

Formation



Programmation et développement

Instr. : Emmanuel Paradis

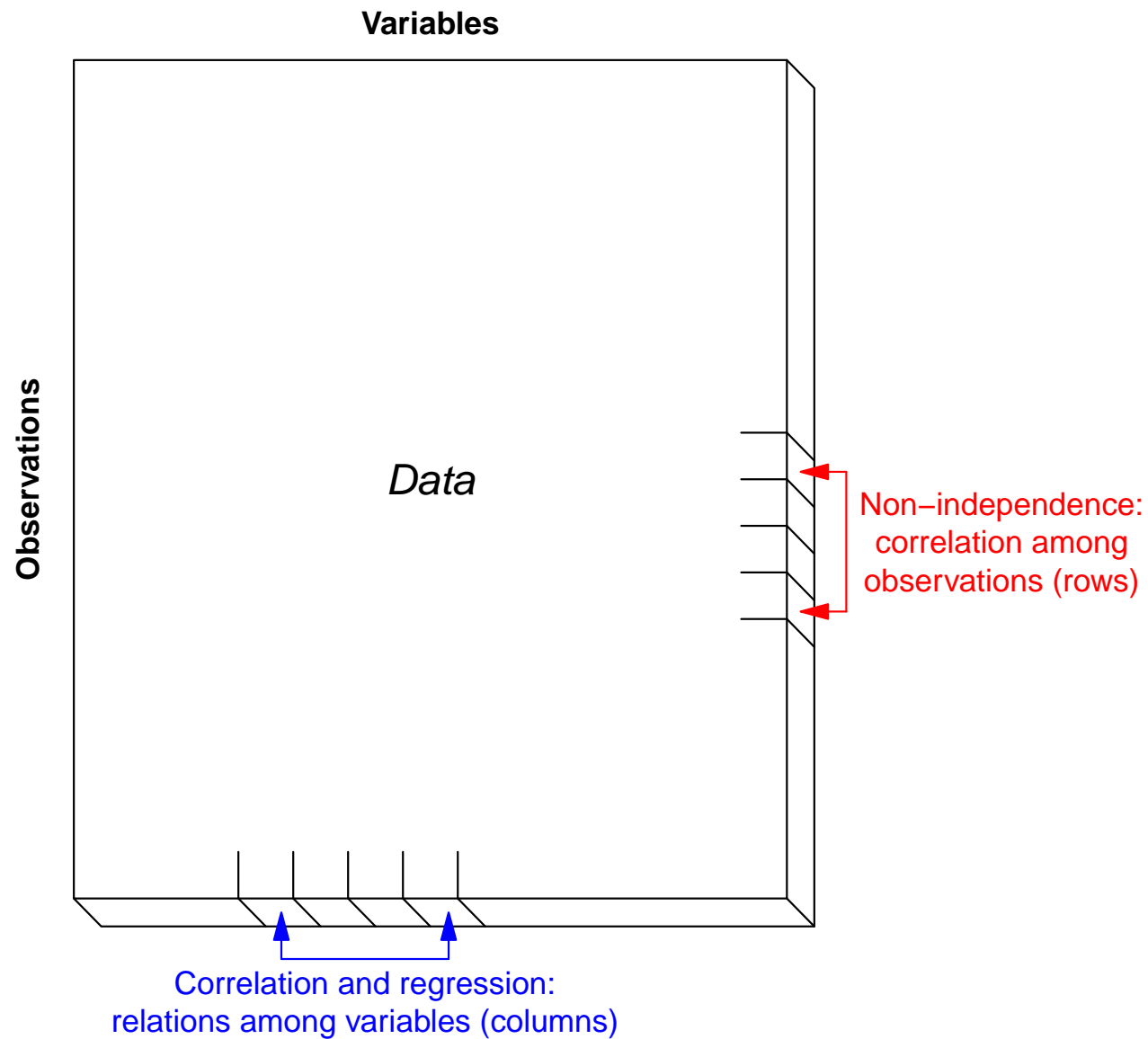
Org. : Lydie Civiletti, Monique Saltel & Christel Gruau

Montpellier, 23–27 septembre 2019

Sommaire

- I Structure des données dans R
- II Programmer des fonctions en R
- III Les expressions
- IV Accéder au système d'exploitation
- V Débogage
- VI Profilage et optimisation
- VII Interface R/C
- VIII Construire un package

I Structure des données dans R



Deux types de variables :

➤ **quantitative**

- continue : taille, masse, température, dosage, ...
- entière (discrète) : effectif de population, nombre de gènes, ...

➤ **qualitative** (catégorielle)

- non-ordonnée : habitat, phénotype, ...
- ordonnée : statut de patient, satisfaction, ...

Les données dans R

Vector

- élément basique des données dans R
- *vector* = une colonne (variable)

```
x <- 1:5
```

Les données dans R

Vector

- élément basique des données dans R
- *vector* = une colonne (variable)

```
x <- 1:5
```

Les données dans R sont :

- un or plusieurs vecteurs
- des *attributes* (info sur les données... qui sont elles-mêmes des données)

1
2
3
4
5

length (= 5)

mode (= "numeric")

"Homo"
"Pan"
"Gorilla"

length (= 3)

mode (= "character")

} **Attributes**

1
2
3
4
5

"Homo"
"Pan"
"Gorilla"

length (= 5)

mode (= "numeric")

length (= 3)

mode (= "character")

} **Attributes**

Cinq modes : numeric,^a character,^b logical,^c complex, raw (bytes)

^a *numeric vector* (var. quantitative), *factor* (var. qualitative)

^b étiquettes (identifiants, “labels of the levels of a factor”, ...)

^c manipulation de données

Cinq types de données *objects* :

- vector
- factor
- matrix
- data frame
- list

factor : un vecteur d'entiers avec l'attribut *levels* (vecteur de mode 'character')

matrix : un vecteur arrangé de façon rectangulaire

data frame : un ensemble de vecteurs et de facteurs de la même longueur (= tableau)

list : un ensemble d'objets (= *vector of objects*)

Un attribut important : la *class*.

mode : attribut basique (intrinsèque)

class : attribut de haut niveau

Un attribut important : la *class*.

mode : attribut basique (intrinsèque)

class : attribut de haut niveau

Indexing : outil fondamental de manipulation des données

[]	vecteurs et facteurs
[,]	matrices et tableaux
[] [[]] \$	listes et tableaux

Un attribut important : la *class*.

mode : attribut basique (intrinsèque)

class : attribut de haut niveau

Indexing : outil fondamental de manipulation des données

[]	vecteurs et facteurs
[,]	matrices et tableaux
[] [[]] \$	listes et tableaux



[] subsetting [[]] extraction

```
> x <- list(1, 1:3)
```

```
> x[1]
```

```
[[1]]
```

```
[1] 1
```

```
> x[[1]]
```

```
[1] 1
```

L'indexation peut être :

- numérique (négative ou positive)
- logique (avec recyclage des indices logiques)
- avec “names” (ou row- et/ou colnames)

L'indexation permet d'extraire, effacer ou remplacer les valeurs de tout type d'object.

