Introduction à la programmation en S

Vincent Goulet

École d'actuariat
Université Laval

Sommaire

Sommaire

Un langage pour «programmer avec des données»

- Développé chez Bell Laboratories
- Langage de programmation complet et autonome
- Inspiré de plusieurs langages, dont l'APL et le Lisp :
  - interprété (et non compilé)
  - sans déclaration obligatoire des variables
  - basé sur la notion de vecteur
  - particulièrement puissant pour les applications mathématiques et statistiques (et donc actuarielles)

Sommaire

Deux «moteurs» ou dialectes du langage S

- Jusqu'à récemment, le plus connu était S-PLUS de Insightful Corporation, maintenant S+ de Timbco Software
- Le leader est maintenant R, ou GNU S, est une version libre (*Open Source*) «*not unlike S*»
- Environnements intégrés de manipulation de données, de calcul et de préparation de graphiques

Sommaire

D’abord des applications en ligne de commande

- S+ possède une interface graphique élaborée qui permet d’utiliser le logiciel sans trop connaître le langage de programmation
- R dispose également d’une interface graphique sous Windows et Mac OS
- L’édition sérieuse de code S bénéficie cependant grandement d’un bon éditeur de texte

- Question 6.2 de la foire aux questions (FAQ) de R : «Devrais-je utiliser R à l'intérieur de Emacs ?»
- Réponse : «Oui, tout à fait»
- Nous apprendrons à utiliser R à l'intérieur de GNU Emacs avec le mode ESS
- Plusieurs autres options disponibles : choisissez celle que vous préférez

Sommaire

Installation de Emacs avec ESS

- Pour une installation simplifiée de GNU Emacs et ESS, consulter le site Internet

  `http://vgoulet.act.ulaval.ca/emacs/`

- Voir l'annexe A du document d'accompagnement pour les plus importantes commandes

Sommaire

Démarrer et quitter R

- Pour démarrer R à l'intérieur de Emacs :
  `M-x R RET`
  puis spécifier un dossier de travail
- Une console R est ouverte dans un *buffer* nommé ∗R∗
- Pour quitter, deux options :
  1. taper `q()` à la ligne de commande
  2. dans Emacs, faire `C-c C-q`

Sommaire

Deux grandes façons de travailler avec R

1. Le code est virtuel et les objets sont réels
2. Le code est réel et les objets sont virtuels

Code virtuel, objets réels

- C'est l'approche qu'encouragent les interfaces graphiques, mais aussi la moins pratique à long terme
- On entre des expressions directement à la ligne de commande pour les évaluer immédiatement
- Les objets créés au cours d'une session de travail sont sauvegardés
- Par contre, le code utilisé pour créer ces objets est perdu lorsque l'on quitte R, à moins de sauvegarder celui-ci dans des fichiers

Code réel, objets virtuels

- C'est l'approche que nous favoriserons
- Le travail se fait essentiellement dans des fichiers de script (de simples fichiers de texte) dans lesquels sont sauvegardées les expressions (parfois complexes !) et le code des fonctions personnelles
- Les objets sont créés au besoin en exécutant le code

Emacs permet un travail efficace

- Démarrer un processus R et spécifier le dossier de travail
- Ouvrir un fichier de script avec `C-x C-f` (pour créer un nouveau fichier, ouvrir un fichier qui n'existe pas)
- Positionner le curseur sur une expression et faire `C-c C-n` pour l'évaluer
- Le résultat apparaît dans le *buffer* ∗R∗

Sommaire

Gestion des projets ou environnements de travail

- R travaille dans un dossier et non avec des fichiers individuels
- Dans R, les objets créés sont conservés en mémoire
  - sauvegardés en quittant l'application, ou
  - avec la commande `save.image()`
- L'environnement de travail (*workspace*) est sauvegardé dans le fichier `.RData` dans le dossier de travail

Gestion des projets ou environnements de travail

- R travaille dans un dossier et non avec des fichiers individuels
- Dans R, les objets créés sont conservés en mémoire
  - sauvegardés en quittant l'application, ou
  - avec la commande `save.image()`
- L'environnement de travail (*workspace*) est sauvegardé dans le fichier `.RData` dans le dossier de travail

Gestion des projets ou environnements de travail

- R travaille dans un dossier et non avec des fichiers individuels
- Dans R, les objets créés sont conservés en mémoire
  - sauvegardés en quittant l'application, ou
  - avec la commande `save.image()`
- L'environnement de travail (*workspace*) est sauvegardé dans le fichier `.RData` dans le dossier de travail

Gestion des projets ou environnements de travail

- R travaille dans un dossier et non avec des fichiers individuels
- Dans R, les objets créés sont conservés en mémoire
  - sauvegardés en quittant l'application, ou
  - avec la commande `save.image()`
- L'environnement de travail (*workspace*) est sauvegardé dans le fichier `.RData` dans le dossier de travail

Gestion des projets ou environnements de travail

- R travaille dans un dossier et non avec des fichiers individuels
- Dans R, les objets créés sont conservés en mémoire
  - sauvegardés en quittant l'application, ou
  - avec la commande `save.image()`
- L'environnement de travail (*workspace*) est sauvegardé dans le fichier `.RData` dans le dossier de travail

Comment déterminer le dossier de travail

- Avec Emacs et ESS on doit spécifier le dossier de travail à chaque fois que l'on démarre un processus R
- Les interfaces graphiques permettent également de spécifier le dossier de travail au lancement de l'application

Sommaire

La première source d'aide

- Rubriques d'aide contiennent une foule d'informations ainsi que des exemples d'utilisation
- Leur consultation est tout à fait essentielle
- Pour consulter la rubrique d'aide de la fonction `foo`, on peut entrer à la ligne de commande

  `> ?foo`
- Dans Emacs, `C-c C-v foo RET` ouvrira la rubrique d'aide de la fonction `foo` dans un nouveau *buffer*

Sommaire

Plusieurs ressources disponibles

- S+ est livré avec quatre livres, mais aucun ne s'avère vraiment utile pour apprendre le langage S
- Plusieurs livres — en versions papier ou électronique, gratuits ou non — ont été publiés sur S+ et/ou R
- Liste exhaustive dans le site du projet R

Sommaire

Sommaire

Affectations et expressions

Toute commande S est soit une *affectation*, soit une *expression*.

- Normalement, une expression est immédiatement évaluée et le résultat est affiché à l'écran :

```
> 2 + 3
[1] 5
> pi
[1] 3.141593
> cos(pi/4)
[1] 0.7071068
```

Affectations et expressions

- Lors d'une affectation, une expression est évaluée, mais le résultat est stocké dans un objet (variable) et rien n'est affiché à l'écran.
- Le symbole d'affectation est <- (ou ->).

```
> a <- 5
> a
[1] 5
> b <- a
> b
[1] 5
```

Deux symboles d'affectation à éviter

- L'opérateur =
  - peut porter à confusion.
- Le caractère _
  - permis dans S-Plus, mais plus dans R depuis la version 1.8.0
  - emploi fortement découragé
  - rend le code difficile à lire
  - dans le mode ESS de Emacs, taper ce caractère génère carrément ␣<-␣.

Deux symboles d'affectation à éviter

- L'opérateur =
  - peut porter à confusion.
- Le caractère _
  - permis dans S-Plus, mais plus dans R depuis la version 1.8.0
  - emploi fortement découragé
  - rend le code difficile à lire
  - dans le mode ESS de Emacs, taper ce caractère génère carrément ␣<-␣.

Astuce

Pour affecter le résultat d'un calcul dans un objet et en même temps voir ce résultat, placer l'affectation entre parenthèses.

L'opération d'affectation devient alors une nouvelle expression :

```
> (a <- 2 + 3)
[1] 5
```

Sommaire

Caractères permis dans les noms d'objets

- Les lettres a–z, A–Z
- Les chiffres 0–9
- Le point «.»
- «_» est maintenant permis dans R, mais son utilisation est découragée.

Caractères permis dans les noms d'objets

- Les lettres a–z, A–Z
- Les chiffres 0–9
- Le point «.»
- «_» est maintenant permis dans R, mais son utilisation est découragée.

Caractères permis dans les noms d'objets

- Les lettres a–z, A–Z
- Les chiffres 0–9
- Le point «.»
- «_» est maintenant permis dans R, mais son utilisation est découragée.

Caractères permis dans les noms d'objets

- Les lettres a–z, A–Z
- Les chiffres 0–9
- Le point «.»
- «_» est maintenant permis dans R, mais son utilisation est découragée.

Règles pour les noms d'objets

- Les noms d'objets ne peuvent commencer par un chiffre.
- Le S est sensible à la casse : `foo`, `Foo` et `F00` sont trois objets distincts.
- Moyen simple d'éviter des erreurs liées à la casse : employer seulement des lettres minuscules.

Règles pour les noms d'objets

- Les noms d'objets ne peuvent commencer par un chiffre.
- Le S est sensible à la casse : `foo`, `Foo` et `F00` sont trois objets distincts.
- Moyen simple d'éviter des erreurs liées à la casse : employer seulement des lettres minuscules.

Noms déjà utilisés et réservés

- Certains noms sont utilisés par le système, aussi vaut-il mieux éviter de les utiliser. En particulier, éviter d'utiliser

  `c`, `q`, `t`, `C`, `D`, `I`, `diff`, `length`, `mean`, `pi`, `range`, `var`.

- Certains mots sont réservés pour le système et il est interdit de les utiliser comme nom d'objet :

  `Inf`, `NA`, `NaN`, `NULL`
  `break`, `else`, `for`, `function`, `if`, `in`, `next`, `repeat`, `return`, `while`.

- Dans S-Plus 6.1 et plus, `T` et `TRUE` (vrai), ainsi que `F` et `FALSE` (faux) sont également des noms réservés.

Noms déjà utilisés et réservés

- Certains noms sont utilisés par le système, aussi vaut-il mieux éviter de les utiliser. En particulier, éviter d'utiliser

  `c`, `q`, `t`, `C`, `D`, `I`, `diff`, `length`, `mean`, `pi`, `range`, `var`.

