Introduction à la programmation en R

Vincent Goulet

Introduction à la programmation en R

Vincent Goulet
École d'actuariat, Université Laval

Avec la collaboration de
Laurent Caron

Cinquième édition révisée

© 2016 Vincent Goulet

Cette création est mise à disposition selon le contrat Attribution-Partage dans les mêmes conditions 4.0 International de Creative Commons. En vertu de ce contrat, vous êtes libre de :

- **partager** — reproduire, distribuer et communiquer l'œuvre ;
- **remixer** — adapter l'œuvre ;
- utiliser cette œuvre à des fins commerciales.

Selon les conditions suivantes :

Attribution — Vous devez créditer l'œuvre, intégrer un lien vers le contrat et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens possibles, mais vous ne pouvez suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.

Partage dans les mêmes conditions — Dans le cas où vous modifiez, transformez ou créez à partir du matériel composant l'œuvre originale, vous devez diffuser l'œuvre modifiée dans les même conditions, c'est à dire avec le même contrat avec lequel l'œuvre originale a été diffusée.

Code source

Code informatique des sections d'exemples

Code source du document

ISBN 978-2-9811416-6-8
Dépôt légal – Bibliothèque et Archives nationales du Québec, 2016
Dépôt légal – Bibliothèque et Archives Canada, 2016

Couverture

Le hibou en couverture est un harfang des neiges *(Bubo scandiacus)*, l'emblème aviaire du Québec. Ce choix relève également d'un clin d'œil à la couverture de Braun et Murdoch (2007).

Introduction

Le système R connaît depuis plus d'une décennie une progression remarquable dans ses fonctionnalités, dans la variété de ses domaines d'application ou, plus simplement, dans le nombre de ses utilisateurs. La documentation disponible suit la même tangente : plusieurs maisons d'édition proposent dans leur catalogue des ouvrages — voire des collections complètes — dédiés spécifiquement aux utilisations que l'on fait de R en sciences naturelles, en sciences sociales, en finance, etc. Néanmoins, peu d'ouvrages se concentrent sur l'apprentissage de R en tant que langage de programmation sous-jacent aux fonctionnalités statistiques. C'est la niche que nous tâchons d'occuper.

Le présent ouvrage se base sur des notes de cours et des exercices utilisés à l'École d'actuariat de l'Université Laval. L'enseignement du langage R est axé sur l'exposition à un maximum de code — que nous avons la prétention de croire bien écrit — et sur la pratique de la programmation. C'est pourquoi les chapitres sont rédigés de manière synthétique et qu'ils comportent peu d'exemples au fil du texte. En revanche, le lecteur est appelé à lire et à exécuter le code informatique se trouvant dans les sections d'exemples à la fin de chacun des chapitres. Ce code et les commentaires qui l'accompagnent reviennent sur l'essentiel des concepts du chapitre et les complémentent souvent. Nous considérons l'exercice d'« étude active » consistant à exécuter du code et à voir ses effet comme essentiel à l'apprentissage du langage R.

Afin d'ancrer la présentation dans un contexte concret, plusieurs chapitres proposent également d'entrée de jeu un problème à résoudre. Nous fournissons des indications en cours de chapitre et la solution complète à la fin. Afin d'être facilement identifiables, ces éléments de contenu se présentent dans des encadrés de couleur contrastante et marqués des symboles ⚙, ϟ et 💡.

Pour télécharger le code R des sections d'exemples directement depuis le site du projet dans GitHub, utiliser le lien fourni à la page précédente.

Un symbole de lecture vidéo dans la marge (comme ci-contre) indique

qu'une capsule vidéo est disponible dans la chaîne YouTube de l'auteur sur le sujet en hyperlien.

Certains exemples et exercices trahissent le premier public de ce document : on y fait à l'occasion référence à des concepts de base de la théorie des probabilités et des mathématiques financières. Les contextes actuariels demeurent néanmoins peu nombreux et ne devraient généralement pas dérouter le lecteur pour qui ces notions sont moins familières. Les réponses de tous les exercices se trouvent en annexe. En consultation électronique, le numéro d'un exercice est un hyperlien vers sa réponse, et vice versa.

On trouvera également en annexe de brèves introductions à l'éditeur de texte GNU Emacs et à l'environnement de développement intégré RStudio, un court exposé sur la planification d'une simulation en R, ainsi que des conseils sur l'administration d'une bibliothèque de packages R.

Nous tenons à remercier M. Mathieu Boudreault pour sa collaboration dans la rédaction des exercices et Mme Mireille Côté pour la révision linguistique de la seconde édition.

Vincent Goulet
Québec, avril 2016

Table des matières

Introduction v

1 Présentation du langage R 1
- 1.1 Bref historique 1
- 1.2 Description sommaire de R 2
- 1.3 Interfaces 3
- 1.4 Stratégies de travail 4
- 1.5 Éditeurs de texte et environnements intégrés 5
- 1.6 Anatomie d'une session de travail 10
- 1.7 Répertoire de travail 10
- 1.8 Consulter l'aide en ligne 11
- 1.9 Où trouver de la documentation 11
- 1.10 Exemples 12
- 1.11 Exercices 13

2 Bases du langage R 15
- 2.1 Commandes R 16
- 2.2 Conventions pour les noms d'objets 18
- 2.3 Les objets R 20
- 2.4 Vecteurs 24
- 2.5 Matrices et tableaux 25
- 2.6 Listes 28
- 2.7 *Data frames* 30
- 2.8 Indiçage 32
- 2.9 Exemples 35
- 2.10 Exercices 46

3 Opérateurs et fonctions 49
- 3.1 Opérations arithmétiques 50

3.2 Opérateurs 51
3.3 Appels de fonctions 52
3.4 Quelques fonctions utiles 53
3.5 Structures de contrôle 61
3.6 Fonctions additionnelles 62
3.7 Exemples 64
3.8 Exercices 72

4 Exemples résolus 75
4.1 Calcul de valeurs actuelles 76
4.2 Fonctions de masse de probabilité 77
4.3 Fonction de répartition de la loi gamma 79
4.4 Algorithme du point fixe 81
4.5 Suite de Fibonacci 83
4.6 Exercices 85

5 Fonctions définies par l'usager 89
5.1 Définition d'une fonction 91
5.2 Retourner des résultats 91
5.3 Variables locales et globales 92
5.4 Exemple de fonction 92
5.5 Fonctions anonymes 93
5.6 Débogage de fonctions 94
5.7 Styles de codage 95
5.8 Exemples 97
5.9 Exercices 101

6 Concepts avancés 105
6.1 Argument '`...`' 106
6.2 Fonction `apply` 107
6.3 Fonctions `lapply` et `sapply` 110
6.4 Fonction `mapply` 112
6.5 Fonction `replicate` 113
6.6 Classes et fonctions génériques 114
6.7 Exemples 117
6.8 Exercices 126

7 Fonctions d'optimisation 131
7.1 Fonctions d'optimisation et de calcul de racines 132
7.2 Astuce Ripley 133

7.3 Pour en savoir plus 134
7.4 Exemples 134
7.5 Exercices 138

8 Génération de nombres aléatoires 141
8.1 Générateurs de nombres aléatoires uniformes 141
8.2 Simulation de variables aléatoires non uniformes 142
8.3 Exemples 144
8.4 Exercices 145

A GNU Emacs et ESS : la base 147
A.1 Mise en contexte 147
A.2 Installation 148
A.3 Description sommaire 149
A.4 *Emacs-ismes* et *Unix-ismes* 149
A.5 Commandes de base 151
A.6 Anatomie d'une session de travail (bis) 155
A.7 Configuration de l'éditeur 156
A.8 Aide et documentation 157

B RStudio : une introduction 159
B.1 Installation 159
B.2 Description sommaire 159
B.3 Projets 161
B.4 Commandes de base 162
B.5 Anatomie d'une session de travail (ter) 162
B.6 Configuration de l'éditeur 164
B.7 Aide et documentation 164

C Planification d'une simulation en R 167
C.1 Contexte 167
C.2 Première approche : avec une boucle 168
C.3 Seconde approche : avec `sapply` 168
C.4 Variante de la seconde approche 172
C.5 Gestion des fichiers 172
C.6 Exécution en lot 173
C.7 Conclusion 174

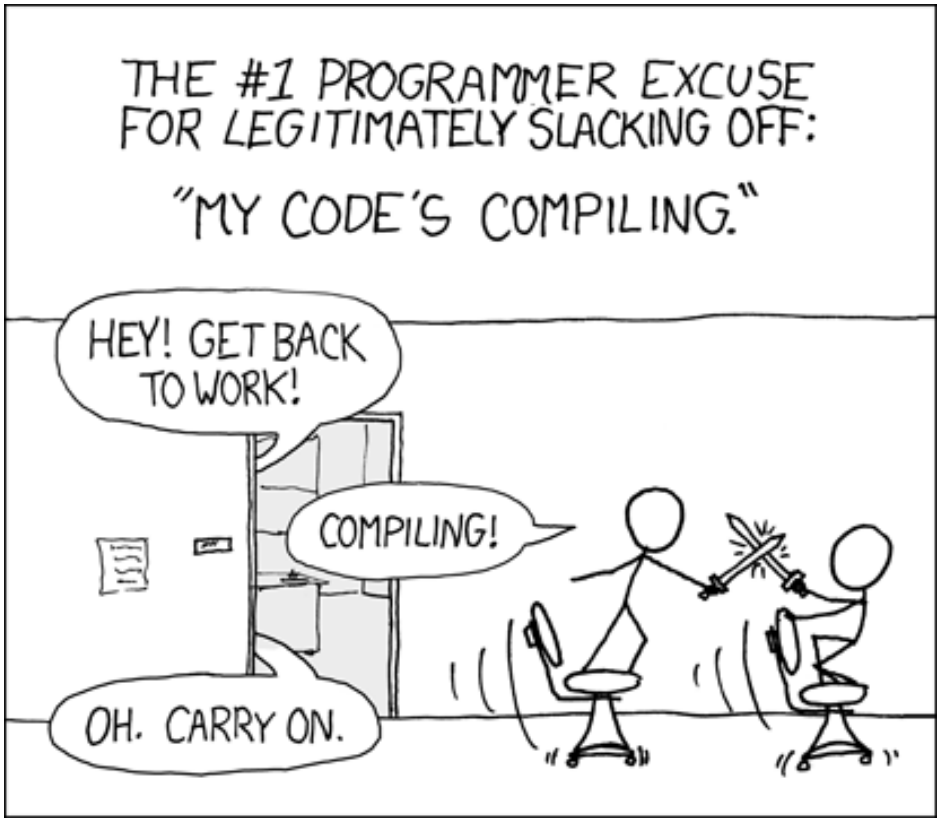
D Installation de packages dans R 175

Réponses des exercices 179

Chapitre 2 179
Chapitre 3 180
Chapitre 4 181
Chapitre 5 183
Chapitre 6 187
Chapitre 7 189
Chapitre 8 191

Bibliographie 193

Index 195



Tiré de XKCD.com

1 Présentation du langage R

Objectifs du chapitre

- Comprendre ce qu'est un langage de programmation interprété.
- Connaître la provenance du langage R et les principes ayant guidé son développement.
- Mettre en place sur son poste de travail un environnement de développement en R.
- Démarrer une session R et exécuter des commandes simples.
- Utiliser des fichiers de script R de manière interactive.
- Créer, modifier et sauvegarder ses propres fichiers de script R.

1.1 Bref historique

À l'origine fut le S, un langage pour « programmer avec des données » développé chez Bell Laboratories à partir du milieu des années 1970 par une équipe de chercheurs menée par John M. Chambers. Au fil du temps, le S a connu quatre principales versions communément identifiées par la couleur du livre dans lequel elles étaient présentées : version « originale » (*Brown Book ;* Becker et Chambers, 1984), version 2 (*Blue Book ;* Becker et collab., 1988), version 3 (*White Book ;* Chambers et Hastie, 1992) et version 4 (*Green Book ;* Chambers, 1998) ; voir aussi Chambers (2000) et Becker (1994) pour plus de détails.

Dès la fin des années 1980 et pendant près de vingt ans, le S a principalement été popularisé par une mise en œuvre commerciale nommée S-PLUS. En 2008, Lucent Technologies a vendu le langage S à Insightful Corporation, ce qui a effectivement stoppé le développement du langage par ses auteurs originaux. Aujourd'hui, le S est commercialisé de manière relativement confidentielle sous le nom Spotfire S+ par TIBCO Software.

