaque variables explicatives en présence de
toutes les autres variables du modèle. Comparez les résultats des deux analyses suivantes :

  anova(toto1, by="margin")
  anova(toto2, by="margin")
```

6.2. Analyse canonique de redondance **partielle**
6.2.1. Exemple d'analyse canonique de redondance **partielle** par algèbre matricielle

```
# Exemple : calculer la contribution partielle (unique) de la variable 'Profondeur'
# à la variation de Y, en présence des autres variables de la matrice X.

# Lire les données, etc.

  Table11.3 = read.table( file.choose() )        # Lire le fichier "Table_11-3.txt"
  Table11.3.mat = as.matrix(Table11.3)
  Y.mat=Table11.3.mat[,1:6]
  X.mat=Table11.3.mat[,7:10]
  Y = scale(Y.mat, center=TRUE, scale=FALSE) # ou : Y = apply(Y.mat,2,scale,center=T,scale=F)
  X = scale(X.mat, center=TRUE, scale=TRUE)  # ou : X = apply(X.mat,2,scale,center=T,scale=T)
  Y
  X

# Former la matrice XX contenant la variable 'Profondeur' et la
# matrice de covariables W contenant les trois autres variables de X:

  XX = X[,1]
  W = X[,2:4]

# Régresser la profondeur (XX) sur les covariables (W) ;
# calculer les résidus de cette régression:

  library(MASS)
  invWprW = ginv(t(W) %*% W)
  projW = W %*% invWprW %*% t(W)
  Xres.W = XX - (projW %*% XX)

# Calculer l'analyse canonique (RDA) de Y par Xres.W:

  invXprX = ginv(t(Xres.W) %*% Xres.W)
  projXres.W = Xres.W %*% invXprX %*% t(Xres.W)
  YhatXX = projXres.W %*% Y
  YhatXX.cov = cov(YhatXX)
  YhatXX.eig = eigen(YhatXX.cov)
  Z.mat = YhatXX %*% YhatXX.eig$vectors[,1]

# Examiner la valeur propre et la position des objets sur l'axe canonique:
  YhatXX.eig$values[1]
  Z.mat

# Exprimer la valeur propre canonique comme une fraction de la variance totale de Y
# (valeur propre = 0.083, P = 0.001) :

  EigvalPercent = YhatXX.eig$values[1] / sum(diag(cov(Y)))
  EigvalPercent

# Cette valeur propre standardisée indique la contribution partielle de la variable 'Profondeur' à
# l'explication de la variance de Y en présence des autres variables de la matrice X (R² semipartiel).
```

6.2.2. Exemple d'analyse canonique de redondance **partielle** à l'aide de 'vegan'
#
Calculer la contribution partielle (unique) de la variable 'Profondeur' à l'explication de la variance
de Y en présence des autres variables de la matrice X.

```
  rda.out = rda(Table11.3[1:6] ~ Depth + Condition(Coral+Sand+Other), data = Table11.3)
# ou encore (mêmes résultats) :
  rda.out = rda(Y.mat, X.mat[,1], X.mat[,2:4])
  anova(rda.out)
```

```
Permutation test for rda under reduced model

Model: rda(X = Y.mat, Y = X.mat[, 1], Z = X.mat[, 2:4])
          Df    Var      F N.Perm Pr(>F)
Model      1 9.3407 12.322    999  0.002 **
Residual   6 4.5481
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
```

Calcul du R^2 semipartiel ; il n'y a pas de R^2 ajusté en RDA partielle

```
  RsquareAdj(rda.out)
```

6.3. Analyse canonique des correspondances
Attention : il y a des fonctioncs cca() dans d'autres bibliothèques R, en particulier dans ade4.

```
# Exemple 1 de CCA: araignées des dunes des Pays-Bas
  library(vegan)
# Télécharger d'abord les deux tableaux de données (réponse et explicatives)
  spiders.spe = read.table( file.choose() )      # Lire le fichier "Spiders_28x12_spe.txt"
  spiders.env = read.table( file.choose() )      # Lire le fichier "Spiders_28x4_env.txt"
  ( spider.cca = cca(spiders.spe, spiders.env) )
  plot(spider.cca, scaling=1)
  summary(spider.cca, scaling=1, axes=3)
  anova(spider.cca)

# Exemple 2 de CCA: les cafés de Neuchâtel (D. Borcard)
  library(vegan)
  cafes.spe = read.table( file.choose() )        # Lire le fichier "Cafes_10x6_spe.txt"
  cafes.env = read.table( file.choose() )        # Lire le fichier "Cafes_10x3_env.txt"
  ( cafes.cca = cca(cafes.spe, cafes.env) )
  plot(cafes.cca, scaling=2)
  summary(cafes.cca, scaling=2, axes=3)
  anova(cafes.cca)
```

7. Analyse de variance multivariable (manova) à deux facteurs par RDA

Utilisation de la RDA pour réaliser une analyse de variance multivariable (manova) à 2 facteurs.

Les deux facteurs croisés (A, B) sont représentés par des contrasts de Helmert. L'interaction des deux facteurs est représentée par des variables qui sont le produit des variables codant pour les facteurs principaux. Propriétés : (1) la somme de chaque variable codante est 0 ; (2) les variables codantes sont orthogonales entre elles (leur produit scalaire est 0). Les variables représentant A sont orthogonales aux variables représentant B ainsi qu'aux variables représentant l'interaction A*B.

1. Lire le tableau de données (24 lignes, 5 colonnes). Examinez la structure de ce fichier.

```
table.sp = read.table( file.choose() )          # Lire le fichier "Table_sp.txt"
table.sp
```

Étiquettes des lignes : les 3 niveaux de A et les 4 niveaux de B, ainsi que les répétitions (1, 2).

2. Générer les variables représentant les deux facteurs

La fonction 'gl' générera une variable représentant le facteur A (3 niveaux et 8 répétitions)

```
A = gl(3, 8)
```

'gl' générera une variable représentant B (4 niveaux et 2 répétitions, le tout répété 3 fois)

```
B = gl(4, 2, length=24)
```

Vérifier si les valeurs produites correspondent bien à l'agencement des facteurs dans le tableau

3. Créer les contrastes de Helmert pour les facteurs A et B ainsi que pour l'interaction A*B

Étudier les fichiers d'aide des fonctions 'model.matrix', 'contrast' et 'contr.helmert'

```
helmert = model.matrix(~ A*B, contrasts=list(A="contr.helmert", B="contr.helmert"))
```

Examiner ce fichier. Quelles colonnes représentent A ? B ? l'interaction ?

