

Programmer avec R

par

Odile Wolber

Un des points forts de R est dans la possibilité de programmer, de façon très simple (y compris par rapport à SAS), une suite d'analyses qui peuvent être exécutées successivement. R possède ainsi des particularités qui rendent la programmation accessible à des non-spécialistes.

Dans cette dernière partie sur R, nous allons présenter quelques notions simples de programmation, et notamment la manière de programmer ses propres fonctions. En tant que nouvel utilisateur de R, et si vous aviez déjà une expérience de programmation antérieure – notamment en langage C -, il vous est possible de participer à l'évolution de R en proposant de nouveaux outils d'analyse : cette question sera abordée dans le paragraphe 3 sur la création de packages.

1. Boucles, traitements conditionnels et vectorisation

Comme avec tout langage de programmation, on dispose avec R des boucles et des traitements conditionnels. On peut faire exécuter plusieurs instructions si elles sont encadrées par des accolades.

1. 1. Traitements conditionnels

Le traitement conditionnel if - else

```
if (condition)
{
  instruction(s)
}
else
{
  instruction(s)
}
```

Les traitements conditionnels peuvent être utilisés à l'intérieur d'autres constructions.

Exemples :

```
x <- if (cond==TRUE) 3.14 else 2.71
```

Supposons que l'on a un vecteur x et pour les éléments de x qui ont la valeur a, on va donner la valeur 0 à une autre variable y, sinon 1. On crée d'abord un vecteur y de même longueur que x :

```
x=c("a","b","c","a","a","b","b","c","c")
y=numeric(length(x))
for (i in 1:length(x)) if (x[i]=="a") y[i]=0 else y[i]=1
y
[1] 0 1 1 0 0 1 1 1 1
```

Il est également possible de construire des vecteurs à partir d'expressions conditionnelles, en utilisant la fonction « ifelse ».

```
x <- rnorm(10)
y <- ifelse(x>0, 1, -1)
z <- ifelse(x>0, 1, ifelse(x<0, -1, 0))
x
[1] 0.9674838 -0.1230696 0.7856278 -0.1663115 -1.2440741 1.0271240 0.2776376
1.2458897 -0.6505648 0.0000000
y
```

```
[1] 1 -1 1 -1 -1 1 1 1 -1 -1
z
[1] 1 -1 1 -1 -1 1 1 1 -1 0
```

La fonction switch

```
switch(EXPR,...)
```

EXPR expression évaluant un nombre ou une chaîne de caractères
... liste d'alternatives en fonction de la valeur de l'expression

Exemple : on affiche « Bonjour » si la variable caractère prend la valeur a, « Gutten Tag » si elle prend la valeur b, « Hello » si elle prend la valeur c, « Konnichi wa » si elle prend la valeur d.

```
x<- letters[floor(1+runif(1,0,4))]
y<-switch(x, a='Bonjour', b='Gutten Tag', c='Hello', d='Konnichi wa')
x
[1] "d"
y
[1] "Konnichi wa"
```

1.2. Boucles

La boucle for (compteur)

```
for (i in 1:n)
{
  instruction(s)
}
```

On peut utiliser des traitements conditionnels à l'intérieur de la boucle. Un traitement conditionnel particulièrement intéressant est `if (condition) {break}` qui permet de sortir de la boucle avant que le compteur n'atteigne la dernière valeur si (condition) est vérifiée :

```
for (i in 1:n) {
  ...
  if (...) {next} # instruction facultative :si la condition est vérifiée, on effectue le traitement
  qui suit
  ...
  if (...) {break} # instruction facultative : si la condition est vérifiée, on sort de la boucle
  avant que i=10
}
```

Exemple : l'exemple qui suit est tiré du polycopié de cours d'Arthur Tenenhaus.

Considérons un jeu de données X défini par n observations et p variables. Soient x et y deux observations. La fonction $K(x,y) = \exp(-\frac{\|x-y\|^2}{2\sigma^2})$ fournit un indice de similarité entre les observations x et y . L'objectif est d'écrire une fonction qui à toute matrice X associe la matrice K .

$$\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & & \vdots \\ x_{n1} & \dots & \dots & x_{np} \end{pmatrix} \rightarrow \begin{pmatrix} K(x_1, x_1) & K(x_1, x_2) & \dots & K(x_1, x_n) \\ K(x_2, x_1) & K(x_2, x_2) & \dots & K(x_2, x_n) \\ \vdots & \vdots & \dots & \vdots \\ K(x_n, x_1) & \vdots & \dots & K(x_n, x_n) \end{pmatrix} \text{ où } x_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix}$$