Ce qui a fortement contribué à la perte d'influence de S-PLUS, c'est une nouvelle mise en œuvre du langage développée au milieu des années 1990. Inspirés à la fois par le S et par Scheme (un dérivé du Lisp), Ross Ihaka et Robert Gentleman proposent un langage pour l'analyse de données et les graphiques qu'ils nomment R (Ihaka et Gentleman, 1996). À la suggestion de Martin Maechler de l'ETH de Zurich, les auteurs décident d'intégrer leur nouveau langage au projet GNU [1], faisant de R un logiciel libre.

Ainsi disponible gratuitement et ouvert aux contributions de tous, R gagne rapidement en popularité là même où S-PLUS avait acquis ses lettres de noblesse, soit dans les milieux académiques. De simple dérivé « *not unlike S* », R devient un concurrent sérieux à S-PLUS, puis le surpasse lorsque les efforts de développement se rangent massivement derrière le projet libre. D'ailleurs John Chambers place aujourd'hui ses efforts de réflexion et de développement dans le projet R (Chambers, 2008).

1.2 Description sommaire de R

R est un environnement intégré de manipulation de données, de calcul et de préparation de graphiques. Toutefois, ce n'est pas seulement un « autre » environnement statistique (comme SPSS ou SAS, par exemple), mais aussi un langage de programmation complet et autonome.

Tel que mentionné précédemment, le R est un langage principalement inspiré du S et de Scheme (Abelson et collab., 1996). Le S était à son tour inspiré de plusieurs langages, dont l'APL (autrefois un langage très prisé par les actuaires) et le Lisp. Comme tous ces langages, le R est *interprété*, c'est-à-dire qu'il requiert un autre programme — l'*interprète* — pour que ses commandes soient exécutées. Par opposition, les programmes de langages *compilés*, comme le C ou le C++, sont d'abord convertis en code machine par le compilateur puis directement exécutés par l'ordinateur.

Cela signifie donc que lorsque l'on programme en R, il n'est pas possible de plaider l'attente de la fin de la phase de compilation pour perdre son temps au travail. Désolé !

Le programme que l'on lance lorsque l'on exécute R est en fait l'interprète. Celui-ci attend que l'on lui soumette des commandes dans le langage R, commandes qu'il exécutera immédiatement, une à une et en séquence.

Par analogie, Excel est certes un logiciel de manipulation de données, de mise en forme et de préparation de graphiques, mais c'est aussi au sens large

1. http://www.gnu.org

un langage de programmation interprété. On utilise le langage de programmation lorsque l'on entre des commandes dans une cellule d'une feuille de calcul. L'interprète exécute les commandes et affiche les résultats dans la cellule.

Le R est un langage particulièrement puissant pour les applications mathématiques et statistiques (et donc actuarielles) puisque précisément développé dans ce but. Parmi ses caractéristiques particulièrement intéressantes, on note :

- langage basé sur la notion de vecteur, ce qui simplifie les calculs mathématiques et réduit considérablement le recours aux structures itératives (boucles `for`, `while`, etc.) ;
- pas de typage ni de déclaration obligatoire des variables ;
- programmes courts, en général quelques lignes de code seulement ;
- temps de développement très court.

1.3 Interfaces

R est d'abord et avant tout une application n'offrant qu'une invite de commande du type de celle présentée à la figure 1.1. En soi, cela n'est pas si différent d'un tableur tel que Excel : la zone d'entrée de texte dans une cellule n'est rien d'autre qu'une invite de commande [2], par ailleurs aux capacités d'édition plutôt réduites.

- Sous Windows, une interface graphique plutôt rudimentaire est disponible. Elle facilite certaines opérations tel que l'installation de packages externes, mais elle offre autrement peu de fonctionnalités additionnelles pour l'édition de code R.
- L'interface graphique de R sous macOS est la plus élaborée. Outre la console présentée à la figure 1.1, l'application `R.app` comporte de nombreuses fonctionnalités, dont un éditeur de code assez complet.
- Sous Unix et Linux, R n'est accessible que depuis la ligne de commande du système d'exploitation (terminal). Aucune interface graphique n'est offerte avec la distribution de base de R.

Peu importe la plateforme utilisée — quoique dans une moindre mesure sous macOS — nous recommandons d'interagir avec R par le biais d'un éditeur de texte pour programmeur ou d'un environnement de développement intégré ; voir la section 1.5.

2. Merci à Markus Gesmann pour cette observation.

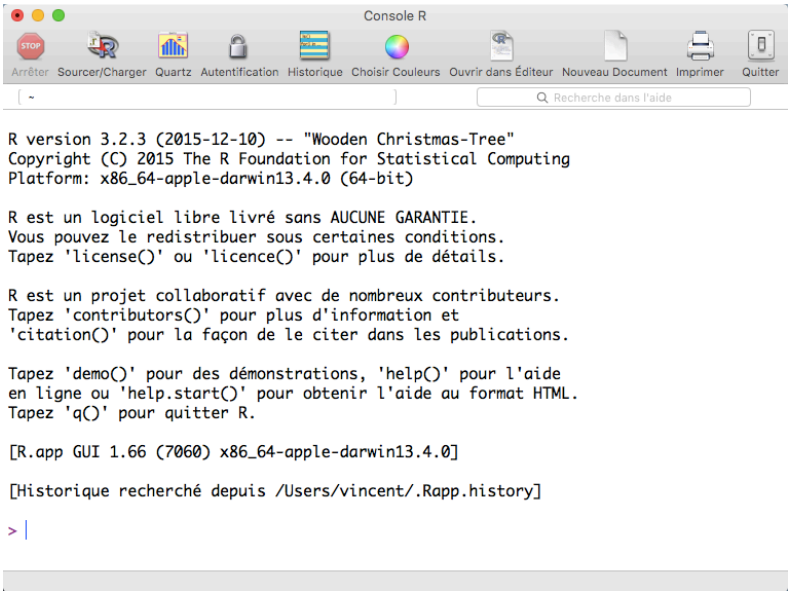


FIG. 1.1 – Fenêtre de la console sous macOS au démarrage de R

1.4 Stratégies de travail

Dans la mesure où R se présente essentiellement sous forme d'une invite de commande, il existe deux grandes stratégies de travail avec cet environnement statistique.

1. On entre des expressions à la ligne de commande pour les évaluer immédiatement :

```
> 2 + 3
[1] 5
```

   On peut également créer des objets contenant le résultat d'un calcul. Ces objets sont stockés en mémoire dans l'espace de travail de R :

```
> x <- exp(2)
> x
[1] 7.389056
```

   Lorsque la session de travail est terminée, on sauvegarde une image de l'espace de travail sur le disque dur de l'ordinateur afin de pouvoir conserver les objets pour une future séance de travail :

```
> save.image()
```

Par défaut, l'image est sauvegardée dans un fichier nommé `.RData` dans le dossier de travail actif (voir la section 1.7) et cette image est automatiquement chargée en mémoire au prochain lancement de R, tel qu'indiqué à la fin du message d'accueil :

```
[Sauvegarde de la session précédente restaurée]
```

Cette approche, dite de « code virtuel et objets réels » a un gros inconvénient : le code utilisé pour créer les objets n'est pas sauvegardé entre les sessions de travail. Or, celui-ci est souvent bien plus compliqué que l'exemple ci-dessus. De plus, sans accès au code qui a servi à créer l'objet `x`, comment savoir ce que la valeur `7.389056` représente au juste ?

2. L'approche dite de « code réel et objets virtuels » considère que ce qu'il importe de conserver d'une session de travail à l'autre n'est pas tant les objets que le code qui a servi à les créer. Ainsi, on sauvegardera dans ce que l'on nommera des *fichiers de script* nos expressions R et le code de nos fonctions personnelles. Par convention, on donne aux fichiers de script un nom se terminant avec l'extension `.R`.

   Avec cette approche, les objets sont créés au besoin en exécutant le code des fichiers de script. Comment ? Simplement en copiant le code du fichier de script et en le collant dans l'invite de commande de R. La figure 1.2 illustre schématiquement ce que le programmeur R a constamment sous les yeux : d'un côté son fichier de script et, de l'autre, l'invite de commande R dans laquelle son code a été exécuté.

   La méthode d'apprentissage préconisée dans cet ouvrage suppose que le lecteur utilisera cette seconde approche d'interaction avec R.

1.5 Éditeurs de texte et environnements intégrés

Dans la mesure où l'on a recours à des fichiers de script tel qu'expliqué à la section précédente, l'édition de code R est rendue beaucoup plus aisée avec un bon éditeur de texte pour programmeur ou un environnement de développement intégré (*integrated development environment*, IDE).

- Un éditeur de texte est différent d'un traitement de texte en ce qu'il s'agit d'un logiciel destiné à la création, l'édition et la sauvegarde de fichiers textes purs, c'est-à-dire dépourvus d'information de présentation et de mise en forme. Les applications Bloc-notes sous Windows ou TextEdit sous macOS sont deux exemples d'éditeurs de texte simples.

```
## Fichier de script simple contenant des expressions R pour
## faire des calculs et créer des objets.
2 + 3

## Probabilité d'une loi de Poisson(10)
x <- 7
10^x * exp(-10) / factorial(x)

## Petite fonction qui fait un calcul trivial
f <- function(x) x^2

## Évaluation de la fonction
f(2)
```

```
R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

[...]

> ## Fichier de script simple contenant des expressions R pour
> ## faire des calculs et créer des objets.
> 2 + 3
[1] 5
>
> ## Probabilité d'une loi de Poisson(10)
> x <- 7
> 10^x * exp(-10) / factorial(x)
[1] 0.09007923
>
> ## Petite fonction qui fait un calcul trivial
> f <- function(x) x^2
>
> ## Évaluation de la fonction
> f(2)
[1] 4
```

FIG. 1.2 – Fichier de script (en haut) et invite de commande R dans laquelle les expressions R ont été exécutées (en bas). Les lignes débutant par # dans le fichier de script sont des commentaires ignorés par l'interprète de commandes.

Toute analogie est boîteuse, mais celle-ci peut néanmoins s'avérer utile pour illustrer l'approche « code réel, objets virtuels ». Avec un tableur comme Excel, que choisiriez-vous si l'on vous proposait les options suivantes :

1. ne sauvegarder que les valeurs calculées d'une feuille de calcul ;
2. ne sauvegarder que les formules d'une feuille de calcul, quitte à les réévaluer par la suite pour obtenir les valeurs correspondantes.

La seconde, sûrement… Or, sauvagarder l'espace de travail R dans un fichier `.RData` (l'approche « objets réels, code virtuel ») correspond à la première option, alors que sauvegarder le code source des fichiers de script correspond à la seconde.
Là où l'analogie est boîteuse, la raison pour laquelle on n'a jamais à se poser la question avec un tableur, c'est que le progiciel évalue constamment et en temps réel les formules pour afficher les résultats dans les cellules.

- Un éditeur de texte pour programmeur saura en plus reconnaître la syntaxe d'un langage de programmation et assister à sa mise en forme : indentation automatique du code, marquage des mots-clés, manipulation d'objets, etc.
- Un éditeur compatible avec R réduira, entre autres, l'opération de copier-coller à un simple raccourci clavier.

Le lecteur peut utiliser le logiciel de son choix pour l'édition de code R. Certains offrent simplement plus de fonctionnalités que d'autres.

- Dans la catégorie des éditeurs de texte, nous recommandons le vénérable et très puissant éditeur pour programmeur GNU Emacs. À la question 6.2 de la foire aux questions de R (Hornik, 2013), « Devrais-je utiliser R à l'intérieur de Emacs ? », la réponse est : « Oui, absolument. »

  En effet, combiné avec le mode ESS (*Emacs Speaks Statistics*), Emacs offre un environnement de développement aussi riche qu'efficace. Entre autres fonctionnalités uniques à cet éditeur, le fichier de script et l'invite de commandes R sont regroupés dans la même fenêtre, comme on peut le voir à la figure 1.3.

