Les fonctions

Thibault LAURENT

27 Octobre 2014

Ce document a été généré directement depuis RStudio en utilisant l'outil Markdown. La version .pdf se trouve ici.

Résumé

Après avoir présenté les principes des fonctions, nous nous arrêterons un moment sur les structures de contrôle (**if/else**, **for**, **while**, **repeat**). Les structures de contrôle ne sont pas propres aux fonctions (elles peuvent s'utiliser en-dehors des fonctions), mais elles font partie des bases de la programmation. Nous verrons également plusieurs outils spécifiques au langage R, qui seront très utiles pour programmer ses propres fonctions.

1. Le principe des fonctions

A l'instar des macros SAS, les fonctions R contiennent des instructions qui sont exécutés les unes à la suite des autres à partir d'arguments fournis par l'utilisateur. Une fois les instructions exécutées, elles renvoient généralement un résultat qui peut prendre la forme de graphiques ou de calculs.

1.1. Les fonctions prédéfinies

Jusqu'à présent, nous avons utilisé des fonctions prédéfines dans R. Certaines de ces fonctions permettent de créer des objets (*c()*, *rep()*, *seq()*, *factor()*, etc.), de faire des calculs statistiques (*sum()*, *mean()*, *var()*, *quantile()*, etc.), d'importer des données (*read.table()*, *read.csv2()*, etc.). Nous verrons tout au long du cours encore beaucoup d'autres fonctions, prédéfinies ou bien incluses dans des packages R, qui nous permettront d'utiliser des outils statistiques sophistiqués. La classe de ces objets particuliers est :

```
class(var)
```

```
## [1] "function"
```

La spécificité de ces objets est leur structure :

```
str(var)
```

```
## function (x, y = NULL, na.rm = FALSE, use)
```

Le résultat ci-dessus montre que la fonction *var()* a un argument obligatoire **x** (on voit que c'est obligatoire car il n'est pas suivi d'une valeur par défaut) et des arguments optionnels **y**, **na.rm**, **use** qui prennent des valeurs par défaut. Pour visualiser le code de la fonction, il suffit de faire :

```
print(var)
```

ou simplement :

```
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x0000000006155c90>
## <environment: namespace:stats>
```

1.2. Créer ses fonctions

Maintenant, on va essayer de créer notre propre fonction. Tout comme les objets qu'on créé au cours d'une session, les fonctions que l'on créé ne sont pas sauvegardées à la fin d'une session. Il sera donc nécessaire, lors d'une session ultérieure, de les soumettre préalablement au logiciel. Pour cela, il suffit d'enregistrer la fonction dans un fichier portant l'extension **.R** et de soumettre ce fichier par la commande *source()* avant de l'utiliser.

Pour créer une fonction, on utilise la syntaxe générale suivante :

```
ma.fonction<-function(arg1,arg2,arg3)
{
# vérification
 all(is.numeric(arg1),is.numeric(arg2),is.numeric(arg3))
# début instructions
 a=arg+arg2
 res=a^2+arg3
# fin instructions
# Et si on veut retourner un objet en sortie, on utilise return()
return(res)
}
```

Les éléments importants dans une fonction sont :

- **ma.fonction** : nom de la fonction.
- **arg1, arg2, arg3** : les arguments d'entrée. Il peut y en avoir autant que l'on souhaite et il peut s'agir d'objets de tous types (vecteurs, matrices, jeux de données, etc).
- **vérification** : il s'agit d'une étape facultative, mais qui permet de vérifier que l'utilisateur a correctement rempli les arguments d'entrée. Ici, la fonction *all()* permet de vérifier que les 3 arguments d'entrée sont tous de type **numeric**.

- **instructions** : les instructions peuvent prendre plusieurs formes. Nous nous restreindrons dans cet E-thème à celles en langage R, mais dans de nombreuses fonctions vous trouverez des instructions de type *.Internal()*, *.Call()* ou *UseMethod()* qui correspondent à des appels de fonctions écrites dans un autre langage (Fortran, C, etc.).
- **res** : une fonction ne peut "renvoyer" qu'un seul objet. C'est pourquoi lorsqu'on a plusieurs informations à retourner, on pourra utiliser des objets de classe **list**. Une autre solution pour les utilisateurs avertis, sera de définir et créer sa propre classe d'objets, mais ce n'est pas quelque chose que nous aborderons dans ce cours. La fonction *return()* n'est pas obligatoire, mais elle permet de marquer clairement la fin d'une fonction car en plus de retourner l'objet mis entre parenthèses, elle arrête l'évaluation de la fonction.

Exemple : on souhaite créer une fonction appelée **rate** qui calcule le taux d'accroissement moyen annuel d'un produit dont le prix est passé de p1 à p2 entre la date t1 et la date t2. Pour cela, la fonction aura :

- en arguments d'entrée : le prix **p1** à la date **t1**, le prix **p2** à la date **t2**.
- en argument de sortie : le taux d'accroissement **r** du produit et la durée de la période d'observation **d** (qui vaut t2-t1).

Pour faire cela, on propose le code suivant :

```
rate <- function(p1, t1, p2, t2)
{
 # étape de vérification
 stopifnot(all(is.numeric(p1),is.numeric(t1),is.numeric(p2),is.numeric(t2)))
 # instructions
 duree<-(t2-t1)
 r <- (p2/p1)^(1/duree)-1
 res <- list(r = r, d = duree) # résultat sous forme de list
 }
```

Remarque : dans cet exemple, on a pas utilisé la fonction *return()*. Dans ce cas, R retourne la dernière expression qui a été évaluée, ici **res**.

Application :

- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 100 € à 500 € entre 2000 et 2010 ?
- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 200 € à 100 € entre 2000 et la fin du 1er semestre de 2012 ?

```
# application 1
(res1<-rate(100, 2000, 500, 2010))
```

```
## $r
## [1] 0.1746189
##
## $d
## [1] 10
```

```r
# application 2
(res2<-rate(200, 2000, 100, 2012.5))
```

```
## $r
## [1] -0.05394235
##
## $d
## [1] 12.5
```

Dans le premier exemple, le taux d'accroissement annuel moyen est égal à 17.5% sur une période de 10 ans. Dans le second exemple, le taux d'accroissement annuel moyen est de -5.4% sur une période de 12 ans et demi.