On remarque que la matrice K est symétrique.

```

noyau_gaussien = function(X, sigma)
{
  K = matrix(0, dim(X)[1], dim(X)[1])
  for (i in 1 : dim(X)[1])
  {
    for (j in i : dim(X)[1])
    {
      K[i,j]=(sum((abs(X[i, ]-X[j, ]))^2))
      K[j,i]=K[i,j]
    }
  }
  K=exp(-K/(2*(sigma^2)))
  return(K)
}

X
     [,1] [,2] [,3] [,4] [,5]
[1,]  1   5   9  13  17
[2,]  2   6  10  14  18
[3,]  3   7  11  15  19
[4,]  4   8  12  16  20

K=noyau_gaussien(X,2)
K
     [,1] [,2] [,3] [,4]
[1,] 1.000000000 0.5352614 0.0820850 0.003606563
[2,] 0.535261429 1.0000000 0.5352614 0.082084999
[3,] 0.082084999 0.5352614 1.0000000 0.535261429
[4,] 0.003606563 0.0820850 0.5352614 1.000000000

```

La boucle while

Les instructions qui se trouvent entre les accolades sont exécutées tant qu'elles sont vérifiées.

```

while (condition)
{
  instruction(s)
}

```

Exemple : on programme le calcul du factoriel d'un nombre entré par l'utilisateur¹. On va procéder de deux manières, d'abord en utilisant la boucle for, puis la boucle while.

```

factoriel1 = function(nombre)
{
  facto = 1
  for (i in 1 : nombre)
    facto=facto*i
  print(paste("Le factoriel de", nombre, "est", facto))
}

factoriel1(5)
[1] "Le factoriel de 5 est 120"

```

¹ Cette fonction existe déjà : il s'agit de `factorial()`, disponible dans le package de base.

```
factoriel2 = function(nombre)
{
  facto = 1
  i = 1
  while (i <= nombre)
  {
    facto = facto*i
    i = i+1
  }
  print(paste("Le factoriel de", nombre, "est", facto))
}
factoriel2(5)
[1] "Le factoriel de 5 est 120"
```

La boucle repeat

Les instructions qui se trouvent entre les accolades sont exécutées tant que la condition testée avec if (condition) n'est pas vérifiée.

```
repeat
{
  instruction(s)
  if (condition) {break} # si la condition est vérifiée, sortie de la boucle. Instruction
                        # obligatoire pour que l'on sorte de la boucle
  instruction(s)
}
```

1. 2. La vectorisation

En R, de nombreuses commandes utilisent des vecteurs ou des tables, ce qui permet pratiquement de programmer sans boucle. Ceci est important car les expressions vectorisées sont non seulement plus simples, mais aussi plus efficaces d'un point de vue informatique, particulièrement pour les grosses quantités de données.

Exemple :

Considérons l'addition de deux vecteurs de même longueur. Dans certains langages, il faudrait procéder avec une boucle, en créant le vecteur z au préalable à cause de l'utilisation de l'indexation, comme ci-dessous :

```
x<-c(2,5,7,8,2)
y<-c(1,3,2,7,9)
z<-numeric(max(length(x),length(y)))
z
[1] 0 0 0 0 0
for (i in 1:length(z)) z[i]<-x[i]+y[i]
z
[1] 3 8 9 15 11
```

Avec R, on peut écrire tout simplement :

```
z<-x+y
```

De même, l'indexation logique permet dans de nombreux cas d'éviter les exécutions conditionnelles (if...else).

Exemple :

Considérons le vecteur y : y[i]=0 si x[i]<0 et y[i]=1 sinon. On peut procéder avec une boucle et un traitement conditionnel comme ci-dessous :

```

x=rnorm(20);x
[1] -0.4594066 0.1219876 0.1874905 0.6895805 0.4135442 -0.8815913
[7] -0.4143397 0.9979605 -1.9664918 -1.4440129 0.8016411 -0.8473184
[13] 0.5603225 1.2481871 -1.3540264 0.5045687 -0.5370558 -0.9083900
[19] 1.0418681 0.3695007
for (i in 1:length(x))
{
  if (x[i]<0)
  y[i]=0
  else
  y[i]=1
}
y
[1] 0 1 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 1 1

```

On peut éviter la boucle et l'exécution conditionnelle en procédant comme suit :

```

y[x<0]=0;y[x>=0]=1
y
[1] 0 1 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 1 1

```

1. 3. Les fonctions de type « apply »

Les fonctions de type "apply" évitent également dans certains cas d'utiliser des boucles.

Fonction apply

La fonction apply applique une fonction aux lignes ou aux colonnes d'une matrice.

`apply(x, margin, FUN,...)`

x matrice
margin indique si l'action doit être appliqué aux lignes (1), aux colonnes (2) ou les deux (c(1,2))
FUN fonction (ou opérateur spécifié entre doubles quotes) à appliquer
... arguments supplémentaires éventuels pour FUN.

Exemple : on veut calculer la somme par ligne et par colonne de la matrice X.

```

x = 1:20
X=matrix(x, 4, 5)
X
     [,1] [,2] [,3] [,4] [,5]
[1,]  1   5   9  13  17
[2,]  2   6  10  14  18
[3,]  3   7  11  15  19
[4,]  4   8  12  16  20
apply(X,1,sum) # somme par ligne
[1] 45 50 55 60
apply(X,2,sum) # somme par colonne
[1] 10 26 42 58 74

```

Fonction lapply

La fonction lapply() s'applique à une liste. La syntaxe est similaire à celle d'apply et le résultat retourné est une liste. On utilisera lapply si on doit réaliser plusieurs analyses identiques sur des fichiers différents de données.