```
apply(helmert[,2:12], 2, sum)           # Vérification de la propriété 1 des contrastes de Helmert
t(helmert[,2:12]) %*% helmert[,2:12]    # Vérification de la propriété 2; ou: cor(helmert[,2:12])
```

4. Tester l'interaction à l'aide de la RDA. Les facteurs A et B forment la matrice des covariables

```
interaction.rda = rda(table.sp, helmert[,7:12], helmert[,2:6])
anova(interaction.rda, model="direct")      # Interaction significative?
```

5. Tester le facteur principal A par RDA. B et l'interaction forment la matrice des covariables

```
factorA.rda = rda(table.sp, helmert[,2:3], helmert[,4:12])
anova(factorA.rda, model="direct")                          # Le facteur A est-il significatif?
```

6. Tester le facteur principal B par RDA. A et l'interaction forment la matrice des covariables

```
factorB.rda = rda(table.sp, helmert[,4:6], helmert[,c(2:3, 7:12)])
anova(factorB.rda, model="direct")                          # Le facteur B est-il significatif?
```

Pouvez-vous calculer le R-carré correspondant à l'explication par le facteur A ? le facteur B ?

8. Sélection des variables explicatives en RDA

8.1. Sélection progressive à l'aide de ***forward.sel()*** de la bibliothèque 'adespatial'

Cette fonction utilise deux fichiers: les variables réponse et les variables explicatives.

```
library(adespatial)
# Examinez le fichier de documentation de la fonction forward.sel()
  ?forward.sel

# Charger les deux tableaux de données (variables réponse et explicatives)

  spiders.spe = read.table( file.choose() )      # Lire le fichier "Spiders_28x12_spe.txt"
  spiders.env = read.table( file.choose() )      # Lire le fichier "Spiders_28x15_env.txt"
```

La fonction ***forward.sel()*** produit un tableau montrant l'inclusion pas à pas des variables explicatives. L'utilisateur décidera à partir de ce tableau quelles variables il conservera pour la RDA par "vegan".

```
  res=forward.sel(spiders.spe, spiders.env)
```

Méthode rapide pour fournir à la fonction ***rda()*** de vegan les colonnes (variables) de "spiders.env" à conserver. Exemple où l'on conserve les 3 premières variables :

```
  rda(spiders.spe, spiders.env[, res$order[1:3]])
```

8.2. Sélection progressive, régressive ou pas à pas à l'aide de ***ordistep()*** ou ***ordiR2step()***

Les fonctions ***ordistep()*** et ***ordiR2step()*** sont disponibles dans vegan.
La fonction ***ordiR2step()*** produit les même sélection que ***forward.sel()*** décrit à la section 8.1.

Examinez le fichier de documentation et faites tourner les exemples.

```
?ordistep                                  # ou :  ?ordiR2step
```

9. Détendançage spatial

Avant de réaliser des analyses spatiales, il est important dans la plupart des cas d'éliminer la tendance spatiale linéaire qui pourrait être présente dans les variables-réponse. Tel est le cas de l'analyse dbMEM (section 10). Une tendance linéaire dans des données représente habituellement un processus qui a une taille supérieure à celle de la zone d'étude. Un tel processus ne peut être efficacement modélisé en utilisant seulement les données provenant de la zone d'étude.

1. Détendançage par régression, fonction ***lm()***. Un exemple univariable.

```
var1 = c(1:20)
var2 = rnorm(20,0,3)
var3 = var1 + var2
plot(var1, var3)        # Examiner la variable 'var3' produite en fonction de 'var1' (abscisse)

model.lm = lm(var3 ~ var1)
resid = residuals(model.lm)
plot(var1, resid)       # Examiner les résidus en fonction de 'var1' (abscisse)
```

==========

2. Détendançage par régression multivariable, manuel éq. 10.16 (algèbre matricielle).
Un exemple multivariable : les données d'oribates disponibles dans {vegan}

```
library(vegan)
data(mite)                          # Composition de la communauté, 70 x 35 espèces
data(mite.xy)                       # Coordonnées des sites, 70 x 2
Y = mite
X = mite.xy

# Centrer Y et X
Y = scale(Y, center=TRUE, scale= FALSE)
X = scale(X, center= TRUE, scale= FALSE)
# Équation matricielle pour la régression multivariable et le calcul des résidus
mite.res = Y - (X %*% solve(t(X)%*%X) %*% t(X) %*% Y)
```

==========

3. Détendançage de données multivariables par régression, fonction ***lm()***
Un exemple multivariable : les données d'oribates

```
mite.lm = lm(as.matrix(mite) ~ ., mite.xy)
mite.resid = residuals(mite.lm)
head(mite.resid)

# Vérifier que les deux résultats sont identiques

sum(abs(mite.res - mite.resid))         # Les deux séries de résidus ne diffèrent pas
diag(cor(mite.res, mite.resid))         # Les deux séries de résidus ont des corrélations de 1
```

10. Analyse spatiale (ou temporelle) dbMEM (appelée autrefois *analyse PCNM*)[2]

```
# Charger les bibliothèques R requises pour cet exercice.
# Il faut charger les bibliothèques R dans l'ordre indiqué: ade4 avant adegraphics. Ainsi, certaines fonctions de ade4 qui ont le même nom que des fonctions de adegraphics seront cachées par les fonctions de adegraphics. À terme, les fonctions de adegraphics remplaceront les fonctions de ade4 de même nom et celles-ci seront éliminées.

library(ade4)
library(adegraphics)
library(adespatial)

# Charger les données d'oribates disponibles dans {vegan}

library(vegan)
data(mite)                  # Données de communautés disponibles dans "vegan"
data(mite.xy)               # Coordonnées géographiques disponibles dans "vegan"

# Transformation de Hellinger des données d'espèces (Travaux pratiques, p. 16)
mite.hel <- decostand(mite, "hellinger")

# Lire le fichier "mite_env.txt" disponible dans le dossier des travaux pratiques
mite_env = read.table( file.choose() )

# Tracer une carte grossière des 70 points d'échantillonnage
plot(mite.xy, asp=1)
text(mite.xy, labels=rownames(mite.xy), pos=3)

# ========

# Analyse dbMEM par étapes

# 1. Détendancer les données d'espèces  (Travaux pratiques, p. 33)

mite.lm = lm(as.matrix(mite.hel) ~ as.matrix(mite.xy))
mite.resid = residuals(mite.lm)
```

[2] La fonction ***pcnm()*** de vegan produit des *PCNM classiques*, pour lesquelles les valeurs propres ne correspondent pas aux indices *I* de Moran; les vecteurs propres qui modélisent la corrélation spatiale positive sont, cependant, parfaitement corrélés aux fonctions dbMEM de même rang. Les deux types jouent le même rôle lorsqu'ils sont utilisés comme variables explicatives dans un modèle de régression ou de RDA. Typiquement, pour *n* sites, les fonctions PCNM produisent un nombre de fonctions propres avec valeurs propres positives égal à environ 2*n*/3. Le dernier tiers de ces 2*n*/3 fonctions ont des *I* de Moran négatifs, ce qui indique qu'elles modélisent de la corrélation spatiale négative. La fonction ***pcnm()*** de vegan peut utiliser des poids, ce qui lui permet de construire des variables PCNM qui demeurent orthogonales lorsqu'elles sont utilisées en analyse canonique des correspondances (fonction ***cca()*** de vegan, section 6.3 des présents *Travaux pratiques*).

2. Construire les fonctions dbMEM à l'aide de la fonction ***dbmem()***. Notez que ***dbmem()*** peut utiliser *soit* les coordonnées xy des sites, *soit* une matrice de distances géographiques.
Avec la valeur par défaut de MEM.autocor, ***dbmem()*** ne calcule que les vecteurs propres qui modélisent la corrélation spatiale positive. MEM.autocor propose cependant d'autres options.