Les indices sont un vecteur.

```
> x <- 5:10
> x[2:3]           # extraction
[1] 6 7
> x[-1]           # deletion
[1] 6 7 8 9 10
> x[6] <- 20      # replacement
> x
[1] 6 7 8 9 10 20
```

```
> x <- 5:10
> x[2:3] # extraction
[1] 6 7
> x[-1] # deletion
[1] 6 7 8 9 10
> x[6] <- 20 # replacement
> x
[1] 6 7 8 9 10 20
> x[c(TRUE, FALSE)] # extraction (recycling)
[1] 5 7 9
```




```
> x <- 5:10
> x[2:3]                                # extraction
[1] 6 7
> x[-1]                                 # deletion
[1] 6 7 8 9 10
> x[6] <- 20                            # replacement
> x
[1] 6 7 8 9 10 20
> x[c(TRUE, FALSE)]                    # extraction (recycling)
[1] 5 7 9
> names(x) <- letters[1:6]
> x["a"]                               # extraction
a
5
> x["a"] <- 100                         # replacement
> x
  a    b    c    d    e    f
100  6    7    8    9   20
```

Ceci se généralise à :

- facteurs
- matrices et tableaux : deux dimensions donc deux jeux d'indices
 - `x[, 1:2]` ou `x[1:2,]` pour sélectionner toutes les lignes ou colonnes

Ceci se généralise à :

- facteurs
- matrices et tableaux : deux dimensions donc deux jeux d'indices
 - $x[, 1:2]$ ou $x[1:2,]$ pour sélectionner toutes les lignes ou colonnes
 -  les matrices sont des vecteurs et les tableaux des listes
- lists
 - opérateurs spéciaux `[[` et `$` pour extraction à partir d'une liste

Concepts avancés

Toutes les données dans R sont des *vecteurs* :

- *atomique* (vecteur, matrice, array, facteur),
- *générique* (tableau, liste) qui sont des vecteurs d'objets.

Rappel sur les vecteurs atomiques :

- deux *attributs intrinsèques* : “mode”, “length”
- cinq *modes* : numeric, character, logical, raw, complex

```
> x <- 1:8
> attributes(x)
NULL
> M <- matrix(1:8, 4)
> attributes(M)
$dim
[1] 4 2
> A <- array(1:8, c(2, 2, 2))
> attributes(A)
$dim
[1] 2 2 2

> F <- factor(1:8)
> attributes(F)
$levels
[1] "1" "2" "3" "4" "5" "6" "7" "8"

$class
[1] "factor"
```

Les objets x , M , A et F :

- contiennent les mêmes valeurs (les entiers de 1 à 8)
- des attributs différents qui caractérisent ces objets

Les objets x , M , A et F :

- contiennent les mêmes valeurs (les entiers de 1 à 8)
- des attributs différents qui caractérisent ces objets

Par exemple, dans un modèle linéaire, x et F seront traités différemment.

Les objets x , M , A et F :

- contiennent les mêmes valeurs (les entiers de 1 à 8)
- des attributs différents qui caractérisent ces objets

Par exemple, dans un modèle linéaire, x et F seront traités différemment.

```
> L <- list(rnorm(2), 1:5)
```

```
> L
```

```
[[1]]
```

```
[1] -0.5946724 -0.5575795
```

```
[[2]]
```

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

```
> mode(L)
```

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


La syntaxe de '[' est la même pour les vecteurs et les listes.

```
> DF <- data.frame(1, 2)
> mode(DF)
[1] "list"
```

Un tableau est un cas particulier de liste avec uniquement des vecteurs et des facteurs tous de même longueur.

La syntaxe de '[', '[[', et '\$' est la même pour les listes et les tableaux.

```
> attributes(L)
NULL
> attributes(DF)
$names
[1] "X1" "X2"

$row.names
```

```
[1] 1
```

```
$class
```

```
[1] "data.frame"
```

Attributs fréquemment utilisés : `names`, `dim`, `dimnames`, `class`.

Les attributs sont extraits ou modifiés avec :

- `attr` (individuellement)
- `attributes` (collectivement sous forme de liste)
- une fonction spécifique si elle existe (`class`, `levels`, `names`, `dim`, ...)
- `structure` (plutôt dans une fonction) :

```
> structure(1:6, dim = c(3, 2))
```

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

Il y a deux stratégies pour créer une structure de données sous R : avec une liste ou avec des attributs. Exemple : matrice de distances calculée avec `dist` :

```
> d <- dist(M)
> d
```

	1	2	3
2	1.414214		
3	2.828427	1.414214	
4	4.242641	2.828427	1.414214

```
> str(d)
Class 'dist'  atomic [1:6] 1.41 2.83 4.24 1.41 2.83 ...
 ..- attr(*, "Size")= int 4
 ..- attr(*, "Diag")= logi FALSE
 ..- attr(*, "Upper")= logi FALSE
 ..- attr(*, "method")= chr "euclidean"
 ..- attr(*, "call")= language dist(x = M)
> attr(d, "Size")
[1] 4
```

Exemple : objet retourné par `lm` :

```
> str(lm(rnorm(8) ~ x))
```

```
List of 12
```


```
$ coefficients : Named num [1:2] -1.254 0.332
```

```
..- attr(*, "names")= chr [1:2] "(Intercept)" "x"
```

```
$ residuals : Named num [1:8] 1.022 -1.41 -1.061 0.264 0.49
```

```
..- attr(*, "names")= chr [1:8] "1" "2" "3" "4" ...
```

```
....
```

 Les attributs sont généralement supprimés au cours d'une extraction (sauf éventuellement `names`, `dim`, `dimnames`) :

```
> d[1:3]
```

```
[1] 1.414214 2.828427 4.242641
```

```
> attributes(d[1:3])
```

```
NULL
```

... mais (*cf.* ci-dessous pour l'explication) :

```
> attributes(F[1:2])
```

```
$levels
```

```
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"
```

```
$class
```

```
[1] "factor"
```

Exercices I

1. Créer une structure d'objet comprenant deux matrices de dissimilarité (l'une symétrique et l'autre non). On fera un effort pour éviter les redondances d'information.
2. Donner une classe à cette structure d'objet et produire un exemple avec $n = 5$ observations.

II Programmer des fonctions en R

Les arguments

Définition d'une fonction sans argument :

```
> foo <- function() cat("Hello World\n")  
> foo()  
Hello World
```

Un argument est “local” à la fonction :

```
> foo <- function(x) print(x)  
> x <- 1  
> foo(2) # idem que foo(x = 2)  
[1] 2  
> print(x)
```

```
[1] 1
```

Définir un argument par défaut (= *option*) :

```
> foo <- function(x = 2) print(x)
```

```
> foo() # == foo(2) == foo(x = 2)
```

```
[1] 2
```

```
> foo(4) # == foo(x = 4)
```

```
[1] 4
```

Option “non définie” à l’avance.

1. Calculée dans la définition de la fonction :

```
> args(write)
```

```
function (x, file = "data", ncolums = if (is.character(x))  
  1 else 5, append = FALSE, sep = " ")
```


2. Définir l'argument comme `NULL`. La fonction `write` pourrait être :

```
> write
function (x, file = "data", ncolumns = NULL,
         append = FALSE, sep = " ")
{
  if (is.null(ncolumns))
    ncolumns <- if (is.character(x)) 1 else 5
  ....
}
```

Plus utile pour les arguments avec plusieurs choix (où `if ... else` pas forcément facile à utiliser).