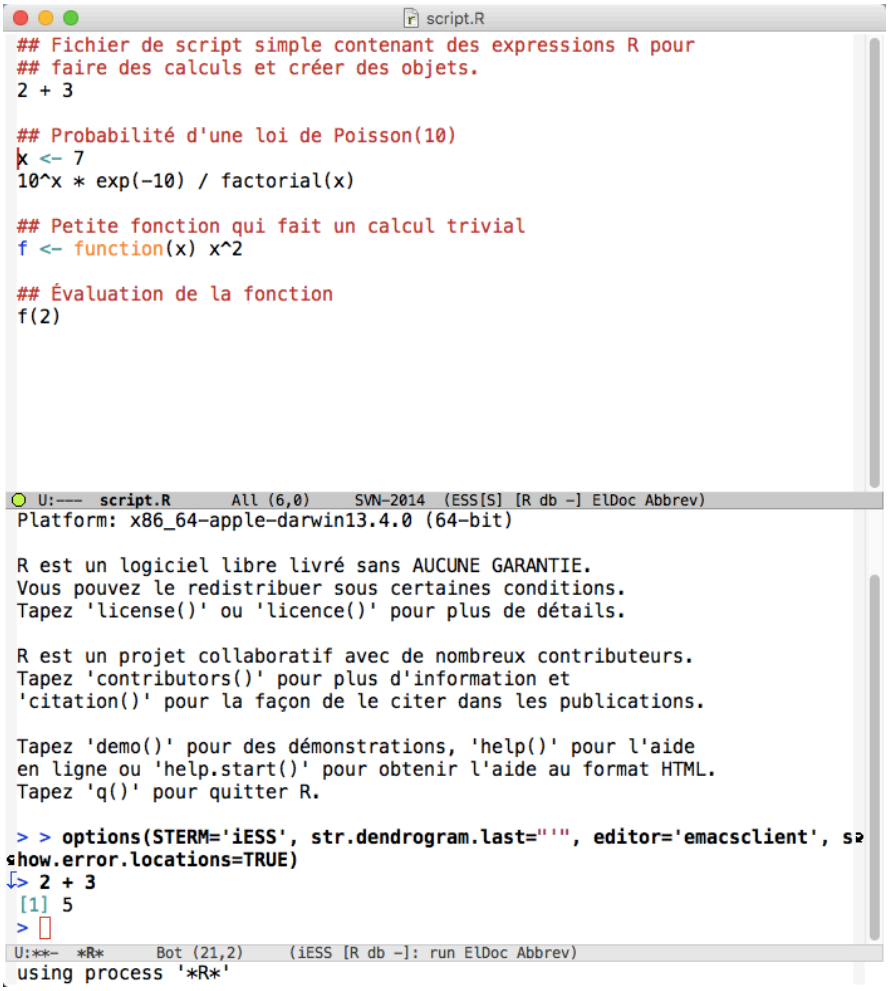


FIG. 1.3 – Fenêtre de GNU Emacs sous macOS en mode d'édition de code R. Dans la partie du haut, on retrouve le fichier de script de la figure 1.2 et dans la partie du bas, l'invite de commandes R.

Emblême du logiciel libre, Emacs est disponible gratuitement et à l'identique sur toutes les plateformes supportées par R, dont Windows, macOS et Linux.

Consulter l'annexe A pour en savoir plus sur GNU Emacs et apprendre les commandes essentielles pour y faire ses premiers pas.

- Emacs est toutefois un logiciel difficile à apprivoiser, surtout pour les personnes moins à l'aise avec l'informatique.

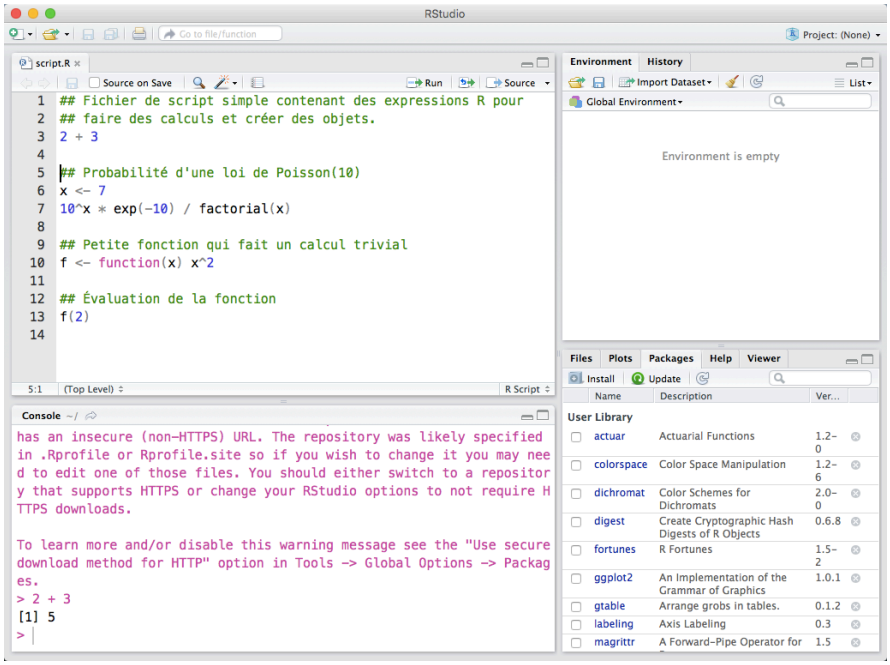


FIG. 1.4 – Fenêtre de RStudio sous macOS dans sa configuration par défaut. Dans le sens des aiguilles d'une montre, on y retrouve : le fichier de script de la figure 1.2 ; la liste des objets de l'environnement (vide, ici) ; une interface pour charger des packages ; l'invite de commandes R.

- RStudio est un environnement de développement intégré (IDE) créé spécifiquement pour travailler avec R. Sa popularité connaît une progression foudroyante depuis 2014. Il permet de consulter dans une interface conviviale ses fichiers de script, la ligne de commande R, les rubriques d'aide, les graphiques, etc. ; voir la figure 1.4.

  RStudio est disponible sur les plateformes Windows, macOS et Linux.

- Il existe plusieurs autres options pour éditer efficacement du code R — et le Bloc-notes de Windows n'en fait *pas* partie ! Nous recommandons plutôt :

  - sous Windows, l'éditeur Notepad++ additionné de l'extension NppToR (Redd, 2010), tous deux des logiciels libres, ou le partagiciel WinEdt muni de l'extension libre R-WinEdt (Ligges, 2003) ;
  - sous macOS, tout simplement l'éditeur de texte très complet intégré à l'application `R.app`, ou alors l'éditeur de texte commercial TextMate (essai gratuit de 30 jours) ;

– sous Linux, Vim et Kate semblent les choix les plus populaires dans la communauté R, après Emacs et RStudio.

1.6 Anatomie d'une session de travail

Dans ses grandes lignes, une session de travail avec R réunit les étapes ci-dessous.

1. Démarrer une session R en cliquant sur l'icône de l'application si l'on utilise une interface graphique ou RStudio, ou alors en suivant la procédure appropriée pour travailler avec son éditeur de texte.
2. Ouvrir un fichier de script existant ou en créer un nouveau à l'aide de l'éditeur de texte de son choix.
3. Bâtir graduellement un fichier de script en y consignant le code R que l'on souhaite sauvegarder et les commentaires qui permettront de s'y retrouver plus tard. Tester les commandes à la ligne de commande. Au cours de la phase de développement, on fera généralement de nombreux aller-retours entre la ligne de commande et le fichier de script.
4. Sauvegarder son fichier de script et quitter l'éditeur ou l'environnement de développement.
5. Si nécessaire — et c'est rarement le cas — sauvegarder l'espace de travail de la session R avec `save.image()`. En fait, on ne voudra sauvegarder nos objets R que lorsque ceux-ci sont très longs à créer comme, par exemple, les résultats d'une simulation.
6. Quitter R en tapant `q()` à la ligne de commande ou en fermant l'interface graphique par la procédure usuelle. Encore ici, la manière de procéder est quelque peu différente dans GNU Emacs ; voir l'annexe A.

Les étapes 1 et 2 sont interchangeables, tout comme les étapes 4, 5 et 6. L'annexe A explique plus en détails la procédure à suivre lorsque l'on utilise GNU Emacs. L'annexe B fait de même lorsque l'on utilise l'environnement intégré RStudio.

1.7 Répertoire de travail

Le répertoire de travail (*workspace*) de R est le dossier par défaut dans lequel le logiciel : 1) va rechercher des fichiers de script ou de données ; et 2) va sauvegarder l'espace de travail dans le fichier `.RData`. Le répertoire de travail est déterminé au lancement de R.

- Les interfaces graphiques de R démarrent avec un répertoire de travail par défaut. Pour le changer, utiliser l'entrée appropriée dans le menu `Fichier` (Windows) ou `Divers` (macOS). Consulter aussi les foires aux questions spécifiques aux interfaces graphiques (Ripley et Murdoch, 2013 ; Iacus et collab., 2016) pour des détails additionnels sur la gestion des répertoires de travail.
- Dans RStudio, on change le répertoire de travail via le menu `Session`.
- Avec GNU Emacs, la situation est un peu plus simple puisque l'on doit spécifier un répertoire de travail chaque fois que l'on démarre un processus R ; voir l'annexe A.

1.8 Consulter l'aide en ligne

Les rubriques d'aide des diverses fonctions de R 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
```

ou

```
> help(foo)
```

1.9 Où trouver de la documentation

La documentation officielle de R se compose de six guides accessibles depuis le menu `Aide` des interfaces graphiques ou encore en ligne dans le site du projet R[3]. Pour le débutant, seuls *An Introduction to R* et, possiblement, *R Data Import/Export* peuvent s'avérer des ressources utiles à court terme.

Plusieurs livres — en versions papier ou électronique, gratuits ou non — ont été publiés sur R. On en trouvera une liste exhaustive dans la section Documentation du site du projet R.

Depuis plusieurs années maintenant, les ouvrages de Venables et Ripley (2000, 2002) demeurent des références standards *de facto* sur les langages S et R. Plus récent, Braun et Murdoch (2007) participe du même effort que le présent ouvrage en se concentrant sur la programmation en R plutôt que sur ses applications statistiques.

3. http://www.r-project.org

1.10 Exemples

```
### Générer deux vecteurs de nombres pseudo-aléatoires issus
### d'une loi normale centrée réduite.
x <- rnorm(50)
y <- rnorm(x)

### Graphique des couples (x, y).
plot(x, y)

### Graphique d'une approximation de la densité du vecteur x.
plot(density(x))

### Générer la suite 1, 2, ..., 10.
1:10

### La fonction 'seq' sert à générer des suites plus générales.
seq(from = -5, to = 10, by = 3)
seq(from = -5, length = 10)

### La fonction 'rep' sert à répéter des valeurs.
rep(1, 5)        # répéter 1 cinq fois
rep(1:5, 5)      # répéter le vecteur 1,...,5 cinq fois
rep(1:5, each = 5) # répéter chaque élément du vecteur cinq fois

### Arithmétique vectorielle.
v <- 1:12        # initialisation d'un vecteur
v + 2            # additionner 2 à chaque élément de v
v * -12:-1       # produit élément par élément
v + 1:3          # le vecteur le plus court est recyclé

### Vecteur de nombres uniformes sur l'intervalle [1, 10].
v <- runif(12, min = 1, max = 10)
v

### Pour afficher le résultat d'une affectation, placer la
### commande entre parenthèses.
( v <- runif(12, min = 1, max = 10) )

### Arrondi des valeurs de v à l'entier près.
( v <- round(v) )

### Créer une matrice 3 x 4 à partir des valeurs de
### v. Remarquer que la matrice est remplie par colonne.
```

```
( m <- matrix(v, nrow = 3, ncol = 4) )

### Les opérateurs arithmétiques de base s'appliquent aux
### matrices comme aux vecteurs.
m + 2
m * 3
m ^ 2

### Éliminer la quatrième colonne afin d'obtenir une matrice
### carrée.
( m <- m[,-4] )

### Transposée et inverse de la matrice m.
t(m)
solve(m)

### Produit matriciel.
m %*% m               # produit de m avec elle-même
m %*% solve(m)        # produit de m avec son inverse
round(m %*% solve(m)) # l'arrondi donne la matrice identité

### Consulter la rubrique d'aide de la fonction 'solve'.
?solve

### Liste des objets dans l'espace de travail.
ls()

### Nettoyage.
rm(x, y, v, m)
```

1.11 Exercices

1.1 Démarrer une session R et entrer une à une les expressions ci-dessous à la ligne de commande. Observer les résultats.

