R pour la modélisation et le traitement de données - une petite introduction en 8 TP

Christian Jost - module 3M8BMCEM et M2 MSE - 2012/13

30 septembre 2012

Table des matières

Chapitre 1

R - une première introduction

1.1 Prise en main

Le logiciel **R** est déjà installé sur vos machines : vous le trouvez dans le menu `Démarrer - Neuro modélisation`. Après démarrage vous verrez une fenêtre s'ouvrir qui donne quelques informations sur **R** et qui finit avec un `>` : **R** attend que vous lui dites quoi faire par une ligne de commande. Dans ce TP vous aurez un petit aperçu de ce qu'on peut faire avec **R**, le mieux sera de taper tous les exemples sur votre ordinateur et d'essayer de comprendre ce que fait ce logiciel qui nous accompagnera durant tout le module.

1.1.1 Utiliser R pour les calculs de base

Vous pouvez vous servir de **R** comme d'une calculatrice

```
R> 11*13                          # une simple multiplication, # indique un commentaire
R> pi                    # R connaît le chiffre π, notez que R utilise le point décimal
R> sin(pi/2); cos(pi/2); tan(pi/3);log(10); log10(100);
```

le ; vous permet d'enchaîner plusieurs commandes sur la même ligne. Est-ce que vous comprenez les résultats de **R** ? Pourquoi $\cos(\pi/2)$ n'est-il pas 0 ? En fait, il l'est, parce que 6.123234e-17 = 6.123234 $\cdot 10^{-17}$ est TRES petit.

Mais **R** va bien au-delà d'une simple calculatrice. Vous pouvez mémoriser toutes vos valeurs dans des variables, par exemple

```
R> x = 2*3                                  # garde le résultat dans x sans l'afficher
R> x                                                # affiche ensuite ce résultat
R> log(x)/x                                                  # un calcul avec x
R> y <- 13                                  # <- est synonyme à =, 13 est attribué à y
R> x*x^y                              # un calcul plus compliqué, ^ représente l'exposant
R> ls()                                     # vous affiche les noms des variables défini
```

R connait l'ordre d'exécution des opérateurs,

```
R> 5*3+4                                                # n'est pas la même chose que
R> 5*(3+4)
R> x*x^y                                                # n'est pas la même chose que
R> (x*x)^y
```

Enfin, **R** peut faire des comparaison ($==, <, \leq, >, \geq$) en répondant vraie (T=TRUE) ou faux (F=FALSE),

```
R> 3 < 5                                                      # la réponse est TRUE
R> x > y                                  # c'est FALSE d'après les définitions ci-dessus
R> (x+7) == y                                                          # c'est TRUE
```

Ceci sera utile quand on veut changer ou ignorer des données sous certaines conditions.

1.1.2 Les vecteurs - une structure pour gérer les données et pour faire du graphisme

R sert en premier lieu à manipuler et analyser des données, c'est-à-dire des grandes quantités de mesures numériques (longueurs, poids, ...) ou d'observations qualitatives (couleur, génotype, ...). Une

structure de base pour gérer ces données sont les vecteurs qu'on crée en **R** avec la commande `c(mesure1, mesure2, ...)`. Par exemple, on a mesuré chaque semaine le poids d'un rat

```
R> poids = c(13.5, 20.7, 30.5, 38.1, 41.5)     # met les poids (g) dans le vecteur poids
R> poids                                                  # affiche les valeurs saisies
R> length(poids)                                          # compte le nombre de mesures
R> poidskg = poids * 0.001; poidskg                       # convertit les mesures en kg
```

On peut afficher seulement certaines valeurs de poids en indiquant leurs positions,

```
R> poids[3]                                                    # valeur à la position 3
R> poids[c(1,3,5)]          # les valeurs en première, troisième et cinquième position
```

Que faire si vous vous rendez compte que le poids en semaine 2 était de 21.1 au lieu de 20.7 ? Retaper toute la commande ? Il y a trois autres possibilités : a) vous pouvez utiliser la flèche "haut" (↑) pour rappeler les anciennes commandes jusqu'à ce que vous retrouviez la commande `poids = ...` pour corriger la faute, ou b) vous remplacez l'élément faux par `poids[2] = 21.1`, ou c) vous utilisez la commande

```
R> fix(poids)                       # ouvre une fenêtre où vous pouvez corriger la valeur
R> poids                           # vérifier la correction après fermeture de la fenêtre
```

On veut maintenant créer un second vecteur `temps` contenant les semaines correspondant aux poids. La solution simple serait la commande

```
R> temps = c(1,2,3,4,5)
R> temps                                                    # afficher le résultat
```

mais il y a mieux :

```
R> temps = seq(1,5,by=1); temps   # crée le même vecteur, seq dit de faire une séquence
R> temps = seq(1,length(poids), by=1); temps                  # encore plus sophistiqué
R> temps = 1:5 ; temps          # ou le plus rapide, 1:5 est un raccourci pour seq(1,5)
```

Comment est-ce qu'on peut savoir que l'option `by=1` permet d'indiquer à `seq` de faire une séquence avec des pas de 1 en 1 ? Pour toutes les fonctions il y a une aide disponible que vous appelez par

```
R> help(seq)
```

Ces aides sont souvent mal écrit, le mieux est de regarder les exemples à la fin de la page d'aide et de les adapter à ce qu'on veut faire.

R vous permet de faire des graphiques sur mesure. La fonction `plot(x,y)` prend en premier argument un vecteur avec les valeurs de l'abscisse x et en second argument un vecteur y avec les valeurs associées de l'ordonnée. Essayons :

```
R> plot(temps, poids)   # ouvre une fenêtre avec un tracé du poids en fonction du temps
```

Le graphique n'est pas encore très joli, ajoutons un titre et les noms des axes,

```
R> plot(temps, poids, main="La croissance d'un rat",        # taper Entrée avant la fin
+  xlab="Temps (semaines)", ylab="Poids (g)")       # ne pas saisir le +, il indique la
suite
```

Vous pouvez donc saisir une commande sur plusieurs lignes, **R** vous le signale par un + qui ne disparaît qu'après avoir saisi la parenthèse qui ferme la commande.

```
R> plot(temps,
+ poids,main = "La croissance"                                 # continuer la commande
+ )                                                             # terminer la commande
```

On a maintenant pesé un second rat pendant 5 semaines et on voudrait mettre les deux rats sur le même graphique : comment faire ?

```
R> poids2 = c(10.5, 19.5, 33.7,43.3, 46.9)                      # poids du second rat
R> plot(temps, poids)                                             # retrace le graphe
R> lines(temps, poids2)            # trace une ligne qui ajoute les poids du second rat
```

On a un petit problème avec les valeurs de l'ordonnée, elle n'est pas suffisamment large pour afficher tous les poids du second rat. On peut imposer à `plot` des échelles précis avec les arguments optionnels `xlim` (pour l'abscisse) et `ylim` (pour l'ordonnée),

```
R> plot(temps, poids, ylim=c(0,50))     # retrace le graphe avec une ordonnée de 0 à 50
R> lines(temps, poids2)                                 # ajoute les poids du second rat
R> points(temps, poids2, pch=5)                                  # mettre des losanges
```

```
R> text(3,30,"Rat 1"); text(1.5,45,"Rat 2");          # ajouter du text dans le graphe
```
Dans la dernière commande les deux premiers chiffres indiquent les coordonnées x et y où mettre le text. Attention, la commande `text` rajoute votre texte au graphique existant. Si le texte n'est pas bien placé il faut d'abord refaire le graphique avec `plot(...)` avant de redessiner le texte au bon endroit.

1.1.3 Etude d'une fonction

Utilisons ces fonctions graphiques pour étudier la forme d'une fonction que nous retrouverons dans les dynamiques des pistes de fourmis. Imaginons un pont en Y qui relie le nid à deux sources de nourriture. Les fourmis déposent du phéromone en revenant de se ravitailler, il y a donc concentrations C_g et C_d sur les branches gauche et droite respectivement. Les fourmis venant du nid choisissent la branche gauche avec une probabilité qui dépend des deux concentrations, $choix_g(C_g, C_d)$. Des études ont montré que cette probabilité a la forme

$$choix_g(C_g, C_d) = \frac{(k + C_g)^n}{(k + C_g)^n + (k + C_d)^n}$$

avec les paramètres $k = 5$ et $n = 2$. k représente l'attractivité d'une branche non-marquée, c'est-à-dire, plus k est grande, plus la différence de concentration entre gauche et droite doit être grand pour changer significativement la probabilité d'aller d'un côté. Le rôle de n sera étudié plus bas. Nous voulons tracer cette fonction pour $C_d = 1$ en fonction de C_g. Il faut donc créer deux vecteurs, le premier contenant les valeurs de C_g pour lesquelles on veut calculer les valeurs de la fonction, et le second les valeurs $choix_g(C_g, C_d)$ correspondantes.

```
R> cg = seq(0,10,by=0.5); cg                    # créer et afficher les valeurs de cg
R> choixg = rep(0,length(cg))      # initialiser/créer le vecteur choixg, rep = répéter
```
Cette nouvelle commande `rep` crée un vecteur où 0 est répété autant de fois qu'il y a d'éléments dans `cg`, donc 21. Maintenant on peut remplir ce vecteur avec les valeurs de la fonction,
```
R> choixg[1] = (5+cg[1])^2/((5+cg[1])^2 + (5+1)^2)          # probabilité pour Cg = 0
R> choixg[2] = (5+cg[2])^2/((5+cg[2])^2 + (5+1)^2)              # proba pour Cg = 0.5
R> choixg[3] = (5+cg[3])^2/((5+cg[3])^2 + (5+1)^2)              # proba pour Cg = 1.0
```
...
C'est à répéter 21 fois, donc très ennuyeux à faire. Mais vous vous rappelez peut-être que dans vos cours de programmation il y avait des boucles `for`. Elles existent aussi dans **R**.
```
R> for(i in 1:length(cg)) {                      # la boucle laisse i aller de 1 à 21
+ choixg[i] = (5+cg[i])^2/((5+cg[i])^2 + (5+1)^2);              # proba pour Cg = cg[i]
+ }                                                          # terminer la commande for
R> choixg                                                 # afficher les valeurs de choixg
```

Maintenant nous pouvons tracer notre courbe,
```
R> plot(cg, choixg, main="Fonction de choix",
+ xlab="Concentration", ylab="Proba de choisir la gauche")   # donner des noms aux axes
```

En fait, quel rôle joue ce paramètre n ? Pour répondre on pourrait refaire les commandes ci-dessus en remplaçant partout $n = 1$, ensuite $n = 3$, ... C'est beaucoup de travail qu'on peut simplifier en écrivant un petit script.

1.1.4 Les scripts - cela facilite le travail

Un script, c'est tout simplement un fichier texte qui contient des commandes **R** que vous pouvez ensuite exécuter en faisant un copier/coller ou avec la commande `source("nomDuFicher")`. Pour enregistrer ce type de fichier ouvrez d'abord votre « Poste de travail » et créez sur le disque dur `D` un répertoire au nom de l'UE (par exemple `3M7BS15M-TPx`) [1]. Ce répertoire contiendra tous les fichiers des TP de ce module, ce qui facilitera à l'ingénieur qui s'occupe de ces machines le nettoyage de fin d'année. Ensuite,

1. Pour information : quand vous quittez **R** et qu'il vous demande s'il faut enregistrer l'espace de travail, la seule chose qu'il enregistrerait (dans un fichier du type binaire) ce sont les variables et vecteurs que vous avez crée durant votre session, mais pas les commandes elles-mêmes. Ces dernières ne peuvent être sauvegardées qu'on les recopiant dans un script que vous sauvez ensuite dans votre répertoire.

sélectionnez dans **R** le menu `File - New script ...`. Copiez/collez ensuite les commandes précédantes (dans la console) dans la fenêtre qui s'ouvre selon le modèle suivant

```
k = 5;
n = 2;
cg = seq(0,10,by=0.5);
choixg = rep(0,length(cg)); # initialiser le vecteur choixg
for(i in 1:length(cg)) {
  choixg[i] = (k+cg[i])^n/((k+cg[i])^n + (k+1)^n)
}
plot(cg, choixg, main="Fonction de choix",
  xlab="Concentration (gauche)", ylab="Proba de choisir la gauche",
  ylim = c(0,1))   # ylim définit l'étendu de l'ordonnée
```

(vous aurez remarqué que dans la console il y avait des `>` et des `+`, il faut les enlever parce qu'ils ne font pas partie des commandes). Enregistrez ce fichier sous le nom `fonctionChoix.R`[2] dans le répertoire que vous venez de créer. Revenez dans **R** dans le menu `File - source` et sélectionnez le fichier créé. S'il n'y a pas de faute de frappe vous devriez voir le même graphe s'afficher, sinon cherchez les fautes de frappe dans le script, resauvez-le et reéxécutez-le. Remarque : sur les PC, vous pouvez aussi éxécuter votre script en sélectionnant les commandes et de le recopier automatiquement dans la console de **R** via le menu `Edition -> Exécuter la ligne ou sélection` (raccourci : `ctrl-R`). Cette façon de travailler est d'habitude plus rapide que de passer par la commande `source(...)`.

Exercice: Faites varier n entre 1 et 5 (en 0.5), exécutez le script pour chaque valeur de n et décrivez comment cette augmentation de n modifie le tracé (en particulier au point $cg = 1.0$). Remarque : ne cherchez pas à superposer les tracés, on verra plus tard comment cela peut se faire.

Remarque : passer tout le temps par le menu `File - Source` prend beaucoup de temps. On peut désigner à **R** un répertoire par défaut via le menu `File - Change dir ...` qui vous laisse sélectionner le dossier `NeuroModelisation`. Vérifiez que cela a marché et regardez le contenu de ce dossier

```
R> getwd()                                          # répertoire actuellement actif
R> list.files()                          # afficher les fichiers dans ce répertoire
R> source("fonctionChoix.R")                                  # éxécuter le script
```

1.1.5 Les chiffres pseudo-aléatoires et une application

Supposons que les concentrations de phéromones sur notre pont en Y sont $C_g = 5$ et $C_d = 1$ (avec $n = 2$ et $k = 5$), la probabilité de choisir la branche gauche est donc de $choixg(5, 1) = 0.735$. Dans une simulation une fourmi a à faire ce choix, comment le programmer ? Il suffit de créer un chiffre aléatoire r (distribué de façon uniforme entre 0 et 1) : si $r \leq 0.735$ on fait tourner la fourmi à gauche, sinon on la fait tourner à droite.

Créer un chiffre aléatoire est une science en soi. Il faut s'assurer qu'il n'y ait aucun lien entre les chiffres créés et qu'ils soient vraiment distribués uniformément. On appelle ces chiffres « pseudo-aléatoires » (parce qu'ils sont créés de façon déterministe, il n'y a paradoxalement rien d'aléatoire là-dedans) et **R** fournit ce type de chiffre pour plusieurs distributions :

```
R> runif(1)                                  # distribué de façon uniforme entre 0 et 1
R> rnorm(1,mean=0, sd=1)    # distribué selon une loi normale (moyenne 0, écart type 1)
R> runif(10)                               # crée un vecteur de 10 chiffres aléatoires
```

2. L'extension `.R` désigne que c'est un script **R** et elle permet à d'autres ordinateurs (Linux ou Mac) de les identifier correctement. Il est recommandé de l'ajouter à tous vos scripts **R**.

Pour implémenter le choix de la fourmi il nous faut une fonction du type « si (r $<$ 0.735) alors (...) sinon (...) ». C'est la fonction `if()`. Regardez d'abord l'aide sur cette fonction

```
R> help("if")
```

Cette aide regroupe toutes les commandes pour l'algorithmique (les choix, les boucles, les interrupteurs), vous y trouvez en particulier la syntaxe pour `if` : if (condition) { commande1 ; commande2 ; ...} else { commande1 ; ...}.

On va s'entrainer avec un autre exemple : la première loi de Mendel prédit que si on croise les F1 de parents homozygotes (`AA` $\times$ `aa`) entre eux les phénotypes de la F2 apparaissent dans la proportion 1 à 3 (phénotype récessif vs. phénotype dominant). Dans 100 individus d'une F2 on a compté 81 phénotypes dominants et 19 phénotypes récessifs et on se demande si ce résultat est compatible avec le ratio prédit. Pour répondre à cette question nous allons écrire un script qui fait cette expérience « in silico ». Ouvrez un nouveau script dans l'éditeur de scripts

```
phenoA = 0; # initialisation de nos compteurs,
phenoa = 0; # R distingue majuscule/minuscule, c'est donc un second
compteur
for (i in 1:100) { # parcourir nos 100 individus
  if (runif(1) > 0.5) { # premier parent donne allele a
    if (runif(1) > 0.5) { # deuxieme parent donne aussi a
      phenoa = phenoa + 1; # genotype aa, on incrémente phenoa
    } else {
      phenoA = phenoA + 1; # genotype aA, on incrémente phenoA
    }
  } else {  # premier parent donne l'allèle A
      phenoA = phenoA + 1; # genotype Aa ou AA, incrémenter phenoA
  }
}
print(c(phenoA, phenoa));
```

Sauvez-le sous le nom "Mendel.R" dans notre répertoire et éxécutez-le. Quel est le résultat ? Répétez la simulation plusieurs fois, est-ce que vous retrouvez les valeurs de la vraie expérience ?