- Certains mots sont réservés pour le système et il est interdit de les utiliser comme nom d'objet :

  `Inf`, `NA`, `NaN`, `NULL`
  `break`, `else`, `for`, `function`, `if`, `in`, `next`, `repeat`, `return`, `while`.

- Dans S-Plus 6.1 et plus, T et TRUE (vrai), ainsi que F et FALSE (faux) sont également des noms réservés.

Noms déjà utilisés et réservés

- Certains noms sont utilisés par le système, aussi vaut-il mieux éviter de les utiliser. En particulier, éviter d'utiliser

  `c`, `q`, `t`, `C`, `D`, `I`, `diff`, `length`, `mean`, `pi`, `range`, `var`.

- Certains mots sont réservés pour le système et il est interdit de les utiliser comme nom d'objet :

  `Inf`, `NA`, `NaN`, `NULL`
  `break`, `else`, `for`, `function`, `if`, `in`, `next`, `repeat`, `return`, `while`.

- Dans S-Plus 6.1 et plus, T et TRUE (vrai), ainsi que F et FALSE (faux) sont également des noms réservés.

TRUE et FALSE dans R

- Dans R, les noms TRUE et FALSE sont également réservés.
- Les variables T et F prennent par défaut les valeurs TRUE et FALSE, respectivement, mais peuvent être réaffectées.

```
> T
[1] TRUE
> TRUE <- 3
Erreur dans TRUE <- 3 : membre gauche de
l'assignation (do_set) incorrect
> (T <- 3)
[1] 3
```

TRUE et FALSE dans R

- Dans R, les noms TRUE et FALSE sont également réservés.
- Les variables T et F prennent par défaut les valeurs TRUE et FALSE, respectivement, mais peuvent être réaffectées.

```
> T

[1] TRUE

> TRUE <- 3

Erreur dans TRUE <- 3 : membre gauche de
l'assignation (do_set) incorrect

> (T <- 3)

[1] 3
```

Sommaire

Tout est un objet

- Tout dans le langage S est un objet, même les fonctions et les opérateurs.
- Les objets possèdent au minimum un mode et une longueur.

Mode et longueur

- Le mode d'un objet est obtenu avec la fonction `mode`.

```
> v <- c(1, 2, 5, 9)
> mode(v)
[1] "numeric"
```

- La longueur d'un objet est obtenue avec la fonction `length`.

```
> length(v)
[1] 4
```

- Certains objets sont également dotés d'un ou plusieurs attributs.

Sommaire

Modes et types de données

- Le mode prescrit ce qu'un objet peut contenir.
- Un objet ne peut donc avoir qu'un seul mode.
- Modes disponibles en S :

`numeric`	nombres réels
`complex`	nombres complexes
`logical`	valeurs booléennes (vrai/faux)
`character`	chaînes de caractères
`function`	fonction
`list`	données quelconques

Sommaire

Longueur

La longueur d'un objet est égale au nombre d'éléments qu'il contient.

- La longueur d'une chaîne de caractères est toujours 1. Un objet de mode `character` doit contenir plusieurs chaînes de caractères pour que sa longueur soit supérieure à 1.

```
> v <- "actuariat"
> length(v)
[1] 1
> v <- c("a", "c", "t", "u", "a", "r", "i",
+     "a", "t")
> length(v)
[1] 9
```

Longueur

La longueur d'un objet est égale au nombre d'éléments qu'il contient.

- La longueur d'une chaîne de caractères est toujours 1. Un objet de mode `character` doit contenir plusieurs chaînes de caractères pour que sa longueur soit supérieure à 1.

```
> v <- "actuariat"
> length(v)

[1] 1

> v <- c("a", "c", "t", "u", "a", "r", "i",
+     "a", "t")
> length(v)

[1] 9
```

Objet vide

- Un objet peut être de longueur 0.
- Doit alors être interprété comme un contenant vide.

```
> v <- numeric(0)
> length(v)
[1] 0
```

Objet vide

- Un objet peut être de longueur 0.
- Doit alors être interprété comme un contenant vide.

```
> v <- numeric(0)
> length(v)
[1] 0
```

Sommaire

Attributs

- Éléments d'information additionnels liés à cet objet.
- Attributs les plus fréquemment rencontrés :

`class`	affecte le comportement d'un objet
`dim`	dimensions des matrices et tableaux
`dimnames`	étiquettes des dimensions des matrices et tableaux
`names`	étiquettes des éléments d'un objet

Sommaire

L’objet spécial NA

NA est fréquemment utilisé pour représenter les données manquantes.

- Son mode est `logical`.
- Toute opération impliquant une donnée NA a comme résultat NA.
- Certaines fonctions (`sum`, `mean`, par exemple), ont par conséquent un argument `na.rm` qui, lorsque TRUE, élimine les données manquantes avant de faire un calcul.
- La fonction `is.na` permet de tester si les éléments d’un objet sont NA ou non.

L'objet spécial NA

NA est fréquemment utilisé pour représenter les données manquantes.

- Son mode est `logical`.
- Toute opération impliquant une donnée NA a comme résultat NA.
- Certaines fonctions (`sum`, `mean`, par exemple), ont par conséquent un argument `na.rm` qui, lorsque TRUE, élimine les données manquantes avant de faire un calcul.
- La fonction `is.na` permet de tester si les éléments d'un objet sont NA ou non.

L'objet spécial NA

NA est fréquemment utilisé pour représenter les données manquantes.

- Son mode est `logical`.
- Toute opération impliquant une donnée NA a comme résultat NA.
- Certaines fonctions (`sum`, `mean`, par exemple), ont par conséquent un argument `na.rm` qui, lorsque TRUE, élimine les données manquantes avant de faire un calcul.
- La fonction `is.na` permet de tester si les éléments d'un objet sont NA ou non.

L'objet spécial NA

NA est fréquemment utilisé pour représenter les données manquantes.

- Son mode est `logical`.
- Toute opération impliquant une donnée NA a comme résultat NA.
- Certaines fonctions (`sum`, `mean`, par exemple), ont par conséquent un argument `na.rm` qui, lorsque TRUE, élimine les données manquantes avant de faire un calcul.
- La fonction `is.na` permet de tester si les éléments d'un objet sont NA ou non.

L'objet spécial NA

NA est fréquemment utilisé pour représenter les données manquantes.

- Son mode est `logical`.
- Toute opération impliquant une donnée NA a comme résultat NA.
- Certaines fonctions (`sum`, `mean`, par exemple), ont par conséquent un argument `na.rm` qui, lorsque TRUE, élimine les données manquantes avant de faire un calcul.
- La fonction `is.na` permet de tester si les éléments d'un objet sont NA ou non.

Sommaire

L’objet spécial NULL

NULL représente «rien», ou le vide.

- Son mode est NULL.
- Sa longueur est 0.
- Différent d’un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - NULL est sans contenant.
- La fonction `is.null` teste si un objet est NULL ou non.

L'objet spécial NULL

NULL représente «rien», ou le vide.

- Son mode est NULL.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - NULL est «pas de contenant».
- La fonction `is.null` teste si un objet est NULL ou non.

L'objet spécial NULL

`NULL` représente «rien», ou le vide.

- Son mode est `NULL`.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - `NULL` est «pas de contenant».
- La fonction `is.null` teste si un objet est `NULL` ou non.

L'objet spécial NULL

NULL représente «rien», ou le vide.

- Son mode est NULL.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - NULL est «pas de contenant».
- La fonction `is.null` teste si un objet est NULL ou non.

L'objet spécial NULL

NULL représente «rien», ou le vide.

- Son mode est NULL.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - NULL est «pas de contenant».
- La fonction `is.null` teste si un objet est NULL ou non.

L'objet spécial NULL

NULL représente «rien», ou le vide.

- Son mode est NULL.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - NULL est «pas de contenant».
- La fonction `is.null` teste si un objet est NULL ou non.

L'objet spécial NULL

`NULL` représente «rien», ou le vide.

- Son mode est `NULL`.
- Sa longueur est 0.
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide ;
  - `NULL` est «pas de contenant».
- La fonction `is.null` teste si un objet est `NULL` ou non.

Sommaire

En S, tout est un vecteur

- Dans un vecteur simple, tous les éléments doivent être du même mode.
- Il est possible (et souvent souhaitable) de donner une étiquette à chacun des éléments d'un vecteur.

```
> (v <- c(a = 1, b = 2, c = 5))

a b c
1 2 5

> v <- c(1, 2, 5)
> names(v) <- c("a", "b", "c")
> v

a b c
1 2 5
```

Et comment crée-t-on ces vecteurs ?

Les fonctions de base pour créer des vecteurs sont

- `c` (concaténation)
- `numeric` (vecteur de mode `numeric`)
- `logical` (vecteur de mode `logical`)
- `character` (vecteur de mode `character`).