```
> ls()
> pi
> (v <- c(1, 5, 8))
> v * 2
> x <- v + c(2, 1, 7)
> x
> ls()
> q()
```

1.2 Ouvrir dans un éditeur de texte le fichier de script contenant le code de la section précédente. Exécuter le code ligne par ligne et observer les résultats. Repéter l'exercice avec un ou deux autres éditeurs de texte afin de les comparer et de vous permettre d'en choisir un pour la suite.

1.3 Consulter les rubriques d'aide d'une ou plusieurs des fonctions rencontrées lors de l'exercice précédent. Observer d'abord comment les rubriques d'aide sont structurées — elles sont toutes identiques — puis exécuter quelques expressions tirées des sections d'exemples.

1.4 Exécuter le code de l'exemple de session de travail R que l'on trouve à l'annexe A de Venables et collab. (2013). En plus d'aider à se familiariser avec R, cet exercice permet de découvrir les fonctionnalités du logiciel en tant qu'outil statistique.

2 Bases du langage R

Objectifs du chapitre

- Écrire et interpréter la syntaxe et la sémantique du langage R.
- Identifier les principaux types de données disponibles dans R.
- Utiliser les divers modes d'objets (en particulier `numeric`, `character` et `logical`) et la conversion automatique de l'un à l'autre.
- Créer et manipuler des vecteurs, matrices, tableaux, listes et *data frames*.
- Extraire des données d'un objet ou y affecter de nouvelles valeurs à l'aide des diverses méthodes d'indiçage.

Pour utiliser un langage de programmation, il faut en connaître la syntaxe et la sémantique, du moins dans leurs grandes lignes. C'est dans cet esprit que ce chapitre introduit des notions de base du langage R telles que l'expression, l'affectation et l'objet. Le concept de vecteur se trouvant au cœur du langage, le chapitre fait une large place à la création et à la manipulation des vecteurs et autres types d'objets de stockage couramment employés en programmation en R.

Énoncé du problème

Une ligue de hockey compte huit équipes. Le classement de la ligue est disponible quotidiennement dans le journal dans le format habituel ; voir le tableau 2.1.
Afin d'effectuer différentes analyses statistiques, on désire intégrer ces données dans un espace de travail R. On doit donc déterminer le type d'objet R approprié pour stocker le classement de la ligue.
Ensuite, on souhaite extraire les valeurs suivantes de l'objet précédemment créé.

Équipe	MJ	V	D	DP	PTS
Washington	55	36	16	3	75
Dallas	56	32	19	5	69
Chicago	57	30	21	6	66
Los Angeles	58	30	22	6	66
St-Louis	56	25	19	12	62
Détroit	57	25	21	11	61
Montréal	56	22	27	7	51
Boston	57	24	31	2	50

Légende : MJ – matchs joués ; V – victoires ; D – défaites ; DP : défaites en prolongation ; PTS – points

TAB. 2.1 – Classement de la ligue de hockey pour le problème à résoudre du chapitre

Énoncé du problème (suite)

a) Le nombre d'équipes de la ligue.

b) La fiche complète de l'équipe de Montréal.

c) La fiche complète de l'équipe à la septième position du classement.

2.1 Commandes R

Tel que vu au chapitre précédent, l'utilisateur de R interagit avec l'interprète R en entrant des commandes à l'invite de commande. Toute commande R est soit une *expression*, soit une *affectation*.

- 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
```

Dans les anciennes versions de S et R, l'on pouvait affecter avec le caractère de soulignement « _ ». Cet emploi n'est plus permis, mais la pratique subsiste dans le mode ESS de Emacs.
Ainsi, taper le caractère « _ » hors d'une chaîne de caractères dans Emacs génère automatiquement ␣<-␣. Si l'on souhaite véritablement obtenir le caractère de soulignement, il suffit d'appuyer deux fois successives sur « _ ».

- 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 <–, c'est-à-dire les deux caractères < et – placés obligatoirement l'un à la suite de l'autre :

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

- Pour affecter le résultat d'un calcul dans un objet et simultanément afficher ce résultat, il suffit de placer l'affectation entre parenthèses pour ainsi créer une nouvelle expression[1] :

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

- Le symbole d'affectation inversé –> existe aussi, mais il est rarement utilisé.
- Éviter d'utiliser l'opérateur `=` pour affecter une valeur à une variable puisque cette pratique est susceptible d'engendrer de la confusion avec les constructions `nom = valeur` dans les appels de fonction.

Que ce soit dans les fichiers de script ou à la ligne de commande, on sépare les commandes R les unes des autres par un point-virgule ou par un retour à la ligne.

1. En fait, cela devient un appel à l'opérateur "`(`" qui ne fait que retourner son argument.

- On considère généralement comme du mauvais style d'employer les deux, c'est-à-dire de placer des points-virgules à la fin de chaque ligne de code, surtout dans les fichiers de script.
- Le point-virgule peut être utile pour séparer deux courtes expressions ou plus sur une même ligne :

```
> a <- 5; a + 2
[1] 7
```

C'est le seul emploi du point-virgule que l'on rencontrera dans cet ouvrage.

On peut regrouper plusieurs commandes en une seule expression en les entourant d'accolades `{ }`.

- Le résultat du regroupement est la valeur de la dernière commande :

```
> {
+     a <- 2 + 3
+     b <- a
+     b
+ }
[1] 5
```

- Par conséquent, si le regroupement se termine par une assignation, aucune valeur n'est retournée ni affichée à l'écran :

```
> {
+     a <- 2 + 3
+     b <- a
+ }
```

- Les règles ci-dessus joueront un rôle important dans la composition de fonctions ; voir le chapitre 5.
- Comme on peut le voir ci-dessus, lorsqu'une commande n'est pas complète à la fin de la ligne, l'invite de commande de R change de >␣ à +␣ pour nous inciter à compléter notre commande.

2.2 Conventions pour les noms d'objets

Les caractères permis pour les noms d'objets sont les lettres minuscules a–z et majuscules A–Z, les chiffres 0–9, le point « . » et le caractère de soulignement « _ ». Selon l'environnement linguistique de l'ordinateur, il peut

être permis d'utiliser des lettres accentuées, mais cette pratique est fortement découragée puisqu'elle risque de nuire à la portabilité du code.

- Les noms d'objets ne peuvent commencer par un chiffre. S'ils commencent par un point, le second caractère ne peut être un chiffre.
- Le R est sensible à la casse, ce qui signifie que `foo`, `Foo` et `FOO` sont trois objets distincts. Un moyen simple d'éviter des erreurs liées à la casse consiste à n'employer que des lettres minuscules.
- Certains noms sont utilisés par le système R, 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 et il est interdit de les utiliser comme nom d'objet. Les mots réservés pour le système sont :

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

  Oui, '`...`' (*point-point-point*) est véritablement un nom d'objet dans R ! Son usage est expliqué à la section 6.1.

- Les variables `T` et `F` prennent par défaut les valeurs `TRUE` et `FALSE`, respectivement, mais peuvent être réaffectées :

```
> T
[1] TRUE
> F
[1] FALSE
```

```
> TRUE <- 3
Error in TRUE <- 3 : membre gauche de l'assignation
(do_set) incorrect
```

```
> (T <- 3)
[1] 3
```

- Nous recommandons de toujours écrire les valeurs booléennes `TRUE` et `FALSE` au long pour éviter des bogues difficiles à détecter.

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

TAB. 2.2 – Modes disponibles et contenus correspondants

2.3 Les objets R

Tout dans le langage R est un objet : les variables contenant des données, les fonctions, les opérateurs, même le symbole représentant le nom d'un objet est lui-même un objet. Les objets possèdent au minimum un *mode* et une *longueur* et certains peuvent être dotés d'un ou plusieurs *attributs*

- 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
```

2.3.1 Modes et types de données

Le mode prescrit ce qu'un objet peut contenir. À ce titre, un objet ne peut avoir qu'un seul mode. Le tableau 2.2 contient la liste des principaux modes disponibles en R. À chacun de ces modes correspond une fonction du même nom servant à créer un objet de ce mode.

- Les objets de mode `"numeric"`, `"complex"`, `"logical"` et `"character"` sont des objets *simples* (*atomic* en anglais) qui ne peuvent contenir que des données d'un seul type.
- En revanche, les objets de mode `"list"` ou `"expression"` sont des objets *récursifs* qui peuvent contenir d'autres objets. Par exemple, une liste peut

contenir une ou plusieurs autres listes ; voir la section 2.6 pour plus de détails.

- La fonction `typeof` permet d'obtenir une description plus précise de la représentation interne d'un objet (c'est-à-dire au niveau de la mise en œuvre en C). Le mode et le type d'un objet sont souvent identiques.

2.3.2 Longueur

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

- La longueur, au sens R du terme, 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 :

```
> v1 <- "actuariat"
> length(v1)
[1] 1
> v2 <- c("a", "c", "t", "u", "a", "r", "i", "a", "t")
> length(v2)
[1] 9
```

- Il faut utiliser la fonction `nchar` pour obtenir le nombre de caractères dans une chaîne :

```
> nchar(v1)
[1] 9
> nchar(v2)
[1] 1 1 1 1 1 1 1 1 1
```

- Un objet peut être de longueur 0 et doit alors être interprété comme un contenant qui existe, mais qui est vide :

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

2.3.3 Objet spécial `NULL`

L'objet spécial `NULL` représente « rien », ou le vide.

- Son mode est `NULL`.

- Sa longueur est 0.
- Toutefois 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.

2.3.4 Valeurs manquantes, indéterminées et infinies

Dans les applications statistiques, il est souvent utile de pouvoir représenter des données manquantes. Dans R, l'objet spécial `NA` remplit ce rôle.

- Par défaut, le mode de `NA` est `logical`, mais `NA` ne peut être considéré ni comme `TRUE`, ni comme `FALSE`.
- 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 valeur `NA` n'est égale à aucune autre, pas même elle-même (selon la règle ci-dessus, le résultat de la comparaison est `NA`) :

```
> NA == NA
[1] NA
```

- Par conséquent, pour tester si les éléments d'un objet sont `NA` ou non il faut utiliser la fonction `is.na` :

```
> is.na(NA)
[1] TRUE
```

La norme IEEE 754 régissant la représentation interne des nombres dans un ordinateur (IEEE, 2003) prévoit les valeurs mathématiques spéciales $+\infty$ et $-\infty$ ainsi que les formes indéterminées du type $\frac{0}{0}$ ou $\infty - \infty$. R dispose d'objets spéciaux pour représenter ces valeurs.

- `Inf` représente $+\infty$.
- `-Inf` représente $-\infty$.
- `NaN` (*Not a Number*) représente une forme indéterminée.
- Ces valeurs sont testées avec les fonctions `is.infinite`, `is.finite` et `is.nan`.

Attribut	Utilisation
`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

TAB. 2.3 – Attributs les plus usuels d'un objet

2.3.5 Attributs

Les attributs d'un objet sont des éléments d'information additionnels liés à cet objet. La liste des attributs les plus fréquemment rencontrés se trouve au tableau 2.3. Pour chaque attribut, il existe une fonction du même nom servant à extraire l'attribut correspondant d'un objet.

- Plus généralement, la fonction `attributes` permet d'extraire ou de modifier la liste des attributs d'un objet. On peut aussi travailler sur un seul attribut à la fois avec la fonction `attr`.
- On peut ajouter à peu près ce que l'on veut à la liste des attributs d'un objet. Par exemple, on pourrait vouloir attacher au résultat d'un calcul la méthode de calcul utilisée :

```
> x <- 3
> attr(x, "methode") <- "au pif"
> attributes(x)
$methode
[1] "au pif"
```

- Extraire un attribut qui n'existe pas retourne `NULL` :

```
> dim(x)
NULL
```

- À l'inverse, donner à un attribut la valeur `NULL` efface cet attribut :

```
> attr(x, "methode") <- NULL
> attributes(x)
NULL
```

2.4 Vecteurs

En R, à toutes fins pratiques, *tout* est un vecteur. Contrairement à certains autres langages de programmation, il n'y a pas de notion de scalaire en R ; un scalaire est simplement un vecteur de longueur 1. Comme nous le verrons au chapitre 3, le vecteur est l'unité de base dans les calculs.

- Dans un vecteur simple, tous les éléments doivent être du même mode. Nous nous restreignons à ce type de vecteurs pour le moment.
- 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`).
- 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
```

  Ces étiquettes font alors partie des attributs du vecteur.
- L'indiçage dans un vecteur se fait avec les crochets `[ ]`. On peut extraire un élément d'un vecteur par sa position ou par son étiquette, si elle existe (auquel cas cette approche est beaucoup plus sûre).

```
> v[3]
c
5
> v["c"]
c
5
```

  La section 2.8 traite plus en détail de l'indiçage des vecteurs et des matrices.

Astuce

Dans un vecteur simple, tous les éléments doivent être du même mode. Or, les informations du classement de la ligue comportent à la fois des chaînes de caractères et des nombres. De plus, le classement se présente sous forme d'un tableau à deux dimensions, alors qu'un vecteur n'en compte qu'une seule. Le vecteur simple n'est donc pas le type d'objet approprié pour stocker le classement de la ligue.