On va regarder la distribution de `phenoA` en répétant la simulation 1000 fois avec le script "Mendel-Simu.R" :

```
tousLesPhenoA = rep(0,1000); #initialiser le vecteur des résultats
for (k in 1:1000) {
    phenoA = 0;
    phenoa = 0;
    for (i in 1:100) {
      if (runif(1) > 0.5) {
        if (runif(1) > 0.5) {
           phenoa = phenoa + 1;
        } else {
            phenoA = phenoA + 1;
        }
      } else {
           phenoA = phenoA + 1;
      }
    }
    tousLesPhenoA[k] = phenoA;
}
```

Visualisez maintenant le résultat via la ligne de commande

```
R> hist(tousLesPhenoA)                          # histogramme du nombre des phénotypes A
R> hist(tousLesPhenoA,freq=F)                           # avec des fréquences relatives
```

Est-ce que la valeur expérimentale de 81 est compatible avec cet histogramme ? Intuitivement il devrait suffire de voir la proportion de valeurs simulées qui sont plus grandes ou égales à 81,

```
R> plusGrand81 = 0                              # initialiser le compteur de ces valeurs
R> for (k in 1:1000) {                 # parcourir toutes les valeurs de tousLesPhenoA
```

```
+  if (tousLesPhenoA[k] >= 81) { plusGrand81 = plusGrand81 + 1}                    #
+ }                                                                  # fin de la boucle
R> plusGrand81/1000                          # calcul de la proportion de valeurs >= 81
```
Avec-vous compris les commandes ? Comment interprétez-vous ce résultat ?

1.1.6 Au-delà des vecteurs : les matrices

Parfois on aimerait garder nos valeurs dans une structure plus compliquée que des vecteurs, par exemple regrouper les poids de nos deux rats dans un tableau qui contient en première colonne les semaines, en deuxième colonne les poids du premier rat et en troisième colonne ceux du deuxième rat. C'est ce qu'on appelle une matrice qu'on peux créer, par exemple, en "collant" nos trois vecteurs précédents ensemble,

```
R> nosRats = cbind(temps,poids,poids2); nosRats
R> is.matrix(nosRats)                 # est-ce que le résultat est une matrice pour R?
```

Pour utiliser les valeurs dans une matrice il faut indiquer dans quelle ligne et quelle colonne elles se trouvent (l'ordre est important !)

```
R> nosRats[2,3]                  # le poids du rat 2 (ligne 2) en semaine 2 (colonne 3)
R> nosRats[3,]                                    # les poids en semaine 3 (colonne)
R> nosRats[3,2:3]                                    # la même chose sans la semaine
R> nosRats[,2]                                             # tous les poids du rat 1
```

On peut donc retracer notre graphe en se servant de cette matrice

```
R> plot(nosRats[,1],nosRats[,3], main="Nos Rats",type='l')                # poids rat 2
R> lines(nosRats[,1],nosRats[,2]); points(nosRats[,1],nosRats[,2])
```

(attention, le `'l'` dans la commande précédante n'est pas le chiffre 1 mais la lettre l comme *line*).

Une autre commande pour créer une matrice (par exemple pour l'initialiser) est

```
R> matrix(0,3,5)                          # crée une matrice de 0 à 3 lignes, 5 colonnes
R> matrix(1:15,3,5)                                 # rempli des chiffres 1, 2, ...15
R> matrix(1:15,3,5,byrow=T)                                          # rempli par ligne
```

On se servira fréquemment des matrices.

1.2 Entraînement

Exercice: attribuez 25 à v et 13 à w, calculez leur moyenne et attribuez-la à u. Ecrivez commandes et résultats dans l'encadré.

Exercice: Calculer avec **R** les expressions : a) $\frac{2^7}{2^7-1}$, b) $\sin^2(\pi/7)$ (attention, `sin^2(pi/7)` ne marche pas), c) $\frac{2^7}{2^7-1} + 3\sin^2(\pi/7)$ (en utilisant copier/coller).

Exercice: Créer un vecteur x contenant les valeurs $(0, \pi/10, 2\pi/10, \ldots, 2\pi)$, et tracer ensuite un graphique contenant x en abscisse et $sin(x)$ en ordonnée. Donnez un nom au graphique et aux axes. Tracez ensuite un graphique contenant $cos(x)$ en abscisse et $sin(x)$ en ordonnée. Qu'est-ce que vous observez ?

Exercice: Explorer à l'aide d'un script la fonction $g(N) = \frac{aN}{1+ahN}$ (prenez au début $a = 1$, $h = 0.1$ et variez $N \in (0, 5)$) : quelle est sa forme (enregistrer le graphique dans un fichier png) ? Qu'est-ce que représentent a et h géométriquement ?

Exercice: Créer la matrice $B = \begin{bmatrix} 0 & 0 & 3.5 & 3.1 \\ 0.7 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.8 & 0.3 \end{bmatrix}$ et additionner la 2ème avec la 4ème colonne.

1.3 Informations supplémentaires

1.3.1 Enregistrer et utiliser les graphiques de R

Pour utiliser un graphique de **R** dans vos rapports vous pouvez tout simplement faire un `copier` dans la fenêtre graphique de **R** et un `coller` dans le traitement de texte de votre choix (par exemple Open Office ou, si vous insistez sur l'utilisation de produits Micro$oft, Word).

Vous pouvez aussi sélectionner la fenêtre du graphique et passer par le menu `Fichier -> Sauver sous -> jpeg` ou `png` pour l'enregistrer dans le format `jpeg` ou `png` (le dernier format est très utile pour des graphismes statistiques parce qu'il comprime toute l'information sans perte).

Enfin, vous pouvez enregistrer les gaphiques via des commandes spécifiques sur la ligne de commande. Par exemple, pour l'enregistrer dans un fichier `"monFichier.jpg"` vous devez avec une première commande ouvrir le fichier, ensuite effectuer toutes le commandes pour créer le graphique, et ensuite fermer le fichier. Concrètement cela donne

```
R> jpeg(filename="monFichier.jpg")                          # ouvrir le (nouveau) fichier
R> plot(nosRats[,1],nosRats[,3], main="Nos Rats",type='l',
+ xlab="Temps", ylab="Poids (g)")                           # graphique des premiers rats
R> lines(nosRats[,1],nosRats[,2]); points(nosRats[,1],nosRats[,2])      # seconds rats
R> dev.off()                                               # fermer le fichier graphique
```

(en fait, `dev` vient du mot anlgais "device" qui désigne dans ce contexte le destinataire des commandes graphiques : une fenêtre sur l'écran, un fichier, l'imprimante, ...). Pour faire un `png` c'est la commande `png(filename=...)`. Et quoi faire si vous ne savez pas ou **R** a enregistré ce fichier ? La commande `getwd()` (= get working directory) vous le dira.

Finalement, un autre format moins connu par l'utilisateur des PC mais qui représente le language universel des imprimantes et de linux et le postscript. Pour enregistrer votre graphiqu dans un fichier postscript la commande la plus simple est

```
R> dev.copy2eps(filename="monFichier.eps")                                    #
```
qui envoye le graphique qui est actuellement dans la fenêtre graphique dans ce fichier. Pour imprimer directement une fenêtre graphique vous pouvez utiliser la commande
```
R> dev.print()
```

1.3.2 Quelques mots sur R

R (R Development Core Team, 2010) est un graticiel du type « open source », c'est-à-dire il est entièrement gratuit et tout le monde peut télécharger les sources du code à partir duquel il a été créé. Il y a une communauté dans le monde entier qui l'utilise et qui continue à développer de nouvelles fonctionnalités (gratuitement). Pour télécharger une version pour votre ordinateur, rendez-vous sur
`http://www.r-project.org`

R est en fait basé sur un language nommé S qui a été définit par J. Chambers dans les laboratoires de Bell. Ce language a d'abord été implémenté dans le logiciel commercial S-plus qui fonctionne avec exactement les mêmes commandes que **R** mais offre en plus une interface graphique (par exemple un tableur pour saisir les données). Donc, tous ce que vous apprenez pour R sera utilisable dans S-plus. Il faut aussi ajouter que la façon de structurer les données dans **R** est un standard international pour tous les logiciels statistique, c'est-à-dire la philosophie que vous acquérez en utilisant **R** vous servira aussi pour comprendre comment faire des analyses (et l'interprétation) dans un logiciel comme Systat ou SPSS.

Pour plus d'informations sur les idées du « open source », voir
`http://www.gnu.org/`

Chapitre 2

Les modèles de croissance discrètes

Thème de ce TP : exploration des dynamiques temporelles d'un modèle de croissance discrète et construction d'un diagramme de bifurcation.

2.1 Programmer dans R et graphismes

R n'est pas seulement un logiciel pour faire des analyses statistiques avec une interface graphique (`Rcmdr`), c'est également un logiciel de programmation pour faire des expériences virtuelles et la modélisation. Vous en avez déjà eu un premier aperçu durant le TP sur la méthode du bootstrap. Rappelons vite ce que vous y aviez appris.

- Vous pouvez passer vos commandes directement à **R** par la ligne de commande dans la console, par exemple pour apprendre plus sur **R** vous y tapez
  ```
  R> citation()     # tout après le signe # est un commentaire, non-interprété par R
  ```
  et la réponse vous dit comment le citer dans un rapport où vous avez utilisé **R**. Une commande est toujours suivi d'une parenthèse, même si rien n'est écrit entre les parenthèses.
- Un programme s'écrit d'habitude dans un script **R** (`Fichier - nouveau script`) dans lequel vous écrivez des commandes que vous exécutez ensuite en les copiant-collant ou (dans Windoze) en sélectionnant la commande et en appuyant simultanément sur `Ctrl` et `R`. Ce script peut être enregistré comme n'importe quel fichier du type `texte`, en lui donnant un nom qui se termine par `.R` (par exemple `monSecondScript.R`).
- Dans ce script vous pouvez écrire des commandes, par exemple créer des vecteurs qui contiennent un certain nombre de chiffres
  ```
  v1 = c(1,3,4,7)     # "connecter" (c) les chiffres 1, 3, 4 et 7 en un seul vecteur
  v2 = rep(2,4)       # vecteur qui contient le chiffre 2 quatre fois (rep = repeat)
  v2                                                    # affichier le résultat
  ```
  (écrire les commandes dans le script et ensuite les copier-coller).
- Avec ces vecteurs vous pouvez faire des calculs vectoriels :
  ```
  v1 + v2                           # addition des deux vecteurs élément par élément
  v1 * v2                     # multiplication des deux vecteurs élément par élément
  3 * v2                      # multiplication du vecteur élément par élément avec 3
  v2[3] = 1                     # changer l'élément dans la troisième position de v2
  v2
  ```
  (écrire les commandes dans le script et ensuite les copier-coller dans **R** - comprenez-vous ce que vous voyez apparaître ?).
- Parcourir les éléments d'un vecteur avec une boucle `for` et changer leurs valeurs
  ```
  for (pos in 1:4) {
    v2[pos] = pos*v1[pos]
  }
  v2                                                    # affichier le résultat
  ```

Passons maintenant à une nouvelle chose, les graphiques dans **R**. On fera les commandes directement dans la console pour explorer. La commande `plot` prend en premier argument un vecteur avec les valeurs

de l'abscisse $(x_1, x_2, \ldots)$ et en second argument un vecteur avec les valeurs de l'ordonnée $(y_1, y_2, \ldots)$ correspondant pour tracer un graphiques avec des points à ces positions (x_i, y_i)

```
R> plot(v1, v2)                                          # tracer v2 en fonction de v1
R> plot(v1, v2, type="l")   # tracer les points joints par des lignes droites, l est la
lettre l
R> plot(v1, v2, type="b")                                  # tracer lignes et points
R> plot(v1,v2, type="b",xlab="Axe x",ylab="Axe y",main="Mon graphique")     # enjoliver
```

Si vous voyez dans la console un `+` cela veut dire que **R** attend que vous terminez la commande en cours, par exemple

```
R> plot(v1, v2                              # il manque encore la fin de la parenthèse
+ )                                           # le + s'affiche, tapez simplement )
```

Si à un moment vous n'arrivez pas à sortir du `+` appuyez sur la touche `Ech`.

Il y a des commandes pour ajouter d'autres éléments sur un graphique déjà existant et ouvert

```
plot(v1, v2, type="b", ylim=c(0,50))              # forcer l'ordonnée de couvrir 0 à 50
lines(v1, 1.5*v2, col="blue")            # ajouter une courbe au-dessus de la prèmière
points(c(2,2.5,3),c(38,35,38))                                  # tracer des points
lines(c(2.1,2.4,2.6,2.9),c(32,30,30,32))                    # ajouter d'autres lignes
text(2.5,25,"smile")                                           # et ajouter du texte
```

(écrire dans le script et copier-coller ligne par ligne pour voir et comprendre ce qui se passe).

2.2 Croissance Logistique discrète : dynamiques et diagramme de bifurcation

Un modèle classique pour décrire la dynamique d'une population dont la reproduction est parfaitement synchronisée et qui prend en compte les effets limitant de la densité dépendance est le modèle logistique en forme discrète (aussi appelé le modèle de Ricker),

$$x_{t+1} = x_t \exp\left(r\left(1 - \frac{x_t}{K}\right)\right), \tag{2.1}$$

avec taux d'accroissement r et capacité de charge K. Veuillez noter que j'ai choisi ce modèle provenant du contexte écologique parce qu'il est le plus simple pour bien comprendre les phénomènes de bifurcation et du chaos déterministe, mais les principes que vous allez découvrir s'appliquent aussi bien aux phénomènes collectifs des insectes sociaux, à la régulation cardiaque ou autres phénomènes plus généraux.

1. Ouvrez le fichier `LogDiscret.R`, analysez ce qu'il fait et modifiez-le ensuite pour calculer et afficher (en fonction du temps) la dynamique de ce modèle avec condition initiale $x_0 = 50$, $r = 1.8$, $K = 500$ et pour un temps $t = 50$. Quelle dynamique observez-vous ? Déterminez en particulier la dynamique transitoire et la dynamique stationnaire.

2. Faites varier r de 1 à 3 (pas 0.1) : quelles type de dynamiques (transitoires et stationnaires) observez-vous en faisant ces variations ? Déterminez plus précisément les valeurs critiques de r où le type de dynamique change.

3. **Sensibilité aux conditions initiales** : en modifiant le fichier `LogDiscret.R` simulez les prédictions de la population au temps $t = 30$ avec $r = 2.5$ et $r = 2.9$, en faisant varier la condition initiale x_0 de 45 à 50 (pas 1) (K toujours fixé à 500). Notez ces valeurs dans le tableau ci-dessous.

x_0	x_{30} pour $r = 2.5$	x_{30} pour $r = 2.9$
45		
46		
47		
48		
49		
50		

Qu'est-ce que vous observez ? Comment interprétez-vous ces observations ? Comment interprétez-vous là-dedans le fameux « effet de papillon » ?

Petit exo optionnel pour ceux qui commencent à bien maîtriser R : Visualisez toutes les courbes pour un même r dans un même graphique et interprétez par rapport à la possibilité de faire des prédictions à court terme (quelques pas en avant) ou à long terme.

4. **Diagramme de bifurcation** : construisez le diagramme de bifurcation en fonction du paramètre r (c'est-à-dire les dynamiques en régime stationnaire en fonction de r) en complétant le fichier `BifurcationIncomplet.R`. Pour faire cela, faites varier r de 0 à 3.5 (pas à déterminer, il définira la résolution de votre diagramme, commencez avec un pas large de ≈ 0.5), itérez l'équation (2.1) suffisamment longtemps pour atteindre l'état stationnaire ($T = 300$ devrait être suffisant, vous pouvez vérifier pour quelques r à l'aide de votre script `LogDiscret.R`), et affichez avec la commande `points(...)` suffisamment de points pour représenter la dynamique stationnaire pour chaque r. Comment interprétez-vous ce diagramme de bifurcation ?

Rendez-moi un **petit** rapport (en binôme, max. 2 pages, format pdf) avec vos réponses et ajoutez-y le code de votre script avec lequel vous avez fait le diagramme de bifurcation. Faites particulièrement attention que le graphique soit « joli ». N'oubliez pas de mettre vos noms aussi bien sur le rapport comme dans le fichier script (en commentaire). Le tout est à envoyer par email.

2.3 Entrainement

1. Tracez une courbe de recrutement exponentielle avec le modèle discret $N_{t+1} = RN_t$ ($R = 1.2$). Superposez la-dessus la dynamique du modèle Malthusienne continue

$$\frac{dN}{dt} = rN, N(0) = N_0 \Leftrightarrow N(t) = N_0 \exp(rt).$$

Quel r devez-vous prendre pour avoir une superposition parfaite ? Expliquez pourquoi.

2. Reprenons le modèle (2.1) et tracez l'effectif moyen (en régime stationnaire, calculé sur 100 valeurs) en fonction de r (de 1 à 3.5). Qu'est-ce que vous observez (comparer aux informations ci-dessous) ? Imaginez que la population est un poisson exploité en danger d'extinction, est-ce que ce graphique et le diagramme de bifurcation vous donnent des idées sur son risque d'extinction ?
3. Testez numériquement si le paramètre K est aussi un paramètre de bifurcation.