Indiçage d'un vecteur

- Se fait avec [ ].
- On peut extraire un élément d'un vecteur par
  - sa position ou
  - son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]

c
5

> v["c"]

c
5
```

Indiçage d'un vecteur

- Se fait avec `[ ]`.
- On peut extraire un élément d'un vecteur par
  - sa position ou
  - son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]

c
5

> v["c"]

c
5
```

Indiçage d'un vecteur

- Se fait avec `[ ]`.
- On peut extraire un élément d'un vecteur par
  - sa position ou
  - son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]

c
5

> v["c"]

c
5
```

Indiçage d'un vecteur

- Se fait avec `[ ]`.
- On peut extraire un élément d'un vecteur par
  - sa position ou
  - son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]

c
5

> v["c"]

c
5
```

Indiçage d'un vecteur

- Se fait avec `[ ]`.
- On peut extraire un élément d'un vecteur par
  - sa position ou
  - son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]

c
5

> v["c"]

c
5
```

Sommaire

Une matrice est un vecteur ?

Une matrice ou, de façon plus générale, un tableau (*array*) n'est rien d'autre qu'un vecteur doté d'un attribut `dim`.

- À l'interne, une matrice est donc stockée sous forme de vecteur.
- La fonction de base pour créer des matrices est `matrix`.
- La fonction de base pour créer des tableaux est `array`.

Une matrice est un vecteur ?

Une matrice ou, de façon plus générale, un tableau (*array*) n'est rien d'autre qu'un vecteur doté d'un attribut `dim`.

- À l'interne, une matrice est donc stockée sous forme de vecteur.
- La fonction de base pour créer des matrices est `matrix`.
- La fonction de base pour créer des tableaux est `array`.

Une matrice est un vecteur ?

Une matrice ou, de façon plus générale, un tableau (*array*) n'est rien d'autre qu'un vecteur doté d'un attribut `dim`.

- À l'interne, une matrice est donc stockée sous forme de vecteur.
- La fonction de base pour créer des matrices est `matrix`.
- La fonction de base pour créer des tableaux est `array`.

Une matrice est un vecteur ?

Une matrice ou, de façon plus générale, un tableau (*array*) n'est rien d'autre qu'un vecteur doté d'un attribut `dim`.

- À l'interne, une matrice est donc stockée sous forme de vecteur.
- La fonction de base pour créer des matrices est `matrix`.
- La fonction de base pour créer des tableaux est `array`.

Remplissage d'une matrice

- Important : les matrices et tableaux sont remplis en faisant d'abord varier la première dimension, puis la seconde, etc.

```
> matrix(1:6, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6, nrow = 2, ncol = 3,
+        byrow = TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Indiçage d'une matrice

- On extrait les éléments d'une matrice en précisant leurs positions sous la forme (ligne, colonne) dans la matrice, ou encore leurs positions dans le vecteur sous-jacent.

```
> (m <- matrix(c(40, 80, 45, 21, 55, 32),
+              nrow = 2, ncol = 3))
     [,1] [,2] [,3]
[1,]   40   45   55
[2,]   80   21   32
> m[1, 2]
[1] 45
> m[3]
[1] 45
```

Fusion verticale de matrices

- La fonction `rbind` permet de fusionner verticalement deux matrices (ou plus) ayant le même nombre de colonnes.

```
> n <- matrix(1:9, nrow = 3)
> rbind(m, n)
     [,1] [,2] [,3]
[1,]   40   45   55
[2,]   80   21   32
[3,]    1    4    7
[4,]    2    5    8
[5,]    3    6    9
```

Fusion horizontale de matrices

- La fonction `cbind` permet de fusionner horizontalement deux matrices (ou plus) ayant le même nombre de lignes.

```
> n <- matrix(1:4, nrow = 2)
> cbind(m, n)
     [,1] [,2] [,3] [,4] [,5]
[1,]   40   45   55    1    3
[2,]   80   21   32    2    4
```

Sommaire

Un vecteur très général

Une liste est un type de vecteur spécial dont les éléments peuvent être de n'importe quel mode, y compris le mode `list` (ce qui permet d'emboîter des listes).

- La fonction de base pour créer des listes est `list`.
- Généralement préférable de nommer les éléments d'une liste : plus simple et sûr d'extraire les éléments par leur étiquette.

Un vecteur très général

Une liste est un type de vecteur spécial dont les éléments peuvent être de n'importe quel mode, y compris le mode `list` (ce qui permet d'emboîter des listes).

- La fonction de base pour créer des listes est `list`.
- Généralement préférable de nommer les éléments d'une liste : plus simple et sûr d'extraire les éléments par leur étiquette.

Un vecteur très général

Une liste est un type de vecteur spécial dont les éléments peuvent être de n'importe quel mode, y compris le mode `list` (ce qui permet d'emboîter des listes).

- La fonction de base pour créer des listes est `list`.
- Généralement préférable de nommer les éléments d'une liste : plus simple et sûr d'extraire les éléments par leur étiquette.

Indiçage d'une liste

- L'extraction des éléments d'une liste peut se faire de deux façons :
  1. avec des doubles crochets `[[ ]]`
  2. par leur étiquette avec `nom.liste$etiquette.element`.

Indiçage d'une liste

- L'extraction des éléments d'une liste peut se faire de deux façons :
  1. avec des doubles crochets `[[ ]]`
  2. par leur étiquette avec `nom.liste$etiquette.element`.

Indiçage d'une liste

- L'extraction des éléments d'une liste peut se faire de deux façons :
  1. avec des doubles crochets `[[ ]]`
  2. par leur étiquette avec `nom.liste$etiquette.element`.

Sommaire

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Liste ou matrice ? Un peu des deux !

- L'analyse de données — la régression linéaire, par exemple — repose sur les *data frames*.
- Liste de classe `data.frame` dont tous les éléments sont de la même longueur.
- Généralement représenté sous forme d'un tableau à deux dimensions (visuellement similaire à une matrice).
- Plus général qu'une matrice puisque les colonnes peuvent être de modes différents (`numeric`, `complex`, `character` ou `logical`).
- Créé avec la fonction `data.frame` ou `as.data.frame`.
- Moins important lors de l'apprentissage du langage de programmation.

Sommaire

Quatre façons d'indicer un vecteur

Dans tous les cas, l'indiçage se fait avec des crochets `[ ]`.

1. Avec un vecteur d'entiers positifs. Les éléments se trouvant aux positions correspondant aux entiers sont extraits du vecteur, dans l'ordre. C'est la technique la plus courante.

```
> letters[c(1:3, 22, 5)]
[1] "a" "b" "c" "v" "e"
```

2. Avec un vecteur d'entiers négatifs. Les éléments se trouvant aux positions correspondant aux entiers négatifs sont alors éliminés du vecteur.

```
> letters[c(-(1:3), -5, -22)]
 [1] "d" "f" "g" "h" "i" "j" "k" "l" "m"
[10] "n" "o" "p" "q" "r" "s" "t" "u" "w"
[19] "x" "y" "z"
```

3. Avec un vecteur booléen. Le vecteur d'indiçage doit alors être de la même longueur que le vecteur indicé. Les éléments correspondant à une valeur TRUE sont extraits du vecteur, alors que ceux correspondant à FALSE sont éliminés.

```
> letters > "f" & letters < "q"
 [1] FALSE FALSE FALSE FALSE FALSE FALSE
 [7]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE  TRUE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
> letters[letters > "f" & letters < "q"]
 [1] "g" "h" "i" "j" "k" "l" "m" "n" "o"
[10] "p"
```

4. Avec une chaîne de caractères. Utile pour extraire les éléments d'un vecteur à condition que ceux-ci soient nommés.

```
> x <- c(Rouge = 2, Bleu = 4, Vert = 9,
+ Jaune = -5)
> x[c("Bleu", "Jaune")]
 Bleu Jaune
    4    -5
```

Sommaire

Une liste non exhaustive

- Principaux opérateurs arithmétiques, fonctions mathématiques et structures de contrôles offertes par le S.
- Liste loin d'être exhaustive.
- Consulter aussi la section See `Also` des rubriques d'aide.

Sommaire

L'unité de base est le vecteur

- Les opérations sur les vecteurs sont effectuées élément par élément :

```
> c(1, 2, 3) + c(4, 5, 6)
[1] 5 7 9
> 1:3 * 4:6
[1]  4 10 18
```

Recyclage des vecteurs

- Si les vecteurs impliqués dans une expression arithmétique ne sont pas de la même longueur, les plus courts sont recyclés.
- Particulièrement apparent avec les vecteurs de longueur 1 :

```
> 1:10 + 2
 [1]  3  4  5  6  7  8  9 10 11 12
> 1:10 + rep(2, 10)
 [1]  3  4  5  6  7  8  9 10 11 12
```

Longueur du plus long vecteur multiple de celle des autres vecteurs

- Les vecteurs les plus courts sont recyclés un nombre entier de fois :

```
> 1:10 + 1:5 + c(2, 4)
 [1]  4  8  8 12 12 11 11 15 15 19
> 1:10 + rep(1:5, 2) + rep(c(2, 4), 5)
 [1]  4  8  8 12 12 11 11 15 15 19
```

Sinon...

- Recyclage un nombre fractionnaire de fois et un avertissement est affiché :

```
> 1:10 + c(2, 4, 6)
 [1]  3  6  9  6  9 12  9 12 15 12
Message d'avis :
la longueur de l'objet le plus long n'est
pas un multiple de la longueur de l'objet
le plus court in: 1:10 + c(2, 4, 6)
```

Sommaire

Opérateurs mathématiques et logiques les plus fréquemment employés

Ordre décroissant de priorité des opérations.

^ ou **	puissance
-	changement de signe
* /	multiplication, division
+ -	addition, soustraction
%*% %% %/%	produit matriciel, modulo, division entière
< <= == >= > !=	plus petit, plus petit ou égal, égal, plus grand ou égal, plus grand, différent de
!	négation logique
& |	«et» logique, «ou» logique

Sommaire

Comment spécifier les arguments d'une fonction

- Pas de limite pratique au nombre d'arguments.
- Arguments peuvent être spécifiés dans l'ordre établi dans la définition de la fonction.
- Plus prudent et fortement recommandé de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.
- Nécessaire de nommer les arguments s'ils ne sont pas appelés dans l'ordre.
- Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié.

Comment spécifier les arguments d'une fonction

- Pas de limite pratique au nombre d'arguments.
- Arguments peuvent être spécifiés dans l'ordre établi dans la définition de la fonction.
- Plus prudent et fortement recommandé de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.
- Nécessaire de nommer les arguments s'ils ne sont pas appelés dans l'ordre.
- Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié.

Comment spécifier les arguments d'une fonction

- Pas de limite pratique au nombre d'arguments.
- Arguments peuvent être spécifiés dans l'ordre établi dans la définition de la fonction.
- Plus prudent et fortement recommandé de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.
- Nécessaire de nommer les arguments s'ils ne sont pas appelés dans l'ordre.
- Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié.