3. Utiliser `missing` dans la fonction :

```
> plot.default
function (x, y = NULL, .....
{
  ....
  xlabel <- if (!missing(x)) deparse(substitute(x))
  ylabel <- if (!missing(y)) deparse(substitute(y))
}
```

.....

`missing` doit être appelée avant que `y` soit utilisé.

R fait correspondre les noms (*tags*) des arguments abrégés de façon non-ambigüe :

```
> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
          dimnames = NULL)
> matrix(1:10, nr = 2, b = TRUE)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

Pour faire correspondre la valeur d'un argument de mode caractère abrégé (à droite du signe '=') on peut utiliser, à l'intérieur de la fonction, `match.arg` ou `pmatch` :

```
> match.arg("o", c("oui", "non"))
[1] "oui"
> match.arg("n", c("oui", "non"))
[1] "non"
> match.arg("n", c("ncol", "nrow"))
Error in match.arg("n", c("ncol", "nrow")) :
  'arg' should be one of "ncol", "nrow"
> match.arg("nc", c("ncol", "nrow"))
[1] "ncol"

> pmatch("nc", c("ncol", "nrow"))
[1] 1
> pmatch("n", c("ncol", "nrow"))
[1] NA
```

La fonction `cor` utilise les deux :

```
> cor
function (x, y = NULL, use = "everything",
          method = c("pearson", "kendall", "spearman"))
{
  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")
  method <- match.arg(method)
  . . . . .
```

L'argument '...' (dot-dot-dot)

'...' sert à passer des arguments dont le nombre et les noms ne sont pas connus a priori :

```
> c
```

```
function (... , recursive = FALSE) .Primitive("c")
```

```
> args(rm)
```

```
function (... , list = character(0), pos = -1,  
          envir = as.environment(pos), inherits = FALSE)
```

Deux usages principaux du '...' :

```
foo <- function(...) {dots <- list(...); print(dots)}
```

```
foo(1)
```

```
foo(1, 2)
```

```
foo(1, 2, 3)
```

```
foo(1, 2, 3:10) # etc...
```

Passer des arguments d'une fonction à une autre :

```
> myplot <- function(x1, x2, ...) {  
+   matplot(cbind(x1, x2), type = "n")  
+   n <- length(x1)  
+   polygon(c(1:n, n:1), c(x1, rev(x2)), ...)  
+ }  
> x1 <- rnorm(100)  
> x2 <- x1 + runif(length(x1), 3, 5)  
> myplot(x1, x2)  
> myplot(x1, x2, col = "red")  
> myplot(x1, x2, col = "blue", border = NA)
```

‘...’ est très utilisé dans les fonctions génériques :

```
> ?print
```

```
print(x, ...)
```

```
...: further arguments passed to or from other methods.
```

Valeur de retour


Une fonction en R retourne un objet unique ou la valeur `NULL`.

La valeur retournée est soit le résultat de la dernière expression dans la fonction, soit retournée explicitement avec la fonction `return` (qui termine la fonction).

Un objet peut retourner plusieurs objets au cours d'une exécution :

- avec l'opérateur `<<-` (*super assignment*)
- avec la fonction `assign(, pos = 1)`

`invisible` retourne un objet sans qu'il soit imprimé par défaut (exemple : `hist.default`).

 `<<-` ne doit pas être utilisé dans un package pour écrire dans l'espace de travail (*cf.* fonctions récursives).

Environnements et étendue des objets

Les données locales à la fonction sont dans l'***environnement*** de la fonction. Cet environnement est créé à chaque exécution :

```
> f <- function() print(environment())  
> f()  
<environment: 0x8c67c3c>  
> f()  
<environment: 0x8c69ac8>  
> environment()  
<environment: R_GlobalEnv>
```

Les environnements sont des objets...

```
> e <- new.env()  
> e  
<environment: 0x82bfa0c>  
> is.environment(e)  
[1] TRUE
```



... et peuvent être manipulés (*cf.* `?environment` pour les détails).

Un environnement est composé de :

- un cadre (*frame*) avec des noms d'objets (*symbols*),
- un pointeur vers l'environnement parent (*enclosure*).

```
> f <- function() print(parent.env(environment()))  
> f()  
<environment: R_GlobalEnv>
```


Dans un environnement donné, disons e , lorsque $y \leftarrow x$ est exécutée, x est recherché dans e puis, s'il n'est pas trouvé, dans l'environnement parent de e , alors que y est créé dans e (éventuellement effaçant un y pré-existant).

 Avec $y[] \leftarrow x$, y est recherché dans e car '[' est prioritaire sur \leftarrow (*cf.* `?Syntax`).

```
> environment(f) # == .GlobalEnv
<environment: R_GlobalEnv>
> parent.env(environment(f))
<environment: package:stats>
attr(,"name")
[1] "package:stats"
attr(,"path")
[1] "/usr/lib/R/library/stats"
> search()
[1] ".GlobalEnv"          "package:stats"
[3] "package:graphics"    "package:grDevices"
[5] "package:utils"       "package:datasets"
[7] "package:methods"     "Autoloads"
[9] "package:base"
> parent.env(parent.env(environment(f)))
<environment: package:graphics>
attr(,"name")
[1] "package:graphics"
attr(,"path")
```

```
[1] "/usr/lib/R/library/graphics"
```

Quand un objet est cherché, R remonte les environnements parents, puis le *search path*. Cette règle s'appelle ***lexical scoping***.

 L'environnement parent d'une fonction est celui où elle a été définie (et pas celui où elle a été appelée).

```
> environment(lm)  
<environment: namespace:stats>
```

Il est donc souvent avantageux de définir une fonction dans une fonction :

- appel récursif aisé
- modification des objets dans la fonction “mère” (environnement parent)
- pas besoin de la documenter

`get/assign` permettent d'accéder/modifier directement dans un environnement précisé. L'opérateur `'::'` accède directement à un objet dans un package.


```
> log <- 9
> log # == get("log")
[1] 9
> ## get("log", pos = 1) == get("log", env = .GlobalEnv)
> base::log # == get("log", env=as.environment("package:base"))
function (x, base = exp(1))  .Primitive("log")
> ## get("log", pos = 2) ou 3, ...
> log(1)
[1] 0
> log[1]
[1] 9
> letters <- 0
> letters
[1] 0
> base::letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

get permet d'afficher le contenu des opérateurs :

```
> get("+")  
function (e1, e2)  .Primitive("+")
```


Fonctions récursives

Une fonction récursive est une fonction qui peut s'appeler elle-même.

-  Ne pas utiliser pour des fonctions mathématiques récursives simples :
factorielle (`factorial` et `lfactorial` ; attention aux tutos sur Wikipedia),
série de Fibonacci (attention aux benchmarks sur Internet), ...


Fonctions récursives

Une fonction récursive est une fonction qui peut s'appeler elle-même.

-  Ne pas utiliser pour des fonctions mathématiques récursives simples :
factorielle (`factorial` et `lfactorial` ; attention aux tutos sur Wikipedia),
série de Fibonacci (attention aux benchmarks sur Internet), ...
- Utile (voire indispensable) quand il y a des boucles (`for`) emboîtées dont le nombre est indéterminé.