2.4 Informations supplémentaires

Voici un commentaire de May (1975) sur la signification du phénomène étudié : « For population biology in general, and for the temperate-zone insects in particular, the implication is that even if the natural world was 100% predictable, the dynamics of populations with "density dependent" regulation

could nonetheless in some circumstances be indistinguishable from chaos, if the intrinsic growth rate r is large enough. »

Pour ceux avec quelques affinités mathématiques : on peut prouver que la moyenne va rester proche de K même en régime chaotique : on peut démontrer que

$$N_{t+j} = N_t \exp\left(r\left(j - \sum_{i=0}^{j-1} \frac{N_{t+i}}{K}\right)\right)$$

(vous pouvez essayer vous-même, ce n'est pas trop compliqué) ce qui se transforme en

$$\begin{aligned}
\log\left(\frac{N_{t+j}}{N_t}\right) &= rj - \frac{r}{K}\sum_{i=0}^{j-1} N_{t+i} \\
\iff \quad \sum_{i=0}^{j-1} N_{t+i} &= Kj(1 - \frac{1}{rj}\log(N_{t+j}/N_t)) \\
\implies \quad \bar{N} \approx (\sum_{i=0}^{j-1} N_{t+i})/j &= K(1 - \frac{1}{rj}\log(N_{t+j}/N_t)) \approx K \text{ pour grand } j
\end{aligned}$$

Chapitre 3

Les modèles à compartiments et leur simulation

3.1 Du calcul matriciel à une simulation des dynamiques des fourmis entre nid et source de nourriture

Ce TP sera divisé en deux parties. La première partie est un petit entraînement au calcul matriciel dans **R**, et dans la seconde on s'en servira pour simuler et visualiser les dynamiques (en forme d'équations différentielles linéaires) des fourmis entre le nid et une source de nourriture.

3.1.1 Le calcul matriciel dans R

Nous avions brièvement vu les matrices à la fin du premier TP (1.1.6). Reprenons les choses à la base. Une matrice est un tableau à n rangs et m colonnes, par exemple (je me sers des exemples de l'annexe A.2.3 du polycopié) la matrice $n \times m = 2 \times 3$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

Il y a plusieurs façons pour la saisir. Une première est de créer d'abord une matrice 2×3 rempli de 0

```
R> A = matrix(0,nrow=2,ncol=3)        # nrow = "number of rows", ncol pour les colonnes
```

et de la remplir ensuite élément par élément

```
R> A[1,1]=1;
R> A[1,2]=2;
R> A[1,3]=3;
R> A[2,1]=3;
R> A[2,2]=2;
R> A[2,3]=1;
R> A                                                    # afficher la matrice A
```

Une seconde façon est de donner les valeurs directement à la commande `matrix()` en forme d'un vecteur et en indiquant qu'il faut remplir la matrice ligne par ligne,

```
R> B = matrix(c(2,0,-1,-1,-2,-3),nrow=2,ncol=3,byrow=T)
R> B
```

B est aussi une matrice 2×3,

```
R> dim(B)
```

on peut donc faire une addition (qui se fait élément par élément entre deux matrices)

```
R> A+B
```

Exercice: Notez le résultat et comparez au polycopié p. 59. Avez-vous compris compris comment faire à la main ?

Utilisez maintenant la fonction `data.entry(A)` pour changer l'entrée `A[2,3]` en -1.

Il y a plusieurs opérations possibles entre une matrice et un scalaire (un chiffre dans $\mathbb{R}$),

```
R> 5 * A                                    # multiplier chaque élément de A par 5
R> 5 + A                                        # ajouter 5 à chaque élément de A
R> A / 5                                           # diviser chaque élément par 5
R> A^3                            # qu'est-ce que cela fait? Indiquez-le dans le cadre
```

Le polycopié explique qu'une matrice $A \in \mathbb{R}^{m\times n}$ représente une fonction linéaire f du $\mathbb{R}^n \to \mathbb{R}^m$. Au niveau matriciel cela correspond à la muliplication de la matrice avec un vecteur $v \in \mathbb{R}^m$, par exemple

```
R> vec = matrix(c(3,3,1),nrow=3,ncol=1); vec                    # definition du vecteur
R> A %*% vec       # le signe %*% dit à R qu'il s'agit d'une multiplication matricielle
```

Cela donne un résultat $A\,vec \in \mathbb{R}^2$ et vous pouvez vérifier

```
R> A[1,1]*vec[1]+A[1,2]*vec[2]+A[1,3]*vec[3]                              # premier élément
```

Exercice: Vérifiez aussi le second élément par ce calcul « à la main ». Notez la commande.

La multiplication existe aussi entre deux matrices (elle correspond à la composition de deux fonctions linéaires). Par exemple, définissons une matrice 3×3

```
R> C = matrix(c(6,5,0,8,-1,4,1,0,2),nrow=3,ncol=3,byrow=T)
```

La multiplication AC correspond à la fonction composé

$$\begin{array}{rcll} \mathbb{R}^3 & \longrightarrow & \mathbb{R}^3 & \longrightarrow \mathbb{R}^2 \\ \boldsymbol{x} & \xrightarrow{g} & g(\boldsymbol{x}) & \xrightarrow{f} (f\circ g)(\boldsymbol{x}) \\ \boldsymbol{x} & \xrightarrow{C} & C\boldsymbol{x} & \xrightarrow{A} AC\boldsymbol{x}. \end{array}$$

On a donc $AC \in \mathbb{R}^{2\times 3}$. Vérifiez dans **R** (notez commandes et résultat)

L'élément « identité » correspond à une matrice carré avec des 1 sur la diagonale et des 0 ailleurs

```
R> id3 = diag(1,3,3)                  # consultez help(diag) pour comprendre les arguments
```

et commentez les résultats de

```
R> C %*% id3; id3 %*% C; C
```

L'inverse d'une matrice, valeurs et vecteurs propres

Pour les matrices carrées $A \in \mathbb{R}^{n\times n}$ il existe souvent une matrice inverse A^{-1} tel que $AA^{-1} = A^{-1}A = I_n$. La condition qu'elle existe est que le déterminant de la matrice ne soit pas 0. Essayons avec notre matrice C

```
R> det(C)                                                    # calculer le déterminant
R> solve(C)                                                  # calculer l'inverse de C
```

Exercice: Attribuez l'inverse de C à la variable `Cinv` et vérifiez la relation ci-dessus. Utilisez la commande `zapsmall` pour enlever les broutilles numériques. Notez commandes et résultat

Enfin, dans le cours je vous ai parlé de valeurs et vecteurs propres. Pour une matrice carrée $A \in \mathbb{R}^{n\times n}$ il existe un ensemble de valeurs propres λ_i et de vecteurs propres associés v_i tel que

$$Av_i - \lambda_i v_i = 0 \qquad \forall i \tag{3.1}$$

La commande pour les calculer est `eigen` qui vous donnera une réponse avec deux éléments, `$values` contenant un vecteur avec les valeurs propres et `$vectors` contenant une matrice dont les colonnes sont les vecteurs propres associées. On appelle un tel objet une 'liste'. Essayons avec notre matrice C

```
R> eigen(C)                                          # calcule vecteurs et valeurs propres
R> eigen(C) -> eC                                                  # attribue la liste à eC
R> names(eC)                    # nous dit comment les éléments de la liste s'appellent
R> eC$values                                            # affiche les valeurs propres
R> eC$vectors                                 # affiche la matrice des vecteurs propres
R> eC$vectors[,1]                                   # affiche le premier vecteur propre
```

Exercice: Vérifiez la relation (3.1) en prenant une valeur propre avec son vecteur propre associé après l'autre. Notez commandes et résultats.

3.1.2 La solution d'une équation différientielle linéaire

Rappelons d'abord l'EDO linéaire que nous avons obtenus pour le système des fourmis entre nid, recherche et abbreuvoir

$$\frac{d}{dt}\begin{pmatrix} N(t) \\ R(t) \\ M(t) \end{pmatrix} = \underbrace{\begin{pmatrix} -r_N & r_A & r_M \\ r_N & -(r_C + r_A) & 0 \\ 0 & r_C & -r_M \end{pmatrix}}_{\text{matrice } A} \begin{pmatrix} N(t) \\ R(t) \\ M(t) \end{pmatrix}, \quad \begin{pmatrix} N_0 \\ R_0 \\ M_0 \end{pmatrix} = \begin{pmatrix} 100 \\ 0 \\ 0 \end{pmatrix}$$

ou plus simplement $dx/dt = Ax$ et $x(0) = x_0$ avec $x(t) = \begin{pmatrix} N(t) \\ R(t) \\ M(t) \end{pmatrix}$.

Dans le cours on avait vu que la solution est

$$x(t) = Pe^{Dt}P^{-1}x_0$$

où D est la matrice contenant les valeurs propres sur sa diagonale et P est la matrice dont les colonnes sont les vecteurs propres. Mettons-nous à programmer un petit script qui calcule la solution et la visualise dans un graphe. Testez vos commandes sur la ligne de commande avant de l'ajouter au script.

Première étape : on défini les valeurs de paramètres

```
rn = 0.2;
rc = 1.0;
ra = 1.0/30.0;
rm = 0.5;
Tot = 100;
```

Deuxième étape : on construit la matrice A et on calcule valeurs et vecteurs propres, mettant les premiers dans le vecteur `eval3` et les derniers dans la matrice `P3` (je vous laisse le faire vous-même en vous servant des commandes du chapitre précédant). Calculez aussi l'inverse de `P3` en la mettant dans `P3inv`. Ensuite vérifiez qu'il s'agit bien de la matrice de passage

```
R> zapsmall(P3inv %*% A %*% P3)
R> zapsmall(eval3)
```

Commentez les deux derniers résultats

Troisième étape : préparation des vecteurs pour y stocker les valeurs simulées

```
tempsSimu = 10; # on simulera les dynamiques de 0 à 10 minutes
tempsPas = seq(0,tempsSimu,0.1); # calcul de la solution entre 0 et 10 s par 0.1 s
nombreDePas = length(tempsPas);
rech = rep(0,nombreDePas); # vecteur pour le nombre à la recherche
mang = rep(0,nombreDePas); # vecteur pour le nombre en mangeant
nid = rep(0,nombreDePas);  # vecteur pour le nombre dans le nid
x0 = c(100,0,0); # condition initiale
```

Quatrième étape : une boucle `for` pour calculer la solution à chaque temps $t \in$ `tempsPas`.

```
for (i in 1:nombreDePas) {
  expDt = diag(exp(eval3*tempsPas[i]),3,3); # forme de expDt?
  x = ... # a vous de remplir
  nid[i] = x[1];
  rech[i] = x[2];
  mang[i] = x[3];
}
```

Et cinquième étape : on fait le graphe de la solution

```
plot(tempsPas,nid,main="Dynamiques des fourmis",
     xlab="temps",ylab="Densites",ylim=c(0,Tot),type='l');
lines(tempsPas,rech,lty='dashed');
lines(tempsPas,mang,lty='dotdash');
```

Vous pouvez ajouter une légende.

3.2 Entrainement

Le système qu'on vient de résoudre était homogène. On peut le transformer dans un système à deux équations mais non-homogène (voir le polycopié pour les détails) en remplaçant N par $N_{tot} - R - M$

$$\frac{d}{dt}\begin{pmatrix} R \\ M \end{pmatrix} = \underbrace{\begin{pmatrix} r_N N_{tot} \\ 0 \end{pmatrix}}_{\boldsymbol{b}} + \underbrace{\begin{pmatrix} -(r_A + r_C + r_N) & -r_N \\ r_C & -r_M \end{pmatrix}}_{B}\begin{pmatrix} R \\ M \end{pmatrix}, \quad \begin{pmatrix} R(0) \\ M(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

La solution de ce système est

$$\begin{pmatrix} R(t) \\ M(t) \end{pmatrix} = x^\star - Pe^{Dt}P^{-1}x^\star.$$

($x^\star$ est un équilibre, c'est-à-dire une solution du système $\boldsymbol{b} + Bx^\star = 0 \Leftrightarrow x^\star = -B^{-1}\boldsymbol{b}$.

Théoriquement les dynamiques de ce système devraient être les mêmes comme celles développées ci-dessus. Pour confirmer cela écrivez un script qui fait tous ces calculs et faites un graphique des dynamiques pour les comparer aux dynamiques précédantes. Remarque : vous pouvez recyclez beaucoup de code du dernier chapitre.

3.3 Informations supplémentaires

Il est possible que la matrice $D = P^{-1}AP$ ne soit pas une matrice diagonale mais qu'elle contient certains 1 juste au-dessus de la diagonale. Dans ce cas la formule de la solution reste la même mais il faut calculer $\exp(Dt)$ d'une autre façon, en s'inspirant du développement de Taylor de la fonction exponentielle :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

où on remplace simplement le x par une matrice $A \in \mathbb{R}^{n\times n}$

$$e^A = I_n + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots$$

La bibliothèque `rmutil` (téléchargeable sur le site de **R**) fournit une fonction `mexp(A)` qui calcule cette exponentielle d'une matrice.

Pour savoir quand vous en aurez besoin vérifiez toujours si $P^{-1}AP$ donne bien une matrice diagonale ou non.

Chapitre 4

Les EDO non-linéaires et leur analyse

On va travailler aujourd'hui avec des équations différentielles et comment analyser leurs dynamiques à l'aide d'un ordinateur et d'un outil de programmation (**R**). Pour échauffement on va faire d'abord une analyse graphique du modèle logistique,

$$\frac{dN}{dt} = r\left(1 - \frac{N}{K}\right) N =: f(N)$$

avec les paramètres $r = 0.5\ \text{min}^{-1}$ (taux de croissance) et $K = 110$ individus (capacité de charge ; prenons par exemple pour $N(t)$ la taille d'un agrégat de fourmis au cours du temps, dans ce cas r est le taux de croissance de l'agrégat et K la taille maximale de l'agrégat). Faites d'abord un « joli » graphique de $f(N)$ en fonction de N en faisant varier N de 0 à K (voir le dernier TP pour le rappel des commandes graphiques). Notez commandes et résultat graphique

Ajoutez à ce graphique la ligne de l'abscisse (avec la commande `lines`). Les équilibres du système sont par définition toutes les valeurs de N tel que $dN/dt = f(N) = 0$. Quelles sont les équilibres du système ? Reporter à la main sur le graphique précédant.

Quel est le signe de la variation de l'effectif (dN/dt) pour des valeurs de N comprises entre 0 et K ? Et pour des valeurs $n > K$? Pour quelles valeurs de N l'effectif augmente/diminue-t-il ? Qu'en concluez-vous sur la stabilité (attractivité) des équilibres identifiés ci-dessus ?

4.1 Simulation numérique d'un système dynamique continu

Nous avons vu dans le cours que pour un système dynamique non-linéaire il est d'habitude impossible de trouver une solution analytique, c'est-à-dire pour un problème du genre

$$\frac{d\boldsymbol{x}}{dt} = f(t, \boldsymbol{x}), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0$$

il est souvent impossible de trouver une expression $\boldsymbol{x}(t) = \ldots$ (quelques chose en fonction de t) qui est solution de l'EDO ci-dessus. $\boldsymbol{x}$ peut désigner une seule variable d'état comme le N dans l'équation logistique ci-dessus, mais ça peut aussi être un vecteur $\boldsymbol{x} = (x_1, \ldots, x_n)$ contenant n variables d'état ; dans ce cas $f(t, \boldsymbol{x})$ est également un vecteur à n élements, $(f_1(t, \boldsymbol{x}), \ldots, f_n(t, \boldsymbol{x}))$,

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} f_1(t, \boldsymbol{x}) \\ \vdots \\ f_n(t, \boldsymbol{x}) \end{pmatrix}.$$

Une manière de traiter ce problème de ne pas connaître une solution analytique est de faire des simulations numériques. Dans la première partie de ce TP nous allons explorer le principe de la simulation numérique et les outils fournis par **R** pour un problème dont la solution est connue. Ceci vous permettra de vous rendre compte des possibilités que la simulation nous ouvre, mais aussi ses limites et incertitudes. Dans la deuxième partie on va appliquer ces outils à un problème plus compliqué dont la solution analytique n'est pas connue pour explorer les prédictions du modèle et concevoir des tests expérimentaux afin de valider (ou de rejeter) le modèle.