Comment spécifier les arguments d'une fonction

- Pas de limite pratique au nombre d'arguments.
- Arguments peuvent être spécifiés dans l'ordre établi dans la définition de la fonction.
- Plus prudent et fortement recommandé de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.
- Nécessaire de nommer les arguments s'ils ne sont pas appelés dans l'ordre.
- Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié.

Comment spécifier les arguments d'une fonction

- Pas de limite pratique au nombre d'arguments.
- Arguments peuvent être spécifiés dans l'ordre établi dans la définition de la fonction.
- Plus prudent et fortement recommandé de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.
- Nécessaire de nommer les arguments s'ils ne sont pas appelés dans l'ordre.
- Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié.

Sommaire

Exemple

Définition de la fonction `matrix` :

```
matrix(data = NA, nrow = 1, ncol = 1,
       byrow = FALSE, dimnames = NULL)
```

- Chaque argument a une valeur par défaut (ce n'est pas toujours le cas).
- Ainsi, un appel à `matrix` sans argument résulte en

```
> matrix()
     [,1]
[1,]   NA
```

Exemple

Définition de la fonction `matrix` :

```
matrix(data = NA, nrow = 1, ncol = 1,
       byrow = FALSE, dimnames = NULL)
```

- Chaque argument a une valeur par défaut (ce n'est pas toujours le cas).
- Ainsi, un appel à `matrix` sans argument résulte en

```
> matrix()
     [,1]
[1,]   NA
```

Exemple

Définition de la fonction `matrix` :

```
matrix(data = NA, nrow = 1, ncol = 1,
       byrow = FALSE, dimnames = NULL)
```

- Chaque argument a une valeur par défaut (ce n'est pas toujours le cas).
- Ainsi, un appel à `matrix` sans argument résulte en

```
> matrix()
     [,1]
[1,]   NA
```

- Appel plus élaboré utilisant tous les arguments. Le premier argument est rarement nommé.

```
> matrix(1 :6, nrow = 2, ncol = 3,
+   byrow = TRUE,
+   dimnames = list(c("Gauche", "Droit"),
+     c("Rouge", "Vert", "Bleu"))))
```

```
        Rouge Vert Bleu
Gauche      1    2    3
Droit       4    5    6
```

Sommaire

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Système de classement des fonctions

- Diffère entre S-Plus et R.
- Dans S-Plus, les fonctions sont classées dans des sections d'une bibliothèque (*library*).
- Dans R, un ensemble de fonctions est appelé un package.
- Par défaut, R charge en mémoire quelques packages de la bibliothèque seulement.
- Cela économise l'espace mémoire et accélère le démarrage.
- On charge de nouveaux packages en mémoire avec la fonction `library`.

Manipulation de vecteurs

`seq`	génération de suites de nombres
`rep`	répétition de valeurs ou de vecteurs
`sort`	tri en ordre croissant ou décroissant
`order`	positions dans un vecteur des valeurs en ordre croissant ou décroissant
`rank`	rang des éléments d'un vecteur en ordre croissant ou décroissant
`rev`	renverser un vecteur
`head`	extraction des *n* premières valeurs ou suppression des *n* dernières (R seulement)
`tail`	extraction des *n* dernières valeurs ou suppression des *n* premières (R seulement)
`unique`	extraction des éléments différents d'un vecteur

Recherche d'éléments dans un vecteur

`which`	positions des valeurs TRUE dans un vecteur booléen
`which.min`	position du minimum dans un vecteur
`which.max`	position du maximum dans un vecteur
`match`	position de la première occurrence d'un élément dans un vecteur
`%in%`	appartenance d'une ou plusieurs valeurs à un vecteur

Arrondi

`round`	arrondi à un nombre défini de décimales
`floor`	plus grand entier inférieur ou égal à l'argument
`ceiling`	plus petit entier supérieur ou égal à l'argument
`trunc`	troncature vers zéro de l'argument ; différent de `floor` pour les nombres négatifs

Sommaires et statistiques descriptives

`sum`, `prod`	somme et produit des éléments
`diff`	différences entre les éléments
`mean`	moyenne arithmétique et moyenne tronquée
`var`, `sd`	variance et écart type (sans biais)
`min`, `max`	minimum et maximum d'un vecteur
`range`	vecteur contenant le minimum et le maximum d'un vecteur
`median`	médiane empirique
`quantile`	quantiles empiriques
`summary`	statistiques descriptives d'un échantillon

Sommaires cumulatifs et comparaisons élément par élément

`cumsum`, `cumprod`	somme et produit cumulatif d'un vecteur
`cummin`, `cummax`	minimum et maximum cumulatif
`pmin`, `pmax`	minimum et maximum en parallèle, c'est-à-dire élément par élément entre deux vecteurs ou plus

Opérations sur les matrices

`t`	transposée
`solve`	avec un seul argument (une matrice carrée) : inverse d'une matrice ; avec deux arguments (une matrice carrée et un vecteur) : solution du système d'équation $\boldsymbol{Ax} = \boldsymbol{b}$
`diag`	avec une matrice en argument : diagonale de la matrice ; avec un vecteur en argument : matrice diagonale formée avec le vecteur ; avec un scalaire p en argument : matrice identité $p \times p$
`nrow, ncol`	nombre de lignes et de colonnes d'une matrice

Opérations sur les matrices (suite)

`rowSums`, `colSums`	sommes par ligne et par colonne, respectivement, des éléments d'une matrice ; voir aussi la fonction `apply`
`rowMeans`, `colMeans`	moyennes par ligne et par colonne, respectivement, des éléments d'une matrice ; voir aussi la fonction `apply`
`rowVars`, `colVars`	variance par ligne et par colonne des éléments d'une matrice (S-Plus seulement)

Produit extérieur

La fonction `outer`, dont la syntaxe est

```
outer(X, Y, FUN),
```

applique la fonction `FUN` (`prod` par défaut) entre chacun des éléments de `X` et chacun des éléments de `Y`.

- La dimension du résultat est par conséquent `c(dim(X), dim(Y))`.

- Par exemple : le produit extérieur entre deux vecteurs est une matrice contenant tous les produits entre les éléments des deux vecteurs :

```
> outer(c(1, 2, 5), c(2, 3, 6))
     [,1] [,2] [,3]
[1,]    2    3    6
[2,]    4    6   12
[3,]   10   15   30
```

- L'opérateur `%o%` est un raccourci de `outer(X, Y, prod)`.

- Par exemple : le produit extérieur entre deux vecteurs est une matrice contenant tous les produits entre les éléments des deux vecteurs :

```
> outer(c(1, 2, 5), c(2, 3, 6))
     [,1] [,2] [,3]
[1,]    2    3    6
[2,]    4    6   12
[3,]   10   15   30
```

- L'opérateur `%o%` est un raccourci de `outer(X, Y, prod)`.

Sommaire

Exécution conditionnelle

`if (`*`condition`*`)` *`branche.vrai`* `else` *`branche.faux`*

Si *`condition`* est vraie, *`branche.vrai`* est exécutée, et *`branche.faux`* sinon.

Si l'une ou l'autre de *`branche.vrai`* ou *`branche.faux`* comporte plus d'une expression, les grouper dans des accolades { }.

`ifelse(`*`condition, expression.vrai, expression.faux`*`)`

Fonction vectorisée qui remplace chaque élément TRUE du vecteur *`condition`* par l'élément correspondant de *`expression.vrai`* et chaque élément FALSE par l'élément correspondant de *`expression.faux`*.

Boucles

- Les boucles sont et doivent être utilisées avec parcimonie en S car elles sont généralement inefficaces (particulièrement avec S-Plus).
- Dans la majeure partie des cas, il est possible de vectoriser les calcul pour éviter les boucles explicites.
- Sinon, s'en remettre aux fonctions `apply`, `lapply` et `sapply` pour faire les boucles de manière plus efficace.

Boucles

- Les boucles sont et doivent être utilisées avec parcimonie en S car elles sont généralement inefficaces (particulièrement avec S-Plus).
- Dans la majeure partie des cas, il est possible de vectoriser les calcul pour éviter les boucles explicites.
- Sinon, s'en remettre aux fonctions `apply`, `lapply` et `sapply` pour faire les boucles de manière plus efficace.

Boucles

- Les boucles sont et doivent être utilisées avec parcimonie en S car elles sont généralement inefficaces (particulièrement avec S-Plus).
- Dans la majeure partie des cas, il est possible de vectoriser les calcul pour éviter les boucles explicites.
- Sinon, s'en remettre aux fonctions `apply`, `lapply` et `sapply` pour faire les boucles de manière plus efficace.

Boucles de longueur déterminée

`for (`*`variable`* `in` *`suite`*`)` *`expression`*

Exécuter *`expression`* successivement pour chaque valeur de *`variable`* contenue dans *`suite`*.

Encore ici, on groupera les expressions dans des accolades { }.

À noter que *`suite`* n'a pas à être composée de nombres consécutifs, ni même par ailleurs de nombres.

Boucles de longueur indéterminée

`while (`*`condition`*`)` *`expression`*

Exécuter *`expression`* tant que *`condition`* est vraie.

Si *`condition`* est fausse lors de l'entrée dans la boucle, celle-ci n'est pas exécutée.

Une boucle `while` n'est par conséquent pas nécessairement toujours exécutée.

`repeat` *`expression`*

Répéter *`expression`*. Cette dernière devra comporter un test d'arrêt qui utilisera la commande `break`.

Une boucle `repeat` est toujours exécutée au moins une fois.

Modification du déroulement d'une boucle

`break`

Sortie immédiate d'une boucle `for`, `while` ou `repeat`.

`next`

Passage immédiat à la prochaine itération d'une boucle `for`, `while` ou `repeat`.

Sommaire

Sommaire

Calcul de valeurs présentes

Énoncé

Un prêt est remboursé par une série de cinq paiements, le premier dans un an. Trouver le montant du prêt pour chacune des hypothèses ci-dessous.

a) Paiement annuel de 1 000, taux d'intérêt de 6 % effectif annuellement.

b) Paiements annuels de 500, 800, 900, 750 et 1 000, taux d'intérêt de 6 % effectif annuellement.

c) Paiements annuels de 500, 800, 900, 750 et 1 000, taux d'intérêt de 5 %, 6 %, 5,5 %, 6,5 % et 7 % effectifs annuellement.