Le résultat est donné sous forme de **list** car à l'intérieur de la fonction, l'objet **res** est un objet de classe **list** composé de deux élément de types **numeric**.

```r
class(res1)
```

```
## [1] "list"
```

```r
str(res1)
```

```
## List of 2
##  $ r: num 0.175
##  $ d: num 10
```

Lors de la construction d'une fonction, l'utilisateur peut donner à certains arguments des valeurs par défaut en leur attribuant d'office une valeur. Dans notre exemple, on attribue à **t1** la valeur 2010 et **t2** la valeur 2014. Dans ce cas, on permute les arguments d'entrée de telle sorte que les arguments "facultatifs" soient placés après les autres arguments. Par exemple :

```r
rate <- function(p1, p2, t1=2010, t2=2014)
{
 duree=(t2-t1)
 r = (p2/p1)^(1/duree)-1
 res=list(r = r, d = duree)
 return(res)
}
```

Application :

- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 100 € à 200 € entre 2010 et 2014 ?

```r
# exemple 3
rate(100,200)
```

```
## $r
## [1] 0.1892071
##
## $d
## [1] 4
```

1.3. Variable globale VS Variable locale

Dans la fonction **rate** créée précedemment, les objets **p1**, **p2**, **t1**, **t2**, **duree**, **r** et **res** sont toutes des variables locales car elles ont été définies à l'intérieur de la fonction **rate** et ne peuvent être qu'utilisées à l'intérieure de celle-ci. Par exemple, la commande suivante renverra un message d'erreur car dans l'environnement global de R, l'objet **duree** n'est pas connu :

```
print(duree)
```

Maintenant, considérons la fonction suivante :

```
f<-function(x)
{return(x+a^2)}
```

On constate que l'objet **a** n'a pas été défini à l'intérieur de la fonction. R va donc aller chercher si cet objet existe à l'extérieur. Si cet objet existe en variable globale, il sera utilisé; en revanche, s'il n'existe pas, il y aura un message d'erreur. Par exemple :

```
a=3
f(5)
```

```
## [1] 14
```

Si l'objet existe à la fois en variable locale et variable globale, c'est la variable locale qui sera utilisée en priorité à l'intérieur de la fonction :

```
f<-function(x,a=2)
{return(x+a^2)}
f(5)
```

```
## [1] 9
```

2. Les structures de contrôle

Dans le cadre de l'écriture de fonctions, quelque soit le langage de programmation, on retrouvera les structures de contrôle **for**, **if/else** et **while**. On en profitera pour voir également des structures de contrôle propre au langage R.

2.1 L'expression if/else

a. Syntaxe

L'expression **if** est utilisée lorsque l'on souhaite effectuer des opérations sur des éléments ayant une (ou plusieurs) caractéristique(s) particulière(s).

La syntaxe est (veillez à bien respecter l'emplacement du **else**, juste après l'accolade fermante du **if**) :

```
if(<condition(s)>)
 {<instruction 1>
 }else
 {<instruction 2>}
```

Si la (les) condition(s) renvoie la valeur **TRUE**, l'instruction 1 sera exécutée. Dans le cas contraire, ce sera l'instruction 2 qui sera exécutée.

Remarque : le **else** n'est pas obligatoire. On peut simplement avoir :

```
if(<condition(s)>)
 {<instruction 1>}
```

b. Un premier exemple

On cherche à savoir si dans un vecteur **x** de taille $n = 25$ simulé selon une $\mathcal{N}(0, 1)$, il existe des valeurs qui en valeurs absolues sont supérieures à 1.96 :

```
x=rnorm(25) # on simule le vecteur gaussien

if(any(abs(x)>1.96))    # voir explication ci-dessous
 {
  print("Il existe au moins une valeur extrême")
 }else
  {
   print("Il n'y a pas de valeurs extrêmes")
  }
```

```
## [1] "Il existe au moins une valeur extrême"
```

Pourquoi avons-nous écrit *if(any(abs(x)>1.96))* et pas simplement *if(abs(x)>1.96)* ? Si on fait juste :

```
abs(x)>1.96
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE
```

on obtient un vecteur de booléen de la même taille que **x** avec la valeur **TRUE** pour les valeurs de **x** qui sont supérieures à 1.96 en valeur absolue et **FALSE** sinon. Or, l'expression **if** ne peut pas s'appliquer à un vecteur de booléen comme l'indique le message suivant :

```
if(abs(x)>1.96)
 {
  print("Il existe au moins une valeur extrême")
 }
```

```
## Warning in if (abs(x) > 1.96) {: the condition has length > 1 and only the
## first element will be used
```

qui signifie que lorsqu'on met un vecteur de booléen à l'intérieur d'une condition **if**, seule la première valeur du vecteur de booléen comptera.

Que fait exactement la fonction *any()* ? Elle prend comme argument d'entrée un vecteur de booléen et renvoit la valeur **TRUE** si la valeur **TRUE** apparaît au moins une fois dans le vecteur de booléen.

```r
any(abs(x)>1.96)
```

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

c. Vérifier deux conditions en utilisant et/ou

- L'opérateur **&** correspond à l'opérateur logique "et".
- L'opérateur | correspond à l'opérateur logique "ou".

Les règles lorsqu'on teste deux conditions sont les suivantes. Pour le **&**, il faut que les deux conditions soient vérifiées en même temps pour obtenir la valeur **TRUE** :

```r
a=5
b=-5
(abs(a)>4)&(abs(b)>4)  # les deux conditions sont vérifiées
```

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

```r
(a>4)&(b>4) # la 2nde condition n'est pas vérifiée
```

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

```r
(a<4)&(b<4) # la 1e condition n'est pas vérifiée
```

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

```r
(a==4)&(b==4) # aucune condition n'est vérifiée
```

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

Pour le |, il suffit qu'il y ait au moins une condition de vérifiée pour obtenir la valeur **TRUE**:

```r
(abs(a)>4)|(abs(b)>4) # les deux conditions sont vérifiées
```

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

```r
(a>4)|(b>4) # au moins une des deux conditions (la 1e) est vérifiée
```

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

```r
(a<4)|(b<4) # au moins une des deux conditions (la 2nde) est vérifiée
```

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

```r
(a==4)|(b==4) # aucune condition n'est vérifiée
```

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

d. Vérifier plusieurs conditions en utilisant et/ou

Lorsqu'on teste successivement des conditions à la suite, on effectue des tests 2 par 2, en commençant de la gauche vers la droite. Par exemple :

```r
FALSE&TRUE|TRUE
```

