Université Nice Sophia Antipolis
Faculté des Sciences

Programmation avec R - Hiver 2015
Arnaud Malapert & Jean-Paul Roy
L2 MASS - Faculté des Sciences

http://www.i3s.unice.fr/~malapert/org/teaching/introR.html

Modules. Polynômes.

library()

Les modules de R

• Le nom d'un fichier R se termine par .R et ne contient que des lettres minuscules, des chiffres et des tirets. Aucun espace !

essai2.R essai_tortue.R ~~Essai tortue.R~~

• Un module est un fichier xxxxxx.R écrit en R et contenant :
  - des définitions
  - des instructions (par exemple d'affichage)

• Un module peut être destiné :

- à être directement **exécuté**. Lorsqu'il est court et effectue une action ré-utilisable, on parle souvent d'un ***script***.

ou
et

- à être **utilisé** par un autre module. Il exporte alors un certain nombre de fonctionnalités.

Un exemple de script

• Un programme R peut être exécuté sur la ligne de commande du système d'exploitation : Unix (Mac, Linux) ou Windows.

• Exemple au Terminal Unix de MacOS-X :

fichier monscript.R

```
#!/usr/bin/Rscript

cat("Hello World !\n")
version
```

chemin vers Rscript

} *instructions*

on le rend exécutable

on l'exécute

```
$ chmod u+x monscript.R
$ ./monscript.R
Hello World !
[1] "x86_64"
```

Extension ou paquet de R

• Les paquets fournissent un mécanisme pour charger à la demande du code, des données et leur documentation.

• Un paquet est contenu dans un sous-répertoire d'un répertoire définissant une bibliothèque de paquets.

• Les paquets ont leur propre espace de noms : les variables peuvent être ou ne pas être visibles pour les utilisateurs. Ainsi, l'utilisateur peut introduire une variable dans son propre espace de noms, qui n'aura rien à voir avec la variable d'un paquet de même nom.

• L'existence d'espaces de noms permet de protéger le programmeur de conflits entre des variables de même nom situées dans des modules distincts. La sécurité avant tout !

• L'écriture d'un paquet R n'est pas au programme, nous nous limiterons a l'utilisation de la commande source.

• Par contre, vous devez retenir comment installer et charger un paquet standard de R.

```
install.packages("TurtleGraphics")
library("TurtleGraphics")
```

Un module polynôme en R

Représentation des polynômes

• Si les polynômes ont beaucoup de coefficients ≠ 0, on opte pour une représentation dense avec des listes de réels :

$2x^4 - 7x^3 - 3x + 5$ ➡ c(5, -3, 0, -7, 2)

N.B. Suivant les puissances croissantes !
Ainsi le coefficient de x^i sera p[i+1]...

• Si au contraire les polynômes ont beaucoup de coefficients = 0, on opte pour une représentation creuse avec des listes de monômes :

$2x^{100} - 3$ ➡ list(c(2, 100), c(-3, 0))

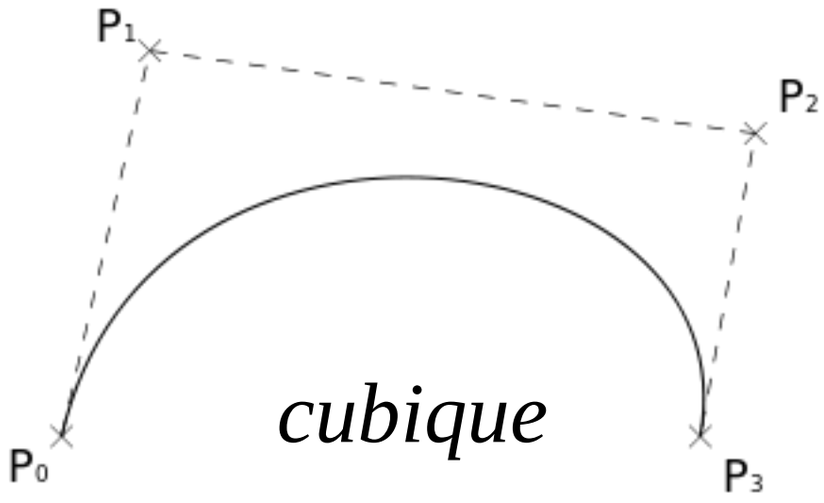
un "monôme"
[coefficient, exposant]