Fonctions récursives


Une fonction récursive est une fonction qui peut s'appeler elle-même.

-  Ne pas utiliser pour des fonctions mathématiques récursives simples : factorielle (`factorial` et `lfactorial` ; attention aux tutos sur Wikipedia), série de Fibonacci (attention aux benchmarks sur Internet), ...
- Utile (voire indispensable) quand il y a des boucles (`for`) emboîtées dont le nombre est indéterminé.

Exemples : simulation d'arbre phylogénétique avec `ape::rtree` ; lister les fichiers dans une archive `.zip` (`?unzip`).

Fonctions récursives

Une fonction récursive est une fonction qui peut s'appeler elle-même.


-  Ne pas utiliser pour des fonctions mathématiques récursives simples : factorielle (`factorial` et `lfactorial` ; attention aux tutos sur Wikipedia), série de Fibonacci (attention aux benchmarks sur Internet), ...
- Utile (voire indispensable) quand il y a des boucles (`for`) emboîtées dont le nombre est indéterminé.

Exemples : simulation d'arbre phylogénétique avec `ape::rtree` ; lister les fichiers dans une archive `.zip` (`?unzip`).

- La fonction récursive est généralement incluse dans une autre fonction qui est appelée par l'utilisateur.

Fonctions récursives

Une fonction récursive est une fonction qui peut s'appeler elle-même.

-  Ne pas utiliser pour des fonctions mathématiques récursives simples : factorielle (`factorial` et `lfactorial` ; attention aux tutos sur Wikipedia), série de Fibonacci (attention aux benchmarks sur Internet), ...
- Utile (voire indispensable) quand il y a des boucles (`for`) emboîtées dont le nombre est indéterminé.

Exemples : simulation d'arbre phylogénétique avec `ape::rtree` ; lister les fichiers dans une archive `.zip` (`?unzip`).

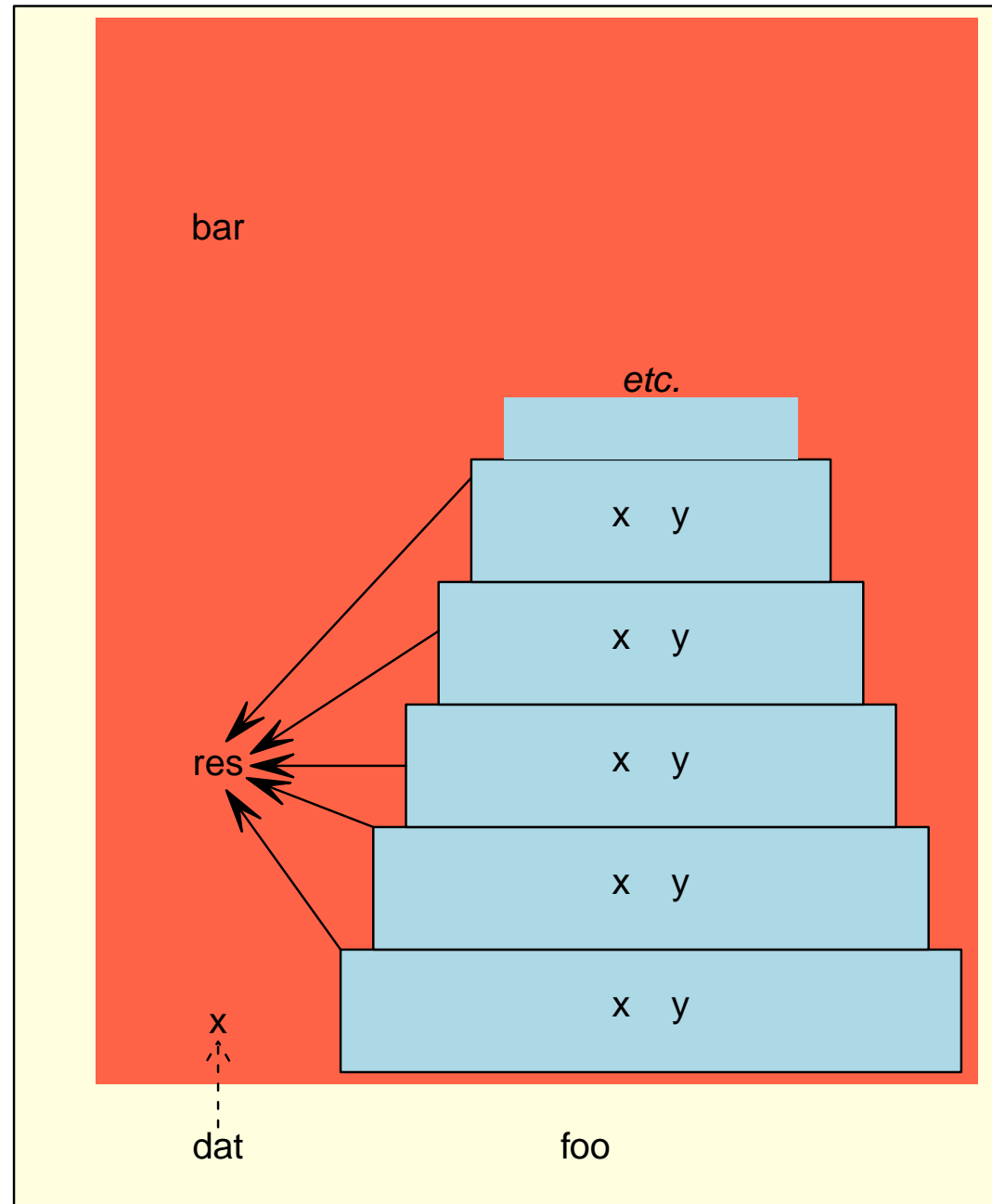
- La fonction récursive est généralement incluse dans une autre fonction qui est appelée par l'utilisateur.
- La récursion est contrôlée par une commande conditionnelle (`if` ou autre) pour éviter des récursions infinies.

Code

```
foo <- function(x) {  
  bar <- function(x) {  
    y <- ..... # des calculs  
    res <-<- c(res, ....  
    if (.....) bar(y)  
  }  
  res <- numeric() # selon le mode  
  bar(x)  
  res  
}  
  
dat <- ....  
foo(dat)
```



En mémoire



Fonctions génériques et méthodes (S3)

Une fonction ***générique*** redirige (*dispatch*) l'appel vers une fonction appropriée appelée *method* :

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x37a2108>
<environment: namespace:base>
```

`print(x)` va appeler la fonction `print.class` où `class` est la classe de `x`. Toutes les méthodes disponibles pour une générique donnée sont affichées avec la fonction `methods` :

```
> methods(print)
[1] print.acf*
[2] print.anova*
```

.....

Si `x` n'a pas de classe ou si la méthode correspondante n'existe pas, `print.default` est utilisée. Si cette dernière n'existe pas, une erreur est retournée.

Si `x` est de classe `c("cl1", "cl2", etc...)`, `print(x)` appelle `print.cl1`, puis, si `print.cl1` est introuvable, `print.cl2`, etc..., jusqu'à `print.default`. `NextMethod` passe directement à la méthode de la classe suivante.

Cela peut être utile pour bénéficier de méthodes déjà écrites. Par exemple, si l'on crée une classe de distances (symétriques) `"mydist"`, `print` utilisera `print.default` si `print.mydist` n'existe pas. Par contre si la classe est définie comme `c("mydist", "dist")`, `print.dist` sera utilisée. (On dit que `"mydist"` *hérite* de la classe `"dist"`.)

Les principales fonctions génériques sont `print`, `summary` et `plot`.

Écrire une méthode nécessite de faire attention à plusieurs points :

- Tous les arguments de la générique doivent être inclus *dans le même ordre*, y compris ‘...’ s’il est dans la générique.
- Une méthode doit être documentée ainsi :

```
\method{print}{titi}(x, ...)
```

qui sera imprimé dans la doc :

```
## S3 method for class 'titi' :
```

```
print(x, ...)
```

Beaucoup d’opérateurs de R sont des fonctions génériques.