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

est équivalent à :

```r
(FALSE&TRUE)|TRUE
```

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

mais qui donne un résultat différent de :

```r
FALSE&(TRUE|TRUE)
```

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

Conseil : lorsqu'on teste plus de deux conditions, il est très important d'utiliser les parenthèses en fonction des priorités qu'on donne aux conditions.

e. Utiliser les fonctions *all()* et *any()*

Lorsqu'on souhaite vérifier que plusieurs conditions sont toutes égales à **TRUE**, on utilise la fonction *all()*. Par exemple, on souhaite vérifier que le vecteur **sexe** ne contient que les modalités **male** ou **female**

```r
x<-factor(c("male","male","female","male"))
all(x%in%c("male","female"))
```

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

Si au souhaite vérifier qu'il n'y aucune valeur égale à la modalité **autre**, on peut utiliser la négation avec l'opérateur **!** :

```r
!all(x%in%"autre")
```

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

On souhaite savoir s'il existe au moins une valeur égale à **autre** :

```r
any(x%in%"autre")
```

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

2.2 Les expressions for et while

Les expressions **for** et **while** permettent l'exécution répétitive d'instructions. Dans le premier cas, on connaît le nombre de fois où on va répéter les instructions. Dans le second cas, on ne sait pas nécessairement le nombre de fois, mais en revanche on va donner une condition d'arrêt. L'inconvénient est que si la condition d'arrêt n'arrive pas, le programme peut tourner en boucles sans jamais s'interrompre...

a. Syntaxe de for

```
for(<ind> in <vecteur>)
 {
  <instructions>
 }
```

ind est la variable de boucle qui va prendre successivement toutes les valeurs de **vecteur**. **vecteur** prend généralement la forme *1:n*, mais il peut aussi être un vecteur de **character**. Pour chacune de ces valeurs, les instructions seront répétées.

Exemple:

```r
for(k in 1:4)
 {
  print(k)
 }
```

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

Les boucles peuvent s'emboîter entre elles. Voici un petit exemple :

```r
for(k in letters[1:3])
 {for(j in 1:2)
  {print(paste(k,":",j))}
 }
```

```
## [1] "a : 1"
## [1] "a : 2"
## [1] "b : 1"
## [1] "b : 2"
## [1] "c : 1"
## [1] "c : 2"
```

Cet outil est utile lorsque l'on souhaite "remplir" un tableau de résultat. On peut également l'utiliser pour du calcul vectoriel ou matriciel, mais il faut savoir que lorsque les données commencent à être volumineuses, le temps de calcul peut devenir très long.

b. Application

L'objectif est de créer une fonction **simul** qui prend comme arguments d'entrée un nombre **n** et un nombre **B**. A l'intérieur de la fonction, on répète **B** fois l'opération suivante :

- simulation d'un vecteur **x** de taille **n** selon une $\mathcal{N}(0,1)$
- on vérifie si oui ou non il existe au moins un élément de **x** supérieur en valeur absolue à 1.96
- On comptabilise sur les **B** boucles le pourcentages de boucles où le phénomène s'est produit.

Une fois la fonction exécutée, on utilisera des boucles pour comparer les résultats de sortie en fonction de **n** (qui prendra les valeurs 10, 50, 100 et 200) et **B** (qui prendra les valeurs 10, 50 et 100).

Code de la fonction

```
simul<-function(n,B)
 {
  # vérification
  stopifnot(is.numeric(n),is.numeric(B))

  # initialisation
  res=0 # le phénomène s'est produit 0 fois au début

  for (b in 1:B)
   {x<-rnorm(n) # simulation d'un vecteur x de taille 10
     if(any(abs(x)>1.96))
      {
       res<-res+1   # on ajoute 1 à chaque fois que la condition est vérifiée
      }
   }
   print(paste("Le phénomène s'est produit",res/B*100,"% de fois"))
 }  # fin de la fonction
```

Test sur la fonction

```
for(n in c(10,50,100,200))
 {for(B in c(10,50,100))
  {print(paste("n=",n,"et B=",B))
   simul(n,B)
  }
 }
```

```
## [1] "n= 10 et B= 10"
## [1] "Le phénomène s'est produit 20 % de fois"
## [1] "n= 10 et B= 50"
## [1] "Le phénomène s'est produit 40 % de fois"
## [1] "n= 10 et B= 100"
## [1] "Le phénomène s'est produit 44 % de fois"
## [1] "n= 50 et B= 10"
## [1] "Le phénomène s'est produit 90 % de fois"
## [1] "n= 50 et B= 50"
## [1] "Le phénomène s'est produit 92 % de fois"
## [1] "n= 50 et B= 100"
```

```
## [1] "Le phénomène s'est produit 89 % de fois"
## [1] "n= 100 et B= 10"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n= 100 et B= 50"
## [1] "Le phénomène s'est produit 94 % de fois"
## [1] "n= 100 et B= 100"
## [1] "Le phénomène s'est produit 99 % de fois"
## [1] "n= 200 et B= 10"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n= 200 et B= 50"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n= 200 et B= 100"
## [1] "Le phénomène s'est produit 100 % de fois"
```

c. Syntaxe de while

```
<initialisation paramètre(s)>
while(<Test sur paramètre(s)>)
 {
  <instructions>
  <Mise à jour paramètre(s)>
 }
```

Exemple : on veut savoir au bout de combien de tirage d'un nombre simulé selon une $\mathcal{N}(0,1)$, on dépasse la valeur 1.96

```
eps<-rnorm(1) # initialisation
k=1
while(eps<1.96)
  {eps<-rnorm(1) # mise à jour du paramètre testé
   k<-k+1}
print(k)
```

```
## [1] 32
```

d. Syntaxe alternative de while avec repeat

Dans la syntaxe précédente, d'abord on teste quelque chose et ensuite on agit. On peut vouloir d'abord agir et ensuite tester. Pour cela, on utilise la commande **repeat**.

```
<initialisation paramètre(s)>
 repeat
 {<Mise à jour paramètre(s)>
  <instructions>
  if(<Test sur paramètre(s)>)
    {break}
 }
```

Exemple : on reprend l'exemple précédent

```r
k=0  # initialisation
repeat
{eps<-rnorm(1) # mise à jour du paramètre testé
 k<-k+1
 if(eps>=1.96){break}
}
print(k)
```

```
## [1] 14
```

Remarque : l'instruction **break** utilisée après la condition **if** peut aussi être utilisée à l'intérieur d'une fonction ou à l'intérieur d'une boucle **for**/**while**. Elle permet d'interrompre à tout moment un calcul dès lors qu'un critère d'arrêt a été défini dans le **if**. Par exemple, on souhaite effectuer un calcul qui dure au plus 1 seconde. Au-delà de ce temps, on souhaite arrêter le calcul.

