UNIVERSITÉ LAVAL

Faculté des sciences et de génie
École d'actuariat

Introduction à la programmation en S

Vincent Goulet

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

- Pendant longtemps, le plus connu était S-Plus/S-PLUS/S+
- Leader maintenant R, ou GNU S, 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+ possèdait une interface graphique élaborée qui permettait d’utiliser le logiciel sans trop connaître le langage de programmation
- Interface graphique pour R sous Windows et Mac OS
- Aussi des interfaces simplifiées pour R : R Commander, Excel, . . .
- Édition sérieuse de code bénéficie grandement d’un bon éditeur de texte
- Plusieurs options disponibles : choisissez celle que vous préférez

Mon choix d'éditeur

- Question 6.2 de la foire aux questions (FAQ) de R :
  - Devrais-je utiliser R à l'intérieur de Emacs ?
  - Oui, tout à fait
- À peu près la seule option identique sur toutes les plateformes
- J'utiliserai R à l'intérieur de GNU Emacs avec le mode ESS

Sommaire

Démarrer et quitter R

- Interfaces graphiques
  - cliquer sur l'icône 
- Emacs
  - démarrer R à l'intérieur de Emacs avec
    `M-x R RET`
    puis spécifier un dossier de travail
  - une console R est ouverte dans un *buffer* nommé `*R*`
- Pour quitter :
  - taper `q()` à la ligne de commande
  - fermer la fenêtre
  - 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

- Approche encouragée par les interfaces graphiques, mais 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
- Le code utilisé pour créer ces objets est perdu lorsque l'on quitte R

Code réel, objets virtuels

- Approche que nous favoriserons
- Travail fait essentiellement dans des fichiers de script
- Fichiers de script contiennent des expressions (parfois complexes !) et le code des fonctions personnelles
- Les objets sont créés au besoin en exécutant le code

Anatomie d'une session de travail

- Démarrer un processus R
- Ouvrir un fichier de script dans son éditeur préféré
- Exécuter les expressions une à une ou en bloc
- Allers-retours entre la console R et le fichier de script
- Emacs rend ce processus très simple

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

Rarement nécessaire de sauvegarder les objets

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

Rarement nécessaire de sauvegarder les objets

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

Rarement nécessaire de sauvegarder les objets

Comment déterminer le dossier de travail

- Pas très naturel avec les interfaces graphiques
  - Windows : menu `File|Change dir...`
  - OS X : menu `Divers|Changer de Répertoire de travail`
- Toujours disponible à la ligne de commande :
  `setwd()`
- Emacs : spécifier le dossier de travail au démarrage de R

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`
- Emacs : `C-c C-v foo RET`

Sommaire

Sommaire

Affectations et expressions

- Toute commande R est soit une **affectation**, soit une **expression**
- 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

- Toute commande R est soit une **affectation**, soit une **expression**
- 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 (suite)

- **Affectation** : expression évaluée, mais
  1. résultat stocké dans un objet (variable)
  2. rien n'est affiché à l'écran
- Symbole d'affectation : `<-` (ou `->`)

```
> a <- 5
> a

[1] 5

> b <- a
> b

[1] 5
```

- Éviter d'utiliser `=`

Astuce

- Dans le mode ESS de Emacs, la touche _ génère ␣<-␣
- Appuyer deux fois pour obtenir le caractère _

Astuce

- Pour affecter le résultat d'un calcul **et** voir le résultat, placer l'affectation entre parenthèses
- Opération d'affectation devient alors une nouvelle expression :

```
> (a <- 2 + 3)

[1] 5
```

Sommaire

Caractères permis dans les noms d'objets

- Lettres a–z, A–Z
- Chiffres 0–9
- Point «.»
- Caractère de soulignement «_»

Règles pour les noms d'objets

- Noms d'objets ne peuvent commencer par un chiffre
- R est sensible à la casse : `foo`, `Foo` et `FOO` sont trois objets distincts
- Moyen simple d'éviter des erreurs liées à la casse : employer seulement des minuscules

Noms déjà utilisés et réservés

- Éviter d'utiliser les noms

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

- Noms réservés :

  `Inf`, `NA`, `NaN`, `NULL`, `TRUE`, `FALSE`
  `break`, `else`, `for`, `function`,
  `if`, `in`, `next`, `repeat`, `return`, `while`
  `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`,
  `...`, `..1`, `..2`, etc.