```
> methods("-")
```

```
[1] -.Date      -.POSIXt
```

```
> methods("[")
```

```
[1] [.acf*           [.AsIs             [.bibentry*
```

```
[4] [.data.frame     [.Date             [.difftime
```

```
[7] [.Dlist          [.factor           [.formula*
```

```
[10] [.getAnywhere*   [.hexmode          [.listof
```

```
[13] [,nonStructure-method [.noquote      [.numeric_version
[16] [.octmode              [.pdf_doc*     [.person*
[19] [.POSIXct              [.POSIXlt     [.raster*
[22] [.roman*               [.simple.list [.table
[25] [.terms*               [.ts*      [.tskernel*
[28] [.warnings
```

see `'?methods'` for accessing help and source code

Il est donc possible de construire un opérateur `[]` qui respecte la classe :

```
> get(".Date")
function (x, ..., drop = TRUE)
{
  cl <- oldClass(x)
  class(x) <- NULL
  val <- NextMethod("[")
  class(val) <- cl
  val
}

> library(ape)
> get(".multiPhylo")
function (x, i)
{
  oc <- oldClass(x)
  class(x) <- NULL
  structure(x[i], TipLabel = attr(x, "TipLabel"), class = oc)
}
```


Exercices II

1. Écrire des méthodes `print`, `summary` et `plot` pour la structure créée au I.
2. Écrire une fonction `fit.mixture` qui ajuste un modèle de mixture de deux distributions exponentielles par maximum de vraisemblance. La fonction de vraisemblance sera maximisée avec `nlm`. On définira notamment une fonction `dev` qui calculera la déviance (rappel : $dev = -2 \ln L$). Où sera placée `dev` par rapport à `fit.mixture` ? Justifier votre choix.
3. Modifier `fit.mixture` afin de pouvoir contrôler l'optimisation depuis cette fonction.

III Les expressions

Toutes les commandes lues par R sont transformées en **expression** : c'est l'action de *parsing*. Les expressions syntactiquement correctes sont ensuite exécutées.

Il est possible (et utile) de pouvoir construire une expression sans qu'elle soit exécutée.

- trouver des dérivées avec `deriv` ou `D` :

```
> D(expression(log(x)), "x")
```

```
1/x
```

- ajouter des annotations mathématiques sur un graphe (`?plotmath`) ;

Une expression se crée avec `expression` ou `parse`.

```
1. > e <- expression(rnorm(5))
> e
expression(rnorm(5))
> eval(e)
[1] -0.4482620  1.9125654  0.5736916 -0.8275439  0.5217176
```

2. parse :

(a) par défaut tapée dans la console :

```
> parse()
?1 + 3
expression(1 + 3)
```

(b) depuis un fichier avec du code R avec `parse(file =)`

(c) des chaînes de caractères converties avec `parse` et l'option `text` :

```
> a <- paste(1, 2, sep = " + ")
> e2 <- parse(text = a)
> eval(e2)
[1] 3
```

Une expression est un objet constitué comme une liste :

```
> e3 <- expression(x <- rnorm(5))
> e3[2] <- expression(y <- runif(5))
> rm(x, y)
> eval(e3)
> x
[1]  1.2714243 -0.8285636  1.1887390 -1.9306594  0.8050333
> y
[1] 0.2794713 0.9192962 0.1273966 0.8500026 0.4434050
```

Mais bizarrement :

```
> is.list(e3)
[1] FALSE
> is.vector(e3)
[1] TRUE
```

On notera la nuance :

```
> str(e3[1])  
  expression(x <- rnorm(5))  
> str(e3[[1]])  
  language x <- rnorm(5)
```

L'opérateur '[' permet d'accéder aux éléments élémentaires du langage :

```
> for (i in 1:3) print(e3[[1]][[i]])  
'<- '  
x  
rnorm(5)  
> str(e3[[1]][[3]][[1]])  
  symbol rnorm  
> str(e3[[1]][[3]][[2]])  
  num 5
```

Pour construire un élément du langage (*statement*), on utilisera `quote` à la place de `expression` :

```
> str(quote(rnorm(2)))  
  language rnorm(2)  
> str(expression(rnorm(2)))  
  expression(rnorm(2))  
  
> mode(expression(rnorm(2)))  
[1] "expression"  
> mode(quote(rnorm(2)))  
[1] "call"  
> mode(expression(rnorm(2))[[1]])  
[1] "call"
```

La différence entre les deux est subtile :

```
> eval(quote(1:5))
```

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

```
> eval(expression(1:5))
```

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

substitute

`substitute` permet de spécifier l'environnement où R doit chercher les objets nommés dans une expression.

Cette expression doit être de mode `"call"` et n'est pas évaluée.

`substitute` a peu d'utilité en dehors d'une fonction.

```
> foo <- function(x)
+   cat("argument:", deparse(substitute(x)), "\n")
> foo(E)
argument: E
```



`deparse` est indispensable si `x` n'est pas un nom simple :

```
> fooB <- function(x) cat("argument:", substitute(x), "\n")
> fooB(E)
argument: E
```



```
> fooB(x + 1)
argument: Error in cat(list(...), file, sep, fill, labels, append=FALSE,
  argument 2 (type 'language') cannot be handled by 'cat'
> foo(x + 1)
argument: x + 1
```

```
> plot.default
.....
      xlabel <- if (!missing(x))
        deparse(substitute(x))
.....
```

`substitute` est utile pour ajouter des annotations sur un graphe « à la volée » (*R pour les débutants*, p. 45).

Exercices III

1. Soit la fonction

$$f(x) = \frac{1}{1+x^3} \ln(2x+10).$$

Calculer $\partial f / \partial x$ pour $x = 1, 2, \dots, 100$. Quelle utilité pratique peut-on faire de ce genre de calculs ?

2. Créer et initialiser des listes nommées L_1, L_2, \dots, L_n telles que L_n a n éléments.

IV Accéder au système d'exploitation

Les listes `.Platform`, `R.version` et `.Machine` et les fonctions `Sys.info()` et `Sys.getenv()` (retournant des vecteurs avec noms) donnent des informations sur le système.

```
> .Platform$OS.type
[1] "unix"
> R.version$platform
[1] "x86_64-pc-linux-gnu"
> .Machine$double.eps
[1] 2.220446e-16
> Sys.info()["user"]
      user
"paradis"
> Sys.getenv()[c("HOME", "R_HOME")]
HOME           /home/paradis
R_HOME         /usr/lib/R
```

`system` exécute une commande système passée avec une chaîne de caractères.

`unlink` efface un fichier.