Exemple : on souhaite savoir au bout de combien de temps de simulation d'une $\mathcal{N}(0,1)$, on dépasse la valeur 4. En théorie, la probabilité de dépasser 4 vaut 0.000032, soit 1 fois tous les 31250 tirages. Par précaution, on souhaite arrêter le calcul dès lors que le temps dépasse 1 seconde. Analyser et répéter le code ci-dessous plusieurs fois.

```r
ptm <- proc.time()  # temps de référence
# initialisation avant la boucle
eps<-rnorm(1)
k=1
# début boucle
while(eps<4)          # 1ère condition sur la valeur seuil
{# mise à jour des paramètres
 eps<-rnorm(1)
 k=k+1
 temps.eps<-(proc.time() - ptm)[3] # mise à jour du temps de calcul
 if(temps.eps>1){break} # 2ème condition sur le temps
}
# fin boucle et affichage du résulat
if(temps.eps>1)
 {print(paste("Le calcul a été interrompu (temps dépassé) après", k, "simulations"))
  }else
  {print(paste("La valeur 4 a été atteinte au bout de :", k, "simulations en",
               temps.eps,"secondes."))}
```

```
## [1] "La valeur 4 a été atteinte au bout de : 37596 simulations en 0.49 secondes."
```

2.3 La fonction ifelse()

La syntaxe de la fonction **ifelse()** est la suivante :

```r
ifelse(test, yes, no)
```

- Le premier argument d'entrée de cette fonction est un vecteur de booléen de taille n.
- Les deuxième et troisième arguments d'entrée peuvent être des scalaires ou des vecteurs qui seront mis automatiquement à la taille n (procédé qui ce fait comme nous l'avons vu dans le chapitre sur les vecteurs).

- La fonction **ifelse()** renvoit un vecteur de taille n dont les composantes seront celles de **yes** lorsque **test** vaut **TRUE** et celles de **no** lorsque **test** vaut **FALSE**.

Exemple 1 : pour créer une variable qualitative à partir d'une variable quantitative.

```
(x<-rnorm(10))
```

```
##  [1] -0.84993579  0.84765473 -0.71018478  0.85973483 -0.83069918
##  [6]  0.34178355 -1.78437395 -0.01344621  1.09170692  0.33692102
```

```
(signe=ifelse(x>0,"+","-"))
```

```
##  [1] "-" "+" "-" "+" "-" "+" "-" "-" "+" "+"
```

Remarque : la fonction **ifelse()** ne doit pas être confondue avec l'expression **if/else**. En effet, la fonction **ifelse()** est équivalente aux commandes suivantes qui ne fait pas intervenir l'expression **if/else** :

```
signe=rep("+",length(x))
signe[!(x>0)]="-"
signe
```

```
##  [1] "-" "+" "-" "+" "-" "+" "-" "-" "+" "+"
```

Exemple 2 : pour transformer une variable quantitative.

```
ifelse(x>=0,sqrt(x),0)
```

```
## Warning in sqrt(x): NaNs produced
```

```
##  [1] 0.0000000 0.9206817 0.0000000 0.9272189 0.0000000 0.5846226 0.0000000
##  [8] 0.0000000 1.0448478 0.5804490
```

3. Trucs et astuces pour créer et utiliser ses fonctions

3.1. La fonction stopifnot()

Lorsqu'on fait des vérifications sur les paramètres d'entrée, la fonction *stopifnot()* permet de vérifier plusieurs conditions à la fois. Elle s'utilise ainsi :

```
stopifnot(<condition 1>, <condition 2>, ...)
```

qui est équivalent à l'instruction suivante :

```
if(!<condition 1> | !<condition 2> | ...) stop()
```

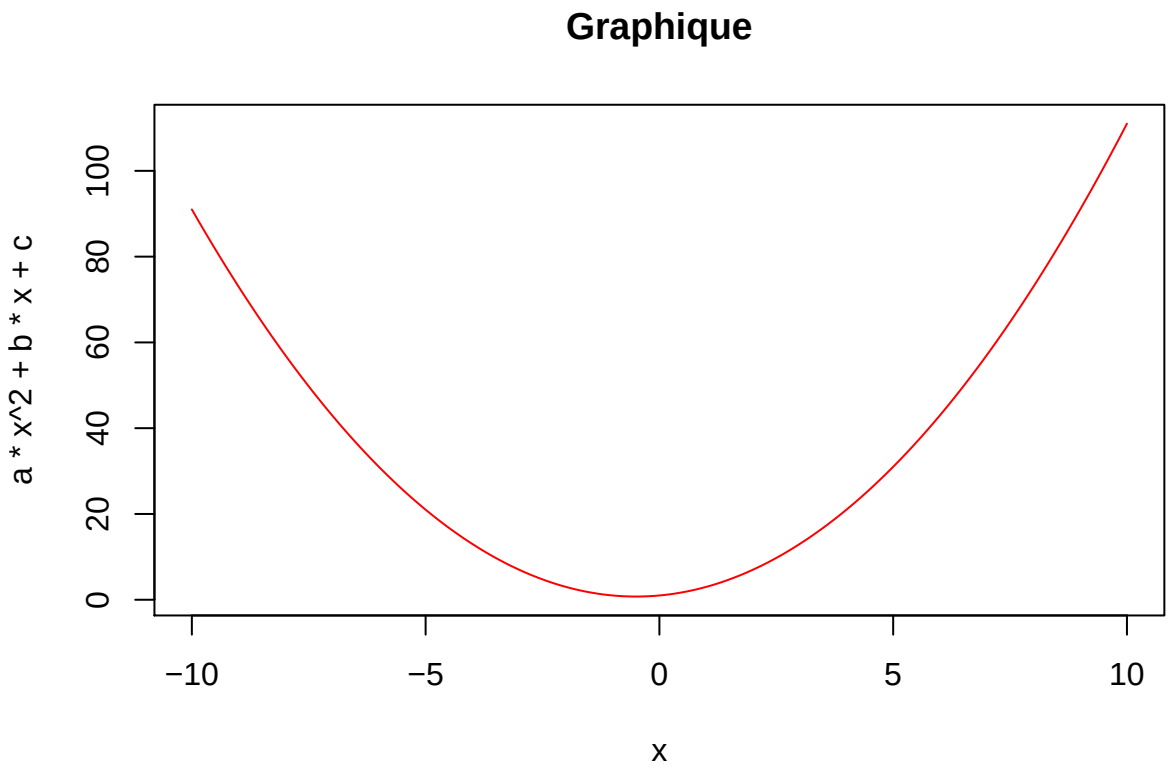
3.2. Pourquoi mettre ... comme argument d'entrée d'une fonction ?