Solution

De manière générale, la valeur présente d'une série de paiements $P_1, P_2, \ldots, P_n$ à la fin des années $1, 2, \ldots, n$ est

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j,$$

a) Un seul paiement annuel, un seul taux d'intérêt.

Cas spécial

$$P \sum_{j=1}^{n} (1+i)^{-j}$$

En S:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

a) Un seul paiement annuel, un seul taux d'intérêt.

Cas spécial

$$P \sum_{j=1}^{n} (1+i)^{-j}$$

En S:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

a) Un seul paiement annuel, un seul taux d'intérêt.

Cas spécial

$$P \sum_{j=1}^{n} (1+i)^{-j}$$

En S:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

a) Un seul paiement annuel, un seul taux d'intérêt.

Cas spécial

$$P \sum_{j=1}^{n} (1+i)^{-j}$$

En S:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

a) Un seul paiement annuel, un seul taux d'intérêt.

Cas spécial

$$P \sum_{j=1}^{n} (1+i)^{-j}$$

En S:

```
> 1000 * sum((1 + 0.06)^(-(1:5)))
[1] 4212.364
```

□

b) Différents paiements annuels, un seul taux d'intérêt.

On a, cette fois,

$$\sum_{j=1}^{n} (1+i)^{-j} P_j$$

En S:

```
> sum(c(500, 800, 900, 750, 1000) *
+     (1 + 0.06)^(-(1:5)))

[1] 3280.681
```

b) Différents paiements annuels, un seul taux d'intérêt.

On a, cette fois,

$$\sum_{j=1}^{n} (1+i)^{-j} P_j$$

En S:

```
> sum(c(500, 800, 900, 750, 1000) *
+       (1 + 0.06)^(-(1:5)))

[1] 3280.681
```

b) Différents paiements annuels, un seul taux d'intérêt.

On a, cette fois,

$$\sum_{j=1}^{n} (1+i)^{-j} P_j$$

En S:

```
> sum(c(500, 800, 900, 750, 1000) *
+       (1 + 0.06)^(-(1:5)))

[1] 3280.681
```

b) Différents paiements annuels, un seul taux d'intérêt.

On a, cette fois,

$$\sum_{j=1}^{n} (1+i)^{-j} P_j$$

En S:

```
> sum(c(500, 800, 900, 750, 1000) *
+       (1 + 0.06)^(-(1:5)))

[1] 3280.681
```

□

c) Différents paiements annuels, différents taux d'intérêt.

On doit utiliser la formule générale

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j$$

En S :

```
> sum(c(500, 800, 900, 750, 1000)/
+ cumprod(c(1.05, 1.06, 1.055, 1.065, 1.07)))

[1] 3308.521
```

c) Différents paiements annuels, différents taux d'intérêt.

On doit utiliser la formule générale

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j$$

En S :

```
> sum(c(500, 800, 900, 750, 1000)/
+ cumprod(c(1.05, 1.06, 1.055, 1.065, 1.07)))

[1] 3308.521
```

c) Différents paiements annuels, différents taux d'intérêt.

On doit utiliser la formule générale

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j$$

En S :

```
> sum(c(500, 800, 900, 750, 1000)/
+ cumprod(c(1.05, 1.06, 1.055, 1.065, 1.07)))

[1] 3308.521
```

c) Différents paiements annuels, différents taux d'intérêt.

On doit utiliser la formule générale

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j$$

En S :

```
> sum(c(500, 800, 900, 750, 1000)/
+ cumprod(c(1.05, 1.06, 1.055, 1.065, 1.07)))
```

```
[1] 3308.521
```

c) Différents paiements annuels, différents taux d'intérêt.

On doit utiliser la formule générale

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j$$

En S :

```
> sum(c(500, 800, 900, 750, 1000)/
+ cumprod(c(1.05, 1.06, 1.055, 1.065, 1.07)))

[1] 3308.521
```

Sommaire

Fonctions de probabilité

Énoncé

Calculer toutes ou la majeure partie des probabilités des deux lois de probabilité ci-dessous. Vérifier que la somme des probabilités est bien égale à 1.

a) Binomiale

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, \ldots, n.$$

b) Poisson

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad x = 0, 1, \ldots,$$

où $x! = x(x-1)\cdots 2 \cdot 1$.

Solution

a) Binomiale$(10, 0{,}8)$.

```
> n <- 10
> p <- 0.8
> x <- 0:n
> choose(n, x) * p^x * (1 - p)^rev(x)
 [1] 0.0000001024 0.0000040960 0.0000737280
 [4] 0.0007864320 0.0055050240 0.0264241152
 [7] 0.0880803840 0.2013265920 0.3019898880
[10] 0.2684354560 0.1073741824
> sum(choose(n, x) * p^x * (1 - p)^rev(x))
[1] 1
```

Solution

a) Binomiale$(10, 0{,}8)$.

```
> n <- 10
> p <- 0.8
> x <- 0:n
> choose(n, x) * p^x * (1 - p)^rev(x)
 [1] 0.0000001024 0.0000040960 0.0000737280
 [4] 0.0007864320 0.0055050240 0.0264241152
 [7] 0.0880803840 0.2013265920 0.3019898880
[10] 0.2684354560 0.1073741824
> sum(choose(n, x) * p^x * (1 - p)^rev(x))
[1] 1
```

b) Poisson(5).

On calcule les probabilités en $x = 0, 1, \ldots, 10$ seulement.

```
> lambda <- 5
> x <- 0:10
> exp(-lambda) * (lambda^x/factorial(x))
 [1] 0.006737947 0.033689735 0.084224337
 [4] 0.140373896 0.175467370 0.175467370
 [7] 0.146222808 0.104444863 0.065278039
[10] 0.036265577 0.018132789

> x <- 0:200
> exp(-lambda) * sum((lambda^x/factorial(x)))

[1] 1
```

b) Poisson(5).

On calcule les probabilités en $x = 0, 1, \ldots, 10$ seulement.

```
> lambda <- 5
> x <- 0:10
> exp(-lambda) * (lambda^x/factorial(x))

 [1] 0.006737947 0.033689735 0.084224337
 [4] 0.140373896 0.175467370 0.175467370
 [7] 0.146222808 0.104444863 0.065278039
[10] 0.036265577 0.018132789

> x <- 0:200
> exp(-lambda) * sum((lambda^x/factorial(x)))

[1] 1
```

Sommaire

Fonction de répartition de la loi gamma

La loi gamma est fréquemment utilisée pour la modélisation d'événements ne pouvant prendre que des valeurs positives et pour lesquels les petites valeurs sont plus fréquentes que les grandes. Nous utiliserons la paramétrisation où la fonction de densité de probabilité est

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}, \quad x > 0,$$

où

$$\Gamma(n) = \int_0^\infty x^{n-1} e^{-x}\, dx = (n-1)\Gamma(n-1).$$

Énoncé

Il n'existe pas de formule explicite de la fonction de répartition de la loi gamma.
Néanmoins, pour α entier et $\lambda = 1$ on a

$$F(x; \alpha, 1) = 1 - e^{-x} \sum_{j=0}^{\alpha-1} \frac{x^j}{j!}.$$

a) Évaluer $F(4; 5, 1)$.

b) Évaluer $F(x; 5, 1)$ pour $x = 2, 3, \ldots, 10$ en une seule expression.

Solution

a) Une seule valeur de x, paramètre α fixe.

```
> alpha <- 5
> x <- 4
> 1 - exp(-x) * sum(x^(0:(alpha - 1))/
+ gamma(1:alpha))
[1] 0.3711631
```

Vérification avec la fonction interne pgamma :

```
> pgamma(x, alpha)
[1] 0.3711631
```

Solution

a) Une seule valeur de x, paramètre α fixe.

```
> alpha <- 5
> x <- 4
> 1 - exp(-x) * sum(x^(0:(alpha - 1))/
+ gamma(1:alpha))
[1] 0.3711631
```

Vérification avec la fonction interne pgamma :

```
> pgamma(x, alpha)
[1] 0.3711631
```

□

Astuce

On peut aussi éviter de générer essentiellement la même suite de nombres à deux reprises en ayant recours à une variable intermédiaire.
L'affectation et le calcul final peuvent se faire dans une seule expression.

```
> 1 - exp(-x) * sum(x^(-1 + (j <- 1:alpha))/
+ gamma(j))

[1] 0.3711631
```

b) Plusieurs valeurs de x, paramètre α fixe.

C'est un travail pour la fonction `outer`.

```
> x <- 2:10
> 1 - exp(-x) *
+ colSums(
+   t( outer(x, 0:(alpha - 1), "^") )
+   /gamma(1:alpha)
+ )

[1] 0.05265302 0.18473676 0.37116306
[4] 0.55950671 0.71494350 0.82700839
[7] 0.90036760 0.94503636 0.97074731
```

Sommaire

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Algorithme du point fixe

- Problème classique : trouver la racine d'une fonction g, c'est-à-dire le point x où $g(x) = 0$.
- Souvent possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$.
- Solution appelée point fixe.
- L'algorithme du calcul numérique du point fixe d'une fonction $f(x)$ est très simple :
  1. choisir une valeur de départ x_0 ;
  2. calculer $x_n = f(x_{n-1})$;
  3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \varepsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \varepsilon$.

Énoncé

Trouver, à l'aide de la méthode du point fixe, la valeur de i telle que

$$a_{\overline{10}|} = \frac{1 - (1 + i)^{-10}}{i} = 8{,}21.$$

Solution

Quelques considérations.

- On doit résoudre

$$\frac{1-(1+i)^{-10}}{8{,}21} = i.$$

- Nous ignorons combien de fois la procédure itérative devra être répétée.
- Il faut exécuter la procédure au moins une fois.
- La structure de contrôle à utiliser dans cette procédure itérative est donc `repeat`.