```
> apropos("^file")
[1] "file"          "file.access"   "file.append"
[4] "file.choose"   "file.copy"     "file.create"
[7] "file.edit"     "file.exists"   "file.info"
[10] "file.path"     "file.remove"   "file.rename"
[13] "file.show"     "file.symlink"  "file_test"
```

Pour accéder à un fichier dans l'installation de R :

```
> system.file()
[1] "/usr/lib/R/library/base"
> system.file(package = "ape")
[1] "/usr/local/lib/R/site-library/ape"
> system.file("data/woodmouse.rda", package = "ape")
[1] "/usr/local/lib/R/site-library/ape/data/woodmouse.rda"
```

Pour utiliser un fichier ou un répertoire temporaire :

```
> tempfile()  
[1] "/tmp/Rtmp4WZzp7/file778557cc"  
> tempdir()  
[1] "/tmp/Rtmp4WZzp7"
```

V Débogage

`browser()` peut être inséré dans une fonction dont l'exécution est alors interrompue, l'utilisateur pouvant examiner l'environnement interne à la fonction.

```
> fix(plot.default) # insérer browser() après "localTitle <- ..  
> plot(rnorm(10))  
Called from: plot.default(rnorm(10))  
Browse[1]> ls()  
[1] "ann"          "asp"          "axes"         "frame.plot"  
[5] "localAxis"    "localBox"     "localTitle"   "localWindow"  
[9] "log"          "main"         "panel.first"  "panel.last"  
[13] "sub"          "type"         "x"            "xlab"  
[17] "xlim"         "y"            "ylab"         "ylim"  
Browse[1]>  
>
```

Commandes disponibles : `c`, `where`, `help`, `n`, `f`, `s`, `r`, `Q`

debug exécute pas-à-pas la fonction avec un nouveau browser.

```
> debug(plot.default)
```

```
> plot(rnorm(10))
```

```
.....
```

Les commandes `c` et `n` sont ici différentes.

Débogage post-mortem

Un exemple très simple :

```
> foo <- function(x) {  
+   x <- log(x)  
+   if (anyNA(x)) stop("NA's not allowed")  
+   sum(x)  
+ }
```

Juste après une erreur, `traceback()` reconstitue la série d'appels de fonctions.

L'option globale `"error"` contrôle le comportement de R quand une erreur se produit. Par défaut :

```
> options("error")  
$error  
NULL
```


Deux possibilités :

1. `options(error = recover)` permet de parcourir les environnements ouverts successivement jusqu'à l'erreur.
2. `options(error = dump.frames)` sauvegarde un objet nommé `last.dump` dans l'environnement global qui peut être parcouru avec la fonction `debugger()`.

 `traceback()`, `recover()` et `debugger()` donnent des informations sur la *dernière* erreur.

Un exemple (un peu) plus compliqué :

```
> x <- 1:100
> y <- 2*x + rnorm(100, 0, 10)
> mod <- lm(y ~ x)
> newx <- 101:110
> predict(mod, newx)
Error in eval(predvars, data, env) :
  numeric 'envir' arg not of length one
```

VI Profilage et optimisation

Quelques règles simples

1. Éviter les boucles “stupides”

trivial :

```
> x <- numeric(1e6)
> system.time(for (i in 1:1e6) x[i] <- 1)
  user  system elapsed 
0.044    0.000    0.045 
> system.time(x[] <- 1)
  user  system elapsed 
0.004    0.000    0.002
```

un peu moins trivial :

```
> y <- x <- rnorm(1e6)
> system.time(for (i in 1:1e6) if (x[i] < 0) x[i] <- 0)
  user  system elapsed 
0.048   0.000   0.048 
> system.time(y[y < 0] <- 0)
  user  system elapsed 
0.004   0.000   0.004
```

2. Dimensionner les objets a priori (si possible)



```
x <- NULL
for (i in 1:n) {
  y <- .....
  x <- c(x, y)
}
```

```
X <- NULL
X <- rbind(X, y)
```



```
x <- numeric(n)
for (i in 1:n) {
  y <- .....
  x[i] <- y
}
```

```
X <- matrix(NA, n, p)
X[i, ] <- y
```

3. Utiliser outer

Exemple : construire une matrice $W_{n \times n}$ telle $w_{ij} = 1$ si $x_i = x_j$ ($i \neq j$), 0 sinon.

```
> f <- function(x) {  
+   n <- length(x)  
+   w <- matrix(0, n, n)  
+   for (i in 1:(n - 1)) for (j in (i + 1):n)  
+     if (x[i] == x[j]) w[i, j] <- w[j, i] <- 1  
+   w  
+ }  
  
> x <- sample(5, size = 1000, replace = TRUE)  
> system.time(f(x))  
   user  system elapsed  
0.044    0.000    0.042  
  
> fo <- function(x) {  
+   w <- outer(x, x, "==")
```

```

+   mode(w) <- "numeric"
+   diag(w) <- 0
+   w
+ }
> system.time(fo(x))
      user  system elapsed
 0.012    0.004    0.016

```

Exemple : idem mais $w_{ij} = 1$ si $x_i = x_j$ et $y_i \neq y_j$, 0 sinon.

```

fo2 <- function (x, y)
{
  d <- outer(x, x, "==") & outer(y, y, "!=")
  diag(d) <- 0
  d
}

> args(outer)
function (X, Y, FUN = "*", ...)

```

`outer(x, y)`, `outer(x, y, "*")` et `x %o% y` sont identiques (tables de multiplication).


4. Préférer l'indexation numérique à l'indexation avec les noms

Trier des nombres est beaucoup plus rapide que trier des chaînes de caractères. De plus les `names` consomment beaucoup de mémoire.

Peut être important s'il y a plusieurs "structures" associées.

Dans une fonction : arranger et/ou ordonner les données au début du code (éviter de rechercher les correspondances entre objets dans la suite du code).

5. Comme pour tous les langages de programmation...

- éviter les répétitions → objets temporaires
- éviter les boucles surchargées
- simplifier les formules mathématiques
- éviter les calculs inutiles ($1/4 \rightarrow 0.25$)
- arranger le code de façon économique
-  Mettre des commentaires

“Déclasser” (pour surclasser)

La recherche des “methods” peut ralentir fortement les calculs si l’indexation est utilisée dans une boucle (ce qui est souvent le cas). Pourquoi ?

```
> methods ("[")  
[1] [.acf*                [.AsIs  
[3] [.bibentry*           [.data.frame  
....
```

Exemple : calculer une statistique sur les colonnes d’un tableau x :

```
> f <- function(x) {  
+   p <- ncol(x)  
+   res <- numeric(p)  
+   for (i in 1:p) res[i] <- sum(x[, i])  
+   res  
+ }
```


La fonction `g` est une version de `f` où l'argument `x` est “déclassé” :

```
> DF <- as.data.frame(matrix(rnorm(1e7), 1e3, 1e4))
> system.time(of <- f(DF))
  user  system elapsed
0.144   0.000   0.147
> system.time(og <- g(DF))
  user  system elapsed
0.052   0.000   0.052
> identical(of, og)
[1] TRUE
```

La différence sera très souvent plus importante (facteur 10 ou plus).

À vous de trouver le code de `g`.

Profilage

Les outils de profilage de R permettent de savoir :

- le temps passé sur chaque fonction au cours d'une exécution
- la consommation de mémoire
- les allocations de mémoire

`Rprof()` enregistre à intervalle régulier (par défaut 0,02 sec) la fonction en cours d'exécution.