TRUE et FALSE

- Variables T et F prennent par défaut les valeurs TRUE et FALSE, 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 R est un objet, même les fonctions et les opérateurs
- Objets possèdent au minimum un **mode** et une **longueur**
- Certains objets également dotés d'un ou plusieurs **attributs**

Modes et types de données

- Mode prescrit ce qu'un objet peut contenir
- Obtenu avec la fonction `mode` :

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

- Principaux modes :

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

Longueur

- Égale au nombre d'éléments que contient un objet
- Obtenue avec la fonction `length` :

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

Longueur (suite)

- Longueur d'une chaîne de caractères est toujours 1
- Un objet de mode `character` peut contenir plusieurs chaînes de caractères :

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

```
[1] 1
```

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

```
[1] 9
```

Longueur (suite)

- Longueur d'un objet peut être 0
- Contenant vide :

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

L'objet spécial `NULL`

- `NULL` représente «rien», ou le vide
- Mode `NULL`
- Longueur est 0
- Différent d'un objet vide :
  - un objet de longueur 0 est un contenant vide
  - `NULL` est «pas de contenant»
- Fonction `is.null` pour tester si un objet est `NULL`

L'objet spécial `NA`

- Utilisé pour représenter les données manquantes ou l'absence de données
- Mode `logical`
- **Toute** opération impliquant `NA` a comme résultat `NA`
- Certaines fonctions (par ex. `sum`, `mean`) peuvent éliminer les `NA` avant de faire un calcul
- Fonction `is.na` pour tester si un objet est `NA`

Valeurs spéciales IEEE 754

- `Inf` : $+\infty$
- `-Inf` : $-\infty$
- `NaN` : $0/0$, $\infty - \infty$, ...
- Fonctions `is.finite`, `is.infinite`, `is.nan`

Attributs

- Éléments d'information liés à un objet
- Principaux attributs

`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

En R, tout est un vecteur

- Dans un vecteur simple (*atomic*), tous les éléments du même mode
- Possible de donner une étiquette aux 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
```

Fonctions de base pour créer des vecteurs

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

Indiçage

- Se fait avec `[ ]`
- Extraction d'un élément par
  - sa position
  - son étiquette (plus lent, mais plus sûr)