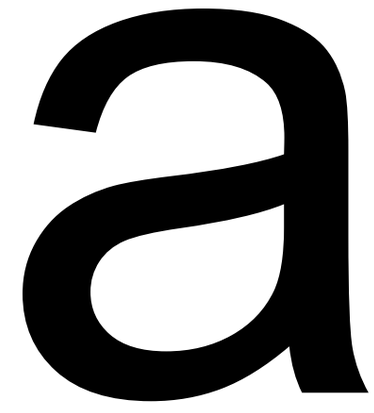
N.B. Suivant les puissances décroissantes !

Un module de polynôme

• Les polynômes sont des objets mathématiques très importants, pour les approximations de fonctions, le graphisme, etc.

• Les caractères par exemple sont dessinés grâce à des polynômes de degré 2 ou 3 (courbes de Bézier).





• Nous allons développer en détail un module polycreux.R contenant les définitions des opérations élémentaires de l'algèbre des polynômes (en représentation creuse).

Définition d'un polynôme

• Un polynôme non nul sera représenté par une liste de monômes (non nuls), suivant les puissances strictement décroissantes.

• En maths :

$$p = 2x^5 - x^4 - 2x$$

$$q = x^4 + 7x^2 - 1$$

• En R:

```
p <- list(c(2,5), c(-1,4),c(-2,1))
q <- list(c(1,4), c(7,2),c(-1,0))
```

• Le polynôme nul sera représenté par une liste vide.

```
is_poly0 <- function(p) length(p) == 0
```

• La fonction poly2str renverra une chaîne de caractères représentant le polynôme.

```
> poly2str(p)
 [1] "2*X^5 + -1*X^4 + -2*X^1"
```

Degré et coefficient d'un polynôme

• Le degré d'un monôme [c,e] est son second élément. Le degré d'un polynôme est celui de son monôme de plus haut degré (en tête).

```
degre <- function(p) {
      if(is_poly0(p)) {return(-Inf)}
      if (is.list(p)) {p <- p[[1]]}
      if (is.vector(p) && length(p) == 2) {return(p[2])}
      else {return(NULL)}
}
```

```
> p[[0]]
[1] 2 5
```

⇒

```
> degre(p[0])
[1] 5
```

⇒

```
>>> degre(p)
5
```

• Le coefficient d'un monôme c(c,e) est son premier élément. Le coefficient (dominant) d'un polynôme est celui de son monôme de tête.

Addition par insertion des monômes

• Commençons par le cas simple : monôme + polynôme. Soit à insérer le monôme cx^e dans le polynôme p, à sa place !

• Comprenons-nous bien : il ne s'agit pas de *modifier* le polynôme p en lui ajoutant cx^e, mais de *construire* le nouveau polynôme cx^e + p.

• Il s'agit d'un algorithme d'insertion :

insérer $5x^3$ dans $p = 2x^5 - x^4 - 2x$

insérer [5,3] dans list(c(2,5),c(-1,4),c(-2,1))

• On cherche donc le premier monôme de degré d ≤ e

• S'il existe déjà un monôme de degré e, attention !

• Pour additionner p1 et p2, il suffit d'additionner chaque monôme de p1 au polynôme p2.

Addition par tri, puis fusion des monômes

• Pour additionner p1 et p2, il suffit d'additionner chaque monôme de p1 au polynôme p2.