```
?dbmem

mite.dbmem = dbmem(mite.xy, silent=FALSE)     # Notez la valeur de troncage
summary(mite.dbmem)

# Examiner les valeurs propres positives
attributes(mite.dbmem)$values

# Extraire les vecteurs dbMEM qui modélisent la corrélation spatiale positive
mite.mem = as.matrix(mite.dbmem)
dim(mite.mem)                  # Combien y a-t-il de vecteurs propres ?

# Tracer la carte des liens qui sont plus longs que la valeur de troncage
s.label(mite.xy, nb = attr(mite.dbmem, "listw"))

# Tracer les 9 premières fonctions dbMEM à l'aide de la fonction s.value() de adegraphics :
s.value(mite.xy, mite.dbmem[,1:9])

# Ce graphique peut aussi être produit à l'aide de la fonction s.value() de ade4 (plus lente); cela
demande cependant d'écrire une boucle for :

par(mfrow=c(3,3))
for(i in 1:9) {
ade4::s.value(mite.xy, mite.mem[ ,i], addaxes=FALSE, include.origin=FALSE,
  sub=paste("dbMEM #",i), csub=1.5)
}

# Calculer et tester les indices I de Moran associés aux vecteurs propres dbMEM.
# On peut vérifier que les valeurs propres sont parfaitement proportionnelles aux I de Moran.

( test <- moran.randtest(mite.dbmem, nrepet = 999) )
plot(test$obs, attr(mite.dbmem, "values"), xlab = "Moran's I", ylab = "Eigenvalues")

# Tracer les valeurs décroissantes du I de Moran correspondant aux fonctions dbMEM successives.
# La ligne rouge représente l'espérance (ou valeur attendue) du I de Moran sous H0.

plot(test$obs, xlab="MEM rank", ylab="Moran's I")
abline(h=-1/(nrow(mite.xy) - 1), col="red")

# 3. Calculer le R² et le R²ajusté du modèle qui inclut toutes les fonctions dbMEM positives

RsquareAdj(rda(mite.resid, mite.mem))
# 4. Sélection pas à pas (p. 28) : identifier les fonctions dbMEM significatives parmi celles qui
modélisent de la corrélation spatiale positive. Le R²ajusté ci-dessus est le critère d'arrêt "adjR2thresh".

( mite.sel = forward.sel(mite.resid, mite.mem, adjR2thresh=0.2455) )

# La liste des fonctions dbMEM retenues se trouve dans le vecteur "mite.sel$order"
( mite.sel$order )

# Obtenir une liste de ces variables en ordre croissant :
```

```
( sel.dbmem = sort(mite.sel$order) )

# Examiner le R²ajusté du modèle contenant les 8 fonctions dbMEM sélectionnées (ligne 8 du tableau):
mite.sel[8,]

# 5. Tracer la carte des dbMEM sélectionnées à l'aide de la fonction s.value() de adegraphics :

s.value(mite.xy, mite.mem[ ,sel.dbmem])

# Ce graphique peut aussi être produit par s.value() de ade4 (plus lente) en écrivant une boucle :

par(mfrow=c(3,3))
for(i in 1:length(sel.dbmem)) {
ade4::s.value(mite.xy, mite.mem[ ,sel.dbmem[i]], addaxes=FALSE, include.origin=FALSE,
sub=paste("dbMEM #", sel.dbmem[i]), csub=1.5) }

# 6. Analyse canonique (RDA) : calculer le modèle spatial dbMEM

mite.rda = rda(mite.resid ~ ., data=as.data.frame(mite.mem[,sel.dbmem]))
anova(mite.rda)
anova(mite.rda, by="axis")

# Combien d'axes du modèle canonique sont significatifs au niveau alpha = 0.05 ?
anova(mite.rda, by="axis", cutoff=0.05)

# 7. Tracer les 3 axes canoniques significatifs à l'aide de s.value() de ade4 (cartes plus grandes)

par(mfrow=c(1,3))
for(i in 1:3) {
ade4::s.value(mite.xy, mite.rda$CCA$u[,i], addaxes=FALSE, include.origin=FALSE,
sub=paste("Canonical axis #",i), csub=1.5) }

# Ces cartes auraient pu être obtenues à l'aide de s.value() de adegraphics. Pouvez-vous le faire ?

# 8. D'autres exemples d'analyses dbMEM sont décrits dans la documentation de la fonction.
?dbmem

# ========
```

11. Partitionnement de la variation

La fonction **varpart** de {vegan} permet de partitionner la variation d'un tableau-réponse **Y** en fonction de 2, 3 ou 4 tableaux explicatifs **X1**, **X2**, **X3**, **X4**. La méthode d'analyse sous-jacente est l'analyse canonique de redondance (RDA). Il n'est pas nécessaire d'éliminer les variables colinéaires des tableaux explicatifs avant la partition. Les transformations disponibles dans **decostand** (décrites dans la section 4 ci-dessus) peuvent être requises dans l'appel à la fonction **varpart**. Comme toujours, on obtiendra la documentation de **varpart** par la commande :

```
library(vegan)
?varpart
```

Les exemples fournis avec la fonction utilisent les données d'acariens du sol de Borcard.

```
data(mite)
data(mite.env)
data(mite.pcnm)
```

Exemple pour **deux tableaux explicatifs**, "mite.env" et "mite.pcnm". "mite" est le tableau réponse
Y. "mite.env" contient des variables qualitatives multiclasses codées sous la forme de facteurs.

Il y a deux façons d'écrire le modèle d'analyse :
1. On peut écrire les noms de fichier l'un après l'autre, en commençant par le tableau réponse **Y**
suivi de la liste des tableaux explicatifs. Le format général est : resultat = varpart(Y, X1, X2)

Le tableau 'mite.env' contient la variable nominale 'Substrate' codée sous forme de facteur. Il faut d'abord créer un nouveau tableau dans lequel cette variable sera recodée en variables binaires. La variable ordinale 'Shrub' contribue de façon essentiellement linéaire à l'explication de 'mite'.

```
mm = model.matrix(~ SubsDens +WatrCont +Substrate +as.numeric(Shrub) +Topo, mite.env)[,-1]
```

La fonction 'model.matrix' crée une première colonne de valeurs '1' qui sont utilisées en régression pour calculer l'ordonnée à l'origine. L'élément [,-1] à la fin de la ligne de commande enlève cette colonne avant de créer l'objet 'mm'.

```
resultat2 = varpart(mite, mm, mite.pcnm, transfo="hel")
resultat2
```

Le paramètre **transfo="hel"** spécifie d'appliquer la transformation de Hellinger au tableau réponse ('mite') avant de calculer le partitionnement de la variation.

2. Les chercheurs qui ont l'habitude d'écrire les modèles de régression avec un tilde (~) dans la fonction 'lm' peuvent utiliser la même forme dans 'varpart'. Il n'est alors pas nécessaire de recoder en variables binaires les variables codées en facteurs avant d'utiliser 'varpart'.