Exemple : on réalise l'ACP de deux fichiers : auto et voitures.

voiture

	Cylindrée	Puissance	Longueur	Largeur	Poids	Vitesse	Prix
ALFASUD_TI_1350	1350	79	393	161	870	165	30570
AUDI_100_L	1588	85	468	177	1110	160	39990
SIMCA_1307_GLS	1294	68	424	168	1050	152	29600
...							
MAZDA_9295	1769	83	440	165	1095	165	27900
OPEL_REKORD_L	1979	100	459	173	1120	173	32700
LADA_1300	1294	68	404	161	955	140	22100

auto

	Cylindrée	Puissance	Vitesse	Poids	Largeur	Longueur
Citroen C2 1.1 Base	1124	61	158	932	1659	3666
Smart Fortwo Coupe	698	52	135	730	1515	2500
Mini 1.6 170	1598	170	218	1215	1690	3625
...						
Land Rover Defender Td5	2495	122	135	1695	1790	3883
Land Rover Discovery Td5	2495	138	157	2175	2190	4705
Nissan X-Trail 2.2 dCi	2184	136	180	1520	1765	4455

On indique d'abord dans la liste nommée ACP les variables sur lesquelles on va réaliser successivement les deux ACP.

```
ACP<-list(auto,voiture)
```

```
lapply(ACP,princomp,cor=TRUE)
```

```
[[1]]
```

```
Call:
```

```
princomp(x = X[[1]], cor = TRUE)
```

Standard deviations:

```
Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6
2.1003018 0.9238018 0.6600484 0.4856651 0.2267966 0.1111369
```

6 variables and 24 observations.

```
[[2]]
```

```
Call:
```

```
princomp(x = X[[2]], cor = TRUE)
```

Standard deviations:

```
Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6 Comp.7
2.2516537 0.9312578 0.6740085 0.5682365 0.4036263 0.2854889 0.2030176
```

7 variables and 18 observations.

Fonction sapply

La fonction sapply est identique à lapply, à deux différences minimes près : il est possible de donner un vecteur ou une matrice en argument principal et la présentation des résultats est différentes.

Exemple : on reprend l'exemple précédent, en utilisant cette fois la fonction sapply.

```
result=sapply(ACP,princomp,cor=TRUE)
```

```
result
```

	[,1]	[,2]
sdev	Numeric,6	Numeric,7
loadings	Numeric,36	Numeric,49
center	Numeric,6	Numeric,7
scale	Numeric,6	Numeric,7
n.obs	24	18
scores	Numeric,144	Numeric,126
call	Expression	Expression

La fonction sapply retourne une liste. Pour afficher les résultats, on peut utiliser l'indexation :

result[[6]]	Comp.1	Comp.2	Comp. 3	Comp. 4	Comp. 5	Comp. 6
Citroen C2 1.1 Base	2.595915393	-0.50997398	0.17914010	-0.16570416	0.207127333	0.031611419
Smart Fortwo Coupe	4.150149792	-1.66590562	-0.27433215	0.92410926	0.029269629	-0.033602334
Mini 1.6 170	1.381906978	-0.81571698	-0.37235974	0.05104958	-0.464039063	0.050992642
...						
Land Rover Defender Td5	1.071682584	0.75142991	0.17613294	1.18245649	0.306776505	-0.008536735
Land Rover Discovery Td5	-0.850645172	1.92032433	1.51814063	0.99971025	-0.310926089	-0.034633766
Nissan X-Trail 2.2 dCi	0.614388049	0.72225714	0.04553763	-0.11663312	0.174152022	0.118669211

2. Ecriture de fonctions

Comme on l'a vu dans l'exemple sur le calcul du factoriel d'un nombre, il est possible d'écrire ses propres fonctions, qui auront les mêmes propriétés que les autres fonctions de R.

Une fonction est définie telle que dans l'exemple suivant :

```
f <- function(x) {
  x^2 + x + 1
}
```

La valeur retournée est la dernière valeur calculée, mais on peut également utiliser la fonction `return`.

```
f <- function(x){
  return(x^2 + x + 1)
}
```

Nous allons détailler dans cette partie la spécification des arguments, les incidences d'une fonction sur les variables de l'environnement global, l'enregistrement et le chargement ultérieur d'une fonction et enfin la détection des erreurs de programmation.

2. 1. Spécification des arguments

Les arguments d'une fonction peuvent être spécifiés de plusieurs manières :

- On peut les spécifier par leurs positions.
Par exemple, si on a une fonction qui prend trois arguments :
`f <- function(arg1, arg2, arg3) {...}`
`f(2,3,4)` exécute la fonction `f` en utilisant les valeurs `arg1=2`, `arg2=3` et `arg3=4`. Il est donc nécessaire de spécifier les arguments dans l'ordre.
- On peut utiliser les noms des arguments. On peut utiliser les arguments dans n'importe quel ordre.
`f(arg2=3, arg3=4, arg1=2) {...}` donne le même résultat que dans l'exemple précédent.
- On peut donner des arguments par défaut :
`f <- function(arg1=2, arg2, arg3) {...}`
`f(arg2=3, arg3=4)` donne également le même résultat que précédemment.