On souhaite créer une fonction qui va appeler une fonction prédéfinie qui a une multitude d'arguments d'entrée (c'est le cas notamment des fonctions graphiques que nous verrons dans le chapitre suivant). Dans ce cas, plutôt que de reécrire tous les arguments d'entrée de la fonction qu'on va utiliser, il suffit d'ajouter parmi les arguments d'entrée de notre fonction, le symbole **...** qu'on re-utilisera au moment de l'appel de la fonction prédéfinie. Par exemple, on souhaite donner comme arguments d'entrée les paramètres **a**, **b** et **c** du polynôme $ax^2 + bx + c$ et représenter cette fonction dans un intervalle borné (argument **borne** égal par défaut à $[-10, 10]$) :

```
myFunc<-function(a, b, c, borne=c(-10,10),...)
  {x<-seq(borne[1],borne[2],0.1)
   plot(x, a*x^2+b*x+c,...)
  }
```

Pour utiliser tous les paramètres définis dans la fonction *plot()* de R, on utilise les **...** comme argument d'entrée qu'on replace lorsqu'on appelle la fonction *plot()*. Voici un exemple d'utilisation :

```
myFunc(1,1,1,main="Graphique",type="l",col="red")
```



3.3. Utiliser les fonctions *apply()*, *lapply()*, *sapply()*, *tapply()*

En terme de clarté dans le code et en temps calcul, on a beaucoup à gagner à utiliser une de ces fonctions plutôt que de faire des boucles. On a déjà vu des exemples d'utilisation de ces fonctions dans le chapitre précédent, mais comme elles sont très importantes dans le langage R, on prend le temps ici pour les appronfondir.

a. La fonction *apply()*

La syntaxe de la fonction *apply()* est la suivante :

```
apply(X, MARGIN, FUN, ...)
```

où **X** est un tableau de donnée de type **array** (typiquement un objet **matrix** ou **data.frame**), **MARGIN** est la(les) dimension(s) sur laquelle (lesquelles) on va appliquer la fonction **FUN**. Si n est le nombre de composantes de **MARGIN**, alors la fonction renvoie un vecteur de taille n.

Exemple 1 : le jeu de données **iris** est un **data.frame** qui est un objet à deux dimensions.

```
dim(iris)
```

```
## [1] 150   5
```

L'espace des individus est représenté en lignes (dimension 1 de taille 150) et l'espace des variables est représenté en colonne (dimension 2 de taille 5). Ici, on va s'intéresser uniquement aux variables quantitatives (de type **numeric**) car il est difficile d'utiliser une même fonction sur des variables quantitatives et qualitatives en même temps. C'est pourquoi on créé l'objet **iris2** qui ne contient que les variables quantitatives :

```
iris2<-iris[,1:4]
```

L'objet **iris2** a deux dimensions. On a donc 2 possibilités pour renseigner l'argument **MARGIN** : **MARGIN=1**, **MARGIN=2**. On va utiliser la fonction *mean()* comme argument **FUN**. Les résultats sont les suivants :

```
apply(iris2, 1, mean)
```

```
##   [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700
##  [12] 2.500 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
##  [23] 2.350 2.650 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725
##  [34] 2.825 2.425 2.400 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675
##  [45] 2.800 2.375 2.675 2.350 2.675 2.475 4.075 3.900 4.100 3.275 3.850
##  [56] 3.575 3.975 2.900 3.850 3.300 2.875 3.650 3.300 3.775 3.350 3.900
##  [67] 3.650 3.400 3.600 3.275 3.925 3.550 3.800 3.700 3.725 3.850 3.950
##  [78] 4.100 3.725 3.200 3.200 3.150 3.400 3.850 3.600 3.875 4.000 3.575
##  [89] 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525 3.525 3.675 2.925
## [100] 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575 4.200 4.850
## [111] 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675 4.525
## [122] 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875
## [144] 4.550 4.550 4.300 3.925 4.175 4.325 3.950
```

```
apply(iris2, 2, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Pour **MARGIN=1**, on a appliqué la fonction *mean()* sur toutes les composantes de la 1ère dimension (i.e. les 150 individus). Pour **MARGIN=2**, on a appliqué la fonction *mean()* sur toutes les composantes de la 2ème dimension (i.e. les 4 variables).

Remarque : dans d'autres logiciels de programmation, on aurait eu tendance à faire des boucles pour faire ce genre de calcul. Par exemple, la commande *apply(iris2, 2, mean)* est équivalente à ce bout de code :

```
res<-matrix(0,ncol(iris2),1)
for(k in 1:ncol(iris2))
  {res[k]<-mean(iris2[,k])}
print(res)
```

```
##          [,1]
## [1,] 5.843333
## [2,] 3.057333
## [3,] 3.758000
## [4,] 1.199333
```

On voit donc qu'on a intérêt à utiliser la fonction *apply()*, plus élégante et moins coûteuse en temps de calcul surtout lorsqu'on emboîte des boucles les unes dans les autres.

Exemple 2 : le jeu de données **iris3** (prédéfini sous R) est un objet de type **array**. Il s'agit du jeu de données **iris**, mais qui a été disposé sous la forme d'un tableau à 3 dimensions :

```
dim(iris3)
```

```
## [1] 50  4  3
```

La première dimension est la dimension des individus (il y en a 50), la seconde est la dimension des variables (il y en a 4) et la troisième dimension est la dimension des espèces (il y en a 3). Pour visualiser ce à quoi ressemble un tel objet, il faudrait imaginer un tableau sous forme d'un cube. Comme R représente difficilement la 3D, lorsqu'on affiche l'objet *print(iris3)*, R nous renvoie 3 tableaux : 1 tableau à 2 dimensions (individus (dimension 1) $\times$ variables (dimension 2)) par espèce (dimension 3). Comme l'objet a 3 dimensions, on a donc 3 possibilités pour renseigner l'argument **MARGIN** : **MARGIN=1**, **MARGIN=2** et **MARGIN=3**, auxquelles viennent s'ajouter les croisements possibles entre dimensions : **MARGIN=c(1,2)**, **MARGIN=c(1,3)** et **MARGIN=c(2,3)**. Bien entendu, parmi toutes ces combinaisons possibles, toutes ne sont pas intéressantes... Ici on souhaiterait connaître la moyenne des variables en fonction des espèces. On s'intéresse donc à la fois à la dimension des variabes (la dimension 2) et celle des espèces (la dimension 3). Pour effectuer la moyenne des variables en fonction des espèces, on fait donc :