```
resultat2 <- varpart(mite, ~ SubsDens + WatrCont + Substrate + as.numeric(Shrub) + Topo,
mite.pcnm, data=mite.env, transfo="hel")
resultat2                    # Same result as above
showvarparts(2)              # Dessiner un diagramme de Venn indiquant le nom de chaque fraction
plot(resultat2, bg=2:3)      # Dessiner un diagramme de Venn indiquant la valeur de chaque fraction
```

Cette nouvelle version de varpart est apparue avec la version 2.3-1 de vegan. Elle produit des diagrammes de Venn colorés sur demande. Le choix des couleurs est spécifié par le paramètre 'bg' (*background colours*).

La signification des fractions 'testables' peut être testée à l'aide de la fonction **rda**.
Par exemple, nous pouvons tester la signification de la fraction [a] comme suit.
La fonction **rda** ne permet pas d'inclure un tableau de covariables se trouvant dans un fichier séparé lorsqu'on utilise la formule avec un tilde (~). Avant l'appel à **rda**, il faut d'abord recoder la variable

nominale du tableau 'mite.env' en variables binaires si cela n'a pas déjà été fait.

```
mm = model.matrix(~ SubsDens +WatrCont +Substrate +as.numeric(Shrub) +Topo, mite.env)[,-1]
```

L'appel à **rda** qui suit contient un appel à **decostand** pour réaliser la transformation de Hellinger ("hel") du tableau-réponse des abondances d'espèces "mite" avant la **rda**.

```
rda.result = rda(decostand(mite, "hel"), mm, mite.pcnm)
anova(rda.result)
```

Exemple pour **trois tableaux explicatifs**. Les deux appels suivants à **varpart** sont équivalents.

1. Il faut d'abord créer des matrices binaires pour recoder les variables qui sont des facteurs :

```
mm1 = model.matrix(~ SubsDens + WatrCont, mite.env)[,-1]
mm2 = model.matrix(~ Substrate + as.numeric(Shrub) + Topo, mite.env)[, -1]

resultat3 = varpart(mite, mm1, mm2, mite.pcnm, transfo="hellinger")
showvarparts(3)          # Pas de couleurs puisque 'bg' n'a pas été spécifié
plot(resultat3, bg=2:4) # Les numéros déterminent les couleurs ; essayez différentes combinaisons
```

Des paramètres additionnels permettent d'imprimer les fractions ayant des valeurs négatives (celles-ci ne sont pas imprimées si on utilise les options par défaut) et de contrôler le nombre de décimales des freactions ('digit') et la taille des caractères ('cex'). Exemple :

```
plot(resultat3, cutoff = -Inf, digits=5, cex = 0.7, bg=5:7)
```

2. Si on préfère écrire le modèle avec un tilde (~) :

```
resultat3 = varpart(mite, ~ SubsDens+WatrCont, ~ Substrate + as.numeric(Shrub) + Topo,
mite.pcnm, data=mite.env, transfo="hel")
showvarparts(3, bg=2:4)
plot(resultat3, bg=5:7)
```

Pour **quatre tableaux explicatifs**, la fonction s'écrit dans le format général suivant :

```
resultat4 = varpart(Y, X1, X2, X3, X4)      # Y = tableau-réponse, X[1 ... 4] = tableaux explicatifs
showvarparts(4, bg=4:7)
```

Truc : on peut utiliser **varpart** pour obtenir le *R*-carré ajusté d'une seule matrice explicative. Il suffit de fournir le nom de la matrice explicative deux fois. Exemple : resultat1 = varpart(Y, X1, X1)

12. Méthodes de groupement et de partitionnement

12.1. Partitionnement par les *K* centroïdes (*K*-means)

# 1. **Préparation du fichier.** Exemple: fichier "Ours.txt" (attention : en-tête à la colonne 1)
Ours = read.table(file.choose(), header=TRUE, row.names=1)
# Éliminer la colonne 5, variable qualitative. K-means demande des variables quantitatives.
Ours.mat = as.matrix(Ours[,1:4])

Centrer et réduire les variables car elles ne possèdent pas les mêmes dimensions physiques.

Ours2 = apply(Ours.mat, 2, scale, center=TRUE, scale=TRUE)

Note: le paramètre '2' indique de calculer le centrage par colonne.
'center=TRUE' commande le centrage, 'scale=TRUE' la réduction.

On peut aussi réaliser le centrage-réduction à l'aide de la fonction decostand() de {vegan} :

Ours2 = decostand(Ours.mat, "stand")

2. **Calculer le partitionnement** en 5 groupes (paramètre 'centers') à l'aide de cclust() de la bibliothèque {cclust}. Cette fonction propose plusieurs méthodes de groupement convexe ; la méthode par défaut est K-means. cclust() n'est pas la meilleure fonction pour K-means car elle ne permet pas de répéter automatiquement l'analyse un grand nombre de fois. Elle demeure cependant intéressante pour d'autres raisons.

library(cclust)
result.cclust.5 = cclust(Ours2, centers=5)
result.cclust.5 # Un bref résumé des résultats
summary(result.cclust.5) # Structure du fichier de sortie où se trouvent les résultats détaillés

#On peut produire des graphiques montrant les groupes sur des diagrammes de dispersion:
plot(Ours.mat[,1:2], col=result.cclust.5$cluster) # Groupement sur graphique des variables 1 et 2
plot(Ours.mat[,3:4], col=result.cclust.5$cluster) # Groupement sur graphique des variables 3 et 4

Extension à l'ACP: calculer une ACP de 'Ours.mat', puis tracer le groupement sur l'ordination :

Ours.pca = rda(Ours2) # ACP calculée par la fonction 'rda' de 'vegan'
plot(summary(Ours.pca)$sites, col=result.cclust.5$cluster)

Calculer des critères d'arrêt par la fonction 'clustIndex' de 'cclust'. Ces critères permettent de
choisir la meilleure partition parmi celles qui correspondent aux différentes valeurs de K.
Milligan & Cooper (1985) recommandent de maximiser l'indice de Calinski-Harabasz (stat. *F*).
Le maximum de 'ssi' est un autre bon indice de la meilleure partition au sens des moindres carrés.

clustIndex(result.cclust.5, Ours2)
clustIndex(result.cclust.5, Ours2, index=c("calinski","ssi")) # Calculer 2 indices seulement
3. **Calculer le partitionnement** en 5 groupes (paramètre 'centers') à l'aide de 'kmeans' de 'stats'.

Cette fonction est plus intéressante parce qu'elle permet de répéter automatiquement l'analyse K-means un grand nombre de fois (paramètre 'nstart'). La fonction trouve la meilleure solution (plus petite valeur de de la somme des carrés des écarts intragroupes, SSC) après 'nstart' répétitions.

resultat.km.5 = kmeans(Ours2, centers=5, nstart=100)
resultat.km.5