Dans ce cas, on ne donne une valeur à `arg1` que lorsque l'on souhaite lui affecter une valeur différente de 2.

- Dans la définition d'une fonction, on peut terminer la liste des arguments par trois points qui représentent les arguments qui n'ont pas été spécifiés. Par exemple, si à l'intérieur d'une fonction, on utilise la fonction `plot`, on peut mettre trois points pour les arguments graphiques que l'utilisateur précisera si besoin est :

```
f <- function(x, ...){  
  plot(x, ...)  
}
```

2. 2. Environnements local et global

Les modifications des variables qui sont effectuées à l'intérieur d'une fonction n'affectent pas les valeurs des variables globales, même si elles ont le même nom.

Exemple : les variables `a` et `b`, créées à l'intérieur de la fonction `f` sont des variables locales. Dès que l'exécution de `f` est terminée, ces variables n'existent plus.

```
f<- function(x,y) {  
  a=x^2  
  b=y^3  
  a+b  
}  
  
f(4,5)  
[1] 141  
  
a  
Erreur : objet "a" non trouvé  
> b  
Erreur : objet "b" non trouvé
```

Si l'on utilise une variable au cours de l'exécution d'une fonction, et que le nom de cette variable désigne un objet dans l'environnement global, la valeur de la variable dans l'environnement global n'est pas utilisée.

Exemple : la variable globale `x` prend la valeur 1. Lors de l'exécution de la fonction `f`, on utilise une variable locale `x` qui prend la valeur 2. Dès que l'exécution de `f` est terminée, l'appel de la variable `x` fait de nouveau référence à la variable globale et retourne la valeur 1.

```
x=1  
f <- function() {  
  x=2; print(x)  
}  
f()  
[1] 2  
x  
[1] 1
```

Si on souhaite créer au cours de l'exécution d'une fonction une variable globale, il faut utiliser la fonction `assign`, dont les arguments sont les suivants :

```
assign(x, value, pos=-1)  
x      nom de variable ( donné entre quotes)  
value  valeur affectée à x  
pos    -1 : environnement courant, .GlobalEnv : environnement global
```


Exemple : au cours de l'exécution de *f*, on crée la variable globale *a*, égale au carré de *x*.

```
f<- function(x,y) {  
  a=x^2  
  assign("a",x^2,pos=.GlobalEnv)  
  a+y^3  
}  
f(4,5)  
[1] 141  
a  
[1] 16
```

Il n'est pas nécessaire de déclarer les variables qui sont utilisées dans une fonction. Si une variable est appelée, R va d'abord chercher dans l'environnement de la fonction un objet qui porte le nom de cette variable. S'il ne trouve pas la variable, la recherche de l'objet se poursuit par « paliers » de l'environnement courant à l'environnement immédiatement supérieur, jusqu'à l'environnement global.

Exemple : lors de l'appel de la fonction *f*, on ne précise pas l'argument *x*. R cherche alors dans l'environnement global une variable qui porte le nom *x*.

```
f <- function() {  
  print(x^2 + x + 1)  
}  
x=3  
f()  
[1] 13
```

2. 3. Enregistrement et chargement ultérieur d'une fonction

Pour pouvoir être exécutée, une fonction créée par l'utilisateur doit être chargée en mémoire. Plusieurs possibilités s'offrent à l'utilisateur, nous allons détailler celle qui consiste à enregistrer la fonction au format ASCII, puis à la charger avec `source()` comme un autre programme :

- On sélectionne le menu Fichier > Nouveau script. Une fenêtre R Editor s'ouvre, dans laquelle on écrit les lignes de programme.

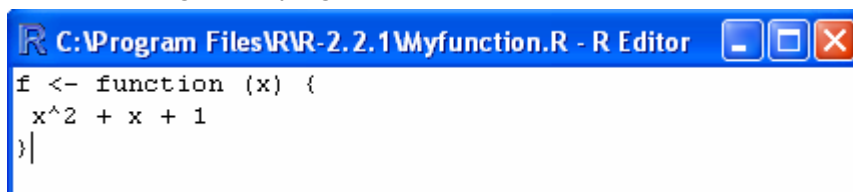
On sauvegarde ensuite ces lignes de programme en passant par le menu Fichier > Sauver sous. Le programme sera écrit dans un fichier sauvé au format ASCII et avec l'extension R.

On pourra ensuite appeler le programme en utilisant l'instruction `source()`. Comme pour toute lecture d'un fichier, il est nécessaire de préciser le chemin d'accès au fichier s'il n'est pas dans le répertoire de travail.