4.1.1 L'équation logistique résolue par la méthode d'Euler

Rappelons-nous la définition de la dérivée,

$$\frac{dx}{dt}(t) = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \approx \frac{x(t + \Delta t) - x(t)}{\Delta t}.$$

Plus Δt est petit, plus l'approximation à la fin est juste. Appliqué à l'équation logistique (par exemple pour décrire le nombre d'individus dans un agrégat de blattes ou de fourmis au cours du temps) ceci nous donne l'approximation

$$\begin{aligned} \frac{N(t + \Delta t) - N(t)}{\Delta t} &\approx r(1 - \frac{N(t)}{K})N(t) \\ \iff N(t + \Delta t) &\approx N(t) + \Delta t\, r(1 - \frac{N(t)}{K})N(t) \end{aligned}$$

Donc, connaissant les valeurs des paramètres r, K et de la valeur de N au temps 0 ($N(0) = n0$), on peut choisir un Δt et calculer de façon récurrente une solution approximative de notre équation logistique pour $N(\Delta t)$. Ensuite, $N(\Delta t)$ connu, vous pouvez calculer $N(2\Delta t)$ - et de suite pour $N(3\Delta t), N(4\Delta t), \ldots$. Ecrivez le code suivant dans un script, comprenez-le (sinon demandez à l'enseignant) et exécutez-le par copier coller, ou écrivez-le directement sur la ligne de commande de **R** ; je vous conseille la première solution en utilisant l'éditeur de script intégré dans **R**.

```
r=0.5;
K=110;
n0=1;

# définir delta t et définir le temps de simulation
deltaT = 2.0;
tempsDeSimulation=30; # durée pdt laquelle on veut simuler la dynamique
pasDeTemps = seq(0,tempsDeSimulation,by=deltaT);
   # vecteur des temps auxquels on simule la solution recursivement

# initialisation du vecteur qui contiendra la solution
# effectif[1] solution au temps 0
# effectif[2] solution au temps deltaT
# effectif[3] solution au temps 2*deltaT ...
effectif = rep(0,length(pasDeTemps));
effectif[1]=n0;  # le remplir avec la condition initiale
# boucle for qui calcule notre équation récurrente
for (i in 2:length(pasDeTemps)) {
  effectif[i] = effectif[i-1]+deltaT*r*(1-effectif[i-1]/K)*effectif[i-1];
```

```
}

# et on fait le graphique de la simulation en donnant les valeurs de
# l'abscisse et de l'ordonnée des points à tracer
plot(pasDeTemps,effectif,main="Croissance Logistique, simulation Euler dt=2",
     xlab = "temps", ylab = "Effectif",type='b');

# Gardons les valeurs des vecteurs dans une variable auxiliaire pour plus tard
pasDeTempsDt2 = pasDeTemps
effectifDt2 = effectif
```

Quelle est la qualité de cette approximation avec $\Delta t = 2$? Comme on connait la solution analytique de l'équation logistique,

$$N(t) = \frac{n_0 K}{(n_0 - exp(-rt)(n_0 - K))},$$

(voir le poly du cours) on peut directement la calculer et comparer à la solution numérique. Continuez donc votre script en remplissant un vecteur `solExact` avec les valeurs de N aux temps $t = 0, 1, \ldots, 30$ (avec une boucle `for`) et superposez cette solution sur la simulation numérique,

```
R> lines(0:30,solExact,col='red')
```

Qu'est-ce que vous observez ? Explorez la différence entre la solution approximative et la solution exacte en variant le $\Delta t = 2.0, 1.0, 0.5, 0.1$.

Exercice: Faites un commentaire (1-2 phrases) sur le rapport entre la qualité de la solution numérique, le Δt et le coût de calcul (nombre de multiplications à effectuer).

4.1.2 Solution avec les outils fournis par R

La méthode d'Euler n'est pas la meilleure façon de trouver une solution approximative (il faut choisir un petit Δt pour avoir une bonne approximation, ce qui coûte cher en temps de calcul). La famille de méthodes de Runge-Kutta donne des algorithmes beaucoup plus précis et moins coûteux. Ces méthodes sont fournies dans **R** par une bibliothèque que vous pouvez installer sur votre ordinateur. Regardons d'abord si cette bibliothèque est déjà installée

```
R> library(deSolve)                                    # charger la bibliothèque
```

Si **R** affiche un message d'erreur que cette bibliothèque est inconnue vous pouvez la télécharger dirèctement via internet. Cliquez dans **R** sur le menu `Packages - Install Package from CRAN`. Votre ordinateur va se connecter sur le site contenant les sources de **R** et vous offrir les bibliothèques disponibles (il faut peut-être d'abord choisir un « miroir », c'est-à-dire un serveur auxiliaire qui contient les installateurs de **R** et de ses bibliothèques ; celui de Lyon fera l'affaire). Sélectionnez `deSolve` et cliquez sur `ok`. Répondez à la fin de garder les fichiers téléchargés, c'est un fichier zip (compacté) que vous pouvez copier sur votre clé USB pour installer cette bibliothèque sur votre ordinateur à la maison (au cas où vous n'auriez pas un accès internet). Dans cette bibliothèque il y a une fonction `lsoda` qui utilise ces méthodes de Runge-Kutta.

Mais avant de pouvoir l'utiliser il faut écrire une fonction dans **R** qui calcule le coté droit de l'équation logistique ($f(N)$). Ecrire une fonction est assez simple. Pour nous entraîner, écrivons par exemple une fonction qui prend deux coordonnées x et y et les convertie en coordonnées polaires on écrit simplement dans un script

```
polaire = function(x,y) {
 r = sqrt(x^2 + y^2)
 cosphi = x/r
```

```
 sinphi = y/r
 phi = atan2(sinphi,cosphi)
 return(c(r, phi))
}
```
et après avoir exécuté ce code vous pouvez vous servir de cette fonction pour calculer des coordonnées polaires
```
R> polaire(1,0)
R> polaire(1,1)
R> polaire(-1,-1)
```
Est-ce que les réponses sont justes ? Est-ce que **R** travaille en degré ou en radiant ?

Pour utiliser la fonction avec la bibliothèque `deSolve` il faut l'écrire dans un format bien précis qui calcule le coté droit de notre EDO. Ecrivez dans le script

```
logistiqueRK = function(t,x,parms) {
  coteDroit = r*(1-x/K)*x;
  return(list(coteDroit))
}
```

Pour pouvoir être utilisé par `deSolve` cette fonction a plusieurs contraintes de syntaxe :
- elle doit avoir trois arguments, le temps t, un vecteur x de la taille du nombre de variables d'état (chez nous c'est 1), et un vecteur *parms* qui peut contenir les valeurs de vos paramètres ; mais vous n'êtes pas obligé de vous servir de *parms* entre les accolades (dans ce dernier cas r et K prendrons les valeurs défini auparavant, c'est ce que j'ai fait ci-dessus).
- le résultat doit être rendu en forme de liste (un format particulier dans **R**).

N'oubliez pas d'enregistrer votre script.

La fonction dans `deSolve` qui implémente la méthode de Runge-Kutta s'appelle `lsoda`. Elle a la structure suivante :

```
lsoda(condInitiale,tempsSimulesVoulus,nomFonctionDuCotéDroit,parms)
```

où `condInitiale` est la condition initiale au temps `0`, $n(0) = n0$, `tempsSimulesVoulus` est un vecteur qui contient les instants de temps auquels on veut obtenir une solution approximative de l'EDO, `nomFonctionDuCotéDroit` est le nom de la fonction définissant le coté droit de notre EDO (et que nous devons programmer nous-même, voir ci-dessus) et `parms` est le vecteur contenant les valeurs des paramètres (chez nous il sera simplement le vecteur nul `c(0)` parce qu'on définit r et K en dehors de la fonction au niveau global). Voici la mise en oeuvre :
```
R> tvoulu=seq(0,30,by=2.0);              # on veut une solution entre 0 et 30, pas 2.0
R> n0 = c(1)                                                  # condition initiale
R> sol=lsoda(n0,tvoulu,logistiqueRK,c(0));               # mettre la solution dans sol
R> dim(sol)
```
La dernière commande vous indique que `sol` est un tableau a 16 lignes et deux colonnes : la première colonne contient les temps `tvoulu` et la seconde les solutions de n à ces temps,
```
R> sol                                                    # afficher le contenu de sol
```
Vous vous rappelez comment extraire ces colonnes du tableau ? L'élément de la $k$ème ligne et $m$ème colonne s'affiche par la commande `sol[k,m]`, et pour afficher toutes les lignes d'une colonne on remplace le k par rien
```
R> sol[,1]                                    # affiche la première colonne (les temps)
R> sol[,2]                                 # affiche la seconde colonne (l'effectif N(t))
```

Exercice: Modifiez votre script pour superposer cette solution avec la solution analytique, la première en noir et la deuxième en rouge. Comment la solution par `lsoda` se compare-t-elle à une solution par la méthode d'Euler avec $\Delta t = 2.0$ (réutilisez les variables `effectifDt2` et `pasDeTempsDt2`) ? Notez vos commandes et le résultat graphique.

Remarque : avec la méthode d'Euler ci-dessus on avait simulé la solution numérique en passant par des pas Δt fixe. Cette simulation par un pas fixe est toujours dangereuse si la trajectoire fait des mouvements très fins. La solution est l'utilisation de méthodes qui adaptent dynamiquement la taille de Δt, le laissant grand quand la trajectoire fait des grands mouvements et le rapetissant quand elle fait des mouvements fines. La fonction `lsoda` fait cette adaptation dynamique du Δt et ensuite elle vous rend la solution interpolée aux temps que vous aviez indiqué dans `tvoulu`.

4.1.3 Exploration du modèle de ravitaillement avec recrutement

Après avoir gagné un peu de confiance dans l'approximation fournis par `lsoda` explorons maintenant un modèle dynamique dont la solution analytique n'est pas connue. On prendra l'exemple du ravitaillement des fourmis avec recrutement. Rappelons les variables d'état et le modèle

$$\begin{aligned} R(t) &= \text{nombre de fourmis à la recherche} \\ M(t) &= \text{nombre de fourmis en train de manger} \\ \frac{dR}{dt} &= cM(N_{tot} - M - R) - r_A R - r_R R \\ \frac{dM}{dt} &= r_R R - r_M M \end{aligned}$$

Les paramètres ont été mesurés expérimentalement (voir le cours pour les détails et interprétations) : $r_R = 1.0$, $r_A = 0.033$, $r_M = 0.5$, $N_{tot} = 100$ et $c = 0.04$.

Nous allons d'abord dessiner les isoclines pour M et pour R dans l'espace de phase (M en abscisse, R en ordonnée). Commencez un nouveau script avec la définition des variables,

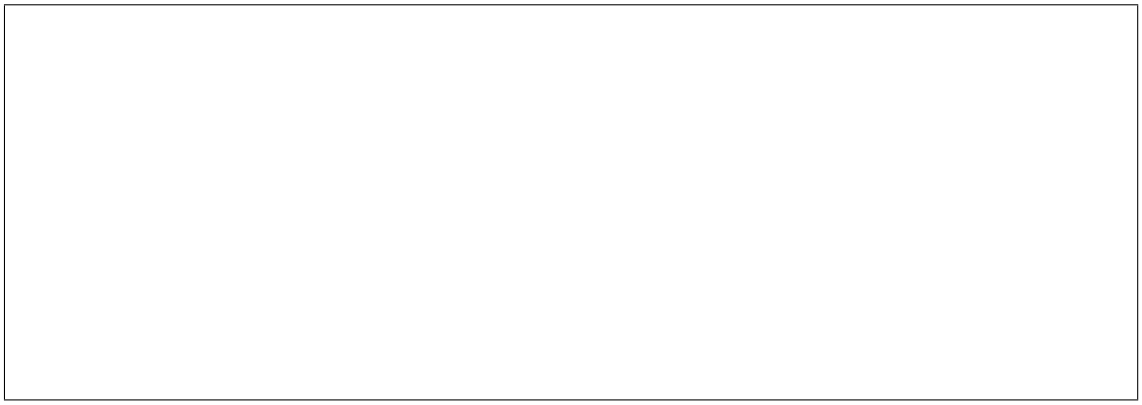
```
c=0.04;
rr=1.0;
ra=0.033;
rm=0.5;
ntot = 100;
```

Vous connaissez maintenant suffisamment les fonctions graphiques `plot` et `lines` pour dessiner les deux isoclines :

$$\begin{aligned} \text{isocline}_R &: \quad R = \frac{cM(N_{tot} - M)}{r_A + r_R + cM} \\ \text{isocline}_M &: \quad R = \frac{r_M}{r_R} M. \end{aligned}$$

Comment s'orientent les flêches principales dans ce graphique ?

Exercice: Dessinez le graphique que **R** vous construit, ajoutez-y à la main les flêches principales et notez le script avec lequel vous l'avez créé

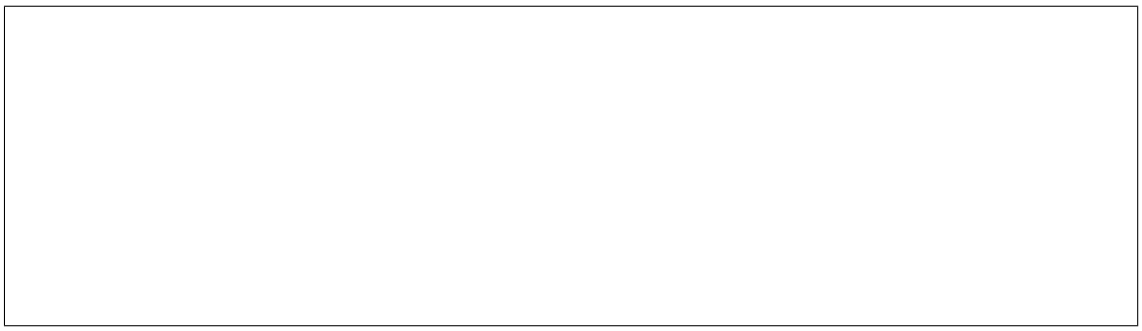


On va maintenant ajouter une trajectoire simulée à ce portrait d'isoclines. Ajoutez d'abord au début de votre fichier une fonction `fourmis(t,x,parms)` qui calcule le coté droit de votre modèle,

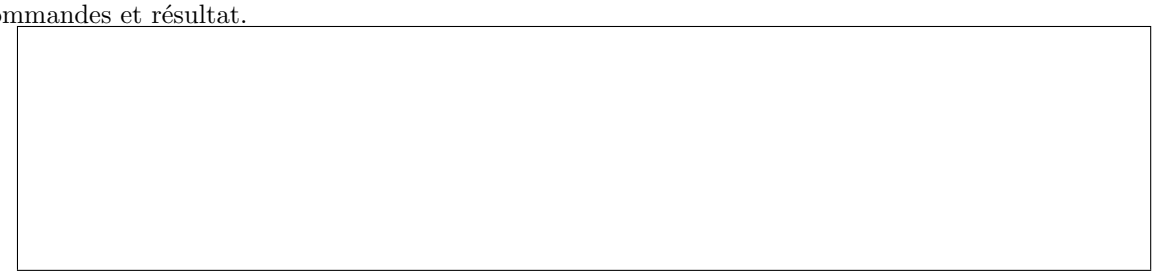
```
fourmis=function(t,x,parms) {
  xp = rep(0,2);
  xp[1]=c*x[2]*(ntot-x[2]-x[1])-ra*x[1]-rr*x[1];
  xp[2]=rr*x[1]-rm*x[2];
  return(list(xp));
}
```

(veuillez noter que $\boldsymbol{x}$ est maintenant un vecteur de taille 2 avec première variable d'état R et seconde variable d'état M). Ajoutez ensuite à votre script des isoclines des commandes qui simulent une solution avec conditions initiales $R_0 = 1, M_0 = 0$ (`condInitiale` va donc être un vecteur de taille 2), une durée de simulation de 0 à 20 (calculez la solution avec 0.5 de pas en pas) et ajoutez cette solution au graphique des isoclines. Choisissez des couleurs différentes pour les isoclines et la trajectoire.

Exercice: Dessinez le graphique avec isoclines et trajectoires et notez le script avec lequel vous avez simulé et affiché la trajectoire. Notez sur le gaphique tous les équilibres (les points (M, R) tel que $dR/dt = 0$ et $dM/dt = 0$).



Quelle est la forme de la dynamique du nombre de chercheuses et de ravitailleuses dans l'espace de temps (le temps t en abscisse, $R(t)$ ou $M(t)$ en ordonnée) ? Utilisez **R** pour les dessiner et notez commandes et résultat.



Est-ce que vous avez bien anticipé la forme des trajectoires ? Est-ce que les trajectoires ont atteint leur équilibre (état stationnaire) ? Sinon, modifiez le temps de simulation pour y arriver.

Après toutes ces préparations nous sommes en mesure d'explorer les prédictions de notre modèle.

1. Explorez la stabilité des deux équilibres en choisissant des conditions initiales très proches de ces équilibres et en observant la dynamique simulée. Résultat ?

2. Variez le paramètre N_{tot} de 100 à 10 (10 de pas en pas). Qu'est-ce qui se passe avec la trajectoire et les équilibres ? En terme biologique, qu'est-ce que le modèle prédit si la taille de la population se réduit ?
3. Avez-vous une idée d'une expérience avec laquelle on pourrait valider ou rejeter cette prédiction ?

4.2 Entrainement

4.2.1 L'effet Allee

Une variante du modèle logistique inclue ce qu'on appelle « l'effet Allee » (voir le poly pour plus d'informations). L'équation est

$$\frac{d}{dt}N = rN\left(1-\frac{N}{K}\right)\left(\frac{N}{\epsilon}-1\right), \quad 0<\epsilon\ll K.$$

Explorez par simulation numérique l'état stationnaire de ce système en fonction de la condition initiale ($N_0 \in (0,\epsilon)$ ou $N_0 \in (\epsilon,\infty)$). Ensuite, tracez le coté droit de l'EDO en fonction de N et interprétez votre résultat graphiquement.