```
-----
K-means clustering with 5 clusters of sizes 5, 6, 5, 5, 8

Cluster means:
  Age.estime   Masse.kg Hauteur.cm Longueur.cm
1  0.9404982 -0.3309587  0.8412004  -0.4314848
2 -0.4429883  0.4064977  0.2937206   0.9624910
3 -1.0359111 -0.9007854 -1.6099529  -0.9332289
4  1.3357801  1.8625307  1.1580161   1.4969575
5 -0.4429883 -0.6991148 -0.4635802  -0.8045206

Clustering vector:
 [1] 5 5 3 3 2 2 2 3 3 2 3 5 5 2 5 5 5 5 2 1 1 4 4 1 1 4 4 4 1

Within cluster sum of squares by cluster:
[1] 1.8493394 3.3117416 0.9739777 0.6940644 4.1964065

Available components:
[1] "cluster"  "centers"  "withinss" "size"
-----
```

Examiner la structure du fichier de sortie. On peut utiliser les sous-fichiers dans des analyses subséquentes.

summary(resultat.km.5)

Calculer la somme des carrés des résidus intragroupes (statistique SCE) :

SCE = sum(resultat.km.5$withinss)
SCE
[1] 11.02553

Calculer des critères d'arrêt par la fonction 'clustIndex' de 'cclust' :

clustIndex(resultat.km.5, Ours2, index=c("calinski","ssi"))

Répéter l'analyse K-means pour K = 2, K = 3, K = 4, K = 5 et comparer la valeur de ces indices.
Les indices ont-ils une valeur maximum à une valeur intermédiaire de K? Quelle est la meilleure partition? Quel indice vous apparaît être le plus utile?

resultat.km.2 = kmeans(Ours2, centers=2, nstart=100)
clustIndex(resultat.km.2, Ours2, index=c("calinski","ssi"))
resultat.km.3 = kmeans(Ours2, centers=3, nstart=100)
clustIndex(resultat.km.3, Ours2, index=c("calinski","ssi"))
resultat.km.4 = kmeans(Ours2, centers=4, nstart=100)
clustIndex(resultat.km.4, Ours2, index=c("calinski","ssi"))
resultat.km.5 = kmeans(Ours2, centers=5, nstart=100)
clustIndex(resultat.km.5, Ours2, index=c("calinski","ssi"))
resultat.km.6 = kmeans(Ours2, centers=6, nstart=100)
clustIndex(resultat.km.6, Ours2, index=c("calinski","ssi"))
resultat.km.7 = kmeans(Ours2, centers=7, nstart=100)
clustIndex(resultat.km.7, Ours2, index=c("calinski","ssi"))

4. **Calculer le partitionnement** pour une gamme de valeurs de K à l'aide de la fonction
'cascadeKM' de la bibliothèque 'vegan'. Cette fonction est une *enveloppe* pour la fonction
'kmeans' de la bibliothèque 'stats' : elle utilise cette fonction de base en lui ajoutant de nouvelles

```
# propriétés. Elle crée plusieurs partitions qui forment une cascade de valeurs de K allant de petite
# (paramètre 'inf.gr') à grande (paramètre 'sup.gr').

resultat.cascadeKM = cascadeKM(Ours2, inf.gr=2, sup.gr=10, iter = 100, criterion = 'ssi')

# Examiner la structure du fichier de sortie :

summary(resultat.cascadeKM)

# L'élément 'partition' contient un tableau montrant à quel groupe chaque objet a été attribué :

resultat.cascadeKM$partition

# L'élément 'results' présente la statistique SCE pour chaque partition ainsi que la valeur du critère
("calinski" or "ssi") qui a été sélectionné. Il est alors facile d'identifier la partition qui maximise le
critère. L'élément 'size' fournit le nombre d'objets dans chaque groupe, pour chaque partition.

# Un graphique montre le groupe attribué à chaque objet de chaque partition ; les partitions forment les
lignes du graphique. Les groupes sont représentés par des couleurs différentes ; il y a 2 couleurs pour
K = 2, trois pour K = 3, et ainsi de suite. Un autre graphique montre les valeurs du critère d'arrêt pour
les différentes valeurs de K :

out.Ours = plot(resultat.cascadeKM)

# L'option 'sort=TRUE' réordonne les objets afin de regrouper, autant que faire se peut, les différents
objets qui appartiennent à chaque groupe :

out.Ours = plot(resultat.cascadeKM, sortg=TRUE)

# L'objet R produit, qui s'appelle 'out.Ours' dans cet exemple, contient un tableau montrant à quel
groupe chaque objet a été attribué, en tenant compte du nouvel ordre des objets.
```

```
# 12.2. Groupement hiérarchique

# Exemple: fichier 'Ours.txt'
# On doit d'abord centrer et réduire les variables de la matrice Ours
# avant de calculer la matrice de distances euclidiennes (voir la section 3).

Ours = read.table(file.choose(), header=TRUE, row.names=1)      # Lire le fichier "Ours.txt"
Ours.mat = as.matrix(Ours[,1:4])
Ours2 = decostand(Ours.mat, "stand")          # Centrer et réduire les variables
Ours.D1 = dist(Ours2, method="eucl")          # Calcul de la distance euclidienne

# Méthodes disponibles dans hclust: "ward.D", "ward.D2", "single", "complete", "average" (=
UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC)" and "centroid" (= UPGMC).

# Groupement agglomératif, méthode UPGMA : hclust(D, method = "average", members=NULL)
Ours.cl = hclust(Ours.D1, method="average")
plot(Ours.cl)                                 # Tracer le dendrogramme
plot(Ours.cl, hang=-1)

# Examinez les fonctions suivantes :
?identify
?rect.hclust
?cutree
?dendrogram                                   # Voir en particulier l'option "horiz"

# Calculer la corrélation cophénétique        # Fichier d'aide :  ?cophenetic
Ours.coph = cophenetic(Ours.cl)               # Distances cophénétiques du dendrogramme
cor(Ours.D1, Ours.coph)                       # Corrélation cophénétique

# =========
# Exemples de groupement hiérarchique : Vis et boulons

# 1. Groupement hiérarchique utilisant seulement les variables binaires

Vis=read.table( file.choose() )               # Lire le fichier "Vis_et_boulons.txt"
Vis.bin=Vis[, 2:9]
library(ade4)
vis.D1=dist.binary (Vis.bin, method=1)        # Calcul de D = sqrt(1 – similarité de Jaccard)
vis.cl=hclust(vis.D1, method="average")
plot(vis.cl)

# 2. Lire une matrice de distance carrée (en ASCII) pré-calculée par un autre programme

vis.S15=read.table( file.choose() )           # Lire le fichier "Vis_D=1-S15.txt"
vis.D=as.dist(vis.S15)                        # Attribue le type "dist" à la matrice de distances
vis.cl2=hclust(vis.D, method="average")
plot(vis.cl2)
# =========