Exemple : on veut sauvegarder dans un fichier R la fonction *f* :

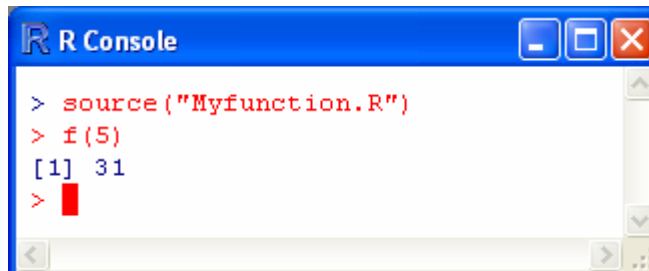
```
f <- function (x) {  
  x^2 + x + 1  
}
```

On saisit ces lignes de programmes dans la fenêtre R Editor.



On sauve le programme sous le nom « Myfunction.R » dans le répertoire courant.

Dans une session ultérieure, on veut utiliser la fonction *f*. On appelle le programme avec l'instruction `source("Myfunction.R")` et on peut immédiatement l'utiliser pour calculer *f*(5) :



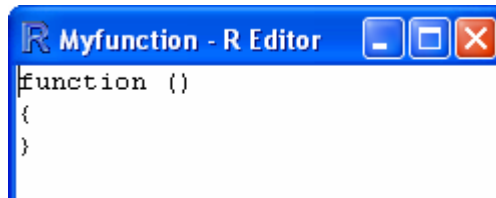
```
> source("Myfunction.R")
> f(5)
[1] 31
>
```

- On peut également utiliser la commande `fix(nom_de_fonction)` pour ouvrir une fenêtre R Editor.

Exemple : on tape sur la console la commande :

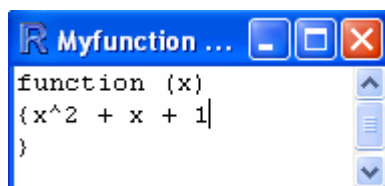
`fix(Myfunction)`

La fenêtre suivante s'ouvre :



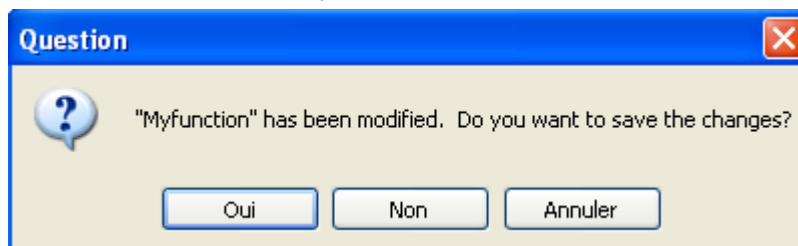
```
function ()
{
}
```

On saisit ensuite la fonction :



```
function (x)
{ x^2 + x + 1 |
}
```

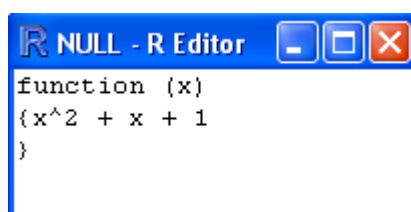
On ferme ensuite la fenêtre. Une fenêtre pop-up s'ouvre alors pour sauver les modifications (mais le script n'est pas sauvé et on ne pourra pas réutiliser la fonction dans une session suivante) :



Avant de fermer la fenêtre, R vérifie la syntaxe. Pour corriger les erreurs, on édite la fonction avec la commande :

`Myfunction <- edit()`

La fenêtre R Editor s'ouvre de nouveau avec le script de la fonction :



```
function (x)
{ x^2 + x + 1
}
```

`Myfunction(2)` retourne la valeur suivante :
[1] 7

- Si l'utilisateur veut que ses fonctions soient chargées au démarrage de R, il peut les enregistrer dans un workspace .RData du répertoire de travail de démarrage. Une autre possibilité est de configurer le fichier '.Rprofile' (taper ?Startup sur la console pour obtenir plus de détails) ou de créer un package (cf. § 3. Créer un package).

2. 4. Détecter les erreurs

Les options warn

Si on pense qu'il peut y avoir des problèmes dans le programme qu'on a réalisé, on peut obtenir une interprétation plus détaillée avec l'option `options(warn=1)`. Les messages de mise en garde sont imprimés dès leur apparition (et non à la fin de l'exécution du programme).

L'option `options(warn=2)` transforme les messages de mise en garde en erreurs et stoppe l'exécution du programme.

La commande debug

La commande debug détaille le déroulement d'une fonction et met en évidence, étape par étape, les éventuelles erreurs de programmation.

Exemples : on reprend la fonction f suivante :

```
f <- function (x) {
  x^2 + x + 1
}
```

On tape d'abord la commande :

```
debug(f)
```

On applique ensuite la fonction f au chiffre 3 :

```
f(3)
```

Les lignes suivantes s'affichent :

```
debugging in : f(3)
debug : {
  x^2 + x + 1
}
Browse[1]>
```

Après chaque « Browse[1] », il faut taper sur la touche enter pour que l'exécution du programme se poursuive :

```
debug: x^2 + x + 1
Browse[1]>
exiting from: f(3)
[1] 13
```

On tape ensuite `undebug(f)` pour qu'au prochain appel de la fonction, on n'ait pas le détail de l'exécution.