```
apply(iris3, c(2,3), mean)
```

```
##          Setosa Versicolor Virginica
## Sepal L.  5.006      5.936     6.588
## Sepal W.  3.428      2.770     2.974
## Petal L.  1.462      4.260     5.552
## Petal W.  0.246      1.326     2.026
```

Remarque : comme la taille de (dimension 2 $\times$ dimension 3) est 4×3, la fonction *apply()* renvoie bien un objet de dimension 4×3.

b. Les fonctions *lapply()* et *sapply()*

La syntaxe de la fonction lapply() est la suivante :

```
lapply(X, FUN, ...)
```

X est un objet de type **list** (*data.frame* inclus) et renvoie un objet de classe **list** de la même taille que **X**. Contrairement à la fonction *apply()*, il n'est pas nécessaire de préciser la dimension de l'objet sur laquelle on va appliquer la fonction, puisqu'en quelque sorte il n'y qu'une seule dimension. Un *data.frame* est équivalent à une **list** dont les éléments sont les variables du **data.frame** prise une à une. C'est pourquoi

```
length(iris2)
```

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

renvoie la valeur 4. Ainsi, lorsqu'on applique la fonction *lapply()* sur un **data.frame**, on applique la fonction **FUN** sur chacune des variables. Par exemple :

```
lapply(iris2,mean)
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
```

Un objet de type **list** n'est pas nécessairement sympathique à manipuler si on veut ensuite faire du calcul. Du coup, on peut utiliser la fonction *sapply()* (il s'agit d'une version arrangée de la fonction *lapply()*) qui retourne un vecteur ou une **matrix** au lieu d'un objet **list** :

```
sapply(iris2,mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

c. Créer une fonction et la coupler avec *lapply()*

Jusqu'à présent, on a utilisé des fonction prédéfinies (comme la fonction *mean()*) à la place de l'argument **FUN**. On peut si l'on veut, utiliser une fonction programmée par nous-mêmes. Reprenons le jeu de données **iris2**. On souhaite transformer toutes les variables quantitatives en variables qualitatives. Pour cela, on écrit la fonction **f** qui prend comme argument d'entrée une variable x et retourne la variable transformée en 2 classes selon que les valeurs sont au-dessus ou au-dessous de la médiane :

```
f<-function(x)
{res<-ifelse(x>median(x),"++","--")
 return(res)
}
```

Pour appliquer cette fonction à chaque variable de **iris2**, il nous reste plus qu'à appliquer la fonction **sapply()** à l'objet **iris2** en précisant l'argument **FUN=f**.

```
iris2b<-sapply(iris2,f)
head(iris2b)
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## [1,] "--"         "++"        "--"         "--"
## [2,] "--"         "--"        "--"         "--"
## [3,] "--"         "++"        "--"         "--"
## [4,] "--"         "++"        "--"         "--"
## [5,] "--"         "++"        "--"         "--"
## [6,] "--"         "++"        "--"         "--"
```

d. La fonction *tapply()*

La syntaxe de la fonction est la suivante :

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Cette fois-ci, **X** est un vecteur et **INDEX** est une **list** de **factor**.

Exemple : on souhaite calculer la moyenne par espèce de la variable **Sepal.Length** du jeu de données **iris**.

```
tapply(iris$Sepal.Length, list(iris$Species), mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Remarque : comme a l'a vu au chapitre précédent, pour calculer la moyenne par espèce de toutes les variables quantitatives du jeu de données **iris**, il suffit d'utiliser la fonction *sapply()* et prendre comme argument **FUN**, la fonction *tapply()* telle qu'utilisée ci-dessus. On a vu que la fonction *tapply()* a 3 paramètres d'entrée. Dans le **FUN** de *sapply()*, on est pas obligé de préciser le 1er argument qui correspond à la variable renvoyée par la fonction *sapply()*. Les deux autres arguments doivent être précisés après le **FUN**. Voici le résultat :

```
sapply(iris[,1:4], tapply, list(iris$Species), mean)
```

```
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

3.4. Utiliser la fonctions *traceback()* pour debugger une fonction

L'exemple que nous allons prendre est tiré de l'aide la fonction *traceback()*. On créé ici deux fonctions dont la première fait appel à la seconde :

```
# Première fonction
foo <- function(x)
  {print(1); bar(2)}
# Deuxième fonction
bar <- function(x)
  {x + a.variable.which.does.not.exist}
```

Si on fait :

```
foo(2)
```

on obtient un message d'erreur. La fonction *traceback()* permet de retracer l'historique des différentes fonctions appelées qui ont conduit au dernier message d'erreur.

```
traceback()
```

Ici, on a d'abord appellé la fonction *foo()* (étape 1), puis à l'intérieur de celle-ci, on a appelé la fonction *bar()* et il y a eu un bug au niveau de la deuxième ligne.

4. Exercices

On considère ici la table crimin.txt qui décrit pour chaque département de la France métropolitaine le taux de criminalité (variable *crim*), la proportion de cadres supérieurs dans la population active (variable *cadr*) ainsi que la zone dans laquelle se situe le département (variable *region* à 5 modalités : **NO**, **NE**, **C**, **SE** et **SO**). L'exercice consiste à programmer deux fonctions réalisant respectivement l'analyse univarié d'une variable dans chaque région (fonction *univarie()*) et l'étude de la liaison entre les 2 variables, toujours dans chaque région (fonction *bivarie()*). Le jeu de données pourra s'importer sur R avec la fonction *read.table()*.

```
crimin<-read.table("Ressource/crimin.txt")
```

4.1 Fonction *univarie()*

La fonction *univarie()* aura pour arguments d'entrée :

- la table de données **table**,
- le nom de la variable étudiée **var**,
- le nom de la variable catégorielle **facteur**.

A partir de ces 3 arguments, retourner un **data.frame** contenant le nom des modalités de **facteur** en ligne et l'effectif, la moyenne, la médiane, l'écart-type, le minimum et le maximum de la variable **var** en colonnes. En outre, elle devra réaliser le graphique des boîtes à moustaches parallèles.