4.3 Informations supplémentaires

Une fonction dans **R** peut avoir un nombre arbitraire d'arguments. Par exemple, la fonction qui calcule pour `lsoda` le côté droit de l'EDO a trois arguments. Le troisième argument permet de passer les valeurs des paramètres utilisés dans le côté droit, par exemple les paramètres r et K dans la fonction logistique

```
logistique = function(t,x,parms) {
  return(parms[1]*(1-x/parms[2])*x)
}
```

Avec cela on n'a pas besoin de se souvenir toujours que les paramètres dans la fonction *logistique* s'appellent r et K, mais on passe en troisième argument un vecteur contenant les valeurs de r et de K et en second argument le x pour lequel on veut calculer la fonction logistique :

```
R> logistique(0,1,c(0.5,10));logistique(0,10,c(0.5,10));            # quelques exemples
R> logistique(0,1,c(0.5,200)); logistique(0,5,c(2,10));              # autres exemples
```

Pour simuler une solution de l'EDO on appelle `lsoda` maintenant de la façon suivante :

```
R> sol = lsoda(n0, tvoulu, logistique, c(0.1,10.0)
```

Le premier argument dans `logistique`, t, n'est pas utilisé du tout dans notre exemple, mais dans le contexte général des EDO il serait utile dans le cas d'un système non-autonome.

4.3.1 Un algorithme très adaptable pour simuler des EDO

La fonction `lsoda` donne beaucoup d'options pour influencer la précision numérique. Sa forme générale est

```
lsoda(x, tvoulu, fonction, parms, rtol=1e-06, atol=1e-06,trcit=NULL,
        jacfunc=NULL, verbose=FALSE, dllname=NULL, hmin=0, hmax=Inf)
```

Vous connaissez déjà les quatre premiers arguments, et pour toutes les autres il y des valeurs par défaut. Pour plus d'informations consultez l'aide sur `lsoda` et un numéritien pour vous l'expliquer.

4.3.2 L'attracteur de Lorenz

Le système classique qui a démarré la recherche sur les dynamiques non-linéaires est l'attracteur étrange de Edward Lorenz. En travaillant sur une simulation du temps avec les équations suivantes,

$$\begin{aligned}\frac{dx}{dt} &= -10x+10y\\ \frac{dy}{dt} &= 28x-y-xz\\ \frac{dz}{dt} &= -8/3z+xy,\end{aligned}$$

Edward Lorenz a répété une fois en 1961 une simulation numérique en redonnant la même condition initiale, mais par flême il l'a écrit avec une précision moindre (3 chiffres significatifs, 0.506, au lieu de 6, 0.506127). Quelle n'a pas été sa surprise qu'après une heure de simulation la trajectoire suivait un chemin complètement différent de la première simulation. Il avait en fait découvert le premier système chaotique ou attracteur étrange. La moindre variation dans une condition initiale ou un des paramètres rend la prédiction à long terme complètement impossible. On appelle ce système aujourd'hui l'attracteur de Lorenz.

Pour le simuler vous-même, écrivez une fonction pour le coté droit et utilisez `ode` pour simuler la trajectoire pour un temps de 0 à 20 avec des conditions initiales $(0.1, 0.1, 0.1)$. Visualisez d'abord la projection sur le plan (x, y) avec la commande `plot` et en prenant seulement les dynamique de $x(t)$ et de $y(t)$. Ensuite, explorez comment visualiser la solution dans l'espace (x, y, z) avec la commande `cloud(...)` (dans la bibliothèque `lattice`, chargez-la d'abord).

Chapitre 5

Modèles statistiques et leur lien avec les modèles déterministes

Jusqu'à présent on a vue de façon quasiment exclusive des modèles déterministes (c'est-à-dire des modèles en EDO ou en équations de récurrences dont l'évolution temporelle est entièrement défini par les conditions initiales et les paramètres du modèle). Dans le TP d'aujourd'hui on va créer d'abord un modèle statistique du temps de séjour d'une fourmis dans le nid et ensuite regarder le liens entre la dynamique collective d'un millier d'individu et une description déterministe de la sortie du nid.

On va utiliser deux fonctions de **R** que vous ne connaissez probablement pas encore. La fonction

```
R> runif(3)
```

crée un vecteur comprenant 3 chiffres (pseudo-)aléatoires provenant d'une distribution uniforme entre 0 et 1. Nous allons d'habitude créer un seul chiffre aléatoire à la fois, `runif(1)`. La deuxième fonction est la boucle `while` dont on se sert si on ne sait pas à l'avance combien de fois il faut répéter une commande. Par exemple, pour calculer le plus petit n tel que $n! > 10000$ (factoriel de n) je tape

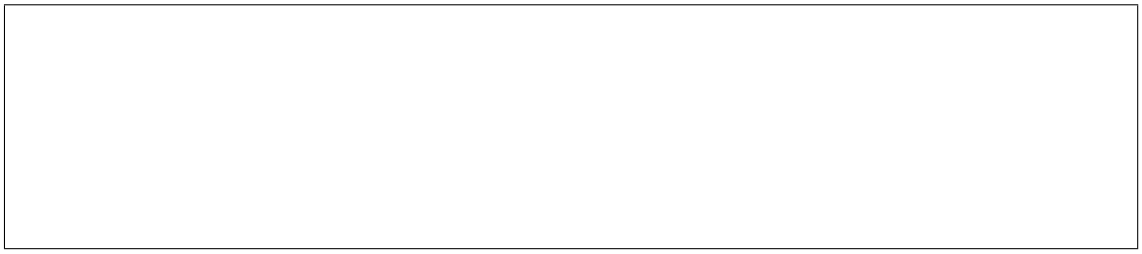
```
R> n=2; nfact=2;
R> while (nfact <= 10000) {n=n+1;nfact=nfact*n;}; n
```

5.1 Le temps de séjour dans le nid

Supposons que nous avons une fourmis dans un nid avec taux $r = 20.0$ (min^{-1}) de sortir. Durant un intervalle de temps $\Delta t < 1$ elle aura donc une probabilité de $r\ \Delta t$ de sortir. Au niveau algorithmique, on peut dire que si `runif(1) < r*Δt` elle sort, sinon elle reste dans le nid et on reéssaye au Δt suivant.

Utilisez la boucle `while` pour calculer pour un individu combien de temps il reste dans le nid avant de sortir (prenez $\Delta t = 0.2$). Notez le code et le résultat d'une simulation

Copiez ensuite le code dans un fichier de l'éditeur de scripts (que vous enregistrez à l'endroit habituel). Astuce : pour afficher à l'écran une valeur quand le programme est à l'intérieur d'une boucle (`for` ou `while`) utilisez la commande `print(...)`. Refaites la simulation 10 fois et notez les temps de séjours. Créez ensuite un vecteur de taille 100 et remplissez-le avec 100 temps de séjour - en faites un graphe qui affiche le temps en fonction du numéro de simulation. Commentaire ?



Pour finir cette partie, faites 1000 simulations pour en estimer la probabilité que le temps de séjour est plus grand que 15 (vous pouvez y ajouter l'intervalle de confiance de cette estiamtion avec la formule de l'approximation normal). Augmentez ensuite le Δt de 0.2 en 0.2 pour déterminer à partir de quelle Δt cette estimation change radicalement.

5.2 Traduction du temps de séjour en variable aléatoire

Notons par d_s la variable aléatoire « temps de séjour dans le nid ». La fonction de densité de probabilité de d_s est donné par $p(d_s) = r\exp(-rt)$. Quel est la vraie probabilité de quitter le nid dans l'intervalle de temps $(0, \Delta t)$? Et dans $(t, t + \Delta t)$ pour $t > 0$? Comparaison avec la section précédante ?

Réflechissez comment on pourrait simuler un temps de séjour dans le nid sans faire appel à un maillage temporel comme avec le Δt dans les simulations. Donnez l'algorithme et implémentez-le dans **R**.

Chapitre 6

Les modèles dans l'espace

6.1 Marche aléatoire dans l'espace 1D

Pour commencer à comprendre les techniques de modélisation qui prennent l'espace explicitement en compte nous allons commencer avec le cas le plus simple : le déplacement d'un animal dans un espace 1D (par exemple, une fourmi qui se déplace en exploration le long d'une branche fine ou une blatte qui se déplace le long du bord d'une grande arène circulaire).

Le modèle de base considère que l'animal se déplace à une vitesse v constante et qu'il change de direction à un taux constant $p_{demiTour}$ (avec unité m^{-1}, càd par unité de distance). Nous allons d'abord étudier la dynamique de ce type de déplacement en traçant la position de l'animal (en ordonnée, au début l'animal est à la position 0) en fonction du temps t (en abscisse). Le script `marcheAleaUneFourmi.R` fait déjà tout et on va se contenter de l'analyser :

- Définition des paramètres `tauxDemiTour` (unité m^{-1}), `vitesse` (unité $\frac{m}{s}$) et le pas de temps de simulation `dt` (unité s). Ces paramètres sont celles d'une fourmi *Messor sanctus* à environ 30°C.
- Calcul de la probabilité de faire un demi tour par unité de temps (veuillez noter que le pas de temps doit être choisi tel que cette probabilité soit beaucoup plus proche de 0 que de 1 pour assurer que la discrétisation du calcul soit une bonne approximation du déplacement continu ; un chiffre autour 0.1 est bon),
- on définit le temps de simulation et on calcule le nombre de pas à simuler, ce qui permet d'initialiser ensuite un vecteur `posBete` qui contiendra la position de l'animal à chaque pas de temps,
- on initialise la direction de l'animal au hasard (veuillez noter l'utilisation de la fonction `runif(n)` qui crée un vecteur de longueur n rempli de chiffres aléatoires uniformément distribués dans l'intervalle $(0, 1)$),
- après définition de la position initiale de l'animal on fait une boucle sur l'ensemble des pas de temps, testant à chaque pas de temps si l'animal fait demi tour ou pas avant de le faire avancer par une distance `dt*vitesse`,
- et finalement on trace la position en fonction du temps.

Faites plusieurs simulations avec ce script, recopiez quelques trajectoires typiques et commentez-les.

6.1.1 Le déplacement net au carré

Recopiez ces commandes dans un nouveau script et modifiez-le pour tracer la distance au carré par rapport à la position initale et au cours du temps. Ajoutez la variable `nombreBetes`, mettez-là à 10 et calculez la moyenne de cette distance au carré sur 10 marches aléatoires (créez autant de vecteurs de données qu'il vous faut pour cela en restant économique). Notez le code modifié :

Tracez maintenant cette distance au carré moyen pour 100 et pour 1000 animaux. Qu'est-ce que vous observez ?

Comme une moyenne empirique n'est qu'une estimation avec une certaine erreur (même si on la calcule sur 1000 valeurs) il faut ajouter sur ce graphique les erreurs standards de chaque distance au carré moyenne estimée. Toujours dans un esprit d'économie de mémoire vive, calculez cet erreur standard par la formule $\text{var}(X) = E(X^2) - E(X)^2$ (X représente une variable aléatoire et cette formule est valable pour n'importe quelle variable aléatoire, $E(...)$ représente la valeur attendue). Ajoutez au graphique les intervalles de confiance en traçant une ligne rouge à moyenne $\pm$ erreur standard (donc un $\text{IC}_{0.68}$ de 68%). Ajoutez enfin une droite de régression et commentez le résultat

Enfin, si un animal se déplace à une vitesse constante, sa distance net au carré devrait augmenter de façon quadratique, au moins avant les premiers demi-tours ! Pour visualiser cette phase de la courbe (qu'on appelle phase balistique) refaite votre graphique pour un temps de simulations de 5 s et avec $dt = 0.1$. Recopiez et annotez le graphique.

6.2 Déplacement de plusieurs animaux simultanément

Après l'exploration des propriétés de la marche aléatoire, passons maintenant à l'exploration de la distribution de plusieurs animaux au cours du temps après les avoir lâchés au même endroit. Reprenez le premier script (marche aléatoire d'un seule animal) qu'on modifiera pour regarder la distribution de plusieurs animaux à un instant donné plutôt que de regarder l'évolution au cours du temps. Il suffit donc d'ajouter le nombre d'animaux (`nombreBetes`) et deux vecteurs `posBete` et `dirBete` de longueur `nombreBetes` qui contiendront l'orientation et la position des animaux à un instant donné. A chaque pas de temps `dt` affichez un histogramme des positions des animaux et regarder défiler cette distribution au cours du temps. Commentez :

Remarque : vous pouvez afficher le temps actuel dans le titre du graphique en donnant à l'option `main` le résultat de la fonction `paste("Temps=",k*dt)` au $k$ème pas dt.

La théorie (voir cours) prédit que la distribution de ces animaux, lâchez au début tous à la position 0, suivra une Gaussienne avec variance $Dt2$ au cours du temps t, où le coefficient de diffusion D se calcule à partir de la vitesse v et du taux de faire demi-tour τ_{demiT} (`tauxDemiTour`) par $D = v/(2\tau_{demiT})$. Tracez l'histogramme en fréquences relatives et superposez là-dessus cette distribution Gaussienne en rouge. Notez le code (3-5 lignes) :

En vous inspirant des tailles potentielles que les colonies de fourmis peuvent atteindre, faites maintenant des simulations avec 1000, 10'000 ou même 100'000 animaux. Qu'est-ce que vous remarquez ?

6.3 L'équation de diffusion et sa simulation

Au lieu d'utiliser un modèle individu centré (qui devient très gourmand avec un grand nombre d'animaux) on pourrait aussi s'appuyer sur la théorie mathématique ci-dessus qui démontre qu'un déplacement simultané de beaucoup d'animaux, chacun faisant une marche aléatoire, peut être décrit par l'équation de diffusion

$$\begin{aligned} \frac{\partial p}{\partial t} &= D\frac{\partial^2 p}{\partial x^2} \\ p(0,x) &= p_0(x) \\ p(t,l_-) &= p(t,l_+) = 0 \end{aligned}$$

avec $p(t,x)$ la densité d'individus au temps t à l'endroit x, $D = v/(2\tau_{demiT})$ le coéfficient de diffusion, $p_0(x)$ la densité initiale à l'endroit x et $p(t,l_-), p(t,l_+)$ les conditions au bord de l'espace (l_-, l_+) qu'on simule (absorbante dans le cas ci-dessus, voir cours). Le seul problème est maintenant de trouver un algorithme numérique qui calcule la solution de cette équation au cours du temps. Nous procéderons comme si nous ne savions pas qu'une solution analytique existe (voir ci-dessus) et nous utiliserons une méthode assez universelle qui s'inspire de la méthode d'Euler vue dans la section 4.1.1.

Comme il y a une approximation de la dérivée temporelle,

$$\frac{\partial p(t,x)}{\partial t} \approx \frac{p(t+dt,x) - p(t,x)}{dt}$$

il en existe une pour la dérivée seconde spatiale

$$\frac{\partial^2 p(t,x)}{\partial x^2} \approx \frac{p(t,x+dx) - 2p(t,x) + p(t,x-dx)}{dx^2}.$$

Combinant les deux on obtient

$$p(t+dt,x) = p(t,x) + dtD\left(\frac{p(t,x+dx) - 2p(t,x) + p(t,x-dx)}{dx^2}\right).$$

Il faut donc discrétiser l'espace (comme on avait discrétisé le temps), disons par exemple qu'on simule le déplacement entre -0.5 et 0.5 m avec un pas d'espace $dx = 0.05$ m. Le vecteur `x` contenant les densités dans chaque longueur dx a donc une longueur $n = 21$, et la case contenant $x = 0$ se trouve à `x[11]`. Les

conditions de bords (réfléchissantes) sont légèrement différentes,

$$p(t+dt, l_-) = p(t, l_-) + dtD\left(\frac{p(t, l_- + dx) - p(t, l_-)}{dx^2}\right)$$
$$p(t+dt, l_+) = p(t, l_+) + dtD\left(\frac{p(t, l_+ - dx) - p(t, l_+)}{dx^2}\right)$$

on les obtient on mettant les densités en dehors de l'espace (l_-, l_+) à la valeur sur les bords, $p(t, l_- - dx) = p(t, l_-)$ et $p(t, l_+ + dx) = p(t, l_+)$.

Avec cela on n'a pas encore tout défini, il nous faut encore définir la condition initiale. Disons qu'on lâche au temps $t = 0$ mille individus à l'endroit $x = 0$, il faut donc remplir le vecteur `x` au bon endroit avec la bonne densité,

```
R> x[11] = 1000/dx
```

Le script `marcheAleaDiffusionIncomplet.R` contient déjà une partie du code nécessaire, il suffit de remplir les ... avec le bon code d'après les informations ci-dessus. Commentez l'évolution de la solution et comparez au résultat avec le modèle individu centré.

Superposez maintenant la solution numérique avec la solution analytique de la dernière section pour juger la qualité de l'approximation numérique. Notez le code

Remarque : l'algorithme ci-dessus est tellement classique dans la littérature qu'il a son propre nom, « Forward Time Centered Space » ou simplement FTCS. Mais comme dans le cas de la simulation d'une EDO l'algorithme (Runge-Kutta) ne converge pas toujours vers la vraie solution (dans ce cas là il fallait surtout choisir un pas de temps dt suffisamment petit), dans notre cas il y a aussi le risque de non-convergence. Mais il y a un critère qui assure que l'algorithme converge vers la bonne solution,

$$\frac{2Ddt}{(dx)^2} \leq 1$$

Vérifiez que cette condition est effectivement suivi dans notre calcul.