```
Rprof()  
## some R codes...  
Rprof(NULL)  
summaryRprof()
```

```
Rprof(memory.profiling = TRUE)  
## some R codes...  
Rprof(NULL)
```

```
summaryRprof(memory = "both")  
summaryRprof(memory = "tseries")  
summaryRprof(memory = "stats")
```

En pratique, utiliser `system.time` au cours de la construction d'une fonction afin de comparer des solutions alternatives. `Rprof` est plutôt utile une fois la fonction terminée.



Ne perdez pas de vue les (futures) applications de vos codes.



N'oubliez pas l'ordre de priorité :

1. précision
2. performance
3. compromis précision/performance (ex. : optimisation et convergence)

Opérations logiques

Opérateurs de comparaison :

➤ ==

➤ !=

➤ <

➤ >

➤ <=

➤ >=

Comparaison de valeurs numériques et de caractères :

```
> LETTERS > letters
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  
[14] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Opérations sur les vecteurs logiques :

- `&`
- `|`
- `!`
- `all`
- `any`
- `xor`

```
> x <- rnorm(1e6)
> y <- rnorm(1e6)
> any(x > 5)
[1] TRUE
> any(y > 5)
[1] TRUE
> any(x > 6)
[1] FALSE
> any(y > 6)
[1] FALSE
```

```
> any(x > 5 & y > 5)
[1] FALSE
```

Dans une fonction, plus efficace de faire :

```
> test.x <- x > 5
> test.y <- y > 5
> any(test.x & test.y)
[1] FALSE
```

 Les opérateurs `&&` et `||` ne sont utilisés que dans `if (...)` ou `while (...)`.

Fonctions de test : `is.na`, `is.numeric`, ...

```
> which(test.x)
[1] 514908
> which(test.y)
```

```
[1] 843452
```

Les valeurs logiques sont codées en interne avec 0 (FALSE) et 1 (TRUE) :

```
> system.time(tabx <- table(test.x))
```

utilisateur	système	écoulé
0.182	0.012	0.194

```
> tabx
```

```
test.x
```

FALSE	TRUE
999999	1

```
> system.time(tabx2 <- tabulate(test.x + 1L))
```

utilisateur	système	écoulé
0.003	0.004	0.006

```
> tabx2
```

```
[1] 999999      1
```

`tabulate` est une version plus simple et plus rapide de `table` (univariée uniquement) :

```
> args(tabulate)
function (bin, nbins = max(1L, bin, na.rm = TRUE))
```

Deux autres possibilités (très efficaces) :

```
> length(which(test.x))
[1] 1
> sum(test.x)
[1] 1
```


Exercices VI

1. Reprendre la matrice de voisinage w calculée avec f_0 . Calculer

$$\sum_{i=1}^n \sum_{j=1}^n w_{ij} x_i x_j$$

Écrire deux fonctions, l'une triviale et l'autre optimale, pour effectuer ce calcul et comparer leur performance.

VII Interface R/C

Interface simple (C et Fortran)

La structure interne (invisible à l'utilisateur) d'un objet est composée de :

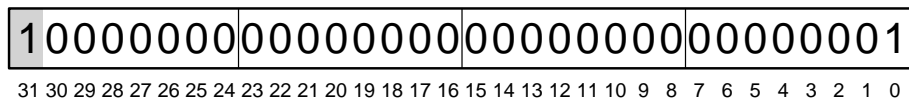
- un nom (symbole),
- les attributs intrinsèques (mode et longueur),
- un pointeur vers les attributs supplémentaires,
- des informations non accessibles (comme la génération de l'objet pour le gestionnaire de mémoire),
- un pointeur vers les données.

Un vecteur atomique dans R est un tableau à une dimension dans C (*1-d array*).

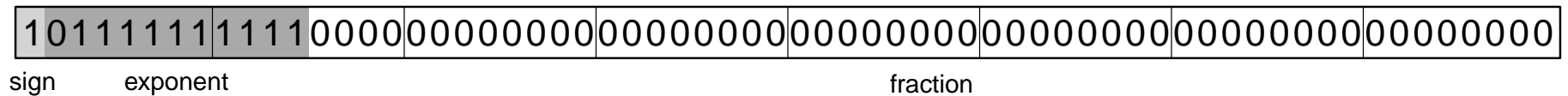
`integer` **et** `double` **doivent** être distingués explicitement (et souvent convertis **avec** `as.integer` **ou** `as.double`).

R ne connaît que deux types de données numériques :

32-bit integer



64-bit floating-point real



Les correspondances de format de données entre R, C, et Fortran sont dans ? .C.

Exemple : faire une somme avec le fichier **sum_C.c** :

```
$ R CMD SHLIB sum_C.c
```

```
> dyn.load("sum_C.so") # .dll sous Windows
```

```
> is.loaded("sum_C")
```

```
[1] TRUE
```

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

```
> .C("sum_C", x, as.integer(10), double(1))
```

```
[[1]]
```

```
[1] 0.1256448 0.2434552 -0.8610338 0.1535991 -1.4853595
```

```
[6] 0.5544852 -0.6986275 -1.0625369 0.2406516 2.6174271
```

```
[[2]]
```

```
[1] 10
```

```
[[3]]
```

```
[1] -0.1722947
```

```
> sum(x)
[1] -0.1722947
```

Fichier **sumF.f** :

```
$ R CMD SHLIB sumF.f
```

```
> dyn.load("sumF.so")
> .Fortran("sumF", x, as.integer(10), double(1))
[[1]]
[1] 0.1256448 0.2434552 -0.8610338 0.1535991 -1.4853595
[6] 0.5544852 -0.6986275 -1.0625369 0.2406516 2.6174271

[[2]]
[1] 10

[[3]]
[1] -0.1722947
```

Interface complexe (C uniquement)

Fichier **sum_Call.c** :

```
R CMD SHLIB sum_Call.c
```

```
> dyn.load("sum_tot_call.so")  
> .Call("sum_tot_call", x)  
[1] -0.1722947
```

Avantages respectifs :

.C : simplicité

.Call : les données ne sont pas dupliquées ; tous les types d'objets de R peuvent être manipulés ; vecteurs longs supportés

Indexation R vs. C


	R	.C	.Call ¹
vecteur	<code>n <- length(x)</code>	²	<code>n = LENGTH(x);</code>
	<code>x[1]</code>	<code>x[0]</code>	<code>xp[0]</code>
	<code>x[n]</code>	<code>x[n - 1]</code>	<code>xp[n - 1]</code>
	<code>x[i] i=1...n</code>	<code>x[i] i=0...n-1</code>	<code>xp[i] i=0...n-1</code>
matrice	<code>n <- nrow(x)</code>	²	<code>n = nrow(x);</code>
	<code>p <- ncol(x)</code>	²	<code>p = ncol(x);</code>
	<code>x[1, 1]</code> ³	<code>x[0]</code>	<code>xp[0]</code>
	<code>x[n, p]</code> ⁴	<code>x[n * p - 1]</code>	<code>xp[n * p - 1]</code>
	<code>x[i, j]</code>	<code>x[i + n * j]</code>	<code>xp[i + n * j]</code>
	<code>i=1...n</code>	<code>i=0...n-1</code>	<code>i=0...n-1</code>
	<code>j=1...p</code>	<code>j=0...p-1</code>	<code>j=0...p-1</code>


¹ `xp` est un pointeur vers `x`

² doit être passé comme argument avec `.C`

³ identique à `x[1]`

⁴ identique à `x[n * p]` ou `x[length(x)]`

 Si le code C retourne un indice i , ne pas oublier de faire $i + 1$ dans R.