Solution

Quelques considérations.

- On doit résoudre

$$\frac{1-(1+i)^{-10}}{8{,}21} = i.$$

- Nous ignorons combien de fois la procédure itérative devra être répétée.
- Il faut exécuter la procédure au moins une fois.
- La structure de contrôle à utiliser dans cette procédure itérative est donc `repeat`.

Solution

Quelques considérations.

- On doit résoudre

$$\frac{1-(1+i)^{-10}}{8{,}21} = i.$$

- Nous ignorons combien de fois la procédure itérative devra être répétée.
- Il faut exécuter la procédure au moins une fois.
- La structure de contrôle à utiliser dans cette procédure itérative est donc `repeat`.

Solution

Quelques considérations.

- On doit résoudre

$$\frac{1-(1+i)^{-10}}{8{,}21} = i.$$

- Nous ignorons combien de fois la procédure itérative devra être répétée.
- Il faut exécuter la procédure au moins une fois.
- La structure de contrôle à utiliser dans cette procédure itérative est donc `repeat`.

Le code.

```
> i <- 0.05
> repeat {
+     it <- i
+     i <- (1 - (1 + it)^(-10))/8.21
+     if (abs(i - it)/it < 1e-10)
+         break
+ }
> i
[1] 0.03756777
> (1 - (1 + i)^(-10))/i
[1] 8.21
```

Le code.

```
> i <- 0.05
> repeat {
+     it <- i
+     i <- (1 - (1 + it)^(-10))/8.21
+     if (abs(i - it)/it < 1e-10)
+         break
+ }
> i
[1] 0.03756777
> (1 - (1 + i)^(-10))/i
[1] 8.21
```

Sommaire

Sommaire

Définition d'une fonction

On définit une fonction de la manière suivante :

`fun <- function(`*`arguments`*`)` *`expression`*

où

- `fun` est le nom de la fonction ;
- *`arguments`* est la liste des arguments, séparés par des virgules ;
- *`expression`* constitue le corps de la fonction, soit une liste d'expressions groupées entre accolades (nécessaires s'il y a plus d'une expression seulement).

Sommaire

Retourner des résultats

- Une fonction retourne tout simplement le résultat de la dernière expression du corps de la fonction.
- Éviter que la dernière expression soit une affectation : la fonction ne retournera rien !
- Autre possibilité : utiliser explicitement la fonction `return`. Rarement nécessaire.
- Utiliser une liste nommée pour retourner plusieurs résultats.

Retourner des résultats

- Une fonction retourne tout simplement le résultat de la dernière expression du corps de la fonction.
- Éviter que la dernière expression soit une affectation : la fonction ne retournera rien !
- Autre possibilité : utiliser explicitement la fonction `return`. Rarement nécessaire.
- Utiliser une liste nommée pour retourner plusieurs résultats.

Retourner des résultats

- Une fonction retourne tout simplement le résultat de la dernière expression du corps de la fonction.
- Éviter que la dernière expression soit une affectation : la fonction ne retournera rien !
- Autre possibilité : utiliser explicitement la fonction `return`. Rarement nécessaire.
- Utiliser une liste nommée pour retourner plusieurs résultats.

Retourner des résultats

- Une fonction retourne tout simplement le résultat de la dernière expression du corps de la fonction.
- Éviter que la dernière expression soit une affectation : la fonction ne retournera rien !
- Autre possibilité : utiliser explicitement la fonction `return`. Rarement nécessaire.
- Utiliser une liste nommée pour retourner plusieurs résultats.

Sommaire

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur `<<-`.
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur `<<-`.
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur `<<-`.
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur $<<-$.
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- ~~On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur <<-.~~
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Variables locales et globales

Les concepts de variable locale et de variable globale existent aussi en S.

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - qu'elle n'apparaît pas dans l'espace de travail ;
  - qu'elle n'écrase pas une variable du même nom dans l'espace de travail.
- ~~On peut définir une variable dans l'espace de travail depuis une fonction avec l'opérateur <<-.~~
- Une fonction définie à l'intérieur d'une autre fonction sera locale à celle-ci. Pratique !

Sommaire

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Fonction à partir du code de point fixe


```
fp <- function(k, n, start=0.05, TOL=1E-10)
{
    i <- start
    repeat
    {
        it <- i
        i <- (1 - (1 + it)^(-n))/k
        if (abs(i - it)/it < TOL)
            break
    }
    i  # ou return(i)
}
```

Sommaire

Fonctions anonymes

- Parfois utile de définir une fonction sans lui attribuer un nom
- C'est une fonction anonyme.
- En général pour des fonctions courtes utilisées dans une autre fonction.

Un exemple

- Calculer la valeur de xy^2 pour toutes les combinaisons de x et y stockées dans des vecteurs du même nom
- Avec `outer` :

```
> x <- 1:3
> y <- 4:6
> f <- function(x, y) x * y^2
> outer(x, y, f)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

Un exemple

- Calculer la valeur de xy^2 pour toutes les combinaisons de x et y stockées dans des vecteurs du même nom
- Avec `outer` :

```
> x <- 1:3
> y <- 4:6
> f <- function(x, y) x * y^2
> outer(x, y, f)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

- La fonction `f` ne sert à rien ultérieurement.
- Utiliser simplement une fonction anonyme à l'intérieur de `outer` :

```
> outer(x, y, function(x, y) x * y^2)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

- La fonction `f` ne sert à rien ultérieurement.
- Utiliser simplement une fonction anonyme à l'intérieur de `outer` :

```
> outer(x, y, function(x, y) x * y^2)
     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

Sommaire

Techniques les plus simples et naïves

- Simples erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules).
- Vérification de la syntaxe lors de la définition d'une fonction.
- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction.
- Permet de déterminer les valeurs des variables dans le déroulement de la fonction.

Techniques les plus simples et naïves

- Simples erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules).
- Vérification de la syntaxe lors de la définition d'une fonction.
- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction.
- Permet de déterminer les valeurs des variables dans le déroulement de la fonction.

Techniques les plus simples et naïves

- Simples erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules).
- Vérification de la syntaxe lors de la définition d'une fonction.
- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction.
- Permet de déterminer les valeurs des variables dans le déroulement de la fonction.

Techniques les plus simples et naïves

- Simples erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules).
- Vérification de la syntaxe lors de la définition d'une fonction.
- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction.
- Permet de déterminer les valeurs des variables dans le déroulement de la fonction.

Exemple

Modification de la boucle du point fixe pour détecter une procédure divergente.

```
repeat
{
    it <- i
    i <- (1 - (1 + it)^(-n))/k
    print(i)
    if (abs((i - it)/it < TOL))
        break
}
```

Avec Emacs et le mode ESS

- S'assurer que toutes les variables passées en arguments à une fonction existent dans l'espace de travail.
- Exécuter successivement les lignes de la fonction avec `C-c C-n`.
- Impossible avec les interfaces graphiques car la fenêtre d'édition de fonctions bloque l'accès à l'interface de commande.

Avec Emacs et le mode ESS

- S'assurer que toutes les variables passées en arguments à une fonction existent dans l'espace de travail.
- Exécuter successivement les lignes de la fonction avec `C-c C-n`.
- Impossible avec les interfaces graphiques car la fenêtre d'édition de fonctions bloque l'accès à l'interface de commande.

Avec Emacs et le mode ESS

- S'assurer que toutes les variables passées en arguments à une fonction existent dans l'espace de travail.
- Exécuter successivement les lignes de la fonction avec `C-c C-n`.
- Impossible avec les interfaces graphiques car la fenêtre d'édition de fonctions bloque l'accès à l'interface de commande.

Sommaire

Styles reconnus par Emacs

C++/Stroustrup

```
for (i in 1:10)
{
    expression
}
```

K&R (1TBS)

```
for (i in 1:10){
    expression
}
```

Whitesmith

```
for (i in 1:10)
    {
    expression
    }
```

GNU

```
for (i in 1:10)
  {
    expression
  }
```

Standard pour la programmation en S

- Style C++, avec les accolades sur leurs propres lignes.
- Une indentation de quatre (4) espaces.
- Pour utiliser ce style dans Emacs, faire

  `M-x ess-set-style RET C++ RET`

  une fois qu'un fichier de script est ouvert.

Standard pour la programmation en S

- Style C++, avec les accolades sur leurs propres lignes.
- Une indentation de quatre (4) espaces.
- Pour utiliser ce style dans Emacs, faire

  `M-x ess-set-style RET C++ RET`

  une fois qu'un fichier de script est ouvert.

Sommaire

Sommaire

Pas un signe de paresse des rédacteurs

- '. . .' est un argument formel dont '. . .' est le nom.
- Signifie qu'une fonction peut accepter un ou plusieurs autres arguments autres que ceux faisant partie de sa définition.
- Contenu de '. . .' n'est ni pris en compte, ni modifié par la fonction.
- Généralement simplement passé tel quel à une autre fonction.
- Voir les définitions des fonctions `apply`, `lapply` et `sapply` pour des exemples.

Pas un signe de paresse des rédacteurs

- '. . .' est un argument formel dont '. . .' est le nom.
- Signifie qu'une fonction peut accepter un ou plusieurs autres arguments autres que ceux faisant partie de sa définition.
- Contenu de '. . .' n'est ni pris en compte, ni modifié par la fonction.
- Généralement simplement passé tel quel à une autre fonction.
- Voir les définitions des fonctions `apply`, `lapply` et `sapply` pour des exemples.

Pas un signe de paresse des rédacteurs

- '. . .' est un argument formel dont '. . .' est le nom.
- Signifie qu'une fonction peut accepter un ou plusieurs autres arguments autres que ceux faisant partie de sa définition.
- Contenu de '. . .' n'est ni pris en compte, ni modifié par la fonction.
- Généralement simplement passé tel quel à une autre fonction.
- Voir les définitions des fonctions `apply`, `lapply` et `sapply` pour des exemples.