# Groupement agglomératif hiérarchique : la méthode de Ward
```

Il existe deux algorithmes différents dans la littérature qui portent le nom de « groupement de Ward ». Dans la fonction ***hculst*** de R version >= 3.1.1, l'algorithme utilisé par l'option "ward.D" *ne met pas en oeuvre* le critère de groupement de Ward (1963) ; c'est aussi le cas de l'option "ward" dans ***hclust*** de R version <= 3.0.3. Après la publication de Murtagh and Legendre (2013), la fonction ***hclust***

a été modifiée dans la version 3.1.1 de R ; la nouvelle option "ward.D2" met en œuvre le critère de Ward (1963). Dans cet algorithme, les dissimilarités sont mises au carré avant le début du groupement. La fonction ***agnes***(*, method="ward") (*AGglomerative NESting*) de {cluster} met également en œuvre le critère de groupement de Ward (1963).

Exemple –

1. Avec la fonction ***agnes***(*, method="ward")

```
library(cluster)                                  # Fichier d'aide : ?agnes
Ours.agnes.Ward.D2 <- agnes(Ours.D1, diss=TRUE, method="ward")
plot(Ours.agnes.Ward.D2)
Ours.agnes.Ward.D2$height                         # Liste des niveaux de fusion
range(Ours.agnes.Ward.D2$height)                  # Plage de variation des niveaux de fusion
```

2. Avec la fonction ***hclust***(*, method="ward.D2")

```
Ours.hclust.Ward.D2 <- hclust(Ours.D1, method="ward.D2")
plot(Ours.hclust.Ward.D2)
Ours.hclust.Ward.D2$height                        # Liste des niveaux de fusion
range(Ours.hclust.Ward.D2$height)                 # Plage de variation des niveaux de fusion
```

3. Avec la fonction ***hclust***(*, method="ward.D")
Note: dans R versions <= 3.0.3, ***hclust***(*, method="ward") produit le même résultat

```
Ours.cl <- hclust(Ours.D1^2, method="ward.D")     # Mise au carré des distances
Ours.cl$height <- sqrt(Ours.cl$height)            # Racine carrée des niveaux de fusion
plot(Ours.cl, hang=-1)                            # Tracer le dendrogramme
Ours.cl$height                                    # Liste des niveaux de fusion
range(Ours.cl$height)                             # Plage de variation des niveaux de fusion
```

Les niveaux de fusion sont-ils les mêmes avec les trois méthodes ? Les plages de variation sont-elles les mêmes ? Pourquoi ces différences ?

=========

```
# Groupement agglomératif des données d'araignées

spiders = read.table( file.choose() )                # Lire le fichier 'Spiders_28x12_spe.txt'
```

1. Calculez la distance de Bray-Curtis à l'aide de ***vegdist()*** (bibliothèque vegan ; voir la section 3.2), puis des groupements agglomératifs à l'aide de ***hclust()*** (method="average") et de ***agnes()*** (method="ward"). Calculez la corrélation cophénétique pour le résultat de ***hclust()***.

2. Calculez la distance de Hellinger : (1) appliquez la transformation de Hellinger au tableau de données (voir la section 4), puis (2) calculez la distance euclidienne sur le tableau transformé (voir la section 3.1). (3) Calculez enfin des groupements agglomératifs à l'aide de ***hclust()*** (method="average") et de ***agnes()*** (method="ward"). Calculez la corrélation cophénétique pour le résultat de ***hclust()***.

3. Cet exercice utilise les données de présence-absence d'espèces. Calculez ensuite la distance de Jaccard sur le tableau de données ; voir la section 3.3. Il n'est pas nécessaire de pré-transformer les données en présences-absences car la fonction ***dist.binary()*** de ade4 s'en charge. Les dissimilarités calculées par ***dist.binary()*** sont en fait des valeurs D = sqrt(1 – Jaccard S). Calculez enfin des groupements agglomératifs à l'aide de ***hclust()*** (method="average") et de ***agnes()*** (method="ward"). Calculez la corrélation cophénétique pour le résultat de ***hclust()***.

Comparer les résultats, de même que les corrélations cophénétiques, obtenus aux #1, 2 et 3. Quelle mesure de dissimilarité produit le dendrogramme qui a la corrélation cophénétique la plus élevée ?

```
# =========
```

13. Corrélogrammes

Des corrélogrammes avec tests de signification des coefficients sont disponibles dans

library(spdep)

Calcul de corrélogrammes pour deux variables du fichier **mite_env.txt** qui se trouve parmi les fichiers de données du cours. Le fichier **mite.env** se trouve égalemennt dans la bibliothèque **vegan**. La variable #1 est *Substrate density* ('SubsDens') et la variable #2 est *Water content* ('WatrCont').

```
# Load the mite data files available in {vegan}
library(vegan)
data(mite.env)                          # Variables environmentales, 70 x 5

subs.density <- mite.env[,1]

# Les coordonnées géographiques des sites se trouvent dans le fichier mite_xy.txt.
# Créer le fichier des sites voisins.

data(mite.xy)                           # Coordonnées géographiques, 70 x 2
nb1 <- dnearneigh(as.matrix(mite.xy), 0, 0.7)

# Calcul du corrélogramme.

mite.correlog <- sp.correlogram(nb1, subs.density, order=14, method="I", zero.policy=TRUE)
print(mite.correlog)
plot(mite.correlog)

# Des tests multiples ont été réalisés. Corriger les probabilités par la méthode de Bonferroni.

print(mite.correlog, "bonferroni")
plot(mite.correlog)

# Le corrélogramme calculé à l'aide de la fonction de corrélation spatiale s'obtient comme suit :

mite.correlog2 <- sp.correlogram(nb1, subs.density, order=14, method="C", zero.policy=TRUE)
print(mite.correlog2)
plot(mite.correlog2)

# Exercice : calculer maintenant le corrélogramme pour la variable 'water.content'.

water.content <- mite.env[,2]
```

14. Variogramme et cartes interpolées par krigeage

```
# 1. Cartographie de base
# Des fonds de carte du monde et des différents pays sont disponibles en R. Exemples:

library(maps)                   # Bibliothèque de base contenant des fonds de cartes
library(mapdata)                # Bibliothèque proposant des cartes spécialisées (par ex. 'rivers')
map('world')      # or map()    # Carte du monde centrée sur l'Europe et l'Afrique
map('world2')                   # Carte du monde centrée sur l'océan Pacifique

# Il y a aussi des cartes par pays, par exemple la Chine

map(region='china')             # Carte de Chine tirée de 'maps', sans les provinces
map('china')                    # Carte de Chine tirée de 'mapdata', avec les provinces

# Produire une carte des capitales du sud-ouest asiatique. Ajouter Taipei pour la perspective
# Les coordonnées des capitales se trouvent dans le fichier "SoutheastAsia_capitals.txt"

capitals = read.table( file.choose() )          # Lire le fichier "SoutheastAsia_capitals.txt"
map(region=
   c('cambodia','laos','myanmar','thailand','vietnam','brunei','indonesia','malaysia','philippines'))
points(capitals[,2], capitals[,1], cex=2, pch=20, col="red")
text(capitals [,2], capitals [,1], rownames(capitals), pos=4)

# Répéter l'exercice en utilisant un fond de carte qui comprend la Chine :
map(region= c('china',
   'cambodia','laos','myanmar','thailand','vietnam','brunei','indonesia','malaysia','philippines'))

=========

#2. Construire in variogramme empirique

library(geoR)

library(vegan)
data(mite)                      # Expèces, 70x35
data(mite.env)                  # Variables environnementales, 70 x 5
data(mite.xy)                   # Coordonnées des sites, 70 x 2

water.content <- as.matrix(mite.env[,2])

vario.water <- variog(coords=mite.xy, data=water.content)
plot(vario.water)

# Ajuster un modèle de variogramme. Sauvegarder ce modèle avant de quitter la fenêtre Eyefit.

x11()      # Mac OS X : ouvrir X11 avant 'eyefit'. Cette opération n'est pas nécessaire en Windows.
model.water <- eyefit(vario.water)      # Bouton «Save» pour sauvegarder le modèle, puis «Quit».