```
> v[3]
```

```
c
5
```

```
> v["c"]
```

```
c
5
```

Sommaire

Une matrice est un vecteur

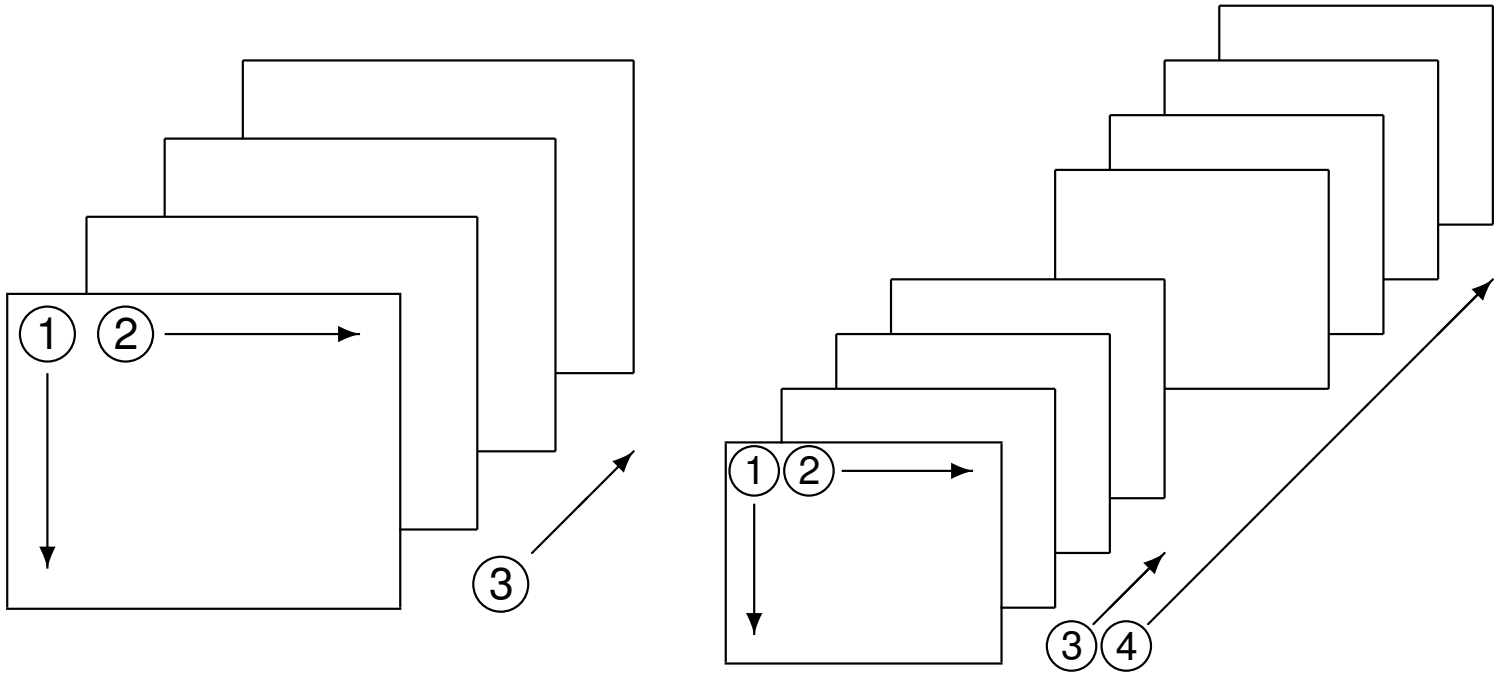
- Matrices et tableaux sont des vecteurs dotés d'un attribut `dim`
- À l'interne, matrice stockée sous forme de vecteur
- Fonction de base pour créer des matrices est `matrix`
- Fonction de base pour créer des tableaux est `array`

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
```

Remplissage des tableaux



Indiçage d'une matrice

- Extraction d'une éléments par sa position (ligne, colonne) dans la matrice, ou encore par sa position dans le vecteur sous-jacent

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

```
     [,1] [,2] [,3]
[1,]   40   45   55
[2,]   80   21   32
```

```
> m[1, 2]
```

```
[1] 45
```

```
> m[3]
```

```
[1] 45
```

Fusion verticale de matrices

- 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

- 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

- Vecteur spécial dont les éléments peuvent être de n'importe quel mode
- Permet d'emboîter des listes $\Rightarrow$ objet *récursif*
- Fonction de base pour créer des listes est `list`
- Conseil : nommer les éléments d'une liste !

Indiçage d'une liste

- Simples crochets `[ ]` retournent une liste
- Doubles crochets `[[ ]]` retournent l'élément seul
- Par l'étiquette avec `liste$element`

Sommaire

Un peu d'une liste et d'une matrice

- Liste de classe `data.frame` dont tous les éléments sont de la même longueur
- Visuellement identique à une matrice
- Plus général qu'une matrice : les colonnes peuvent être de modes différents
- Fonctions `data.frame` ou `as.data.frame`

Sommaire

1. Avec un vecteur d'entiers positifs

Éléments se trouvant aux positions correspondant aux entiers sont extraits, dans l'ordre :

```
> letters[c(1:3, 22, 5)]
```

```
[1] "a" "b" "c" "v" "e"
```