2.5 Matrices et tableaux

Le R étant un langage spécialisé pour les calculs mathématiques, il supporte tout naturellement et de manière intuitive — à une exception près, comme nous le verrons — les matrices et, plus généralement, les tableaux à plusieurs dimensions.

Les matrices et tableaux ne sont rien d'autre que des vecteurs dotés d'un attribut `dim`. Ces objets sont donc stockés, et peuvent être manipulés, exactement comme des vecteurs simples.

- Une matrice est un vecteur avec un attribut `dim` de longueur 2. Cela change implicitement la classe de l'objet pour `"matrix"` et, de ce fait, le mode d'affichage de l'objet ainsi que son interaction avec plusieurs opérateurs et fonctions.
- La fonction de base pour créer des matrices est `matrix` :

```
> 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
```

- La généralisation d'une matrice à plus de deux dimensions est un tableau (*array*). Le nombre de dimensions du tableau est toujours égal à la longueur de l'attribut `dim`. La classe implicite d'un tableau est `"array"`.
- La fonction de base pour créer des tableaux est `array` :

```
> array(1:24, dim = c(3, 4, 2))
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

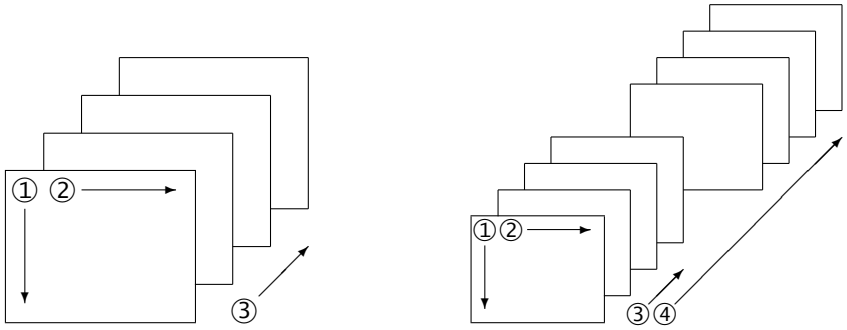
On remarquera ci-dessus que les matrices et tableaux sont remplis en faisant d'abord varier la première dimension, puis la seconde, etc. Pour les matrices, cela revient à remplir par colonne. On conviendra que cette convention, héritée du Fortran, n'est pas des plus intuitives.
La fonction `matrix` a un argument `byrow` qui permet d'inverser l'ordre de remplissage. Cependant, il vaut mieux s'habituer à la convention de R que d'essayer constamment de la contourner.

L'ordre de remplissage inhabituel des tableaux rend leur manipulation difficile si on ne les visualise pas correctement. Imaginons un tableau de dimensions $3 \times 4 \times 5$.

- Il faut voir le tableau comme cinq matrices 3×4 (remplies par colonne !) les unes *derrière* les autres.
- Autrement dit, le tableau est un prisme rectangulaire haut de 3 unités, large de 4 et profond de 5.
- Si l'on ajoute une quatrième dimension, cela revient à aligner des prismes les uns derrière les autres, et ainsi de suite.

La figure 2.1 fournit une représentation schématique des tableaux à trois et quatre dimensions.

Comme pour les vecteurs, l'indiçage des matrices et tableaux se fait avec les crochets `[ ]`.



(a) trois dimensions (b) quatre dimensions

FIG. 2.1 – Représentation schématique de tableaux. Les chiffres encerclés identifient l'ordre de remplissage.

- On extrait un élément d'une matrice en précisant sa position dans chaque dimension de celle-ci, séparées par des virgules :

```
> (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
```

- On peut aussi ne donner que la position de l'élément dans le vecteur sous-jacent :

```
> m[3]
[1] 45
```

- Lorsqu'une dimension est omise dans les crochets, tous les éléments de cette dimension sont extraits :

```
> m[2, ]
[1] 80 21 32
```

- Les idées sont les mêmes pour les tableaux.
- Pour le reste, les règles d'indiçage de vecteurs exposées à la section 2.8 s'appliquent à chaque position de l'indice d'une matrice ou d'un tableau.

Des fonctions permettent de fusionner des matrices et des tableaux ayant au moins une dimension identique.

- 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
```

- 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
```

Astuce

Une matrice convient bien pour stocker un tableau de données. Toutefois, puisque la matrice est en fait un vecteur avec un attribut `dim` de longueur 2, tous les éléments doivent être du même mode, comme c'était le cas avec les vecteurs simples. Impossible dans ce cas d'y stocker le nom des équipes. La matrice n'est toujours pas le type d'objet approprié.

2.6 Listes

La liste est le mode de stockage le plus général et polyvalent du langage R. Il s'agit d'un type de vecteur spécial dont les éléments peuvent être de n'importe quel mode, y compris le mode `list`. Cela permet donc d'emboîter des listes, d'où le qualificatif de *récursif* pour ce type d'objet.

- La fonction de base pour créer des listes est `list` :

```
> (x <- list(size = c(1, 5, 2), user = "Joe", new = TRUE))
$size
[1] 1 5 2

$user
[1] "Joe"

$new
[1] TRUE
```

Ci-dessus, le premier élément de la liste est de mode `"numeric"`, le second de mode `"character"` et le troisième de mode `"logical"`.

- Nous recommandons de nommer les éléments d'une liste. En effet, les listes contiennent souvent des données de types différents et il peut s'avérer difficile d'identifier les éléments s'ils ne sont pas nommés. De plus, comme nous le verrons ci-dessous, il est très simple d'extraire les éléments d'une liste par leur étiquette.
- La liste demeure un vecteur. On peut donc l'indicer avec l'opérateur `[ ]`. Cependant, cela retourne une liste contenant le ou les éléments indicés. C'est rarement ce que l'on souhaite.
- Pour indicer un élément d'une liste et n'obtenir que cet élément, et non une liste contenant l'élément, il faut utiliser l'opérateur d'indiçage `[[ ]]`. Comparer

```
> x[1]
$size
[1] 1 5 2
```

et

```
> x[[1]]
[1] 1 5 2
```

Évidemment, on ne peut extraire qu'un seul élément à la fois avec les crochets doubles `[[ ]]`.

- Petite subtilité peu employée, mais élégante. Si l'indice utilisé dans `[[ ]]` est un vecteur, il est utilisé récursivement pour indicer la liste : cela sélectionnera la composante de la liste correspondant au premier élément du vecteur, puis l'élément de la composante correspondant au second élément du vecteur, et ainsi de suite.

- Une autre — la meilleure, en fait — façon d'indicer un seul élément d'une liste est par son étiquette avec l'opérateur `$` :

```
> x$size
[1] 1 5 2
```

- La fonction `unlist` convertit une liste en un vecteur simple. Elle est surtout utile pour concaténer les éléments d'une liste lorsque ceux-ci sont des scalaires. Attention, cette fonction peut être destructrice si la structure interne de la liste est importante.

2.7 *Data frames*

Les vecteurs, les matrices, les tableaux et les listes sont les types d'objets les plus fréquemment utilisés en programmation en R. Toutefois, un grand nombre de procédures statistiques — pensons à la régression linéaire, par exemple — repose davantage sur les *data frames* pour le stockage des données.

- Un *data frame* est une liste de classe `"data.frame"` dont tous les éléments sont de la même longueur (ou comptent le même nombre de lignes si les éléments sont des matrices).
- Il est généralement représenté sous la forme d'un tableau à deux dimensions. Chaque élément de la liste sous-jacente correspond à une colonne.
- Bien que visuellement similaire à une matrice un *data frame* est plus général puisque les colonnes peuvent être de modes différents ; pensons à un tableau avec des noms (mode `character`) dans une colonne et des notes (mode `numeric`) dans une autre.
- On crée un *data frame* avec la fonction `data.frame` ou, pour convertir un autre type d'objet en *data frame*, avec `as.data.frame`.
- Le *data frame* peut être indicé à la fois comme une liste et comme une matrice.
- Les fonctions `rbind` et `cbind` peuvent être utilisées pour ajouter des lignes ou des colonnes à un *data frame*.
- On peut rendre les colonnes d'un *data frame* (ou d'une liste) visibles dans l'espace de travail avec la fonction `attach`, puis les masquer avec `detach`.

Astuce

La liste permettrait de stocker à la fois le nom des équipes et leurs statistiques puisqu'elle peut contenir des objets de mode différent.
On crée d'abord des vecteurs simples contenant les données de chaque colonne du classement des équipes.

```
> Equipe <- c("Washington", "Dallas", "Chicago",
+             "Los Angeles", "St-Louis", "Détroit",
+             "Montréal", "Boston")
> MJ  <- c(55, 56, 57, 58, 56, 57, 56, 57)
> V   <- c(36, 32, 30, 30, 25, 25, 22, 24)
> D   <- c(16, 19, 21, 22, 19, 21, 27, 31)
> DP  <- c( 3,  5,  6,  6, 12, 11,  7,  2)
> PTS <- c(75, 69, 66, 66, 62, 61, 51, 50)
```

On les combine ensuite sous forme de liste nommée.

```
> list(Equipe = Equipe, MJ = MJ, V = V, D = D,
+      DP = DP, PTS = PTS)
$Equipe
[1] "Washington"  "Dallas"      "Chicago"
[4] "Los Angeles" "St-Louis"    "Détroit"
[7] "Montréal"    "Boston"

$MJ
[1] 55 56 57 58 56 57 56 57

$V
[1] 36 32 30 30 25 25 22 24

$D
[1] 16 19 21 22 19 21 27 31

$DP
[1]  3  5  6  6 12 11  7  2

$PTS
[1] 75 69 66 66 62 61 51 50
```

On constate que R ne présente pas le contenu de la liste sous forme d'un tableau. Ce n'est donc pas le type d'objet le mieux approprié pour stocker un classement. En fait, la liste est un mode de stockage *trop* général pour le type de données dont nous disposons.

Astuce

L'élément distinctif entre un *data frame* et une liste générale, c'est que tous les éléments du premier doivent être de la même longueur et que, par conséquent, R les dispose en colonnes. Nous avons donc ici le type d'objet tout désigné pour stocker des données de modes différents, mais qui se présentent sous forme de tableau à deux dimensions.

2.8 Indiçage

L'indiçage des vecteurs et matrices a déjà été brièvement présenté aux sections 2.4 et 2.5. La présente section contient plus de détails sur cette procédure des plus communes lors de l'utilisation du langage R. On se concentre toutefois sur le traitement des vecteurs.

- L'indiçage sert principalement à deux choses : soit extraire des éléments d'un objet avec la construction `x[i]`, ou les remplacer avec la construction `x[i] <- y`.
- Il est utile de savoir que ces opérations sont en fait traduites par l'interprète R en des appels à des fonctions nommées `[` et `[<-`, dans l'ordre.
- De même, les opérations d'extraction et de remplacement d'un élément d'une liste de la forme `x$etiquette` et `x$etiquette <- y` correspondent à des appels aux fonctions `$` et `$<-`.

Il existe cinq façons d'indicer un vecteur dans le langage R. Dans tous les cas, l'indiçage se fait à l'intérieur de 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 :

```
> x <- c(A = 2, B = 4, C = -1, D = -5, E = 8)
> x[c(1, 3)]
 A  C
 2 -1
```

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

```
> x[c(-2, -3)]
```

```
 A  D  E
 2 -5  8
```

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 :

```
> x > 0
    A     B     C     D     E
 TRUE  TRUE FALSE FALSE  TRUE
> x[x > 0]
A B E
2 4 8
```

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

```
> x[c("B", "D")]
 B  D
 4 -5
```

5. L'indice est laissé vide. Tous les éléments du vecteur sont alors sélectionnés :

```
> x[]
 A  B  C  D  E
 2  4 -1 -5  8
```

   Cette méthode est essentiellement utilisée avec les matrices et tableaux pour sélectionner tous les éléments d'une dimension (voir l'exemple à la page 27). Laisser l'indice vide est différent d'indicer avec un vecteur vide ; cette dernière opération retourne un vecteur vide.

Solution du problème

Nous avons déjà créé à l'Astuce de la page 31 des vecteurs contenant les données des différentes colonnes du classement de la ligue.

Solution du problème (suite)

```
> Equipe
[1] "Washington"  "Dallas"      "Chicago"
[4] "Los Angeles" "St-Louis"    "Détroit"
[7] "Montréal"    "Boston"
> MJ
[1] 55 56 57 58 56 57 56 57
> V
[1] 36 32 30 30 25 25 22 24
> D
[1] 16 19 21 22 19 21 27 31
> DP
[1]  3  5  6  6 12 11  7  2
> PTS
[1] 75 69 66 66 62 61 51 50
```

On crée l'objet `classement` qui contiendra le classement de la ligue avec la fonction `data.frame`. Celle-ci prend en arguments les différents vecteurs de données.