Pas un signe de paresse des rédacteurs

- '. . .' est un argument formel dont '. . .' est le nom.
- Signifie qu'une fonction peut accepter un ou plusieurs autres arguments autres que ceux faisant partie de sa définition.
- Contenu de '. . .' n'est ni pris en compte, ni modifié par la fonction.
- Généralement simplement passé tel quel à une autre fonction.
- Voir les définitions des fonctions `apply`, `lapply` et `sapply` pour des exemples.

Pas un signe de paresse des rédacteurs

- '`...`' est un argument formel dont '`...`' est le nom.
- Signifie qu'une fonction peut accepter un ou plusieurs autres arguments autres que ceux faisant partie de sa définition.
- Contenu de '`...`' n'est ni pris en compte, ni modifié par la fonction.
- Généralement simplement passé tel quel à une autre fonction.
- Voir les définitions des fonctions `apply`, `lapply` et `sapply` pour des exemples.

Sommaire

Sommaires généraux pour matrices et tableaux

La fonction `apply` sert à appliquer une fonction quelconque sur une partie d'une matrice ou, plus généralement, d'un tableau.

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

où

- `X` est une matrice ou un tableau ;
- `MARGIN` est un vecteur d'entiers contenant la ou les dimensions de la matrice ou du tableau sur lesquelles la fonction doit s'appliquer ;
- `FUN` est la fonction à appliquer ;
- '`...`' est un ensemble d'arguments supplémentaires, séparés par des virgules, à passer à la fonction `FUN`.

Sommaires généraux pour matrices et tableaux

La fonction `apply` sert à appliquer une fonction quelconque sur une partie d'une matrice ou, plus généralement, d'un tableau.

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

où

- `X` est une matrice ou un tableau ;
- `MARGIN` est un vecteur d'entiers contenant la ou les dimensions de la matrice ou du tableau sur lesquelles la fonction doit s'appliquer ;
- `FUN` est la fonction à appliquer ;
- '`...`' est un ensemble d'arguments supplémentaires, séparés par des virgules, à passer à la fonction `FUN`.

- Principalement pour calculer des sommaires par ligne (dimension 1) ou par colonne (dimension 2) autres que la somme et la moyenne.
- Utiliser la fonction `apply` plutôt que des boucles puisque celle-ci est plus efficace.

- Principalement pour calculer des sommaires par ligne (dimension 1) ou par colonne (dimension 2) autres que la somme et la moyenne.
- Utiliser la fonction `apply` plutôt que des boucles puisque celle-ci est plus efficace.

Exemples avec une matrice


```
> m
     [,1] [,2] [,3] [,4]
[1,]   54   33   30   17
[2,]    3   46   95   83
[3,]   47    6   56   58
[4,]   18   22   50   36
[5,]   41   41   77   31
> apply(m, 1, var)
[1]  235.0000 1718.9167  590.9167  211.6667
[5]  409.0000
> apply(m, 2, min)
[1]  3  6 30 17
> apply(m, 1, mean, trim = 0.2)
[1] 33.50 56.75 41.75 31.50 47.50
```

Exemples avec une matrice


```
> m
     [,1] [,2] [,3] [,4]
[1,]   54   33   30   17
[2,]    3   46   95   83
[3,]   47    6   56   58
[4,]   18   22   50   36
[5,]   41   41   77   31
> apply(m, 1, var)
[1]  235.0000 1718.9167  590.9167  211.6667
[5]  409.0000
> apply(m, 2, min)
[1]  3  6 30 17
> apply(m, 1, mean, trim = 0.2)
[1] 33.50 56.75 41.75 31.50 47.50
```

Exemples avec une matrice


```
> m
     [,1] [,2] [,3] [,4]
[1,]   54   33   30   17
[2,]    3   46   95   83
[3,]   47    6   56   58
[4,]   18   22   50   36
[5,]   41   41   77   31
> apply(m, 1, var)
[1]  235.0000 1718.9167  590.9167  211.6667
[5]  409.0000
> apply(m, 2, min)
[1]  3  6 30 17
> apply(m, 1, mean, trim = 0.2)
[1] 33.50 56.75 41.75 31.50 47.50
```

Exemples avec une matrice


```
> m
     [,1] [,2] [,3] [,4]
[1,]   54   33   30   17
[2,]    3   46   95   83
[3,]   47    6   56   58
[4,]   18   22   50   36
[5,]   41   41   77   31
> apply(m, 1, var)
[1]  235.0000 1718.9167  590.9167  211.6667
[5]  409.0000
> apply(m, 2, min)
[1]  3  6 30 17
> apply(m, 1, mean, trim = 0.2)
[1] 33.50 56.75 41.75 31.50 47.50
```

Exemple avec un tableau

Si X est un tableau de plus de deux dimensions, alors l'argument passé à FUN peut être une matrice ou un tableau.

```
> dim(arr)
[1] 4 4 5
> apply(arr, 3, det)
[1]  1178800 16153716 14298240 20093933
[5]  6934743
```

Sommaire

Les apply des vecteurs et des listes

- Les fonctions `lapply` et `sapply` permettent d'appliquer une fonction aux éléments d'un vecteur ou d'une liste.
- Syntaxe similaire :

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

Des fonctions très utiles

- `lapply` applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous forme de liste.
- `sapply` est similaire, sauf que le résultat est retourné sous forme de vecteur, si possible.
- Si le résultat de chaque application de la fonction est un vecteur, `sapply` retourne une matrice, remplie comme toujours par colonne.
- Dans un grand nombre de cas, il est possible de remplacer les boucles `for` par l'utilisation de `lapply` ou `sapply`.

Des fonctions très utiles

- `lapply` applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous forme de liste.
- `sapply` est similaire, sauf que le résultat est retourné sous forme de vecteur, si possible.
- Si le résultat de chaque application de la fonction est un vecteur, `sapply` retourne une matrice, remplie comme toujours par colonne.
- Dans un grand nombre de cas, il est possible de remplacer les boucles `for` par l'utilisation de `lapply` ou `sapply`.

Des fonctions très utiles

- `lapply` applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous forme de liste.
- `sapply` est similaire, sauf que le résultat est retourné sous forme de vecteur, si possible.
- Si le résultat de chaque application de la fonction est un vecteur, `sapply` retourne une matrice, remplie comme toujours par colonne.
- Dans un grand nombre de cas, il est possible de remplacer les boucles `for` par l'utilisation de `lapply` ou `sapply`.

Des fonctions très utiles

- `lapply` applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous forme de liste.
- `sapply` est similaire, sauf que le résultat est retourné sous forme de vecteur, si possible.
- Si le résultat de chaque application de la fonction est un vecteur, `sapply` retourne une matrice, remplie comme toujours par colonne.
- Dans un grand nombre de cas, il est possible de remplacer les boucles `for` par l'utilisation de `lapply` ou `sapply`.

Sommaire

Version multidimensionnelle de `sapply`

- Syntaxe :

```
mapply(FUN, ...)
```

- Le résultat est l'application de `FUN` aux premiers éléments de tous les arguments contenus dans '`...`', puis à tous les seconds éléments, et ainsi de suite.
- Ainsi, si `v` et `w` sont des vecteurs,

```
mapply(FUN, v, w)
```

retourne `FUN(v[1], w[1])`, `FUN(v[2], w[2])`, etc.

Version multidimensionnelle de `sapply`

- Syntaxe :

```
mapply(FUN, ...)
```

- Le résultat est l'application de `FUN` aux premiers éléments de tous les arguments contenus dans '`...`', puis à tous les seconds éléments, et ainsi de suite.
- Ainsi, si `v` et `w` sont des vecteurs,

```
mapply(FUN, v, w)
```

retourne `FUN(v[1], w[1])`, `FUN(v[2], w[2])`, etc.

Version multidimensionnelle de `sapply`

- Syntaxe :

```
mapply(FUN, ...)
```

- Le résultat est l'application de `FUN` aux premiers éléments de tous les arguments contenus dans '`...`', puis à tous les seconds éléments, et ainsi de suite.
- Ainsi, si `v` et `w` sont des vecteurs,

```
mapply(FUN, v, w)
```

retourne `FUN(v[1], w[1])`, `FUN(v[2], w[2])`, etc.

Exemple


```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

Exemple


```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

Les éléments de ‘...’ sont recyclés au besoin


```
> mapply(seq, 1:6, 6:8)
[[1]]
[1] 1 2 3 4 5 6

[[2]]
[1] 2 3 4 5 6 7

[[3]]
[1] 3 4 5 6 7 8

[[4]]
[1] 4 5 6

[[5]]
[1] 5 6 7
```

Sommaire

Une fonction pour la simulation

- Fonction enveloppante de `sapply` propre à R.
- Simplifie la syntaxe pour l'exécution répétée d'une expression.
- Usage particulièrement indiqué pour les simulations.
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10 000 simulations avec

  ```
  > replicate(10000, fun(...))
  ```
- Voir l'annexe D du document d'accompagnement.

Une fonction pour la simulation

- Fonction enveloppante de `sapply` propre à R.
- Simplifie la syntaxe pour l'exécution répétée d'une expression.
- Usage particulièrement indiqué pour les simulations.
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10 000 simulations avec

  ```
  > replicate(10000, fun(...))
  ```

- Voir l'annexe D du document d'accompagnement.

Une fonction pour la simulation

- Fonction enveloppante de `sapply` propre à R.
- Simplifie la syntaxe pour l'exécution répétée d'une expression.
- Usage particulièrement indiqué pour les simulations.
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10 000 simulations avec

  ```
  > replicate(10000, fun(...))
  ```

- Voir l'annexe D du document d'accompagnement.

Une fonction pour la simulation

- Fonction enveloppante de `sapply` propre à R.
- Simplifie la syntaxe pour l'exécution répétée d'une expression.
- Usage particulièrement indiqué pour les simulations.
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10 000 simulations avec

  ```
  > replicate(10000, fun(...))
  ```

- Voir l'annexe D du document d'accompagnement.