2. Avec un vecteur d'entiers négatifs

Éléments se trouvant aux positions correspondant aux entiers négatifs sont **éliminés** :

```
> 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

- Vecteur d'indiçage doit être de la même longueur que le vecteur indicé
- Éléments correspondant à `TRUE` sont **extraits**, ceux correspondant `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

- À condition que les éléments 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
- 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

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

Spécification des arguments d'une fonction

- Pas de limite pratique au nombre d'arguments
- Dans l'ordre établi dans la définition de la fonction
- Plus prudent et **fortement recommandé** de spécifier par le nom des arguments, surtout après les deux ou trois premiers
- 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

Spécification des arguments d'une fonction

- Pas de limite pratique au nombre d'arguments
- Dans l'ordre établi dans la définition de la fonction
- Plus prudent et **fortement recommandé** de spécifier par le nom des arguments, surtout après les deux ou trois premiers
- 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

Spécification des arguments d'une fonction

- Pas de limite pratique au nombre d'arguments
- Dans l'ordre établi dans la définition de la fonction
- Plus prudent et **fortement recommandé** de spécifier par le nom des arguments, surtout après les deux ou trois premiers
- 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

Spécification des arguments d'une fonction

- Pas de limite pratique au nombre d'arguments
- Dans l'ordre établi dans la définition de la fonction
- Plus prudent et **fortement recommandé** de spécifier par le nom des arguments, surtout après les deux ou trois premiers
- 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

Spécification des arguments d'une fonction

- Pas de limite pratique au nombre d'arguments
- Dans l'ordre établi dans la définition de la fonction
- Plus prudent et **fortement recommandé** de spécifier par le nom des arguments, surtout après les deux ou trois premiers
- 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

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)
- 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)
- Un appel à `matrix` sans argument résulte en

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

- Appel plus élaboré utilisant tous les arguments (le premier est rarement nommé) :

```
> matrix(16, 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

- Dans R, un ensemble de fonctions est appelé un **package**
- Par défaut, R charge en mémoire quelques packages de la bibliothèque (*library*) 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`	suites de nombres
`rep`	répétition de valeurs ou de vecteurs
`sort`	tri en ordre croissant ou décroissant
`order`	positions des valeurs en ordre croissant ou décroissant
`rank`	rang des éléments en ordre croissant ou décroissant
`rev`	renverser un vecteur
`head`	extraction des *n* premières valeurs ou suppression des *n* dernières
`tail`	extraction des *n* dernières valeurs ou suppression des *n* premières
`unique`	éléments différents

Recherche d'éléments dans un vecteur

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

Arrondi

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

Sommaires et statistiques descriptives

`sum`, `prod`	somme, produit
`diff`	différences
`mean`	moyennes arithmétique et tronquée
`var`, `sd`	variance, écart type
`min`, `max`	minimum, maximum
`range`	minimum **et** maximum
`median`	médiane empirique
`quantile`	quantiles empiriques
`summary`	statistiques descriptives usuelles

Sommaires cumulatifs et comparaisons élément par élément

`cumsum`, `cumprod`	somme cumulative, produit cumulatif
`cummin`, `cummax`	minimum et maximum cumulatif
`pmin`, `pmax`	minimum et maximum élément par élément (parallèle)

Opérations sur les matrices

`t`	transposée
`solve`	inverse et résolution de systèmes d'équations linéaires
`diag`	extraction de la diagonale d'une matrice ; création d'une matrice diagonale ; création d'une matrice identité
`nrow`, `ncol`	nombre de lignes, de colonnes d'une matrice

Opérations sur les matrices (suite)

`rowSums`, `colSums`	sommes par ligne, sommes par colonne
`rowMeans`, `colMeans`	moyennes par ligne, moyennes par colonne

Produit extérieur

- Syntaxe : `outer(X, Y, FUN)`
- Applique la fonction `FUN` (`prod` par défaut) entre chacun des éléments de `X` et chacun des éléments de `Y`
- Dimension du résultat est `c(dim(X), dim(Y))`
- Exemple :

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

- Raccourci : `X %o% Y`

Produit extérieur

- Syntaxe : `outer(X, Y, FUN)`
- Applique la fonction `FUN` (`prod` par défaut) entre chacun des éléments de `X` et chacun des éléments de `Y`
- Dimension du résultat est `c(dim(X), dim(Y))`
- Exemple :

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

- Raccourci : `X %o% Y`

Produit extérieur

- Syntaxe : `outer(X, Y, FUN)`
- Applique la fonction `FUN` (`prod` par défaut) entre chacun des éléments de `X` et chacun des éléments de `Y`
- Dimension du résultat est `c(dim(X), dim(Y))`
- Exemple :

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

- Raccourci : `X %o% Y`

Sommaire

Exécution conditionnelle

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

- Exécuter *`branche.vrai`* si *`condition`* est vraie et *`branche.faux`* sinon
- Lorsqu'une branche comporte plus d'une expression, les grouper dans des accolades { }

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

- Fonction vectorisée
- Pour chaque `TRUE` de *`vecteur.condition`*, retourne l'élément correspondant de *`vecteur.vrai`* et pour chaque `FALSE` l'élément correspondant de *`vecteur.faux`*