 Les indices hors limites sont valides en R, mais pas en C : ajouter des contrôles si risque de dépassement de ces limites.

Vecteurs longs

Si votre machine est équipée d'un processeur 64-bit :

```
> storage.mode(length(integer(1e9)))
```

```
[1] "integer"
```

```
> storage.mode(length(integer(3e9)))
```

```
[1] "double"
```

R ne connaît que deux types de données numériques : “signed 32-bit integer” et “64-bit floating point”. La plus longue “length” *entière* est donc :


```
> .Machine$integer.max  
[1] 2147483647
```

La longueur d'un vecteur long est donc stockée avec un "64-bit floating point" (= double).

```
SEXP rawStreamToDNABin(SEXP x)  
{  
    . . . .  
    PROTECT(x = coerceVector(x, RAWSXP));  
    double N = XLENGTH(x);  
    for (long i = 1; i < N; i++) {  
        . . . .  
    }
```

XLENGTH est remplacé par LENGTH sur les plateformes 32-bit.

C++ avec le package Rcpp

Fichier **sumRcpp.cpp** :

```
> library(Rcpp)
> sourceCpp("sumRcpp.cpp")
> ls()
[1] "sumRcpp"      "sumRcppBis"
> sumRcpp
function (x)
.Primitive(".Call")(<pointer: 0x7f7e31b8bf20>, x)
```

VIII Construire un package

- ./pkgModule5/DESCRIPTION
- ./pkgModule5/NAMESPACE
- ./pkgModule5/man/
- ./pkgModule5/R/Rmodule3.R
- ./pkgModule5/src/sum_C.c
- ./pkgModule5/src/sum_Call.c
- ./pkgModule5/src/sumF.f
- ./pkgModule5/src/sumRcpp.cpp
- ./pkgModule5/src/pkgModule5.c

Le sous-répertoire src/ n'est pas nécessaire si le package n'inclut pas de codes à compiler.

Note : si des routines de BLAS ou LAPACK sont utilisées, il est nécessaire d'ajouter `./pkgModule5/src/Makevars` avec cette ligne :

```
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)
```

DESCRIPTION :

Package: pkgModule5

Version: 0.1

Date: 2017-12-08

Title: Package du Module 5

Author: Emmanuel Paradis <Emmanuel.Paradis@ird.fr>

Maintainer: Emmanuel Paradis <Emmanuel.Paradis@ird.fr>

LinkingTo: Rcpp

Description: pkgModule5.....

License: GPL (>= 2)

pkgModule5.R :

```
sumC <- function(x)
```

```
.C("sum_C", as.double(x), length(x), double(1),  
    PACKAGE = "Rmodule3")[[3]]
```

```
sumF <- function(x)  
  .Fortran("sumF", as.double(x), length(x), double(1),  
           PACKAGE = "Rmodule3")[[3]]
```

```
sumCall <- function(x)  
  .Call("sum_Call", as.double(x), PACKAGE = "Rmodule3")
```

NAMESPACE :

```
useDynLib(pkgModule5, .registration = TRUE)  
export(sumC, sumF, sumCall, sumCpp)
```

```
$ R CMD INSTALL pkgModule5 # as root
```

Temps de calcul datant de 2008 :

```
> x <- rnorm(1e7)
> system.time(sum(x))
  user  system elapsed
0.036   0.000   0.037
> system.time(sumC(x))
  user  system elapsed
0.216   0.160   0.378
> system.time(sumF(x))
  user  system elapsed
0.248   0.148   0.396
> system.time(sumCall(x))
  user  system elapsed
0.096   0.000   0.096
> system.time(sumC(x)) # avec DUP = FALSE...
  user  system elapsed
0.128   0.000   0.126
> system.time(sumF(x)) # avec DUP = FALSE...
```

```
      user  system elapsed
0.144    0.000    0.144
> system.time(sumC(x)) # ... et NAOK = TRUE
      user  system elapsed
0.048    0.000    0.045
> system.time(sumF(x)) # ... et NAOK = TRUE
      user  system elapsed
0.064    0.000    0.063
```

Documenter un package

Exemple : DNABin.Rd dans ape

```
\name{DNABin}
\alias{DNABin}
\alias{print.DNABin}
\alias{summary.DNABin}
\alias{[.DNABin}
\alias{rbind.DNABin}
\alias{cbind.DNABin}
\alias{as.matrix.DNABin}
\title{Manipulate DNA Sequences in Bit-Level Format}
\description{
  These functions help to manipulate DNA sequences [etc...]
}
\usage{
\method{print}{DNABin}(x, \dots)
\method{summary}{DNABin}(object, printlen = 6, digits = 3, \dots)
\method{rbind}{DNABin}(\dots)
\method{cbind}{DNABin}(\dots, check.names = TRUE)
\method{[.DNABin}(x, i, j, drop = TRUE)
\method{as.matrix}{DNABin}(x, \dots)
}
\arguments{
  \item{x, object}{an object of class \code{"DNABin"}}
  \item{\dots}{either further arguments to be passed to or from other
```



```

    methods in the case of \code{print}, \code{summary}, and
    \code{as.matrix}, or a series of objects of class \code{"DNABin"} in
    the case of \code{rbind} and \code{cbind}.)
\item{printlen}{the number of labels to print (6 by default).}
\item{digits}{the number of digits to print (3 by default).}
\item{check.names}{a logical [etc...] (see details).}
\item{i, j}{indices of the rows and/or columns [etc...]}
\item{drop}{logical; if \code{TRUE} [etc...]}
}
\details{
  bla bla [etc...]
}
\value{
  an object of class \code{"DNABin"} in the case of \code{rbind},
  \code{cbind}, and \code{[]}.
}
\author{Emmanuel Paradis \email{Emmanuel.Paradis@mpl.ird.fr}}
\seealso{
  \code{\link{as.DNABin}}, \code{\link{read.dna}},
  \code{\link{read.GenBank}}, \code{\link{write.dna}}

  The corresponding generic functions are documented in the package
  \pkg{base}, e.g., \code{\link[base]{print}}
}
\examples{
data(woodmouse) [etc...] # Les exemples sont utiles !
}
\keyword{manip}

```

Sections optionnelles : `\references`, `\note`

Pour démarrer à partir de la fonction (mais les méthodes ne sont pas formatées correctement) :

```
> prompt(sumC)
```

```
Created file named 'sumC.Rd'.
```

```
Edit the file and move it to the appropriate directory.
```

Une fois tous les objets R (non cachés) documentés :

```
R CMD build pkgModule5
```

```
R CMD check pkgModule5
```



Pour soumettre un package à CRAN, il doit passer le check *sans warnings*.