Dans l'exemple suivant, on donne une valeur au paramètre x qui est erronée (la racine carrée de -1 n'existe pas). La commande debug donne l'affichage suivant :

```
f <- function(x) {
  sqrt(x) + x
}
debug(f)
f(-1)
debugging in: f(-1)
debug: {
  sqrt(x) + x
}
Browse[1]>
debug: sqrt(x) + x
```

```
Browse[1]>
exiting from: f(-1)
[1] NaN
Warning message:
production de NaN in: sqrt(x)
```

Dans les exemples précédents, la commande debug présente un intérêt limité car on n'a qu'une seule instruction. On a autant d'informations en exécutant l'instruction f(-1) :

```
f(-1)
[1] NaN
Warning message:
production de NaN in: sqrt(x)
```

La commande traceback

La commande traceback imprime la liste des fonctions qui ont été appelées lorsque la dernière erreur a eu lieu. Elle présente un intérêt essentiellement pour les programmes longs.

Exemple :

```
f(-1,-2)
Warning message:
production de NaN in: sqrt(y)
> traceback()
1: f(-1)
```

La fonction stop

La fonction « stop » arrête une fonction dès qu'un problème survient (par exemple, s'il y a un problème avec la nature des arguments).

Exemple :

```
saisie.vecteur2<-function(x) {
  if( !is.numeric(x))
    stop("Vous devez saisir un vecteur NUMERIQUE")
  if( !is.vector(x))
    stop("Vous devez saisir un VECTEUR numérique")
  if (length(x)<2)
    stop("le vecteur doit être au moins de dimension 2")
  return("Saisie correcte")
}

saisie.vecteur2("ab")
Erreur dans saisie.vecteur2("ab") : Vous devez saisir un vecteur NUMERIQUE

saisie.vecteur2(3)
Erreur dans saisie.vecteur2(3) : le vecteur doit être au moins de dimension 2

saisie.vecteur2(matrix(1:4,nc=2,nr=2))
Erreur dans saisie.vecteur2(matrix(1:4, nc = 2, nr = 2)) :
Vous devez saisir un VECTEUR numérique
```

3. Créer un package

Un package est un ensemble de fonctions, fichiers et manuels, contenus dans une librairie ou un fichier *.tar.gz ou *.zip. Il y a souvent une confusion entre "package" et "library", due notamment à la commande library utilisée pour charger les packages. Une librairie est un répertoire, qui contient généralement un ou plusieurs packages.

Nous allons voir sur un exemple comment créer son propre package. Pour plus de précisions, on se reportera au manuel « Writing R extensions », disponible à partir du menu d'aide de R.

Exemple : supposons que l'on ait créé un fichier *Functions3.R* qui contient 3 fonctions : *f*, *g* et *h*. On va créer un package en commençant par taper les deux lignes de code suivantes :

```
source("Functions3.R")
package.skeleton("Functions3",c("f","g","h"))
```

Les messages suivants s'inscrivent sur la console :

```
Creating directories ...
Creating DESCRIPTION ...
Creating READMEs ...
Saving functions and data ...
Making help files ...
Created file named './Functions3/man/Functions3.package.Rd'.
Edit the file and move it to the appropriate directory.
Created file named './Functions3/man/f.Rd'.
Edit the file and move it to the appropriate directory.
Created file named './Functions3/man/g.Rd'.
Edit the file and move it to the appropriate directory.
Created file named './Functions3/man/h.Rd'.
Edit the file and move it to the appropriate directory.
Done.
Further steps are described in ./Functions3/README
```

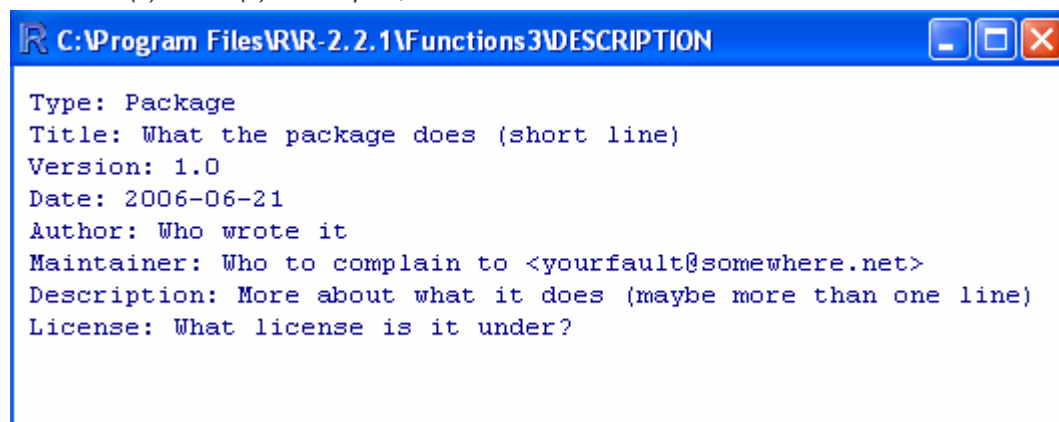
On peut consulter dans le répertoire *R-2.2.1* le dossier *Functions3*. Il contient les fichiers suivants :

```
Functions3/man/Functions3package.Rd
Functions3/man/f.Rd
Functions3/man/g.Rd
Functions3/man/h.Rd
Functions3/man/README
Functions3/R/f.R
Functions3/R/g.R
Functions3/R/h.R
Functions3/src/README
Functions3/DESCRIPTION
Functions3/README
```

Si on avait inclus dans le package une *data.frame*, on aurait également un répertoire *data* contenant un fichier d'extension *rda*.