Boucles

- Les boucles sont et **doivent** être utilisées avec parcimonie en S car elles sont généralement inefficaces
- En général possible de vectoriser les calcul pour éviter les boucles explicites
- Utiliser aussi les fonctions `apply`, `lapply`, `sapply` et `mapply`

Boucles de longueur déterminée

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

- Exécuter *`expression`* pour chaque valeur de *`variable`* contenue dans *`suite`*
- Grouper les expressions dans des accolades { }
- *`suite`* n'a pas à être composée de nombres consécutifs, ni même 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
- `while` pas toujours exécutée

`repeat` *`expression`*

- Répéter *`expression`*
- Nécessite un test d'arrêt avec un `break`
- `repeat` 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. . .

a) Paiement annuel de 1000, taux d'intérêt de 6 % effectif annuellement

b) Paiements annuels de 500, 800, 900, 750 et 1000, taux d'intérêt de 6 % effectif annuellement

c) Paiements annuels de 500, 800, 900, 750 et 1000, taux d'intérêt de 5 %, 6 %, 5,5 %, 6,5 % et 7 % effectifs annuellement

Solution

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

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

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

Cas spécial

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

En R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R:

```
> 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 R :

```
> 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 R :

```
> 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 R :

```
> 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 R :

```
> 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 R :

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

```
[1] 3308.521
```

□

Sommaire

Sommaire

Fonction de répartition de la loi gamma

Nous utilisons 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

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.

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

```
> 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.

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

```
> 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.

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

```
> 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.

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

```
> 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.

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

```
> 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 éviter de générer la même suite de nombres à deux reprises :

```
> 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`.

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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
```

b) Plusieurs valeurs de x, paramètre α fixe.

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

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

```
> 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 $|x_n - x_{n-1}| / |x_{n-1}| < \varepsilon$

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`

Le code.

$$i_n = \frac{1 - (1 + i_{n-1})^{-10}}{8{,}21}, \quad n = 1, 2, \ldots$$

```
> 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.

$$i_n = \frac{1 - (1 + i_{n-1})^{-10}}{8{,}21}, \quad n = 1, 2, \ldots$$

```
> 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` : nom de la fonction
- *`arguments`* : liste des arguments, séparés par des virgules
- *`expression`* : corps de la fonction, suite d'expressions groupées entre accolades

Sommaire

Retourner des résultats

- Une fonction retourne le résultat de la **dernière expression** du corps
- Éviter que la dernière expression soit une affectation : la fonction ne retournera rien !
- Parfois nécessaire d'utiliser explicitement `return`
- Utiliser une liste nommée pour retourner plusieurs résultats

Sommaire

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur $<<-$
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur $<<-$
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur $<<-$
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur $<<-$
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- ~~Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur <<-~~
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

Variables locales et globales

- Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire
  - n'apparaît pas dans l'espace de travail
  - n'écrase pas une variable du même nom dans l'espace de travail
- ~~Possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur <<-~~
- Fonction définie à l'intérieur d'une autre fonction locale à celle-ci

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
}
```

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
}
```

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
}
```

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
}
```

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
}
```

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
}
```

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
- 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
- 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
```

- Avec une fonction anonyme :

```
> 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
- Première approche :
  1. placer des commandes `print` à l'intérieur de la fonction
  2. exécuter la fonction normalement
- Deuxième approche :
  1. définir toutes les variables passées en arguments dans l'espace de travail
  2. exécuter les lignes de la fonction une à une

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
- Première approche :
  1. placer des commandes `print` à l'intérieur de la fonction
  2. exécuter la fonction normalement
- Deuxième approche :
  1. définir toutes les variables passées en arguments dans l'espace de travail
  2. exécuter les lignes de la fonction une à une

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
- Première approche :
  1. placer des commandes `print` à l'intérieur de la fonction
  2. exécuter la fonction normalement
- Deuxième approche :
  1. définir toutes les variables passées en arguments dans l'espace de travail
  2. exécuter les lignes de la fonction une à une

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
- Première approche :
  1. placer des commandes `print` à l'intérieur de la fonction
  2. exécuter la fonction normalement
- Deuxième approche :
  1. définir toutes les variables passées en arguments dans l'espace de travail
  2. exécuter les lignes de la fonction une à une

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
}
```