# 3. Krigeage conventionnel pour 'water.content'. Modèle de variogramme pré-sélectionné

# 3.1. Définir la grille d'interpolation. La zone échantillonnée est un rectangle de 2.5m x 10 m.
# Attention: L'espacement des points de la grille doit être au moins aussi petit que la précision des
# coordonnées des points situés sur le pourtour de la zone à cartographier.

pred.grid <- expand.grid(seq(0, 2.5, 0.05), seq(0, 10, 0.05))
```

```
# 3.2. Interpoler par krigeage ordinaire (abréviation pour ordinary kriging : "ok").

kc.water <- krige.conv(coords=mite.xy, data=water.content, loc= pred.grid,
    krige=krige.control(type.krige="ok", obj.model=model.water))

# 3.3. Tracer la carte des estimations interpolée (krigeage) et celle de la variance de ces estimations.

# Créer la fenêtre graphique qui recevra deux cartes disposées en une ligne et deux colonnes
par(mfrow=c(1,2))

# Dans la portion de gauche, tracer la carte interpolée à l'aide de la fonction image.kriging.
image(kc.water, loc = pred.grid, xlab="X", ylab="Y")

# 3.4. Ajouter les données observées (données centrées-réduites) à la carte interpolée.
# Utiliser la fonction s.value de la bibliothèque ade4:
# cette fonction tracera des carrés vides pour les valeurs plus petites que la moyenne et des carrés
pleins pour les valeurs plus grandes que la moyenne.

library(ade4)
s.value(mite.xy, scale(water.content), add.plot=TRUE, csize=0.8, clegend=0)

# 3.5. Portion de droite de la fenêtre graphique : carte de la variance des estimations du krigeage.

# Définir d'abord les axes
x <- seq(0, 2.5, 0.05)
y <- seq(0, 10, 0.05)

# Écrire les données de variance des estimations dans une matrice
krige.water.var <- matrix(kc.water$krige.var, nrow=length(x), ncol=length(y))

# Tracer la carte de la variance des estimations à l'aide de la fonction image de graphics.
image(x=x, y=y, z=krige.water.var, asp=1, col=heat.colors(30))


# Exercice : cartographier l'abondance de l'espèce #15 par krigeage après une transf. racine carrée.

sp15.sqrt <- sqrt(mite.spe[,15])
```

15. Linear discriminant analysis – Daniel Borcard, 31 mars 2009

```
# Load required packages
library(ade4)
library(vegan)

# Preparation of the Doubs data (package ade4). See: http://fr.wikipedia.org/wiki/Doubs_(rivière)
data(doubs)
spe <- doubs$poi          # Species abundances
env <- doubs$mil          # Environmental data
spa <- doubs$xy           # Geographical coordinates

# Remove empty site 8
spe <- spe[-8,]
env <- env[-8,]
spa <- spa[-8,]

# Correct a mistake in env object
env[6,1] <- 268
env[7,1] <- 324

# Remove the 'das' variable from the env dataset
env = env[,-1]

# Transform the species dataset to allow RDA
spe.hel = decostand(spe, "hellinger")

# Create subset of 3 explanatory variables: altitude, oxygen concentration and
# biological oxygen demand

env.pars2 <- as.matrix(env[,c(1,9,10)])

# Create four groups of sites based on the clustering dendrogram of the fish abundances

gr = cutree(hclust(vegdist(spe.hel,"euc")^2,"ward"),4) ### D^2 ###
gr

# Verify multivariate homogeneity of within-group variances
# using Marti Anderson's betadisper() function {vegan}

env.pars2.d1 <- dist(env.pars2)
env.MHV <- betadisper(env.pars2.d1,gr)
anova(env.MHV)            # Parametric test
permutest(env.MHV)        # Permutational test

      # Variances are NOT homogeneous. Let us try a log
      # transformation on variables alt and dbo
env.pars3 <- cbind(log(env$alt), env$oxy, log(env$dbo))
colnames(env.pars3) <- c("alt.ln", "oxy", "dbo.ln")
row.names(env.pars3) <- row.names(env)
env.pars3.d1 <- dist(env.pars3)
env.MHV2 <- betadisper(env.pars3.d1,gr)
permutest(env.MHV2)

      # This time the variances are homogeneous! We can proceed.
```

```
# Test of significance of among-group dispersion:
# compute Wilks' lambda (eq. 11.42) and test its significance (eq. 11.43).

# Computation of LDA (discrimination)

library(MASS)
env.pars3.df <- as.data.frame(env.pars3)
spe.lda <- lda(gr ~ alt.ln + oxy + dbo.ln, data=env.pars3.df)
spe.lda

# The result object contains the information necessary to interpret the LDA
summary(spe.lda)

# Examine the elements in the output object –
# Group means for the 3 variables
spe.lda$means

# Normalized eigenvectors (matrix C, eq. 11.33) = standardized
# discriminant function coefficients
C <- spe.lda$scaling

# Canonical eigenvalues
spe.lda$svd^2
sum.sl <- sum(spe.lda$svd^2)
proportion.of.trace <- spe.lda$svd^2/sum.sl

# Position of the objects in the space of the canonical variates
F <- predict(spe.lda)$x
# alternative way: F <- scale(env.pars3.df, center=TRUE, scale=FALSE) %*% C

# Classification of the objects
spe.class <- predict(spe.lda)$class

# Posterior probabilities of objects belonging to the groups
spe.post <- predict(spe.lda)$posterior

# Table of prior vs predicted
spe.table <- table(gr, spe.class)
```

```
# Percentage of correct classification
diag(prop.table(spe.table, 1))

# Plot of objects in the space of the canonical variates
# with colors according to their classification
plot(F[,1], F[,2], type="n")
text(F[,1], F[,2], row.names(env), col=c(as.numeric(spe.class)+1))
abline(v=0, lty="dotted")
abline(h=0, lty="dotted")
# Draw 95% ellipses around groups
library(ellipse)
for(i in 1:length(levels(as.factor(gr)))) {
  cov <- cov(F[gr==i,])
  centre <- apply(F[gr==i,], 2, mean)
  lines(ellipse(cov,centre=centre, level=0.95))
}

# Classification of a new object (identification)
# A new object is created with ln(alt)=6.8, oxygen=90 and ln(dbo)=3.2
new = c(6.8, 90, 3.2)
new = as.data.frame(t(new))                # 'new' must be a row table
colnames(new) = colnames(env.pars3)
predict.new <- predict(spe.lda, newdata=new)
predict.new

      # In which group has the new object has been classified?
      # Note: group numbers in vector 'gr' may differ among computer platforms and versions of R.
      # This calculation could have been done in the same way for a whole table of new observations.