Une fonction pour la simulation

- Fonction enveloppante de `sapply` propre à R.
- Simplifie la syntaxe pour l'exécution répétée d'une expression.
- Usage particulièrement indiqué pour les simulations.
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10 000 simulations avec

  ```
  > replicate(10000, fun(...))
  ```

- Voir l'annexe D du document d'accompagnement.

Sommaire

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

Quelques notions de programmation OO

- Tous les objets dans le langage S ont une classe.
- La classe est parfois implicite ou dérivée du mode de l'objet (consulter la rubrique d'aide de `class` pour de plus amples détails).
- Certaines fonctions génériques se comportent différemment selon la classe de l'objet donné en argument.
- Les fonctions génériques les plus fréquemment employées sont `print`, `plot` et `summary`.
- Une fonction générique possède une méthode correspondant à chaque classe qu'elle reconnaît.
- Sinon, une méthode `default` pour les autres objets.

- La liste des méthodes existant pour une fonction générique s'obtient avec `methods` :

```
> methods(plot)
 [1] plot.acf*          plot.data.frame*
 [3] plot.Date*         plot.decomposed.ts*
 [5] plot.default       plot.dendrogram*
 [7] plot.density       plot.ecdf
 [9] plot.factor*       plot.formula*
[11] plot.hclust*       plot.histogram*

[...]

   Non-visible functions are asterisked
```

- À chaque méthode `methode` d'une fonction générique `fun` correspond une fonction `fun.methode`.
- Consulter cette rubrique d'aide et non celle de la fonction générique, qui contient en général peu d'informations.

Astuce

Lorsque l'on tape le nom d'un objet à la ligne de commande pour voir son contenu, c'est la fonction générique `print` qui est appelée.
On peut donc complètement modifier la représentation à l'écran du contenu d'un objet est créant une nouvelle classe et une nouvelle méthode pour la fonction `print`.

Sommaire

Sommaire

Qu'est-ce que Emacs ?

- Emacs est l'Éditeur de texte des éditeurs de texte.
- D'abord et avant tout un éditeur pour programmeurs (avec des modes spéciaux pour une multitude de langages différents).
- Également un environnement idéal pour travailler sur des documents LaTeX, interagir avec R, S-Plus, SAS ou SQL, ou même pour lire son courrier électronique.

Mise en contexte

Emacs est le logiciel étendard du projet GNU («*GNU is not Unix*»), dont le principal commanditaire est la *Free Software Foundation*.

- Distribué sous la GNU *General Public License* (GPL), donc gratuit, ou «libre».
- Le nom provient de «*Editing MACroS*».
- La première version de Emacs a été écrite par Richard M. Stallman, président de la FSF.

Configuration de l'éditeur

Une des grandes forces de Emacs est d'être configurable à l'envi.

- Depuis la version 21, le menu `Customize` rend la configuration aisée.
- Une grande part de la configuration provient du fichier `.emacs` :
  - nommé `.emacs` sous Linux et Unix, Windows 2000 et Windows XP ;
  - sous Windows 95/98/Me, utiliser plutôt `_emacs`.

Emacs-ismes et *Unix-ismes*

- Un *buffer* contient un fichier ouvert («*visited*»). Équivalent d'une fenêtre dans Windows.
- Le *minibuffer* est la région au bas de l'écran Emacs où l'on entre des commandes et reçoit de l'information de Emacs.
- La ligne de mode («*mode line*») est le séparateur horizontal contenant diverses informations sur le fichier ouvert et l'état de Emacs.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'interpréteur (ou invite) de commandes.
- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Ctrl` (`Control`);
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC;
  - `ESC` est la touche Échap (`Esc`) et est équivalente à `Meta`;
  - `SPC` est la barre d'espacement;
  - `RET` est la touche Entrée.

- Le caractère ~ représente le dossier vers lequel pointe la variable d'environnement `$HOME` (Unix) ou `%HOME%` (Windows).
- La barre oblique (`/`) est utilisée pour séparer les dossiers dans les chemins d'accès aux fichiers, même sous Windows.
- En général, il est possible d'appuyer sur TAB dans le *minibuffer* pour compléter les noms de fichiers ou de commandes.

- Le caractère ~ représente le dossier vers lequel pointe la variable d'environnement `$HOME` (Unix) ou `%HOME%` (Windows).
- La barre oblique (`/`) est utilisée pour séparer les dossiers dans les chemins d'accès aux fichiers, même sous Windows.
- En général, il est possible d'appuyer sur TAB dans le *minibuffer* pour compléter les noms de fichiers ou de commandes.

- Le caractère ~ représente le dossier vers lequel pointe la variable d'environnement `$HOME` (Unix) ou `%HOME%` (Windows).
- La barre oblique (`/`) est utilisée pour séparer les dossiers dans les chemins d'accès aux fichiers, même sous Windows.
- En général, il est possible d'appuyer sur `TAB` dans le *minibuffer* pour compléter les noms de fichiers ou de commandes.

Commandes d'édition de base

- Il n'est pas vain de lire le tutoriel de Emacs, que l'on démarre avec

  ```
  C-h t
  ```

- Pour une liste plus exhaustive des commandes Emacs les plus importantes, consulter la *GNU Emacs Reference Card* à l'adresse

  ```
  http://refcards.com/refcards/gnu-emacs/
  ```

- Pour créer un nouveau fichier, ouvrir un fichier n'existant pas.
- Principales commandes d'édition :

`C-x C-f`	ouvrir un fichier
`C-x C-s`	sauvegarder
`C-x C-w`	sauvegarder sous
`C-x k`	fermer un fichier
`C-x C-c`	quitter Emacs

- Pour créer un nouveau fichier, ouvrir un fichier n'existant pas.
- Principales commandes d'édition :

`C-x C-f`	ouvrir un fichier
`C-x C-s`	sauvegarder
`C-x C-w`	sauvegarder sous
`C-x k`	fermer un fichier
`C-x C-c`	quitter Emacs

`C-g`	bouton de panique : quitter !
`C-_`	annuler (pratiquement illimité) ; aussi `C-x u`
`C-s`	recherche incrémentale avant
`C-r`	Recherche incrémentale arrière
`M-%`	rechercher et remplacer

`C-x b`	changer de *buffer*
`C-x 2`	séparer l'écran en deux fenêtres
`C-x 1`	conserver uniquement la fenêtre courante
`C-x 0`	fermer la fenêtre courante
`C-x o`	aller vers une autre fenêtre lorsqu'il y en a plus d'une

Sélection de texte

La sélection de texte fonctionne différemment du standard Windows.

- Les raccourcis clavier standards sous Emacs sont :

`C-SPC`	débute la sélection
`C-w`	couper la sélection
`M-w`	copier la sélection
`C-y`	coller
`M-y`	remplacer le dernier texte collé par la sélection précédente

- Il existe quelques extensions de Emacs permettant d'utiliser les raccourcis clavier usuels de Windows (`C-c`, `C-x`, `C-v`).

Sélection de texte

La sélection de texte fonctionne différemment du standard Windows.

- Les raccourcis clavier standards sous Emacs sont :

`C-SPC`	débute la sélection
`C-w`	couper la sélection
`M-w`	copier la sélection
`C-y`	coller
`M-y`	remplacer le dernier texte collé par la sélection précédente

- Il existe quelques extensions de Emacs permettant d'utiliser les raccourcis clavier usuels de Windows (`C-c`, `C-x`, `C-v`).

Sommaire

Emacs Speaks Statistics

Mode pour interagir avec des logiciels statistiques (S-Plus, R, SAS, etc.) depuis Emacs.

- Voir

  `http://ess.r-project.org/`

  pour la documentation complète.
- Deux modes mineurs : ESS pour les fichiers de script (code source) et `iESS` pour l'invite de commande.
- Une fois installé, le mode mineur ESS s'active automatiquement en éditant des fichiers avec l'extension `.S` ou `.R`.

Démarrer un processus S

- Pour démarrer un processus S et activer le mode mineur `iESS`, entrer l'une des commandes
  - `S`
  - `Sqpe` ou
  - `R`

  dans l'invite de commande de Emacs
- Par exemple, pour démarrer un processus R à l'intérieur de Emacs :

  `M-x R RET`

Raccourcis clavier les plus utiles à la ligne de commande (mode iESS)

`C-c C-e`	replacer la dernière ligne au bas de la fenêtre
`M-h`	sélectionner le résultat de la dernière commande
`C-c C-o`	effacer le résultat de la dernière commande
`C-c C-v`	aide sur une commande S
`C-c C-q`	terminer le processus S

Raccourcis clavier les plus utiles lors de l'édition d'un fichier de script (mode ESS)

`C-c C-n`	évaluer la ligne sous le curseur dans le processus S, puis déplacer le curseur à la prochaine ligne de commande
`C-c C-r`	évaluer la région sélectionnée dans le processus S
`C-c C-f`	évaluer le code de la fonction courante dans le processus S
`C-c C-l`	évaluer le code du fichier courant dans le processus S
`C-c C-v`	aide sur une commande S
`C-c C-s`	changer de processus (utile si l'on a plus d'un processus S actif)

Consultation des rubriques d'aide

- Quelques avantages à lire les rubriques d'aide dans Emacs.
- Modifier l'option R `chmhelp` ainsi :

  ```
  > options(chmhelp = FALSE)
  ```

- Pour que la commande s'exécute automatiquement à chaque lancement de R, entrer la commande dans un fichier nommé `.Rprofile` sauvegardé dans le dossier mentionné dans le résultat de

  ```
  > Sys.getenv("R_USER")
  ```

- Consulter aussi la rubrique d'aide de `Startup`.

Raccourcis clavier utiles lors de la consultation des rubriques d'aide

h	ouvrir une nouvelle rubrique d'aide, par défaut pour le mot se trouvant sous le curseur
n, p	aller à la section suivante (n) ou précédente (p) de la rubrique
l	évaluer la ligne sous le curseur ; pratique pour exécuter les exemples
r	évaluer la région sélectionnée
q	retourner au processus ESS en laissant la rubrique d'aide visible
x	fermer la rubrique d'aide et retourner au processus ESS