Chapitre 7

Tests statistiques sur un ou deux échantillons

Dans le dernier TP, vous avez mesuré vos propres temps de réaction sous différentes conditions expérimentales. On reprendra aujourd'hui ces données pour les analyser dans **R**.

7.1 Lecture des données - les `data.frame`

La première chose à faire est d'indiquer à **R** le répertoire dans lequel se trouve le fichier `tempsReaction2008.txt`. Regardez d'abord le répertoire par défaut quand vous lancez **R**

```
R> getwd()
```

et ensuite, utilisez le menu `File - Change dir ...` pour sélectionner le répertoire contenant le fichier (cela peut être la disquette `A:` ou le répertoire `NeuroModelisation`). La commande

```
R> list.files()
```

vous affiche tous les fichiers dans ce répertoire.

Les données du TP "Temps de réaction" se trouvent dans le fichier `tempsReaction2008.txt`. Lisez ces données avec la commande

```
R> tr = read.table("tempsReaction2008.txt",h=T,dec=",")          # lecture des données
R> tr                                 # afficher les données stockées dans la variable tr
```

L'option `h=T` indique à la fonction que la première ligne du fichier ne contient pas de données mais les noms des colonnes de données (h=header, T=True), et l'option `dec=","` indique que les décimales sont représentées par une virgule (ce qui arrive si vous préparez votre jeux de données dans Excel) : ouvrez ce fichier dans l'éditeur de script pour voir son format.

Dans quel format est-ce que **R** a stocké les données dans la variable `tr` ? Cela à l'air d'un tableau ou d'une matrice. Et effectivement, si on demande les dimensions des la variable `tr`,

```
R> dim(tr)
```

on apprend qu'elle a 24 lignes (ou un chiffre différent, selon le nombre d'étudiants qui ont participé au dernier TP) et 6 colonnes. Mais quand vous demandez à **R** si c'est une matrice

```
R> is.matrix(tr)
```

le logiciel vous répond que non (`F`). En fait, c'est ce qu'on appelle un `data.frame` :

```
R> is.data.frame(tr)                                                    # type de variable?
```

C'est un type de variable en forme de matrice dans laquelle chaque ligne représente les mesures pour un individu et chaque colonne regroupe les mesures du même type (dans le cas présent, le sexe, le temps de réaction simple (en ms) sans distraction, ...). Pour connaître les noms des colonnes, il y a la commande

```
R> names(tr)
```

et pour utiliser une colonne particulière, on ajoute au nom de la variable un $ et le nom de la colonne :

```
R> tr$Sexe                                              # la colonne du sexe des individus
R> tr$simple.sans.dist                                # colonne des mesures sans distraction
```

Par contre, si vous tapez seulement

```
R> Sexe
```

R ne connaît pas (encore) ce nom. Comment rendre accessibles les noms des colonnes comme de simples variables ? C'est la commande

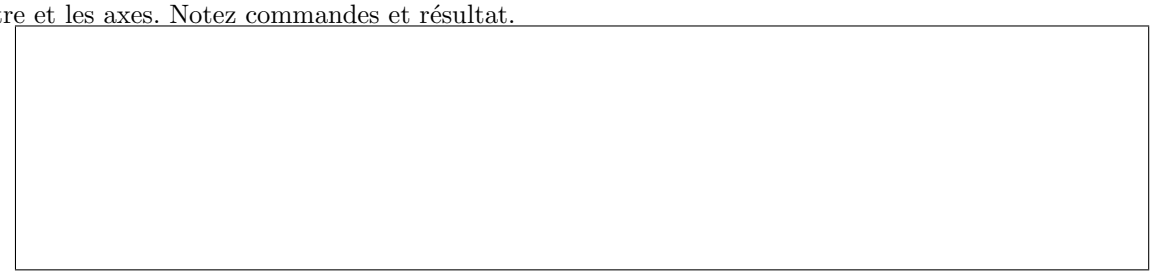
```
R> attach(tr)                          # faire une copie des colonnes en tant que vecteurs
R> Sexe                              # est maintenant directement disponible comme vecteur
R> simple.avec.dist; simple.diff
```

7.2 Visualisation des données

La colonne `simple.sans.dist` contient tout les temps de réaction aussi bien pour les femmes que pour les hommes. Nous aimerions maintenant faire un boxplot pour les femmes et un autre pour les hommes. Comment les séparer ? Une première solution est la commande `split`,

```
R> mesures = c(3.5,2,3,1,4.5)                          # les valeurs mesurées pour ...
R> leSexe = c("H","H","F","F","H")                   # ...pour les hommes ou les femmes
R> split(mesures ,leSexe)
```

Utilisez cette fonction pour créer une nouvelle variable `simpleHF` qui contient deux éléments : les temps de réaction des femmes et ceux des hommes. Tracez ensuite un boxplot avec des noms explicites pour le titre et les axes. Notez commandes et résultat.



En fait, il y aurait une solution encore plus élégante : essayez la commande

```
R> boxplot(simple.sans.dist ~ sexe)
```

Le signe $\sim$ (touches `Alt Gr`, 2 et espace) veut dire « en fonction de ». **R** comprend qu'il faut tracer autant de boîtes à moustaches qu'il y a de catégories dans `sexe`, donc 2. Ce signe $\sim$ permet d'écrire en formule ce que nous voulons tracer ou tester. On y reviendra.

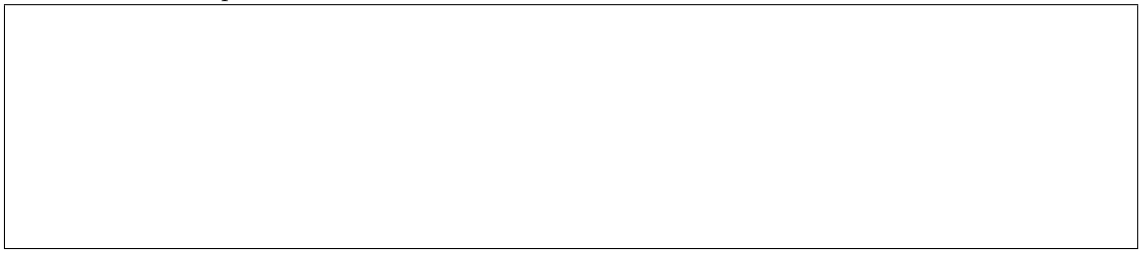
Enfin, on voit de plus en plus dans les publications un nouveau type de boîte à moustache avec des boîtes un peu tordues

```
R> boxplot(simple.sans.dist ~ sexe, notch=T)
```

Dans ce cas, la hauteur du triangle représente une estimation de l'intervalle de confiance à 95% de la médiane, ce qui permet de voir tout de suite s'il y a une vraie différence entre les échantillons ou pas.

7.3 Y a-t-il une différence entre le temps de réaction des femmes et des hommes ?

Reprenez la variable `simpleHF` : on veut tester s'il existe une différence significative (à $\alpha = 0.05$) entre le temps de réaction des femmes et des hommes. La fonction `shapiro.test` permet de tester la normalité des données. Appliquez-la séparément aux temps de réaction des deux sexes. Notez les commandes et le résultat avec son interprétation.



Il faut aussi tester l'homoscédastisticité. La fonction `var.test` permet de le faire. Notez H_0, la commande, le résultat et la conclusion. Est-ce que l'on a le droit d'utiliser un test de Student ? Si oui, la commande s'appelle

```
R> t.test(simpleHF$H,simpleHF$F,var.equal=T)
```

Qu'est-ce que l'option `var.equal` (utilier l'aide de **R** pour répondre) ? Pourquoi est-ce que je l'ai mise à

`T` (=true) ? Notez H_0, le résultat et la conclusion.

Le signe $\sim$ sert de nouveau à simplifier l'écriture : on veut tester si le facteur `Sexe` a une influence sur le temps de réaction, c'est-à-dire qu'on veut étudier la relation `simple.sans.dist` $\sim$ `Sexe`. Effectivement, la commande

```
R> t.test(simple.sans.dist ~ Sexe, var.equal=T)
```

nous donne le même résultat (essayez).

Pour finir cette partie, regardons si la distraction change le temps de réaction de façon significative. C'est un test de student apparié dans lequel on teste si la différence entre le temps de réaction sans distraction et le temps de réaction avec distraction est significativement différente de $\mu = 0$. J'ai mis les différences dans la variable `simple.diff`. Testez la normalité des différences et appliquez ensuite le test de student

```
R> t.test(simple.diff,mu=0)
```

Notez le résultat et l'interprétation. Faites une phrase qui résume la question, vos démarches et le résultat (comme pour un rapport).

7.4 Les interfaces graphiques des logiciels statistiques

Les logiciels statistiques que vous rencontrerez dans les laboratoires ou que vous pouvez acheter dans le commerce (par exemple Systat, MiniTab, SPSS, ...) sont maintenant doté d'une interface graphique qui vous permettent de faire les analyses par le bon choix des menus. Afin de vous donner un aperçu de l'utilisation de ces logiciels on va refaire la même analyse dans une telle interface : Rcommander (ou Rcmdr).

Pour lancer Rcmdr il suffit de taper la commande

```
R> library(Rcmdr)
```

et vous verrez apparaître une nouvelle fenêtre avec des menus. La première chose à faire et de charger les données. Pour cela sélectionnez les menus

```
Données -> Importer des données -> depuis un fichier texte ...
```

Rcmdr vous demande d'abord de donner un nom à ce jeu de données (appelez-le tr2) et affiche ensuite une fenêtre de dialogue avec des options, par exemple pour indiquer si la première ligne contient les noms des colonnes. On prendra les options par défaut (comprenez-vous leur sens ?). Cliquez ok et sélectionnez notre fichier de données des temps de réaction. Vous pouvez maintenant visualiser ces données en cliquant sur `Visualiser` ou les modifier en cliquant sur `Editer`. Faites le second et cliquer sur le nom de la première colonne (`Sexe`) : vous verrez deux cases à cocher, `numeric` s'il s'agit d'une colonne de données mesurées ou `character` s'il s'agit d'un facteur. La colonne `Sexe` devrait être du type facteur. Vérifiez que toutes les autres colonnes sont du type `numeric`.

Vous avez probablement remarqué qu'à la fin des menus les deux fenêtres de Rcmdr se sont remplies avec des commandes **R** : en fait, la fenêtre en bas rapporte la commande que Rcmdr a généré en fonction des menus et options choisi et la fenêtre en haut fonctionne comme la copie script que vous pouvez modifier, rearranger, sauver dans un fichier ou simplement copier/coller dans la fenêtre de **R** pour refaire la même chose.

Pour refaire la boîte à moustache des temps de réaction en fonction du sexe il faut savoir que "Boîte de dispersion" est un synonyme. Sélectionnez ce menu dans le menu `Graphes`, sélectionnez ensuite la colonne `simple.avec.dist` (pour voir un autre jeu de données) et cliquez 'graphe par groupe' pour sélectionner le `Sexe` comme facteur. Cliquez deux fois ok et vous verrez le résultat.

Remarque : Rcmdr ne permet pas de tout faire, par exemple si vous voulez ajouter un titre a votre graphique il faut ajouter l'option `main="Temps de reaction"` dans la fenêtre script et ensuite copier/coller la commande entière dans le fenêtre de commande de **R**. Rcmdr permet donc surtout de faire une première analyse et de trouver le nom des fonctions utilisées et la syntaxe, mais après on fait souvent le travail fin dans **R**.

Avant de passer au test de Student il faut vérifier l'homoscédasticité. Sélectionnez les menus

```
Statistiques -> Variance -> test de Bartlett ...
```

en choisissant les mêmes colonnes comme ci-dessus. Quel est le résultat ? Notez la commande et l'interprétation du résultat

La normalité ne peut pas se tester dans Rcmdr (ou je ne l'ai tout simplement pas encore trouvé), il faudrait donc le faire dans **R**.

Pour finir, sélectionnez les menus
`Statistiques -> Moyennes -> t-test indépendant ...`
et sélectionnez les bonnes colonnes. Je vous laisse choisir vous-mêmes les bonnes options. Quel est le résultat ?

7.5 Les tests non-paramétriques : exemple de l'influence de la température sur le dépot spontané de cadavres chez la fourmi *Messor sancta*

Pour le dernier exemple, nous reprenons les données du premier cours de modélisation dans lequel on avait modélisé le temps pendant lequel une fourmi transporte un cadavre avant un dépôt spontané. Ici, nous voulons tester si ce temps diffère significativement selon que l'expérience est réalisée à 16 °C ou à 30 °C. Les données sont trop nombreuses pour être saisies à la main. Mais je vous les ai envoyées au préalable par email et on peut les lire directement à partir des fichiers. Voici les commandes :

```
R> getwd()                # get working directory, vous indique où R cherche le fichier
```

Si ce n'est pas le bon endroit, sélectionnez donc d'abord le bon répertoire via le menu `Fichier ...`.

```
R> depot = read.table("depotSpont16.dat")                              # lire les données
R> depot16 = depot$V1;                                          # transformer en vecteur ligne
```

Lire de la même façon le fichier `depotSpont30.dat`. Tester ensuite la normalité et l'homoscédasticité des deux jeux de données.

Exercice: Résultats de ces tests ? Est-ce que les données ont une structure normale ?

Sinon, il faut utiliser un test non-paramétrique, par exemple celui de Wilcoxon-Mann-Whitney. Consulter l'aide de `wilcox.test` pour comprendre la syntaxe de cette commande.

Exercice: Selon le test de Wilcoxon-Mann-Whitney, est-ce que les deux échantillons sont significativement différents ? Notez commandes et sortie **R**.

7.6 Comment sauver vos données de R ?

Pour faire les mêmes analyses sur votre ordinateur personnel, il vous faut emporter les données. Vous pouvez copier les fichiers `*.txt` sur votre disquette. Une autre solution est de sauver les variables dans un fichier propre à **R** (qui prendra moins de place est qui sera plus rapide à lire par **R**),

```
R> save(tr,depot16,depot30,file="mesdonnees.rda")
```

et copiez/collez ce fichiers sur votre clé USB et disquette (ou vous l'envoyez par email). Vous pouvez ensuite le recharger à la maison avec la commande `load("mesdonnees.rda")`.

7.7 R et le test du χ^2

Rappelons le premier exemple du cours : on un échantillon de chauve-souris avec 44 mâles et 37 femelles, et on veut savoir si le sexe ratio est de 1. La fonction **R** qui fait ce type de test est `chisq.test`. Si vous lui donnez simplement un vecteur de fréquences il teste par défaut si les fréquences sont égales, la commande est donc

```
R> chisq.test(c(44, 37))                    # tester les sexe ratio des chauve-souris
```

Quand les fréquences attendues ne sont pas égales il faut indiquer les fréquences attendues relatives dans l'option `p=...`. A l'exemple des pois de Mendel (2 allèles dominant-récessif) on attend les phénotypes en fréquences `9:3:3:1`. La commande est donc

```
R> dat = c(152,39,53,6)                                          # lecture des données
R> predProp = c(9,3,3,1)/16                              # fréquences relatives attendues
R> chisq.test(dat,p=predProp)
```

Notez le résultat et l'interprétation.

Remarque : cette analyse peut se faire dans Rcmdr. Charger d'abord le jeu de données "phenoTypes.txt" (attention, la première ligne ne contient pas les noms des colonnes), ensuite faites l'analyse

`Statistiques -> Résumés -> Distribution de Fréquences ...`

sélectionnez la première colonne (V1) et cochez la case "Test chi-deux d'ajustement". La fenêtre suivante vous permettra d'indiquer les proportions attendues.

Si l'argument qu'on donne à `chisq.test(...)` est une matrice **R** l'interprète automatiquement comme un tableau de contingence et fait l'analyse correcte. La seule difficulté est donc de construire cette matrice. A l'exemple si la couleur des cheveux est indépendant du sexe cela donne.

```
R> sexeCouleur = matrix(rbind(c(32,43,16,9),c(55,65,64,16)),nrow=2)
R> sexeCouleur                                          # afficher le tableau pour vérifier
R> chisq.test(sexeCouleur)
```

Notez le résultat et l'interprétation. Remarque : il y a d'autres façons pour construire la matrice, donnez-en au moins une :

7.8 Entraînement

1. **Comparaison à une moyenne connue.** Nous reprenons l'exemple du cours de la température des crabes. La question est : est-elle significativement différente de 24.3 °C (température ambiante) ? Les données sont dans le fichier `crabes.txt`. Lisez-le et mettez les données dans le vecteur 'crabes'. Calculez d'abord les statistiques de base (moyenne, écart type, erreur standard) et visualisez-les sous forme d'un histogramme. Testez la normalité des données et choisissez le test de comparaison (avec 24.3 °C) adapté.

2. **L'effet d'un nouveau engrais sur la croissance des plantes** Dans l'exemple suivant, on a mesuré la hauteur de plantes cultivées avec un engrais habituel et la hauteur de plantes cultivées avec un nouveau super engrais.

```
R> engraisActuel = c(48.2, 54.6, 58.3, 47.8, 51.4, 52.0, 55.2,
+  49.1, 49.9, 52.6);                                                    #
R> engraisNeuf = c(52.3, 57.4, 55.6, 53.2, 61.3, 58.0, 59.8, 54.8);
```

   On veut tester si le super engrais fait pousser plus haut : c'est donc un test unilatéral. Consultez l'aide pour savoir comment dire ça à t.test(). Après avoir vérifié les hypothèses de base, est-ce que le super engrais est vraiment super ? Donnez la commande que vous avez utilisée et l'interprétation du résultat.