```
> (classement <- data.frame(Equipe, MJ, V, D,  DP, PTS))
       Equipe MJ  V  D DP PTS
1  Washington 55 36 16  3  75
2      Dallas 56 32 19  5  69
3     Chicago 57 30 21  6  66
4 Los Angeles 58 30 22  6  66
5    St-Louis 56 25 19 12  62
6     Détroit 57 25 21 11  61
7    Montréal 56 22 27  7  51
8      Boston 57 24 31  2  50
```

On répond ensuite aux questions de traitement des données.

a) Le nombre d'équipes dans la ligue correspond au nombre de lignes de l'objet :

Solution du problème (suite)

```
> dim(classement)[1]
[1] 8
```

ou, comme nous le verrons dans le code informatique de la section suivante,

```
> nrow(classement)
[1] 8
```

b) Pour extraire la fiche complète de l'équipe de Montréal, il suffit d'extraire cette ligne du tableau de données :

```
> ## indiçage par vecteur booléen
> classement[classement$Equipe == "Montréal", ]
    Equipe MJ  V  D DP PTS
7 Montréal 56 22 27  7  51
> ## extraction par une fonction pratique et élégante
> subset(classement, Equipe == "Montréal")
    Equipe MJ  V  D DP PTS
7 Montréal 56 22 27  7  51
```

c) Pour extraire la fiche complète de l'équipe à la septième position du classement, on peut indicer le tableau par position :

```
> classement[7, ]
    Equipe MJ  V  D DP PTS
7 Montréal 56 22 27  7  51
```

2.9 Exemples

```
###
### COMMANDES R
###
```

```
## Les expressions entrées à la ligne de commande sont
## immédiatement évaluées et le résultat est affiché à
## l'écran, comme avec une grosse calculatrice.
1                           # une constante
(2 + 3 * 5)/7               # priorité des opérations
3^5                         # puissance
exp(3)                      # fonction exponentielle
sin(pi/2) + cos(pi/2)       # fonctions trigonométriques
gamma(5)                    # fonction gamma

## Lorsqu'une expression est syntaxiquement incomplète,
## l'invite de commande change de '> ' à '+ '.
2 -                         # expression incomplète
5 *                         # toujours incomplète
3                           # complétée

## Taper le nom d'un objet affiche son contenu. Pour une
## fonction, c'est son code source qui est affiché.
pi                          # constante numérique intégrée
letters                     # chaîne de caractères intégrée
LETTERS                     # version en majuscules
matrix                      # fonction

## Ne pas utiliser '=' pour l'affectation. Les opérateurs
## d'affectation standard en R sont '<-' et '->'.
x <- 5                      # affecter 5 à l'objet 'x'
5 -> x                      # idem, mais peu usité
x                           # voir le contenu
(x <- 5)                    # affecter et afficher
y <- x                      # affecter la valeur de 'x' à 'y'
x <- y <- 5                 # idem, en une seule expression
y                           # 5
x <- 0                      # changer la valeur de 'x'...
y                           # ... ne change pas celle de 'y'

## Pour regrouper plusieurs expressions en une seule commande,
## il faut soit les séparer par un point-virgule ';', soit les
## regrouper à l'intérieur d'accolades { } et les séparer par
## des retours à la ligne.
x <- 5; y <- 2; x + y       # compact; éviter dans les scripts
x <- 5;                     # éviter les ';' superflus
{                           # début d'un groupe
    x <- 5                  # première expression du groupe
    y <- 2                  # seconde expression du groupe
```

```
    x + y                   # résultat du groupe
}                           # fin du groupe et résultat
{x <- 5; y <- 2; x + y}     # valide, mais redondant

###
### NOMS D'OBJETS
###

## Quelques exemples de noms valides et invalides.
foo <- 5                    # valide
foo.123 <- 5                # valide
foo_123 <- 5                # valide
123foo <- 5                 # invalide; commence par un chiffre
.foo <- 5                   # valide
.123foo <- 5                # invalide; point suivi d'un chiffre

## Liste des objets dans l'espace de travail. Les objets dont
## le nom commence par un point sont considérés cachés.
ls()                        # l'objet '.foo' n'est pas affiché
ls(all.names = TRUE)        # objets cachés aussi affichés

## R est sensible à la casse
foo <- 1
Foo
FOO

###
### LES OBJETS R
###

## MODES ET TYPES DE DONNÉES

## Le mode d'un objet détermine ce qu'il peut contenir. Les
## vecteurs simples ("atomic") contiennent des données d'un
## seul type.
mode(c(1, 4.1, pi))         # nombres réels
mode(c(2, 1 + 5i))          # nombres complexes
mode(c(TRUE, FALSE, TRUE)) # valeurs booléennes
mode("foobar")              # chaînes de caractères

## Si l'on mélange dans un même vecteur des objets de mode
## différents, il y a conversion automatique vers le mode pour
## lequel il y a le moins de perte d'information, c'est-à-dire
## vers le mode qui permet le mieux de retrouver la valeur
## originale des éléments.
```

```
c(5, TRUE, FALSE)          # conversion en mode 'numeric'
c(5, "z")                  # conversion en mode 'character'
c(TRUE, "z")               # conversion en mode 'character'
c(5, TRUE, "z")            # conversion en mode 'character'

## La plupart des autres types d'objets sont récursifs. Voici
## quelques autres modes.
mode(seq)                  # une fonction
mode(list(5, "foo", TRUE)) # une liste
mode(expression(x <- 5))   # une expression non évaluée

## LONGUEUR

## La longueur d'un vecteur est égale au nombre d'éléments
## dans le vecteur.
(x <- 1:4)
length(x)

## Une chaîne de caractères ne compte que pour un seul
## élément.
(x <- "foobar")
length(x)

## Pour obtenir la longueur de la chaîne, il faut utiliser
## nchar().
nchar(x)

## Un objet peut néanmoins contenir plusieurs chaînes de
## caractères.
(x <- c("f", "o", "o", "b", "a", "r"))
length(x)

## La longueur peut être 0, auquel cas on a un objet vide,
## mais qui existe.
(x <- numeric(0))          # création du contenant
length(x)                  # l'objet 'x' existe...
x[1] <- 1                  # possible, 'x' existe
X[1] <- 1                  # impossible, 'X' n'existe pas

## L'OBJET SPECIAL 'NULL'
mode(NULL)                 # le mode de 'NULL' est NULL
length(NULL)               # longueur nulle
x <- c(NULL, NULL)         # s'utilise comme un objet normal
x; length(x); mode(x)      # mais donne toujours le vide
```

```
## L'OBJET SPÉCIAL 'NA'
x <- c(65, NA, 72, 88)      # traité comme une valeur
x + 2                       # tout calcul avec 'NA' donne NA
mean(x)                     # voilà qui est pire
mean(x, na.rm = TRUE)       # éliminer les 'NA' avant le calcul
is.na(x)                    # tester si les données sont 'NA'

## VALEURS INFINIES ET INDÉTERMINÉES
1/0                         # +infini
-1/0                        # -infini
0/0                         # indétermination
x <- c(65, Inf, NaN, 88)    # s'utilisent comme des valeurs
is.finite(x)                # quels sont les nombres réels?
is.nan(x)                   # lesquels ne sont «pas un nombre»?

## ATTRIBUTS

## Les objets peuvent être dotés d'un ou plusieurs attributs.
data(cars)                  # jeu de données intégré
attributes(cars)            # liste de tous les attributs
attr(cars, "class")         # extraction d'un seul attribut

## Attribut 'class'. Selon la classe d'un objet, certaines
## fonctions (dites «fonctions génériques») vont se comporter
## différemment.
x <- sample(1:100, 10)      # échantillon aléatoire de 10
                            # nombres entre 1 et 100
class(x)                    # classe de l'objet
plot(x)                     # graphique pour cette classe
class(x) <- "ts"            # 'x' est maintenant une série
                            # chronologique
plot(x)                     # graphique pour les séries
                            # chronologiques
class(x) <- NULL; x         # suppression de l'attribut 'class'

## Attribut 'dim'. Si l'attribut 'dim' compte deux valeurs,
## l'objet est traité comme une matrice. S'il en compte plus
## de deux, l'objet est traité comme un tableau (array).
x <- 1:24                   # un vecteur
dim(x) <- c(4, 6)           # ajoute un attribut 'dim'
x                           # l'objet est une matrice
dim(x) <- c(4, 2, 3)        # change les dimensions
x                           # l'objet est maintenant un tableau

## Attribut 'dimnames'. Permet d'assigner des étiquettes (ou
```

```
## noms) aux dimensions d'une matrice ou d'un tableau.
dimnames(x) <- list(1:4, c("a", "b"), c("A", "B", "C"))
dimnames(x)                # remarquer la conversion
x                          # affichage avec étiquettes
attributes(x)              # tous les attributs de 'x'
attributes(x) <- NULL; x   # supprimer les attributs

## Attributs 'names'. Similaire à 'dimnames', mais pour les
## éléments d'un vecteur ou d'une liste.
names(x) <- letters[1:24]  # attribution d'étiquettes
x                          # identification facilitée

###
### VECTEURS
###

## La fonction de base pour créer des vecteurs est 'c'. Il
## peut s'avérer utile de donner des étiquettes aux éléments
## d'un vecteur.
x <- c(a = -1, b = 2, c = 8, d = 10) # création d'un vecteur
names(x)                             # extraire les étiquettes
names(x) <- letters[1:length(x)]     # changer les étiquettes
x[1]                       # extraction par position
x["c"]                     # extraction par étiquette
x[-2]                      # élimination d'un élément

## La fonction 'vector' sert à initialiser des vecteurs avec
## des valeurs prédéterminées. Elle compte deux arguments: le
## mode du vecteur et sa longueur. Les fonctions 'numeric',
## 'logical', 'complex' et 'character' constituent des
## raccourcis pour des appels à 'vector'.
vector("numeric", 5)       # vecteur initialisé avec des 0
numeric(5)                 # équivalent
numeric                    # en effet, voici la fonction
logical(5)                 # initialisé avec FALSE
complex(5)                 # initialisé avec 0 + 0i
character(5)               # initialisé avec chaînes vides

###
### MATRICES ET TABLEAUX
###

## Une matrice est un vecteur avec un attribut 'dim' de
## longueur 2 une classe implicite "matrix". La manière
## naturelle de créer une matrice est avec la fonction
```

```
## 'matrix'.
(x <- matrix(1:12, nrow = 3, ncol = 4)) # créer la matrice
length(x)                  # 'x' est un vecteur...
dim(x)                     # ... avec un attribut 'dim'...
class(x)                   # ... et classe implicite "matrix"

## Une manière moins naturelle mais équivalente --- et parfois
## plus pratique --- de créer une matrice consiste à ajouter
## un attribut 'dim' à un vecteur.
x <- 1:12                  # vecteur simple
dim(x) <- c(3, 4)          # ajout d'un attribut 'dim'
x; class(x)                # 'x' est une matrice!