# LDA with jackknife-based classification (i.e., leave-one-out cross-validation)

spe.lda.jac <- lda(gr ~ alt.ln + oxy + dbo.ln, data=env.pars3.df, CV=TRUE)
summary(spe.lda.jac)

# Number and percentages of correct classification
spe.jac.class <- spe.lda.jac$class
spe.jac.table <- table(gr,spe.jac.class)
spe.jac.table                    # Classification table (see Manuel p. 629)
diag(prop.table(spe.jac.table,1))

    # The classification success in "spe.jac.table" seems not as good as the result in "spe.table".
    Remember, however, that "spe.table" shows an a posteriori classification of the objects that have
    been used in the computations. It is too optimistic. By comparison, cross-validation results are
    obtained by computing the 'lda' and classification of each object, in turn, with that object taken
    out of the 'lda' calculation. It is more realistic.
```

16. Quelques procédures statistiques usuelles en langage R

Tests disponibles dans la librairie « stats »

t.test: test t de Student. Test *t* pour données appariées: paired=TRUE
pairwise.t.test: comparaisons a posteriori entre groupes par le test *t*
var.test: test *F* du rapport de deux variances
bartlett.test: test de Bartlett, homogénéité des variances
friedman.test: test de Friedman (analyse de variance non-paramétrique sans réplication)
kruskal.test: test de Kruskal-Wallis
ks.test : test de Kolmogorov-Smirnov
mcnemar.test: test de McNemar
power.t.test: puissance du test *t*, 1 ou 2 échantillons
shapiro.test: test de normalité de Shapiro-Wilk
wilcox.test: tests de Wilcoxon. Test *U* de Mann-Whitney: paired=FALSE

chisq.test: analyse de tableau de contingence, statistique khi-carré de Pearson.
On fournit soit deux vecteurs, soit un tableau de contingence.
En option, on peut obtenir un test par permutation.

aov: analyse de variance. Le critère de classification doit être déclaré "as.factor"
summary.aov: résultats de 'aov'
manova: analyse de variance multivariable
summary.manova: résultats de 'manova'

Calcul d'intervalles de confiance

t.test: test *t* (Student) et intervalles de confiance, 1 ou 2 groupes
confint: intervalles de confiance des paramètres d'un modèle ajusté aux données
confint.glm (bibliothèque MASS): intervalles de confiance des paramètres d'un modèle ajusté aux données
norm.ci (bibliothèque boot): intervalles de confiance par approximation normale
boot.ci (bibliothèque boot): intervalles de confiance par bootstrap
print.bootci (bibliothèque boot): impression des intervalles de confiance par bootstrap
intervals (bibliothèque nlme): intervalles de confiance de coefficients

Régression multiple

lm (bibliothèque stats) ajuste des modèles linéaires à des données. On peut l'utiliser pour la régression, l'analyse de variance à un seul critère et l'analyse de covariance (la fonction aov fournit une meilleure sortie pour ces dernières analyses). Cette fonction permet également de réaliser la régression multivariable si le tableau-réponse est multivariable.

Analyse discriminante linéaire

library(MASS)
lda : analyse discriminante linéaire
ldahist : histogramme ou diagramme de densité des groupes
pairs.lda : diagrammes de dispersion pour paires de groupes
plot.lda : diagrammes pour objets de la classe "lda"
predict.lda : classification d'observations multivariables par LDA

library(ade4)
discrimin : analyse discriminante linéaire

library(labdsv)
spcdisc : une forme d'analyse discriminante pour des données d'espèces

library(sonarX)
Cette bibliothèque est disponible sur la page http://esapubs.org/archive/ de l'ESA, à l'intérieur de « Ecological Archives A016-047-S1 »
ldaTest : analyse discriminante linéaire

Analyse des corrélations canoniques

CCorA : fonction disponible sur la page http://www.bio.umontreal.ca/legendre/

Distributions statistiques

Exemple : FDist pour la distribution de F –
Recherchez l'information à propos de la distribution de F :

?FDist

Applications :

pf(F,df1,df2,lower.tail=FALSE) pour trouver la probabilité associée à la loi de *F*
qf(prob,df1,df2,lower.tail=FALSE) pour trouver la valeur de *F* correspondant à une prob.
rf(n,df1,df2) pour obtenir *n* nombre aléatoires tirés de la loi de *F* en question
df(F,df1,df2) pour obtenir la densité de la loi pour une valeur de *F* donnée

Recherchez l'information à propos des autres distributions courantes :

?Normal
?TDist
?Chisquare
?Lognormal
?Binomial
?Multinomial
?NegBinomial
?mvrnorm dans la bibliothèque MASS

17. Utilisation de R comme calculette

Addition simple

```
  1+1
[1] 2
```

Racine carrée d'un nombre

```
  sqrt(10)
[1] 3.162278
```

Créer une série de nombres Information : **?seq**

```
  seq(0,5,length=6)
[1] 0 1 2 3 4 5
```

Générer un graphique Information : **?plot**

```
  plot(sin(seq(0,2*pi,length=100)))
```

Créer un vecteur contenant des nombres

```
  x =  c(2,3,5,2,7,1)
  x

[1] 2 3 5 2 7 1
```

Calculer une moyenne

```
  mean(x)
[1] 3.333333
```

Calculer un écart type

```
  sd(x)
[1] 2.250926
```

Calculer l'erreur-type d'une moyenne

```
  sd(x)/sqrt(6)
[1] 0.9189366
```

Mettre des nombres en ordre croissant Information : **?sort**

```
  sort(x)
[1] 1 2 2 3 5 7
# or try
  y=sort(x)
  y
[1] 1 2 2 3 5 7
```

Générateur de nombres aléatoires — Information : **?rnorm**

```
x=rnorm(10,0,1);
mean(x)
sd(x)

x=rnorm(1000,0,1);
mean(x)
sd(x)
```

Générer un vecteur de zéros — Information : **?matrix**

```
Vector=matrix(0,10,1)
Vector
      [,1]
 [1,]    0
 [2,]    0
 [3,]    0
 [4,]    0
 [5,]    0
 [6,]    0
 [7,]    0
 [8,]    0
 [9,]    0
[10,]    0
```

Créer une boucle réalisant une série d'opérations

```
NumberOfSamples = 100
  SampleSize = 30
  VectorOfValues = matrix(0, NumberOfSamples, 1)
  # Calcul de la moyenne d'échantillons de nombres aléatoires N(0,1)
  for (i in 1: NumberOfSamples) {
     x=rnorm(SampleSize, 0, 1)
     VectorOfValues[i,1] = mean(x)
  }
  VectorOfValues
```