3. **Comparaison entre traffic de fourmis sur un pont de 10mm et sur un pont de 3mm.** Les jeux de données suivants représentent des mesures de flux de fourmis sur un pont de largeur 10mm ou de largeur 3mm. On veut tester s'ils sont significativement différents.

```
R> pont10 = c(54.1, 88.1, 140.25, 88.05, 63.25, 72.0, 56.65, 41.85,
+ 123.15, 82.85, 70.15, 47.3, 58.7, 97.25, 60.6);                       #
R> pont6 = c(51.4, 69.6, 122.8, 76.75, 76.25, 105.8, 71.8, 78.6,
+ 54.5, 43.5, 51.25, 74.85, 33.7, 66.15, 51.8);                         #
```

   Testez la normalité et l'hétéroscédasticité de ces deux jeux de données. Est-ce que vous avez le droit d'appliquer le test-t ? La syntaxe du test-t sera `t.test(pont6,pont10,var.equal=TRUE)`. Qu'est-ce qu'il vous donne comme information/résultat ?

4. **Préparation d'un jeu de données pour analyse dans Rcmdr.** Vous voulez faire la comparaison des temps de dépôts à deux températures différentes dans Rcmdr. Pour cela lisez d'abord les jeu de données correspondant dans un tableur (par exemple Excel) et réarrangez-les en forme d'un `data.frame` (c'est-à-dire une première colonne contenant les mesures, une deuxième colonne contenant le facteur température). Enregistrez ensuite ce tableau sous format text (avec séparateur tabulation), lisez-le dans Rcmdr et faites la comparaison non-paramétrique entre les deux échantillons.

7.9 Informations supplémentaires

Remarque : au lieu d'utiliser un test non-paramétrique, on essaye souvent de transformer les données pour les rendre normales (et pouvoir ainsi appliquer un `t.test()` sur les données transformées). Les transformations habituelles sont : $\log(X+1)$ (données lognormales, par exemple estimations d'effectifs), $\sqrt{X+0.5}$ (données Poissonniennes) ou $\arcsin(\sqrt{X})$ (pour des proportions, ne s'applique donc pas à nos données). Essayez si une de ces transformations rend nos données normales.

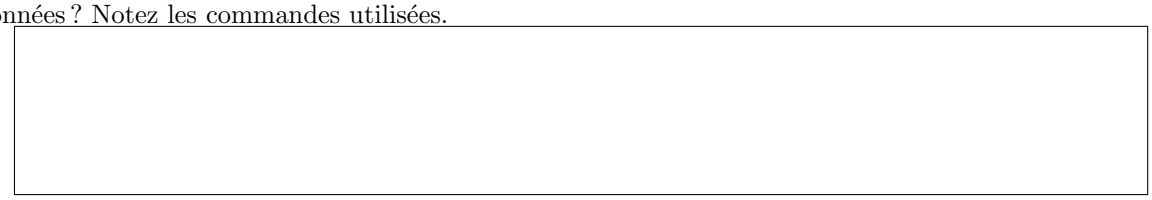
Chapitre 8

Estimation, régression linéaire et le bootstrap

Un rôle important des méthodes statistiques est l'estimation des paramètres d'un système dynamique. Dans ce chapitre on va explorer ce rôle à l'aide de l'exemple du taux de départ d'un compartiment (voir les exemples du chapitre 4 du polycopié modélisation). En particulier, le compartiment en question est l'action de transporter le cadavre d'un congénère (on suppose que cela se fait pour des raisons d'hygiène, il faut éloigner tous les déchets du nid et les agréger quelque part pour diminuer le risque de les rencontrer au hasard). Une fourmi rejoint ce compartiment en prenant un cadavre, et elle le quitte en déposant ce cadavre. C'est cette seconde partie, dépôt de cadavre, qu'on veut modéliser/estimer.

8.1 Les courbes de survie et les modèles à compartiments

Le fichier `depotSpont16.txt` contient les mesures du temps qu'une fourmi *Messor sanctus* passe à transporter un cadavre avant de le déposer. Lisez ces données (`read.table`, qui rend un data.frame), appelez ce data.frame `depot` et faites-en un histogramme. Combien de mesures y a-t-il dans ce jeu de données ? Notez les commandes utilisées.



Vous voyez sur l'histogramme qu'il y a beaucoup de petits temps et quelques temps très long, mais on ne voit pas très bien la structure détaillée de ces durées.

La représentation graphique de durées d'évènements a été bien étudiée dans le contexte des études cliniques (temps avant guérison, temps avant rechute, temps avant la mort) et un des outils qui a émergé est l'analyse de survie. Dans une telle analyse chaque temps de transport représente un évènement. Dans une courbe de survie on les fait tous commencer au même moment et ensuite en trace le nombre d'évènements qui durent encore au cours du temps. Dans notre exemple on trace le nombre de fourmis qui transportent encore au cours de temps, et à chaque fois qu'une fourmi dépose son cadavre ce nombre est diminué par un :

```
tempsDepot  nombreQuiTransporte
9           71
10          70
11          69
12          68
12          67
13          66
13          65
15          64
...         ...
```

Utilisez les commandes `sort` et `seq` pour créer ces deux vecteurs colonne. Faites ensuite la courbe de survie avec la commande `plot`. Notez commandes et résultat graphique.

8.2 Estimation d'un taux de départ d'un compartiment

La courbe de survie, par sa définition, décroit de façon monotone. Le modèle générale décrivant cette décroissance est

$$\frac{dN(t)}{dt} = \lambda(N(t), t)N(t)$$

où $N(t)$ est le nombre de fourmis qui transportent encore et $\lambda(N(t), t)$ est le taux de décroissement qui peut dépendre aussi bien de $N(t)$ comme de t. Le but est d'estimer ce $\lambda(N(t), t)$ à partir des données mesurées.

Retracez d'abord la courbe de survie en mettant l'ordonnée en échelle logarithmique (on pourrait utiliser l'option `log='y'` dans `plot`, mais procédez ici en transformant les valeurs de l'ordonnée manuellement en log et en refaisant le `plot` de façon classique). Notez la commande et commentez le résultat :

Le fait qu'en échelle log-linéaire la courbe de survie devienne quasiment une droite veut dire que $\lambda(N(t), t)$ ne dépend pas de N et ne change pas au cours du temps, $\lambda(N(t), t) = \lambda$ (constant), et qu'il correspond à la valeur absolue de la pente de cette droite. Il suffit donc d'estimer cette pente. La commande dans **R** pour faire cela est `lm(ordonnee ~ abscisse)`. Utilisez cette commande avec vos vecteurs créés ci-dessus, stockez le résultat dans la variable `ResReg` et affichez son contenu (notez les commandes). La commande

```
R> abline(ResReg)
```

tracera la droite de régression dans la courbe de survie (log-linéaire). Sachant que les temps ont été mesurés en secondes, quelle est l'unité de λ ?

Cette commande fait ce qu'on appelle une régression linéaire qui ajuste une droite aux points mesurés de façon à ce que les distances verticales entre points et droite, mises au carré, soient minimales (faites un dessin de cela : est-ce que la régression linéaire $y \sim x$ donne le même résultat que $x \sim y$?). Les coefficients affichés sont l'ordonnée à $x = 0$ (l'ordonnée à l'origine ou intercept) et la pente (qui est négative dans notre cas).

Pour la régression linéaire il existe des formules pour calculer l'erreur standard de ces deux paramètres estimés :

```
R> summary(ResReg)
```

Notez la valeur et l'erreur standard des deux paramètres.

Pour information : les `t value` et `Pr` (p-value) correspondent aux tests statistiques de l'hypothèse nulle que l'intercept et la pente soient égales à 0.

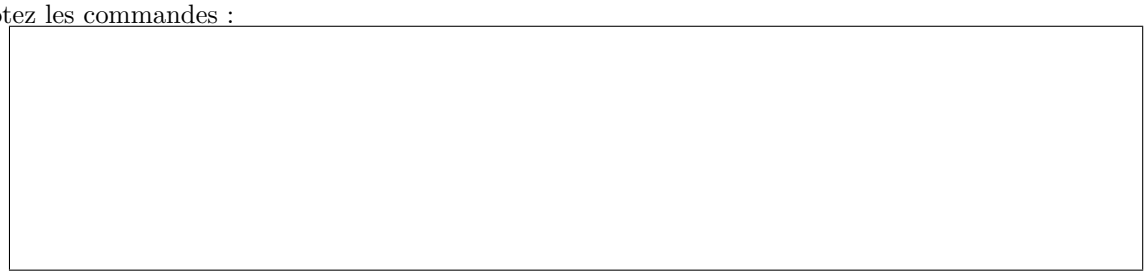
Cependant, ces formules exigent que les valeurs de l'abscisse et de l'ordonnée ont été mesurées indépendemment, ce qui n'est pas le cas dans les courbes de survie où on construit l'ordonnée à partir des valeurs de l'abscisse. Il nous faut donc une autre méthode pour estimer l'erreur standard de la pente (taux de déposer le cadavre).

8.3 Le boostrap : une méthode générale pour estimer l'erreur standard

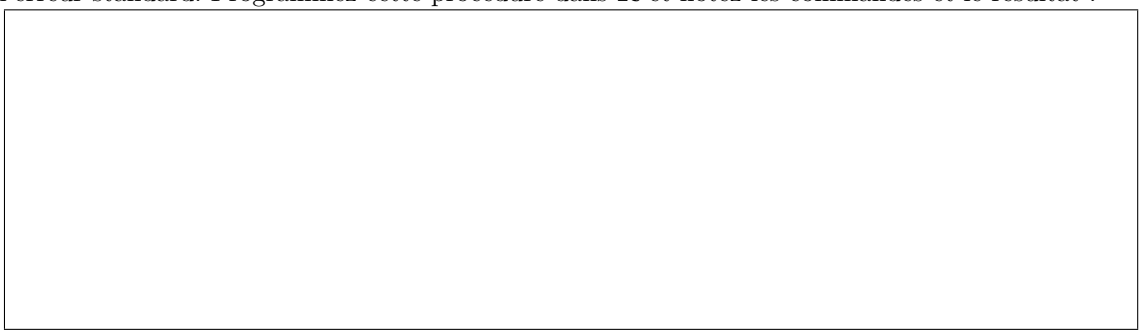
Un jeu de données bootstrap est un vecteur qui contient autant de valeurs que le jeu de données original et qu'on a tiré au hasard là-dedans (avec remise). Par exemple,

```
R> sample(c(2,5,1,3),replace=T)
```

tire au hasard avec remise 4 valeurs du jeu de données `(2,5,1,3)` (répétez la commande plusieurs fois : comprenez-vous ce qu'elle fait ?). Créez un jeu de données bootstrap des temps avant dépôt et estimez le taux de dépot associé (attention, il faut recalculer les vecteurs de l'abscisse et de l'ordonnée associée). Notez les commandes :



Pour estimer l'erreur standard par la méthode du bootstrap il suffit de répéter cette estimation du taux de dépôt pour un jeu de données boostrap environ 300 fois, de stocker les pentes (ou n'importe quel autre paramètre statistique dont on veut estimer l'erreur standard) dans un vecteur et de calculer ensuite leur écart type (`sd(...)`). Le résultat fondamental du bootstrap est que cet écart type est une estimation de l'erreur standard. Programmez cette procédure dans **R** et notez les commandes et le résultat :



Comparez cette erreur standard avec celle de la régression linéaire.

8.4 Entrainement

Le fichier `depotSpont30.txt` contient le temps avant dépôt de cadavres des fourmis de la même espèce mais à une température de 30°C. Estimez le taux de dépôt et son erreur standard.

Chapitre 9

Les analyses de la variance (ANOVA)

Les ANOVA (ou analyses de la variance) généralisent le test t de Student aux cas où il y a plus de 2 moyennes à comparer (au lieu de 2 dans le t-test). Dans ce cas là, il n'est pas légitime d'effectuer plusieurs tests t (par exemple pour 3 moyennes A, B et C, comparer la moyenne A à la moyenne B, puis la B à la C et la A à la C) car les comparaisons ne sont plus indépendantes entre elle (A vs B n'est pas indépendant de A vs C, car A est impliqué dans les deux comparaisons). Lorsqu'on effectue plusieurs comparaisons non indépendantes, l'erreur de type I augmente (la probabilité de rejeter H0 à tort, c'est à dire le risque α, est en réalité plus importante que le 5% que l'on fixe ordinairement). L'ANOVA permet de contourner ce problème en faisant un test global. En situation non paramétrique (c'est à dire sur des données quantitatives absolument pas gaussiennes, non transformables et hétéroscédastiques) il existe un équivalent à l'ANOVA : le test de Kruskal-Wallis.

Pendant le TP, vous devrez remplir toutes les cases dans ce poly et nous le rendre à la fin de la séance. Pour chaque test d'hypothèse veuillez notez en particulier l'hypothèse 0, la valeur de p que vous rend le logiciel et l'interprétation. Tout comme le test de Student, l'ANOVA suppose la normalité de vos échantillons (testée avec la fonction `shapiro.test(...)`). En théorie, il faut également que les variances soient égales (`bartlett.test(...)`). N'oubliez pas lors de ce TP de tester ces hypothèses et de mentionner le résultat dans les cases. A l'avenir, n'oubliez pas de les faire lorsque vous travaillerez avec vos propres données !

Ouvrez le fichier `tp-anova.rda` par la commande `load(...)` (que vous connaissez déjà). Vérifiez par la commande `ls()` si vous disposez bien des objets `engrais, moutons, sourisA` et `sourisB`.

9.1 ANOVA à un facteur

Avant de commencer, veuillez noter que les données sont organisées par variable : ce n'est pas intuitif (vous ne noteriez pas vos données comme ça sous Excel) mais les logiciels de statistique fonctionnent de cette façon. Ainsi, le jeux de données `engrais` contient 2 colonnes, une première `rend` qui contient le rendement des parcelles, et une colonne `fact` (facteur) qui indique l'engrais utilisé (pour **R**, le premier est du type 'numeric' et le deuxième du type 'factor', on reviendra sur cette subtilité). On va d'abord visualiser ces données :

```
R> plot(engrais$fact,engrais$rend)
```

Vous noterez que **R** a reconnu que les données de l'abscisse sont du type facteur et dessine des boîtes à moustaches.

Pour commencer, nous allons chercher à savoir si le rendement `rend` varie en fonction de l'engrais `fact`. Il y a 4 engrais, donc 4 groupes à comparer : c'est bien une situation d'ANOVA à 1 facteur. Tout d'abord, il vous faut vérifier les hypothèses de base, c'est à dire la normalité et l'égalité des variances. Les commandes sont (par exemple) :

```
R> attach(engrais)
R> shapiro.test(rend[1:5])
R> shapiro.test(rend[6:8])
R> shapiro.test(rend[9:12])
```

```
R> shapiro.test(rend[13:16])
R> bartlett.test(list(rend[1:5], rend[6:8], rend[9:12], rend[13:16]))
```
L'inconvénient de la notation par variable sous **R** est qu'il faut lui dire que le premier groupe de `rend` va des valeurs 1 à 5, puis que le deuxième va de 6 à 8, etc ...Mais vous vous souvenez surement du dernier TP : il y a la commande `split` qui simplifie la vie. Le test de Bartlett devient, par exemple,
```
R> bartlett.test(split(rend,fact))
```

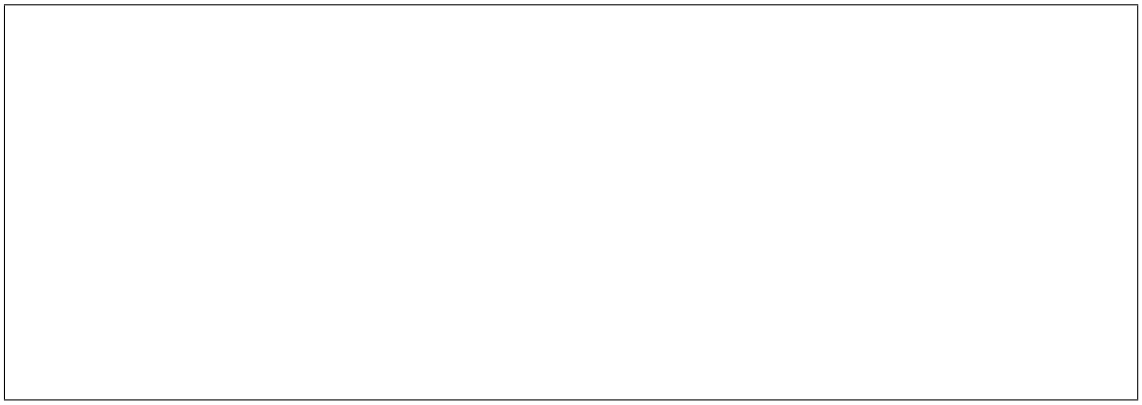
Exercice: Quel est le résultat de ces tests préliminaires (notez un exemple avec l'interprétation) ?