## Les matrices sont remplies par colonne par défaut. Utiliser
## l'option 'byrow' pour remplir par ligne.
matrix(1:12, nrow = 3, byrow = TRUE)

## Indicer la matrice ou le vecteur sous-jacent est
## équivalent. Utiliser l'approche la plus simple selon le
## contexte.
x[1, 3]                    # l'élément en position (1, 3)...
x[7]                       # ... est le 7e élément du vecteur
x[1, ]                     # première ligne
x[, 2]                     # deuxième colonne
nrow(x)                    # nombre de lignes
dim(x)[1]                  # idem
ncol(x)                    # nombre de colonnes
dim(x)[2]                  # idem

## Fusion de matrices et vecteurs.
x <- matrix(1:12, 3, 4)    # 'x' est une matrice 3 x 4
y <- matrix(1:8, 2, 4)     # 'y' est une matrice 2 x 4
z <- matrix(1:6, 3, 2)     # 'z' est une matrice 3 x 2
rbind(x, 1:4)              # ajout d'une ligne à 'x'
rbind(x, y)                # fusion verticale de 'x' et 'y'
cbind(x, 1:3)              # ajout d'une colonne à 'x'
cbind(x, z)                # concaténation de 'x' et 'z'
rbind(x, z)                # dimensions incompatibles
cbind(x, y)                # dimensions incompatibles

## Les vecteurs ligne et colonne sont rarement nécessaires. On
## peut les créer avec les fonctions 'rbind' et 'cbind',
## respectivement.
rbind(1:3)                 # un vecteur ligne
cbind(1:3)                 # un vecteur colonne
```

```
## Un tableau (array) est un vecteur avec un attribut 'dim' de
## longueur supérieure à 2 et une classe implicite "array".
## Quant au reste, la manipulation des tableaux est en tous
## points identique à celle des matrices. Ne pas oublier:
## les tableaux sont remplis de la première dimension à la
## dernière!
x <- array(1:60, 3:5)      # tableau 3 x 4 x 5
length(x)                  # 'x' est un vecteur...
dim(x)                     # ... avec un attribut 'dim'...
class(x)                   # ... une classe implicite "array"
x[1, 3, 2]                 # l'élément en position (1, 3, 2)...
x[19]                      # ... est l'élément 19 du vecteur

## Le tableau ci-dessus est un prisme rectangulaire 3 unités
## de haut, 4 de large et 5 de profond. Indicer ce prisme avec
## un seul indice équivaut à en extraire des «tranches», alors
## qu'utiliser deux indices équivaut à en tirer des «carottes»
## (au sens géologique du terme). Il est laissé en exercice de
## généraliser à plus de dimensions...
x                          # les cinq matrices
x[, , 1]                   # tranches de haut en bas
x[, 1, ]                   # tranches d'avant à l'arrière
x[1, , ]                   # tranches de gauche à droite
x[, 1, 1]                  # carotte de haut en bas
x[1, 1, ]                  # carotte d'avant à l'arrière
x[1, , 1]                  # carotte de gauche à droite

###
### LISTES
###

## La liste est l'objet le plus général en R. C'est un objet
## récursif qui peut contenir des objets de n'importe quel
## mode et longueur.
(x <- list(joueur = c("V", "C", "C", "M", "A"),
           score = c(10, 12, 11, 8, 15),
           expert = c(FALSE, TRUE, FALSE, TRUE, TRUE),
           niveau = 2))
is.vector(x)               # vecteur...
length(x)                  # ... de quatre éléments...
mode(x)                    # ... de mode "list"
is.recursive(x)            # objet récursif

## Comme tout autre vecteur, une liste peut être concaténée
```

```
## avec un autre vecteur avec la fonction 'c'.
y <- list(TRUE, 1:5)       # liste de deux éléments
c(x, y)                    # liste de six éléments

## Pour initialiser une liste d'une longueur déterminée, mais
## dont chaque élément est vide, uitliser la fonction
## 'vector'.
vector("list", 5)          # liste de NULL

## Pour extraire un élément d'une liste, il faut utiliser les
## doubles crochets [[ ]]. Les simples crochets [ ]
## fonctionnent aussi, mais retournent une sous liste -- ce
## qui est rarement ce que l'on souhaite.
x[[1]]                     # premier élément de la liste...
mode(x[[1]])               # ... un vecteur
x[1]                       # aussi le premier élément...
mode(x[1])                 # ... mais une sous liste...
length(x[1])               # ... d'un seul élément
x[[2]][1]                  # 1er élément du 2e élément
x[[c(2, 1)]]               # idem, par indiçage récursif

## Les éléments d'une liste étant généralement nommés (c'est
## une bonne habitude à prendre!), il est souvent plus simple
## et sûr d'extraire les éléments d'une liste par leur
## étiquette.
x$joueur                   # équivalent à a[[1]]
x$score[1]                 # équivalent à a[[c(2, 1)]]
x[["expert"]]              # aussi valide, mais peu usité
x$level <- 1               # aussi pour l'affectation

## Une liste peut contenir n'importe quoi...
x[[5]] <- matrix(1, 2, 2)  # ... une matrice...
x[[6]] <- list(20:25, TRUE)# ... une autre liste...
x[[7]] <- seq              # ... même le code d'une fonction!
x                          # eh ben!
x[[c(6, 1, 3)]]            # de quel élément s'agit-il?

## Pour supprimer un élément d'une liste, lui assigner la
## valeur 'NULL'.
x[[7]] <- NULL; length(x)  # suppression du 7e élément

## Il est parfois utile de convertir une liste en un simple
## vecteur. Les éléments de la liste sont alors «déroulés», y
## compris la matrice en position 5 (qui n'est rien d'autre
## qu'un vecteur, on s'en souviendra).
```

```
unlist(x)                    # remarquer la conversion
unlist(x, recursive = FALSE) # ne pas appliquer aux sous-listes
unlist(x, use.names = FALSE) # éliminer les étiquettes

###
### DATA FRAMES
###

## Un data frame est une liste dont les éléments sont tous de
## même longueur. Il comporte un attribut 'dim', ce qui fait
## qu'il est représenté comme une matrice. Cependant, les
## colonnes peuvent être de modes différents.
(DF <- data.frame(Noms = c("Pierre", "Jean", "Jacques"),
                  Age = c(42, 34, 19),
                  Fumeur = c(TRUE, TRUE, FALSE)))
mode(DF)                     # un data frame est une liste...
class(DF)                    # ... de classe 'data.frame'
dim(DF)                      # dimensions implicites
names(DF)                    # titres des colonnes
row.names(DF)                # titres des lignes (implicites)
DF[1, ]                      # première ligne
DF[, 1]                      # première colonne
DF$Noms                      # idem, mais plus simple

## Lorsque l'on doit travailler longtemps avec les différentes
## colonnes d'un data frame, il est pratique de pouvoir y
## accéder directement sans devoir toujours indicer. La
## fonction 'attach' permet de rendre les colonnes
## individuelles visibles dans l'espace de travail. Une fois
## le travail terminé, 'detach' masque les colonnes.
exists("Noms")               # variable n'existe pas
attach(DF)                   # rendre les colonnes visibles
exists("Noms")               # variable existe
Noms                         # colonne accessible
detach(DF)                   # masquer les colonnes
exists("Noms")               # variable n'existe plus

###
### INDIÇAGE
###

## Les opérations suivantes illustrent les différentes
## techniques d'indiçage d'un vecteur pour l'extraction et
## l'affectation, c'est-à-dire que l'on utilise à la fois la
## fonction '[' et la fonction '[<-'. Les mêmes techniques
```

```
## existent aussi pour les matrices, tableaux et listes.
##
## On crée d'abord un vecteur quelconque formé de vingt
## nombres aléatoires entre 1 et 100 avec répétitions
## possibles.
(x <- sample(1:100, 20, replace = TRUE))

## On ajoute des étiquettes aux éléments du vecteur à partir
## de la variable interne 'letters'.
names(x) <- letters[1:20]

## On génère ensuite cinq nombres aléatoires entre 1 et 20
## (sans répétitions).
(y <- sample(1:20, 5))

## On remplace maintenant les éléments de 'x' correspondant
## aux positions dans le vecteur 'y' par des données
## manquantes.
x[y] <- NA
x

## Les cinq méthodes d'indiçage de base.
x[1:10]                    # avec des entiers positifs
"["(x, 1:10)               # idem, avec la fonction '['
x[-(1:3)]                  # avec des entiers négatifs
x[x < 10]                  # avec un vecteur booléen
x[c("a", "k", "t")]        # par étiquettes
x[]                        # aucun indice...
x[numeric(0)]              # ... différent d'indice vide

## Il arrive souvent de vouloir indicer spécifiquement les
## données manquantes d'un vecteur (pour les éliminer ou les
## remplacer par une autre valeur, par exemple). Pour ce
## faire, on utilise la fonction 'is.na' et l'indiçage par un
## vecteur booléen. (Note: l'opérateur '!' ci-dessous est la
## négation logique.)
is.na(x)                   # positions des données manquantes
x[!is.na(x)]               # suppression des données manquantes
x[is.na(x)] <- 0; x        # remplace les NA par des 0
"[<-"(x, is.na(x), 0)      # idem, mais très peu usité

## On laisse tomber les étiquettes de l'objet.
names(x) <- NULL

## Quelques cas spéciaux d'indiçage.
```

```
length(x)                  # un rappel
x[1:25]                    # allonge le vecteur avec des NA
x[25] <- 10; x             # remplit les trous avec des NA
x[0]                       # n'extraie rien
x[0] <- 1; x               # n'affecte rien
x[c(0, 1, 2)]              # le 0 est ignoré
x[c(1, NA, 5)]             # indices NA retourne NA
x[2.6]                     # fractions tronquées vers 0

## On laisse tomber les 5 derniers éléments et on convertit le
## vecteur en une matrice 4 x 5.
x <- x[1:20]               # ou x[-(21:25)]
dim(x) <- c(4, 5); x       # ajouter un attribut 'dim'

## Dans l'indiçage des matrices et tableaux, l'indice de
## chaque dimension obéit aux mêmes règles que ci-dessus. On
## peut aussi indicer une matrice (ou un tableau) avec une
## matrice. Si les exemples ci-dessous ne permettent pas d'en
## comprendre le fonctionnement, consulter la rubrique d'aide
## de la fonction '[' (ou de 'Extract').
x[1, 2]                    # élément en position (1, 2)
x[1, -2]                   # 1ère rangée sans 2e colonne
x[c(1, 3), ]               # 1ère et 3e rangées
x[-1, ]                    # supprimer 1ère rangée
x[, -2]                    # supprimer 2e colonne
x[x[, 1] > 10, ]           # lignes avec 1er élément > 10
x[rbind(c(1, 1), c(2, 2))] # éléments x[1, 1] et x[2, 2]
x[cbind(1:4, 1:4)]         # éléments x[i, i] (diagonale)
diag(x)                    # idem et plus explicite
```

2.10 Exercices

2.1 a) Écrire une expression R pour créer la liste suivante :

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

$data
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

[[3]]
```

```
[1] 0 0 0

$test
[1] FALSE FALSE FALSE FALSE
```

b) Extraire les étiquettes de la liste.

c) Trouver le mode et la longueur du quatrième élément de la liste.

d) Extraire les dimensions du second élément de la liste.

e) Extraire les deuxième et troisième éléments du second élément de la liste.

f) Remplacer le troisième élément de la liste par le vecteur `3:8`.

2.2 Soit `x` un vecteur contenant les valeurs d'un échantillon :

```
> x
 [1]  7 19 15 12  4 14 18  9 13  1  1  9 11 11  6 12
[17]  4  2 14 19
```

Écrire une expression R permettant d'extraire les éléments suivants.

a) Le deuxième élément de l'échantillon.

b) Les cinq premiers éléments de l'échantillon.

c) Les éléments strictement supérieurs à 14.

d) Tous les éléments sauf les éléments en positions 6, 10 et 12.

2.3 Soit `x` une matrice 10×7 obtenue aléatoirement avec

```
> x <- matrix(sample(1:100, 70), 7, 10)
```

Écrire des expressions R permettant d'obtenir les éléments de la matrice demandés ci-dessous.

a) L'élément $(4, 3)$.

b) Le contenu de la sixième ligne.

c) Les première et quatrième colonnes (simultanément).

d) Les lignes dont le premier élément est supérieur à 50.

3 Opérateurs et fonctions

Objectifs du chapitre

- Tirer profit de l'arithmétique vectorielle caractéristique du langage R dans les calculs.
- Utiliser les opérateurs R les plus courants, notamment pour le traitement des vecteurs, le calcul de sommaires et la manipulation des matrices et tableaux.
- Faire l'appel d'une fonction dans R; concevoir comment les arguments sont passés à la fonction et le traitement des valeurs par défaut.
- Utiliser la fonction `if` pour l'exécution conditionnelle de commandes R.
- Distinguer la construction `if() ... else` de la fonction `ifelse`.
- Faire des boucles en R.
- Choisir entre les opérateurs `for`, `while` et `repeat` lors de la construction d'une boucle R.

Ce chapitre présente les principaux opérateurs arithmétiques, fonctions mathématiques et structures de contrôle disponibles dans R. La liste est évidemment loin d'être exhaustive, surtout étant donné l'évolution rapide du langage. Un des meilleurs endroits pour découvrir de nouvelles fonctions demeure la section `See Also` des rubriques d'aide, qui offre des hyperliens vers des fonctions apparentées au sujet de la rubrique.

Énoncé du problème

On s'intéresse à la somme des résultats du lancer de deux dés. Le premier dé compte 8 faces et le deuxième, 6. On souhaite calculer la moyenne des résultats de la somme supérieurs à 7.