On n'a pas à modifier les fichiers d'extension *R* : ils contiennent le code des fonctions.

Le fichier *DESCRIPTION* doit être modifié. On peut l'ouvrir en passant par le menu *Fichier > Afficher le(s) fichier(s)*. Au départ, il contient les informations suivantes :



On peut également l'ouvrir dans le bloc-notes et ainsi le modifier.

Il faut également modifier les fichiers de documentation d'extension Rd. On peut également les afficher en passant par le menu Fichier > Afficher le(s) fichier(s). Au départ, ils se présentent sous la forme suivante (f.Rd) :

```
\name{f}
\alias{f}
%- Also NEED an 'alias' for EACH other topic documented here.
\title{ ~~function to do ... ~~ }
\description{
  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
f(x)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{x}{ ~~Describe \code{x} here~~ }
}
\details{
  ~~ If necessary, more details than the description above ~~
}
\value{
  ~Describe the value returned
  If it is a LIST, use
  \item{comp1 }{Description of 'comp1'}
  \item{comp2 }{Description of 'comp2'}
  ...
}
\references{ ~put references to the literature/web site here ~ }
\author{ ~~who you are~~ }
\note{ ~~further notes~~

  ~Make other sections like Warning with \section{Warning }{....} ~
}
\seealso{ ~~objects to See Also as \code{\link{help}}, ~~~ }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##-- or do help(data=index) for the standard data sets.

## The function is currently defined as
function (x) {
  x^2 + x + 1
}
}
\keyword{ ~kwd1 }% at least one, from doc/KEYWORDS
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line
```

Les fichiers d'extension Rd sont utilisés pour documenter la fonction sur le modèle standard, tel qu'il apparaît lorsque l'on tape une instruction `help(nom_de_fonction)`.

- Si on veut documenter les 3 fonctions dans la même page, on utilisera l'instruction `alias`. On modifiera la page précédente avec les instructions :

```
\alias{f}
\alias{g}
\alias{h}
```

- On donne une très courte description de ces fonctions (moins d'une ligne) :
`\title{ Basic functions }`
- Une description plus longue sera donnée ensuite après l'instruction `\description` :
`\description{`

Very simple functions that perform elementary arithmetic operations such as adding one, multiplying by two or squaring

- On décrit l'usage de ces fonctions, en indiquant tous les arguments et le cas échéant, les valeurs par défaut :

```
\usage{
f(x)
}
\usage{
g(x,y)
}
\usage{
h(x,y,z)
}
```

- On donne une description des arguments, laquelle peut être beaucoup plus longue que ci-dessous :

```
\arguments{
\item{x}{A number}
\item{y}{A number}
\item{z}{A number}
}
```

- On donne une description plus détaillée des fonctions, des algorithmes utilisées, des erreurs à éviter dans leur utilisation, etc.

```
\details{
The \code{f} returns the sum of  $x^2 + x + 1$ 
The \code{g} returns the sum of  $x^2 + y + 3$ 
The \code{h} returns the sum of  $2 \cdot z^3 + x^2 + 3 \cdot y + 4$ 
}
```

- Après l'instruction `value`, on décrit les résultats des fonctions (classe de l'objet retourné en résultat, etc). On peut ensuite donner des références (`\references`), le nom de l'auteur et son adresse électronique (`\author`), des liens vers d'autres pages ou manuels qui pourraient être utiles pour utiliser les fonctions (`\seealso`) et des mots-clés (`\keyword`).

- Une partie également importante est celle concernant les exemples. Ils doivent pouvoir être recopiés par l'utilisateur, qui les utilisera tels quels pour tester que tout s'est bien passé lors de l'installation du package. On ajoute également des tests avec l'instruction `"stopifnot"` : dans l'exemple suivant, on arrêtera le traitement si $f(2) \neq 7$, $g(2,3) \neq 10$ et $h(2,3,4) \neq 145$.

```
\examples{
f(2)+g(2,3)+h(2,3,4)
(2^2+2+1)+(2^2+3+3)+(2*4^3+2^2+3*3+4)
stopifnot( f(2) == 7 )
stopifnot( g(2,3) == 10 )
stopifnot( h(2,3,4) == 145 )
}
```

Une fois qu'on a complété cette page de documentation, on peut supprimer les pages vides du répertoire `Functions3/man` (en l'occurrence les fichiers `g.Rd` et `h.Rd`).