Nous pouvons passer maintenant aux choses sérieuses et faire notre ANOVA à l'aide de la commande suivante :
```
R> res1=aov(rend~fact)
```
Le sigle `aov` est l'abréviation de « analysis of variance ». Les ANOVA travaillent avec des formules d'un modèle statistique. Ici, la formule dit que l'on veut comparer les valeurs de `rend` lorsqu'elles sont groupées par `fact`, autrement dit on veut étudier `rend` en fonction de `fact`. Le caractère ~ (qui se lit tilde) sert juste à indiquer cette relation, « en fonction de ... ». Le résultat de `aov` n'est pas très parlant, comme vous pouvez le constater en l'affichant :
```
R> res1
```
En fait, pour que **R** donne le résultat de l'ANOVA qui est stocké (pour ne pas dire caché) dans `res1`, il faut le lui demander à l'aide de la fonction `summary` :
```
R> summary(res1)
```

Exercice: Donnez la table d'ANOVA fournie par `summary` et interprétez en quelques mots le résultat en termes biologiques.

9.2 Tests post-hoc

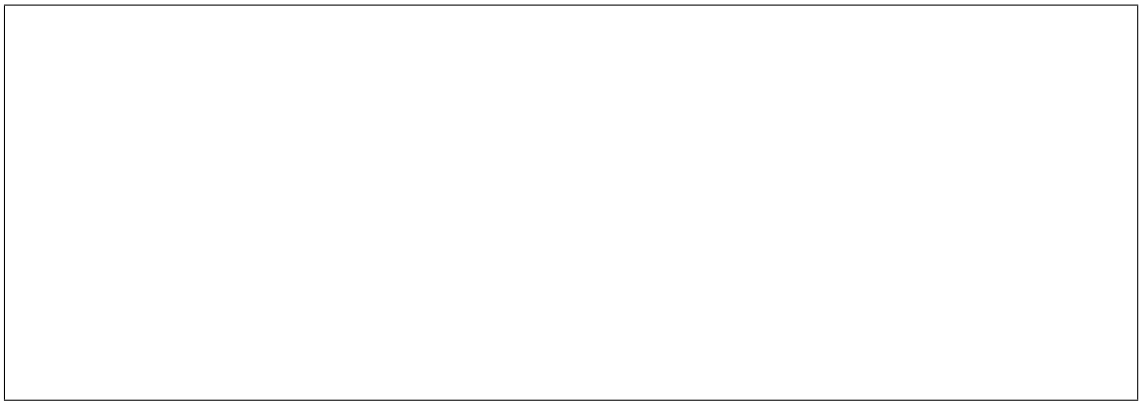
R sait faire le test « honestly significant difference » de Tukey en donnant l'intervalle de confiance, ce qui est suffisant pour conclure (si l'IC n'inclut pas 0, la différence est significative). La syntaxe est la suivante :
```
R> TukeyHSD(res1)
```
Exercice: Notez le résultat et interprétez-le en terme biologique. Représentez les données en boîtes à moustache et indiquer avec des lettres minuscules les différentes populations statistiques identifiées (commandes `text` pour ajouter du texte sur le graphique et `locator(1)` pour identifier les coordonnées d'un point sur le graphique en cliquant dessus).



Vous pouvez directement visualisez le résultat du test de Tukey en le donnant à la fonction plot,

```
R> plot(TukeyHSD(res1))
R> detach(engrais)                    # annuler l'acces directe aux colonnes de engrais
```

Exercice: Ces analyses peuvent aussi être faites dans R-commander, l'interface graphique de **R**. Démarrez-le (`library(Rcmdr)`), sélectionnez le jeu de données `engrais` et faites la même analyse : homoscédasticité, ANOVA et post-hoc. Notez les menus sélectionnés.



9.3 ANOVA à deux facteurs

Le jeux de données `moutons` contient les poids de moutons mérinos d'Arles. Un premier lot de mâles et de femelles a été mesuré en hiver, un second en été. Regardez-les avec la commande

```
R> moutons
```

La commande

```
R> names(moutons)
```

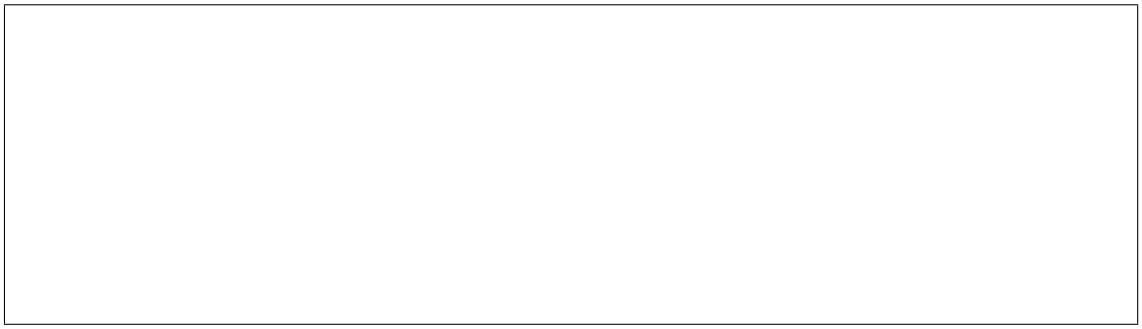
vous indique les noms des colonnes de `moutons` ; `poids`, `sexe` et `saison`. Pour simplifier la suite on va rendre les colonnes directement accessible,

```
R> attach(moutons)
```

Comme d'habitude, il faut avant toute autre analyse vérifier la normalité des données et l'homoscédasticité. Encore, la commande `split` nous aide à extraire le poids des mâles en été, des femelles en été, ... Faites la commande suivante et expliquez ce qu'elle fait. Tester ensuite la normalité et l'hétéroscédasticité.

```
R> split(poids,list(sexe,saison))
```

et représentez les données en boîte à moustache.



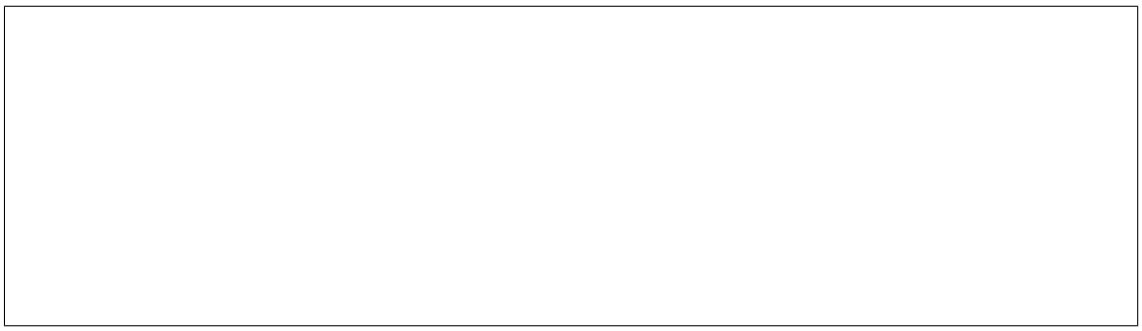
On veut maintenant tester l'effet des deux facteurs `sexe` et `saison` et leur interaction. La commande est

```
R> aov(poids ~ sexe+saison+sexe:saison) -> res2
```

(par le `+` on indique dans le modèle statistique les différents facteurs qu'on veut tester, le signe `:` indiquer « interaction » entre les deux facteurs). Afficher `res2`, pouvez-vous l'interpréter ? C'est difficile, mais de nouveau la commande `summary` fait les calculs pour nous,

```
R> summary(res2)
```

Exercice: Notez et interprétez les résultats. Y a-t-il un effet `sexe` ? Effet `saison` ? Interaction entre `sexe` et `saison` ?



Exercice: Faites la même analyse dans R-commander, notez les menus sélectionné

9.4 ANOVA avec mesures répétées (ANOVA appariée)

Prenons maintenant les données `sourisA`. Il s'agit d'un jeu de données dans lequel 16 souris ont été traîtées avec NaCl (7) ou AP5 (9), et ensuite soumises à 4 séances d'entraînement successives. Comme il s'agit d'un jeux de données provenant d'un « neuro-chercheur » on va évoquer la robustesse et ne pas tester ni l'homoscédasticité ni la normalité. Regardons maintenant ce jeux de données,

```
R> names(sourisA)
R> sourisA
```

En fait, les mesures répétées sont traîtées comme un deuxième facteur

```
   trmt  sujet   seance  explore
1  NaCl  1       s1      65
   ...     ...       ...
7  NaCl  7       s1      83
8  AP5   8       s1      75
   ...     ...       ...
16 AP5   16       s1      74
```

```
17 NaCl  1        s2      30
   ...     ...       ...
49 NaCl  16       s4      28
   ...     ...       ...
64 AP5   16       s4      52
```

Effectivement, il y a eu en total 64 mesures, mais comment interpréter la représentation ? Explication : on va comparer les performances des animaux sur les séances `s1`, `s2`, `s3` et `s4`. On introduit donc un facteur `seance` qui contient simplement le numéro de la séance pour chaque performance. De plus, comme chaque souris est testée à chaque séance (donc 4 fois), il faut dire à **R** de quel traîtement il s'agit : le facteur `trmt` contient le traîtement des souris. Enfin, la variable que l'on teste sera `explore`, qui contient le temps que chaque souris passe à explorer un nouveau objet. Chaque souris apparaîtra donc 4 fois dans le facteur `trmt`, car on a pour chacune d'elle 4 valeurs correspondantes dans `seance` (puisque chacune est testée dans chaque séance).

La syntaxe pour faire une ANOVA à mesures répétées est maintenant :

```
R> aov(explore ~ trmt+seance+trmt:seance+Error(sujet+sujet:seance),
+ data=sourisA) -> resA                                                    #
```

Une fois l'ANOVA calculée, il faut faire comme d'habitude un `summary` pour avoir le résultat. Quelques précisions sur la syntaxe : `trmt` et `seance` sont les deux facteurs étudiés, `trmt:seance` représente l'interaction entre ces deux facteurs. La sous-fonction `Error` est utilisée dans le calcul pour faire comprendre à **R** qu'on est en situation appariée, mieux vaut la traiter comme une "boîte noire".

Exercice: Faites la même analyse pour sourisB. Quel est le résultat? Donnez aussi la commande.

9.5 Entrainement

Le jeu de données `plasmaData.rda`[1] contient les concentrations de Ca^{++} dans le plasma (colonne `plasma`) en fonction d'un traitement hormonal (colonne `treat`, NH = pas de traitement, HT = traitement avec l'hormone) et du sexe des oiseaux étudiés (colonne `sexe`). Testez s'il y a un effet du traitement hormonal, du sexe ou une interaction entre ces deux facteurs.

9.6 Informations supplémentaires

Comment préparer vos données pour les analyser dans **R** ?

9.6.1 Pour une ANOVA à 1 facteur

Prenons l'exemple de l'engrais (4 types d'engrais, on mesure le rendement sur plusieurs réplications). Vos données sont probablement saisies dans un tableur (genre excel) avec une colonne contenant les rendements pour un type d'engrais, donc 4 colonnes:

```
eng1     eng2     eng3     eng4
45       31       32       39
42       34       33       38
39       40       38       41
41                33       44
48
```

1. Ce jeu de données est extrait de l'exemple 12.1 dans Zar (1999).

et **R** en a besoin sous la forme forme de deux vecteurs, le premier (rend) contenant tous les rendement enchaînés,
`45 42 39 41 48 31 34 40 32 33 38 33 39 38 41 44`
et le second (facteur) contenant les facteurs associés,
`A A A A A B B B C C C C D D D D`
pour faire une ANOVA avec la commande
`R> summary(aov(rend ~ as.factor(facteur)))`
Vous avez deux solutions pour préparer ces vecteurs à partir de votre tableau de données: a) préparer les données dans votre tableur, les enregistrer dans un fichier text et les lire ensuite dans **R**, et b) lire le tableau dans **R** et créer les vecteurs en utilisant les commandes de **R**. a) est plus intuitif, mais b) vous donnera plus de souplesse pour travailler dans **R**.

Remarque: le choix de lettres (A, B, C, D) au lieu de chiffres est exprès, ceci indique à **R** qu'il s'agit d'un facteur de type catégorique (type d'engrais) et ne pas d'une variable continue. Si vous désignez les modalités d'un facteurs par des chiffres un logiciel statistique pourrais les interpréter comme des valeurs continues avec l'ordre associé et faire une régression au lieu d'une ANOVA.

Préparer les vecteurs dans le tableur

1. En copiant/collant dans votre tableur créer le tableau suivant:

   ```
   rend fact
   45    A
   42    A
   39    A
   41    A
   48    A
   31    B
   34    B
   40    B
   32    C
   33    C
   38    C
   33    C
   39    D
   38    D
   41    D
   44    D
   ```

   que vous enregistrez ensuite en format 'text' en choisissant comme séparateur entre chiffres le tabulateur. Disons que votre fichier s'appelle `engraisR.txt`.

2. dans **R** lisez ces données avec la commande
   ```
   R> read.table("engraisR.txt",header=T,dec=",") -> don
   R> don
   R> names(don)                    # vérifier que les noms des colonnes sont bien lus
   ```

3. analysez avec la commande
   `R> summary(aov(rend ~ fact, data = don))`
   Remarque: l'option `dec=","` est nécessaire parce que excel version française enregistre les chiffres avec des virgules, tandis que **R** attend le standard international qui est le point.

Préparer les données dans R

1. Créer le vecteur des données avec la commande
   ```
   R> rend = c(45,42,39,41,48,31,43,40,32,33,38,33,39,38,41,44)
   R> rend
   ```
   et le vecteur des traitement en utilisant la fonction `rep` (répéter)
   `R> fact = c(rep("A",5),rep("B",3),rep("C",4),rep("D",4)); fact`

2. et maintenant analysez par la commande
   `R> summary(aov(rend ~ factor(fact)))`

Remarque: `factor` dit à **R** que le vecteur `fact` contient le facteur dont on veut étudier l'effet (sinon `aov` risque de faire une régression simple). Vous auriez déjà pu créer `fact` tel quel par la commande
`R> fact = factor(c(rep(1,5),rep(2,3),rep(3,4),rep(4,4))); fact`
(vous avez vu le changement dans l'affichage du vecteur?).

Pour une ANOVA à 2 facteurs il suffit de créer un vecteur `fact2` contenant les modalités du deuxième facteur associés au vecteur des donnés. Vous faites ensuite l'analyse comme décrit dans les chapitres précédents.

Autre remarque: vous pouvez organiser les données dans un `data.frame` (comme je vous l'avais fournis au début du TP dans le fichier `tp-anova.rda`) via la commande

```
R> data.frame(rend=rend,fact=as.factor(fact)) -> engrais                # créer engrais
R> engrais
R> is.factor(engrais$fact); is.factor(engrais$rend);
R> is.numeric(engrais$rend);                                          # vérifier les formats
```

9.6.2 Pour une ANOVA à mesures répétées

Supposons que vous avez saisis les données dans votre tableur sous la forme suivante

traitement	s1	s2	s3	s4
NaCl	65	30	19	28
NaCl	143	68	53	21
NaCl	61	69	59	50
NaCl	139	79	32	22
NaCl	41	56	30	48
NaCl	119	152	79	20
NaCl	83	72	20	24
AP5	75	86	75	100
AP5	82	65	49	75
AP5	45	58	58	65
AP5	59	45	89	98
AP5	78	100	73	65
AP5	65	89	76	67
AP5	89	75	45	78
AP5	45	72	26	56
AP5	74	65	98	52

mais **R** en a besoin dans la forme donnée sur la page 47.

Vous avez toujours les deux possiblités, créer ce format directement dans votre tableur, ou vous le faites dans **R**. Je ne détaillerai que la seconde variante (la première étant assez triviale).

1. Enregistrez votre tableau sous format 'text' en choisissant comme séparateur entre chiffres le tabulateur. Disons que votre fichiers s'appelle `sourisA.txt`.
2. Lire ce fichier dans **R** avec la commande
   ```
   R> read.table("sourisA.txt",header=T) -> don
   R> don
   R> names(don)                          # vérifier les noms des colonnes sont bien lus
   ```
3. créer les vecteurs `trmt`, `sujet`, `seance` et `explore` avec les commandes
   ```
   R> trmt = as.factor(rep(c(rep("NaCl",7),rep("AP5",9)),4)); trmt
   R> sujet = as.factor(rep(seq(1,16,1),4))
   R> seance = as.factor(c(rep("s1",16),rep("s2",16),rep("s3",16),(rep("s4",16))));
   R> explore = c(don$s1, don$s2, don$s3, don$s4); explore
   ```

Attention, si vous utilisez pour les facteurs des noms plus "français", ne mettez **pas** d'accents: **R** ne les interprète pas correctement et vous affichera des erreurs.

Et, bien sur, vous pouvez les organiser dans un `data.frame` par la commande suivante:
`R> data.frame(trmt=trmt,sujet=sujet,seance=seance,explore=explore) -> sourisA`
(remarque: dans `trmt=trmt`, le premier indique le nom de la colonne dans `sourisA` et le second le nom du vecteur contenant les données. Si vous voulez changez les noms des colonnes dans `sourisA` il faut changer le premier, `trait = trmt`.

Bibliography

May, R. M. (1975). Biological populations obeying difference equations: stable points, stable cycles, and chaos. *Journal of Theoretical Biology*, **51**, 511–524.

R Development Core Team (2010). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Zar, J. H. (1999). *Biostatistical analysis*. Prentice Hall, New Jersey, 4 edition.