1. Concaténer les listes représentant les deux polynômes.
2. Trier la nouvelle liste obtenu par degré décroissant.
3. Fusionner les monômes de même degré.
4. Supprimer les monômes dont le coefficient est nul.

• Remarque : les étapes 3 et 4 de l'algorithme peuvent être réalisé en une seule passe.

• Cet algorithme est robuste puisqu'il permet de sommer un nombre quelconque de polynôme sans supposer qu'il soit bien formé (monômes triés par degré décroissant et sans doublons).

• Cet algorithme peut être amélioré en s'inspirant du tri fusion en supposant que les monômes sont triés par degré décroissant.

Soustraction : polynôme - polynôme

- Il est bien connu qu'une soustraction n'est qu'une addition !

$$p - q = p + (-1) * q$$

- Programmons la multiplication kp d'un polynôme p par un scalaire k.

Il s'agit de la loi externe d'un espace vectoriel :

```
mult_ext <- function(p, k)  lapply(p, function (x) {c(k*x[1], x[2])})

sub <- function(p,q) add(p,mult_ext(q,-1))
```

```
>  poly2str(p)
[1] "2*X^5 + -1*X^4 + -2*X^1"
>  poly2str(mult_ext(p,-2))
[1] "-4*X^5 + 2*X^4 + 4*X^1“
> poly2str(sub(p,q))
[1] "2*X^5 + -2*X^4 + -7*X^2 + -2*X^1 + 1*X^0"
```

Multiplication : polynôme * polynôme

• Pour calculer le produit du polynôme p1 par le polynôme p2, un algorithme consiste à multiplier chaque monôme de p1 par p2 en additionnant les résultats obtenus :

$$\begin{aligned} p_1 \times p_2 &= (a_n x^n + a_{n-1} x^{n-1} + \cdots) p_2 \\ &= (a_n x^n) p_2 + (a_{n-1} x^{n-1}) p_2 + \cdots \end{aligned}$$

• Tout revient donc à savoir multiplier un monôme par un polynôme et à faire l'addition de deux polynômes.

Valeur d'un polynôme en un point

• Soit p un polynôme et x un nombre réel. Comment calculer la valeur de p en x, bien que p ne soit pas une fonction ?

$$p = 2x^5 - x^4 - 2x$$ ⟹ p = [[2,5],[-1,4],[-2,1]]

• Valeur du monôme [2,5] en x ? Il s'agit bien de $2x^5$. Valeur du polynôme p en x : la somme des valeurs en x de ses monômes.

```
polyval <- function(p,x) {
    .....
}
```

mais

```
> p(1)
Erreur : impossible de trouver la fonction "p"
```

• Il reste possible de construire la fonction polynôme associée à p :

```
fpoly <- function(p) {
        return( function(x) {polyval(p, x)})
  }
```

```
> f = fpoly(p)
> f(1)
[1] -1
```

Les fonctions apply

Spécifications

• Les fonctions de la famille apply sont des boucles particulières qui appliquent une fonction en parcourant une structure de données (vecteur, liste, matrice, data frame).

• Il existe plusieurs variantes de ces fonctions selon le type de l'argument et le type souhaité pour le résultat.

• Voici un extrait des résultats de ??apply :

```
base::apply              Apply Functions Over Array Margins
base::lapply             Apply a Function over a List or Vector
base::mapply             Apply a Function to Multiple List or Vector Arguments
base::rapply             Recursively Apply a Function to a List
parallel::mclapply  Parallel Versions of 'lapply' and 'mapply' using Forking
```

• Les fonctions apply ne sont pas nécessairement plus rapides que les boucles classiques, mais plus courtes et plus sécurisées quand elles sont utilisées a bon escient.