Indications

Un algorithme pour la construction de cette fonction est :

1. Vérifier que **var** et **facteur** sont bien des variables de **table**.
2. Vérifier que la variable catégorielle est bien de type **factor**
3. Préparer la table de résultats. On pourra utiliser la fonction *tapply()*
4. Effectuer le graphique
5. Faire "sortir" les résultats. Pour information, les boites à moustaches parallèles se font avec la fonction *boxplot()*. On pourra utiliser les **...** pour utiliser les paramètres optionnels de *boxplot()*

Solution

On propose la solution suivante pour répondre à l'exercice. Bien évidemment, il existe d'autres façons pour y arriver, mais le but est d'essayer d'avoir un code aussi clair et simple que possible. Remarque : on a utilisé les fonctions *stopifnot()* et *tapply()* plutôt que les structures (**if/else** et **for**).

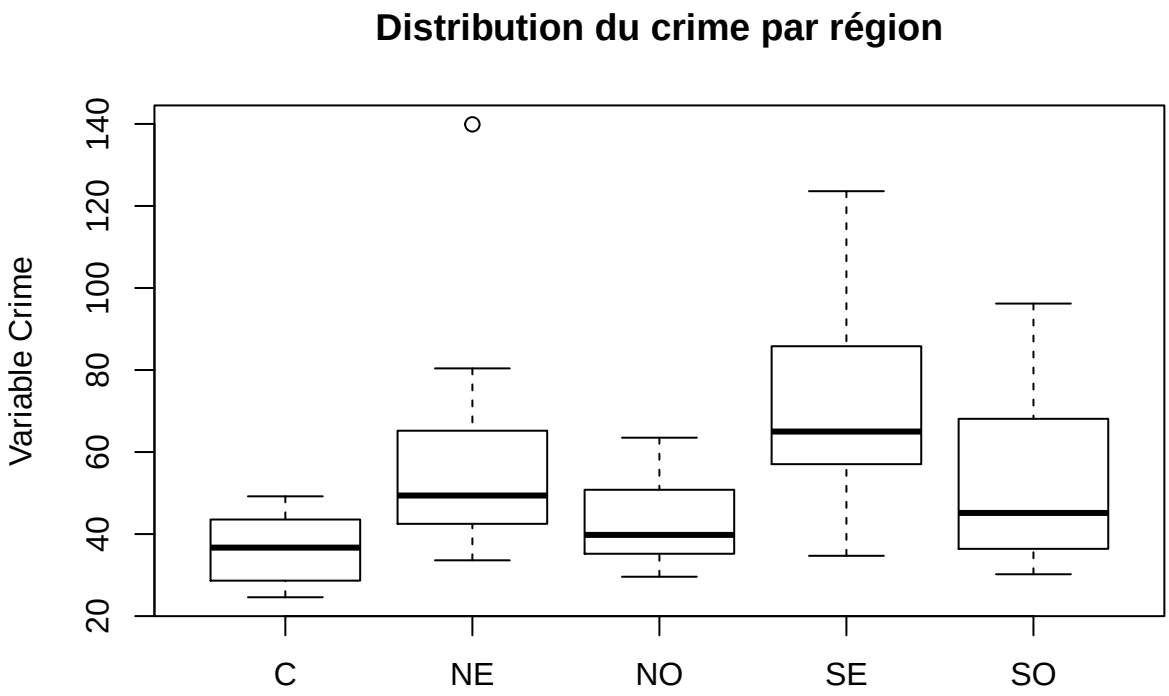
```
univarie<-function(table, var, facteur, ...)
{
 # Vérification 1
  stopifnot(var%in%colnames(table), facteur%in%colnames(table))

 # Vérification 2
  stopifnot(is.factor(table[,facteur]))

 # Création du data.frame
 resultat<-data.frame(n=tapply(table[,var],table[,facteur],length),
            moyenne=tapply(table[,var],table[,facteur],mean),
            mediane=tapply(table[,var],table[,facteur],median),
            ecart.type=tapply(table[,var],table[,facteur],sd),
            minimum=tapply(table[,var],table[,facteur],min),
            maximum=tapply(table[,var],table[,facteur],max))

 # Réalisation du graphique
 boxplot(table[,var] ~ table[,facteur], ...)

 #  sortie
 return(resultat)
}
```

Application On vérifie d'abord que les conditions d'arrêts fonctionnent bien :

```
univarie(crimin,"var1","var2")
univarie(crimin,"crim","cadr")
```

On teste ensuite sur le jeu de données **crimin** :

```
univarie(crimin,"crim","region",
  main="Distribution du crime par région", ylab="Variable Crime")
```




```
##     n  moyenne mediane ecart.type minimum maximum
## C  15 36.24667   36.70   8.531611    24.6    49.2
## NE 30 54.98333   49.40  20.707405    33.6   139.9
## NO 17 42.63529   39.80  10.119964    29.6    63.5
## SE 15 70.95333   65.00  24.251888    34.7   123.6
## SO 18 53.50556   45.15  21.132480    30.2    96.2
```

Quand on écrit une fonction, on essaye de faire en sorte qu'elle puise s'utiliser dans plusieurs cas de figure. Ici, on a codé la fonction pour qu'elle puisse être utiliser sur n'importe quel jeu de données. On pourra tester la fonction *univarie()* sur le jeu de données **iris** :

```
univarie(iris,"Sepal.Length","Species",
  main="Distribution de la variable pétale par espèce", ylab="Variable pétale")
```

4.2 Fonction *bivarie()*

La fonction *bivarie()* aura pour arguments d'entrée :

- la table de données **table**,
- le nom de deux variables quantitatives écrites dans **vars** sous forme d'un vecteur de **character**,
- le nom de la variable catégorielle **facteur**.