On construit ensuite le package en utilisant la commande `R CMD build` :

```
R CMD build Functions3
```

On vérifie les exemples donnés dans la page de documentation en utilisant la commande `R CMD check` :

```
R CMD check Functions3
```

Le fichier final `Functions3.tar.gz` qui sera distribué est créé à partir de la commande :

```
R CMD install Functions3
```

Exercices complémentaires

Exercice 1 Afficher les chiffres

Écrire une fonction permettant d'afficher les n premiers chiffres à l'écran (où n est fixé par l'utilisateur)

Exercice 2 Afficher les chiffres pairs

Écrire une fonction permettant d'afficher les n premiers chiffres pairs à l'écran (où n est fixé par l'utilisateur)

Exercice 3 Matrice identité

Écrire une fonction permettant de construire une matrice identité de dimension n (où n est fixé par l'utilisateur).

Exercice 4 Stockage de coefficients résultant d'analyses

Considérons le jeu de données auto2004_original.txt. On souhaite évaluer l'impact des différentes variables sur la variable cylindrée. On décide donc de faire une régression de la variable cylindrée sur chacune des variables et stocker les coefficients de chacune des régressions dans un vecteur. Écrire une fonction prenant 2 arguments (la variable cible et le tableau des variables explicatives) qui retourne le vecteur de coefficients.

Exercice 5 Echantillonnage

Écrire une fonction permettant de retourner un jeu de données composé d'une sélection aléatoire de n individus extrait d'un jeu de données initial X (où les paramètres n et X sont fixés par l'utilisateur).

Exercice 6 Limite de e^x

$$e^x = \lim_{n \rightarrow +\infty} \sum_{k=0}^n \frac{x^k}{k!}$$

Programmer une fonction qui retourne la valeur approchée de e^x pour des valeurs de x et n données (où x et n sont fournis par l'utilisateur).

Exercice 7 Graphique

Reprenons l'exercice 4 de la session R05. Écrire une fonction permettant de relier chaque observation d'un jeu de données à son centre de gravité. Le nombre de centre de gravité étant fourni par l'utilisateur et les centres de gravité étant calculé préalablement par la méthode des kmeans appliqué aux données projetées sur les deux premières composantes principales de l'ACP.

Correction

Exercice 1 Afficher les chiffres

Première proposition : Boucle for

```
ecriture_for = function(nombre)
{
    for (i in 1 : nombre)
    {
        print(i)
    }
}
```

Deuxième proposition : Boucle while

```
ecriture_while = function(nbre)
{
    i = 1
    while (i <= nbre)
    {
        print(i)
        i = i+1
    }
}
```

Exercice 2 Afficher les chiffres pairs

Première proposition : Boucle for

```
ecriture_pair = function(nombre)
{
    for (i in 1: nombre)
    {
        if (i%%2 == 0)
        {
            print(i)
        }
    }
}
```

Deuxième proposition : Boucle while

```
ecriture_pair = function(nombre)
{
    i = 2
    while (i <= nombre)
    {
        print(i)
        i = i+2
    }
}
```

Exercice 3 Matrice identité

```
identite = function(dimension)
{
  X = matrix(0, dimension, dimension)
  for (i in 1:dimension)
  {
    X[i,i]=1
  }
  return(X)
}
```

Exercice 4 Stockage de coefficients résultant d'analyses

```
coef_reg = function(X, Y)
{
  b = numeric(dim(X)[2])
  for (i in 1 : dim(X)[2])
  {
    result.lm = lm(Y ~ X[, i])
    b[i] = result.lm[[1]][2]
  }
  return(b)
}
```

```
## Applications aux autos
A = read.table("/auto2004_original.txt", header = TRUE, sep = "\t")
variable = A[, 3:7]
cible = A$Cylindree
b = coef_reg(variable, cible)
```

Exercice 5 Echantillonnage

```
aleatoire = function (X, n)
{
  ind = sample(1 :dim(X)[1], n)
  Xnew = X[ind, ]
  return(Xnew)
}
```

Exercice 6 Limite de e^x

```
exponentiel = function(x, n)
{
  numerateur = numeric(n)
  denominateur = numeric(n)
  for (i in 1 : n)
  {
    numerateur[i] = x^i
    denominateur[i] = factorial(i)
  }
  resultat = 1 + sum(numerateur/denominateur)
  return(resultat)
}
```

Exercice 7 Graphique

```
kmeans.plot <- function(X, n)
{
  result.pca <- princomp(X)
  X.pca = result.pca[[6]][, 1:2]
  result.kmeans <- kmeans(X.pca, n)
  plot(X.pca)
  points(result.kmeans$centers, col = 1:3, pch = 7, lwd = 3)
  for (i in 1:n)
  {
    segments(X.pca[result.kmeans$cluster == i, ][, 1],
             X.pca[result.kmeans$cluster == i, ][, 2],
             result.kmeans$centers[i, 1],
             result.kmeans$centers[i, 2],
             col = i
            )
  }
}
```