Sommaire

Sommaire

Pas un signe de paresse des rédacteurs

- ‘. . .’ est un argument formel dont ‘. . .’ est le nom
- Fonction accepte un ou plusieurs arguments autres que ceux faisant partie de sa définition
- Contenu de ‘. . .’ ni pris en compte, ni modifié par la fonction
- Passé tel quel à une autre fonction
- Exemples : `apply`, `lapply`, `sapply`

Exemple

```
> f <- function(x, y) x + y
> g <- function(z, ...) z * f(...)
> g(2, 3, 4)
[1] 14
```

Exemple

```
> f <- function(x, y) x + y
> g <- function(z, ...) z * f(...)
> g(2, 3, 4)
[1] 14
```

Sommaire

Sommaires généraux pour matrices et tableaux

`apply` applique une fonction quelconque sur une partie d'une matrice ou d'un tableau

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

où

- `X` : matrice ou tableau
- `MARGIN` : vecteur de la ou des dimensions sur lesquelles la fonction doit s'appliquer
- `FUN` : fonction à appliquer
- '`...`' : arguments supplémentaires à passer à `FUN`

Sommaires généraux pour matrices et tableaux

`apply` applique une fonction quelconque sur une partie d'une matrice ou d'un tableau

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

où

- `X` : matrice ou tableau
- `MARGIN` : vecteur de la ou des dimensions sur lesquelles la fonction doit s'appliquer
- `FUN` : fonction à appliquer
- '`...`' : arguments supplémentaires à passer à `FUN`

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

```
> 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

- `lapply` et `sapply` appliquent une fonction aux éléments d'un **vecteur** ou d'une **liste**
- Syntaxe similaire à `apply` :

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

Des fonctions très utiles

- `lapply` applique `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
- Souvent 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

- Simplifie la syntaxe pour l'exécution répétée d'une expression
- Particulièrement indiqué pour les simulations
- Si la fonction `fun` fait tous les calculs d'une simulation, on obtient les résultats pour 10000 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
- Classe parfois implicite ou dérivée du mode de l'objet
- Fonctions **génériques** se comportent différemment selon la classe de l'objet donné en argument
- Fonctions génériques les plus fréquemment employées : `print`, `plot`, `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
- Éditeur pour programmeurs avec des modes spéciaux pour une multitude de langages différents
- Environnement idéal pour travailler sur des documents LaTeX, interagir avec R, S+, SAS ou SQL, ou même pour lire son courrier électronique

Mise en contexte

- Logiciel phare 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»
- Nom provient de «*Editing MACroS*»
- Première version écrite par Richard M. Stallman, président de la FSF

Configuration de l'éditeur

- Emacs est configurable à l'envi
- Configuration relativement aisée depuis le menu `Customize`
- Configuration personnelle dans le fichier `.emacs`

Emacs-ismes et Unix-ismes

- Un **buffer** contient un fichier ouvert («*visited*»)
- Le **minibuffer** est la région au bas de la fenêtre 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

Commandes

- 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 `Ctrl` (`Control`)
  - `M` est `Meta` (`Alt` sur un PC, option ou commande sur un Mac)
  - `ESC` est `Échap` et est équivalente à `Meta`
  - `SPC` est la barre d'espacement
  - `RET` est la touche Entrée

- Caractère ~ représente le dossier `$HOME` (Unix) ou `%HOME%` (Windows)
- Barre oblique (/) utilisée pour séparer les dossiers dans les chemins d'accès aux fichiers
- En général possible d'appuyer sur `TAB` dans le *minibuffer* pour compléter les noms de fichiers ou de commandes

Commandes d'édition de base

- Lire le tutoriel de Emacs, que l'on démarre avec

  `C-h t`

- Voir aussi la *GNU Emacs Reference Card* à l'adresse

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

- Pour créer un nouveau fichier, ouvrir un fichier qui n'existe 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

- Raccourcis clavier standards sous Emacs :

  `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

- Des extensions permettent d'utiliser les raccourcis usuels (`C-c`, `C-x`, `C-v`)

Sommaire

Emacs Speaks Statistics

- 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
- Le mode mineur `ESS` s'active automatiquement en éditant des fichiers `.S` ou `.R`

Démarrer un processus S

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)

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