A partir de ces 3 arguments, retourner un **data.frame** contenant le nom des modalités de **facteur** en ligne et l'effectif ainsi que le coefficient de corrélation des variables **vars** en colonnes. En outre, elle devra réaliser le graphique pour chaque modalité de la variable catégorielle, le nuage de points des 2 variables (la première étant représentée sur l'axe des abscisses la seconde sur l'axe des ordonnées). On rappelle que la formule de corrélation linéaire de Pearson s'écrit :

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x}).(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}.\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} = \frac{(\sum_{i=1}^{n} x_i y_i) - \bar{x}\bar{y}}{\sqrt{(\sum_{i=1}^{n} x_i^2) - \bar{x}^2}.\sqrt{(\sum_{i=1}^{n} y_i^2) - \bar{y}^2}}$$

Indications

Un algorithme pour la construction de cette fonction est :

- 1. Vérifier que **vars** est un vecteur de taille 2 correspondant à des noms de colonnes de **table**.
- 2. Vérifier que **facteur** est une variable de **table** de type **factor**.
- 3. Identifier les modalités de **facteur**, faire une boucle **for** pour calculer le coefficient de corrélation et ajouter un nuage de points au graphique à chaque étape de la boucle. Pour information, les nuages de points se font avec la fonction *plot()*. On pourra utiliser les **...** pour utiliser les paramètres optionnels de *plot()*
- 4. Faire "sortir" les résultats.

Solution

Dans cet exemple, il n'est pas trivial de faire appel à la famille des fonctions *apply()* dans la mesure où on fait un calcul sur 2 variables en même temps. Du coup, on passe par l'utilisation de la boucle **for**. Il faut noter qu'on a pris le soin de déterminer à l'avance le nombre de calcul qu'on allait faire (qui dépend du nombre de modalités) et on a donc mis la matrice de résultats dans la bonne dimension avant de faire la boucle.

```
bivarie<-function(table, vars, facteur, ...)
{
 # Vérification 1
  stopifnot(length(vars)==2, all(vars%in%colnames(table)))

 # Vérification 2
  stopifnot(facteur%in%colnames(table),is.factor(table[,facteur]))

 # on identifie les modalités et leur nombre
 modalites<-levels(table[,facteur])
 n.modalites<-length(modalites)

 # on prépare la matrice contenant les effectifs
 # et les coeff. de corrélation par modalité
 res<-matrix(0, n.modalites, 2)

 # options pour représenter les graphiques
 op = par(mfrow=c(ceiling(sqrt(n.modalites)), floor(sqrt(n.modalites))),
          mar=c(2,3,2,2),oma=c(1,1,0,0), mgp=c(2,.4,0), cex.main=.75,
          cex.axis=.8, cex=.8, cex.lab=.8)

 # on boucle
```

```
 for(k in 1:n.modalites)
  {# on identifie les indices associés à la modalité k
   ind.k<-which(table[,facteur]==modalites[k])
   # on remplit les effectifs
   res[k,1]<-length(ind.k)
   # on calcule le coefficient de corrélation
   res[k,2]<-cor(table[ind.k,vars[1]], table[ind.k,vars[2]])
   # on dessine le nuage de points correspondants
   plot(table[ind.k,vars[1]], table[ind.k,vars[2]],
        main=paste("Modalité",modalites[k],"(r =",round(res[k,2],2),")"), ...)
  }

 # on retourne le résultat sous forme de data.frame
 return(data.frame(mod=factor(modalites),n=res[,1],cor=res[,2]))
}
```

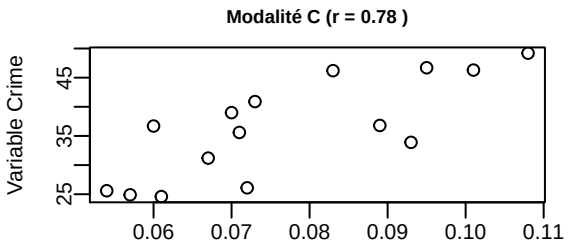
Application On vérifie d'abord que les conditions d'arrêts fonctionnent bien :

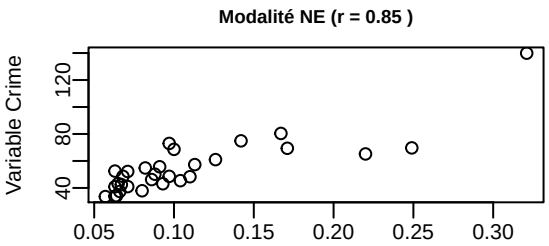
```
bivarie(crimin,c("var1","var2"),"region")
bivarie(crimin,vars="crim",facteur="cadr")
```

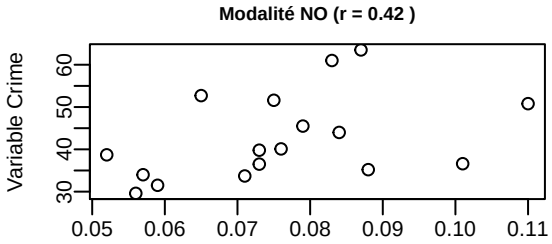
On teste ensuite sur le jeu de données **crimin** :

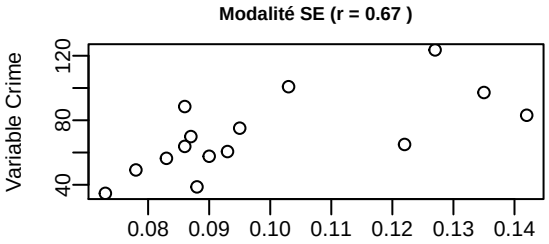
```
bivarie(crimin,c("cadr","crim"),"region",
  xlab="Pourcentage de cadre", ylab="Variable Crime")
```

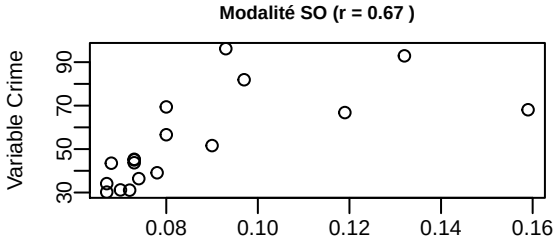
```
##   mod  n       cor
## 1   C 15 0.7816209
## 2  NE 30 0.8530509
## 3  NO 17 0.4222893
## 4  SE 15 0.6727890
## 5  SO 18 0.6714933
```


Modalité C (r = 0.78)
Variable Crime
25
35
45
0.06
0.07
0.08
0.09
0.10
0.11


Modalité NE (r = 0.85)
Variable Crime
40
80
120
0.05
0.10
0.15
0.20
0.25
0.30


Modalité NO (r = 0.42)
Variable Crime
30
40
50
60
0.05
0.06
0.07
0.08
0.09
0.10
0.11


Modalité SE (r = 0.67)
Variable Crime
40
80
120
0.08
0.09
0.10
0.11
0.12
0.13
0.14


Modalité SO (r = 0.67)
Variable Crime
30
50
70
90
0.08
0.10
0.12
0.14
0.16