Cas d'utilisation : calcul de moyennes

- Supposons que nous voulions calculer un vecteur des moyennes d'élèves dont les notes sont stockées dans une liste.

```
>  li <- list("Ariane"=c(10, 9, 12), "Maxime"=c(14, 14, 12), "Tarek"=c(15, 19, 10))
> print(li)
$Ariane
[1] 10  9 12

$Maxime
[1] 14 14 12

$Tarek
[1] 15 19 10
```

Étudions trois solutions différentes :

- Deux boucles imbriquées ;
- Une boucle et une fonction vectorisée ;
- Fonction sapply et une fonction vectorisée ;

Moyennes : boucles imbriquées

- La première boucle parcours la liste d'élèves.
- La deuxième boucle calcule la moyenne de chaque élève.
- Le vecteur de moyennes peut être construit a la volée ou mieux pré-alloué. Quand c'est possible, il est préférable de pré-allouer la mémoire.

```
res <- c()
for(notes in li) {
  acc <- 0
  for(x in notes) {
    acc <- acc + x
  }
  acc <- acc/length(notes)
  res <- append(res, acc)
}
```

```
res <- numeric(length(li))
 for(i in seq_along(li)) {
   for(x in li[[i]]) {
     res[i] <- res[i] + x
   }
   res[i] <- res[i]/length(li[[i]])
 }
```

```
> print(res)
 [1] 10.33333 13.33333 14.66667
```

- Extraction d'une sous-liste :

```
> li[1]
$Ariane
[1] 10  9 12
```

- Extraction d'un élément de la liste:

```
> li[[1]]
[1] 10  9 12
```

Moyennes : boucle et vectorisation

- La première boucle parcours la liste d'élèves.
- La fonction vectorisée mean calcule la moyenne de chaque élève.
- La question de la pré-allocation n'est toujours pas résolue.

```
 res <- c()
for(notes in li) {
  res <- append(res, mean(notes))
}
```

```
res <- numeric(length(li))
for(i in seq_along(li)) {
  res[i] <- mean(li[[i]])
}
```

Moyennes : apply et vectorisation

- La lisibilité est accrue : une seule ligne de code !
- On ne se soucie plus de la pré-allocation.
- Cerise sur le gâteau : le vecteur est nommé !

```
> sapply(li, mean)
  Ariane   Maxime    Tarek
10.33333 13.33333 14.66667
```

```
> lapply(li,mean)
$Ariane
[1] 10.33333

$Maxime
[1] 13.33333

$Tarek
[1] 14.66667
```

Moyennes : apply et vectorisation

- Supposons maintenant que les notes sont dans une matrice.

```
> mat <- matrix(c(10, 9, 12, 14, 14, 12,
15, 19, 10), ncol=3)
> colnames(mat) <- c("Ariane",
"Maxime", "Tarek")
> rownames(mat) <- c("CC", "CT", "PR")
> print(mat)
   Ariane Maxime Tarek
CC     10     14    15
CT      9     14    19
PR     12     12    10
```

```
> apply(mat, MARGIN=1, mean)
      CC       CT       PR
13.00000 14.00000 11.33333
```

```
> apply(mat, MARGIN=2, mean)
  Ariane   Maxime    Tarek
10.33333 13.33333 14.66667
```

- Attention, la fonction appliquée peut aussi renvoyer un vecteur.

```
> apply(mat, MARGIN=1, summary)
           CC   CT    PR
Min.    10.0  9.0 10.00
1st Qu. 12.0 11.5 11.00
Median  14.0 14.0 12.00
Mean    13.0 14.0 11.33
3rd Qu. 14.5 16.5 12.00
Max.    15.0 19.0 12.00
```

Comment choisir sa fonction apply ?

• En pratique, on doit considérer les points suivants pour faire son choix :

- Le type de donnée de l'entrée
- Que veut-on faire ? Quelle fonction va-t-on appliquer ?
- Le sous-ensemble de données (lignes, colonnes, cellules, …) sur lesquelles on applique la fonction.
- Quel type de donnée souhaite-t'on en sortie ?

• Il existe souvent plusieurs manières équivalentes de réaliser le même traitement.