3.1 Opérations arithmétiques

L'unité de base en R 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
```

- Si les vecteurs impliqués dans une expression arithmétique ne sont pas de la même longueur, les plus courts sont *recyclés* de façon à correspondre au plus long vecteur. Cette règle est particulièrement apparente 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
```

- Si la longueur du plus long vecteur est un multiple de celle du ou des autres vecteurs, ces derniers sont recyclés un nombre entier de fois :

```
> 1:10 + 1:5 + c(2, 4)  # vecteurs recyclés 2 et 5 fois
 [1]  4  8  8 12 12 11 11 15 15 19
> 1:10 + rep(1:5, 2) + rep(c(2, 4), 5)  # équivalent
 [1]  4  8  8 12 12 11 11 15 15 19
```

- Sinon, le plus court vecteur est recyclé un nombre fractionnaire de fois, mais comme ce résultat est rarement souhaité et provient généralement d'une erreur de programmation, un avertissement est affiché :

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

Opérateur	Fonction
`$`	extraction d'une liste
`^`	puissance
`-`	changement de signe
`:`	génération de suites
`%*% %% %/%`	produit matriciel, modulo, division entière
`* /`	multiplication, division
`+ -`	addition, soustraction
`< <= == >= > !=`	plus petit, plus petit ou égal, égal, plus grand ou égal, plus grand, différent de
`!`	négation logique
`& &&`	« et » logique
`|, ||`	« ou » logique
`-> ->>`	assignation
`<- <<-`	assignation

TAB. 3.1 – Principaux opérateurs du langage R, en ordre décroissant de priorité

Astuce

Grâce à la propriété d'opération élément par élément de R, il devrait être possible — en fait, il est souhaitable — de résoudre le problème sans avoir recours à des boucles.

3.2 Opérateurs

Le tableau 3.1 présente les opérateurs mathématiques et logiques les plus fréquemment employés, en ordre décroissant de priorité des opérations. Le tableau contient également les opérateurs d'assignation et d'extraction présentés au chapitre précédent ; il est utile de connaître leur niveau de priorité dans les expressions R.

Les opérateurs de puissance (`^`) et d'assignation à gauche (`<-`, `<<-`) sont évalués de droite à gauche ; tous les autres de gauche à droite. Ainsi, `2^2^3` est `2^8`, et non `4^3`, alors que `1 - 1 - 1` vaut `-1`, et non `1`.

3.3 Appels de fonctions

Les opérateurs du tableau 3.1 constituent des raccourcis utiles pour accéder aux fonctions les plus courantes de R. Pour toutes les autres, il faut appeler la fonction directement. Cette section passe en revue les règles d'appels d'une fonction et la façon de spécifier les arguments, qu'il s'agisse d'une fonction interne de R ou d'une fonction personnelle (voir le chapitre 5).

- Il n'y a pas de limite pratique quant au nombre d'arguments que peut avoir une fonction.
- Les arguments d'une fonction peuvent être spécifiés selon l'ordre établi dans la définition de la fonction. Cependant, il est beaucoup plus prudent et *fortement recommandé* de spécifier les arguments par leur nom, avec une construction de la forme `nom = valeur`, surtout après les deux ou trois premiers arguments.
- L'ordre des arguments est important ; il est donc nécessaire de les nommer 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 dans l'appel de la fonction.

Par exemple, la définition de la fonction `matrix` est la suivante :

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

- La fonction compte cinq arguments : `data`, `nrow`, `ncol`, `byrow` et `dimnames`.
- Ici, chaque argument a une valeur par défaut (ce n'est pas toujours le cas). Ainsi, un appel à `matrix` sans argument résulte en une matrice 1×1 remplie par colonne (sans importance, ici) de l'objet `NA` et dont les dimensions sont dépourvues d'étiquettes :

```
> 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
```

3.4 Quelques fonctions utiles

Le langage R compte un très grand nombre (des milliers !) de fonctions internes. Cette section en présente quelques-unes seulement, les fonctions de base les plus souvent utilisées pour programmer en R et pour manipuler des données.

Pour chaque fonction présentée dans les sections suivantes, on fournit un ou deux exemples d'utilisation. Ces exemples sont souvent loin de couvrir toutes les utilisations possibles d'une fonction. La section 3.7 fournit des exemples additionnels, mais il est recommandé de consulter les diverses rubriques d'aide pour connaître toutes les options des fonctions.

3.4.1 Manipulation de vecteurs

`seq` génération de suites de nombres

```
> seq(1, 9, by = 2)
[1] 1 3 5 7 9
```

`seq_len` version plus rapide de `seq` pour générer la suite des nombres de 1 à la valeur de l'argument

```
> seq_len(10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

`rep` répétition de valeurs ou de vecteurs

```
> rep(2, 10)
 [1] 2 2 2 2 2 2 2 2 2 2
```

`sort` tri en ordre croissant ou décroissant

```
> sort(c(4, -1, 2, 6))
[1] -1  2  4  6
```

`rank` rang des éléments d'un vecteur dans l'ordre croissant ou décroissant

```
> rank(c(4, -1, 2, 6))
[1] 3 1 2 4
```

`order` ordre d'extraction des éléments d'un vecteur pour les placer en ordre croissant ou décroissant

```
> order(c(4, -1, 2, 6))
[1] 2 3 1 4
```

`rev` renverser un vecteur

```
> rev(1:10)
 [1] 10  9  8  7  6  5  4  3  2  1
```

`head` extraction des n premiers éléments d'un vecteur ($n > 0$) ou suppression des n derniers ($n < 0$)

```
> head(1:10, 3); head(1:10, -3)
[1] 1 2 3
[1] 1 2 3 4 5 6 7
```

`tail` extraction des n derniers éléments d'un vecteur ($n > 0$) ou suppression des n premiers ($n < 0$)

```
> tail(1:10, 3); tail(1:10, -3)
[1]  8  9 10
[1]  4  5  6  7  8  9 10
```

`unique` extraction des éléments différents d'un vecteur

```
> unique(c(2, 4, 2, 5, 9, 5, 0))
[1] 2 4 5 9 0
```

3.4.2 Recherche d'éléments dans un vecteur

Les fonctions de cette sous-section sont toutes illustrées avec le vecteur

```
> x

[1]  4 -1  2 -3  6
```

`which` positions des valeurs `TRUE` dans un vecteur booléen

```
> which(x < 0)
[1] 2 4
```

`which.min` position du minimum dans un vecteur

```
> which.min(x)
[1] 4
```

`which.max` position du maximum dans un vecteur

```
> which.max(x)
[1] 5
```

`match` position de la première occurrence d'un élément dans un vecteur

```
> match(2, x)
[1] 3
```

`%in%` appartenance d'une ou plusieurs valeurs à un vecteur

```
> -1:2 %in% x
[1]  TRUE FALSE FALSE  TRUE
```

3.4.3 Arrondi

Les fonctions de cette sous-section sont toutes illustrées avec le vecteur

```
> x

[1] -3.6800000 -0.6666667  3.1415927  0.3333333
[5]  2.5200000
```

`round` arrondi à un nombre défini de décimales (par défaut 0)

```
> round(x)
[1] -4 -1  3  0  3
> round(x, 3)
[1] -3.680 -0.667  3.142  0.333  2.520
```

`floor` plus grand entier inférieur ou égal à l'argument

```
> floor(x)
[1] -4 -1  3  0  2
```

`ceiling` plus petit entier supérieur ou égal à l'argument

```
> ceiling(x)
[1] -3  0  4  1  3
```

`trunc` troncature vers zéro ; différent de `floor` pour les nombres négatifs

```
> trunc(x)
[1] -3  0  3  0  2
```

3.4.4 Sommaires et statistiques descriptives

Les fonctions de cette sous-section sont toutes illustrées avec le vecteur

```
> x

 [1] 14 17  7  9  3  4 25 21 24 11
```

`sum`, `prod` somme et produit des éléments d'un vecteur

```
> sum(x); prod(x)
[1] 135
[1] 24938020800
```

`diff` différences entre les éléments d'un vecteur (opérateur mathématique ∇)

```
> diff(x)
[1]   3 -10   2  -6   1  21  -4   3 -13
```

`mean` moyenne arithmétique (et moyenne tronquée avec l'argument `trim`)

```
> mean(x)
[1] 13.5
```

`var`, `sd` variance et écart type (versions sans biais)

```
> var(x)
[1] 64.5
```

`min`, `max` minimum et maximum d'un vecteur

```
> min(x); max(x)
[1] 3
```

```
[1] 25
```

`range` vecteur contenant le minimum et le maximum d'un vecteur

```
> range(x)
[1]  3 25
```

`median` médiane empirique

```
> median(x)
[1] 12.5
```

`quantile` quantiles empiriques

```
> quantile(x)
  0%  25%  50%  75% 100%
 3.0  7.5 12.5 20.0 25.0
```

`summary` statistiques descriptives d'un échantillon

```
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    3.0     7.5    12.5    13.5    20.0    25.0
```

Astuce

La fonction `mean` permettra de calculer la moyenne des résultats supérieurs à 7.

3.4.5 Sommaires cumulatifs et comparaisons élément par élément

Les fonctions de cette sous-section sont toutes illustrées avec le vecteur

```
> x

[1] 14 17  7  9  3
```

`cumsum, cumprod` somme et produit cumulatif d'un vecteur

```
> cumsum(x); cumprod(x)
[1] 14 31 38 47 50
[1]    14   238  1666 14994 44982
```

`cummin, cummax` minimum et maximum cumulatif

```
> cummin(x); cummax(x)
[1] 14 14  7  7  3
[1] 14 17 17 17 17
```

`pmin`, `pmax` minimum et maximum élément par élément (en parallèle) entre deux vecteurs ou plus

```
> pmin(x, 12)
[1] 12 12  7  9  3
> pmax(x, c(16, 23,  4, 12,  3))
[1] 16 23  7 12  3
```

3.4.6 Opérations sur les matrices

Les fonctions de cette sous-section sont toutes illustrées avec la matrice

```
> x

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

`nrow`, `ncol` nombre de lignes et de colonnes d'une matrice

```
> nrow(x); ncol(x)
[1] 2
[1] 2
```

`rowSums`, `colSums` sommes par ligne et par colonne, respectivement, des éléments d'une matrice ; voir aussi la fonction `apply` à la section 6.2

```
> rowSums(x)
[1] 6 4
```

`rowMeans`, `colMeans` moyennes par ligne et par colonne, respectivement, des éléments d'une matrice ; voir aussi la fonction `apply` à la section 6.2

```
> colMeans(x)
[1] 1.5 3.5
```

`t` transposée

```
> t(x)
     [,1] [,2]
[1,]    2    1
[2,]    4    3
```

`det` déterminant

```
> det(x)
[1] 2
```

`solve` 1) avec un seul argument (une matrice carrée) : inverse d'une matrice ; 2) avec deux arguments (une matrice carrée et un vecteur) : solution du système d'équations linéaires $\mathbf{Ax} = \mathbf{b}$

```
> solve(x)
     [,1] [,2]
[1,]  1.5   -2
[2,] -0.5    1
> solve(x, c(1, 2))
[1] -2.5  1.5
```

`diag` 1) avec une matrice en argument : diagonale de la matrice ; 2) avec un vecteur en argument : matrice diagonale formée avec le vecteur ; 3) avec un scalaire p en argument : matrice identité $p \times p$

```
> diag(x)
[1] 2 3
```

3.4.7 Produit extérieur

La fonction `outer` calcule le produit extérieur entre deux vecteurs. Ce n'est pas la fonction la plus intuitive à utiliser, mais elle s'avère extrêmement utile pour faire plusieurs opérations en une seule expression tout en évitant les boucles. La syntaxe de `outer` est :

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

Le résultat est l'application la fonction `FUN` (`"*"` par défaut) entre chacun des éléments de `X` et chacun des éléments de `Y`, autrement dit

```
FUN(X[i], Y[j])
```

pour toutes les valeurs des indices `i` et `j`.

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

- Lorsque `FUN` est un opérateur arithmétique du tableau 3.1, on place le symbole entre guillemets : `"*"`, `"+"`, `"<="`, etc.
- L'opérateur `%o%` est un raccourci de `outer(X, Y, "*")`.

Astuce

Nous avons trouvé ici la fonction clé pour résoudre notre problème sans faire de boucles. La fonction `outer` permet de calculer facilement tous les résultats possibles de la somme de deux dés :

```
> x <- 1:8         # résultats possibles du premier dé
> y <- 1:6         # résultats possibles du second dé
> outer(x, y, "+") # sommes de toutes les combinaisons

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    2    3    4    5    6    7
[2,]    3    4    5    6    7    8
[3,]    4    5    6    7    8    9
[4,]    5    6    7    8    9   10
[5,]    6    7    8    9   10   11
[6,]    7    8    9   10   11   12
[7,]    8    9   10   11   12   13
[8,]    9   10   11   12   13   14
```

3.5 Structures de contrôle

Les structures de contrôle sont des commandes qui permettent de déterminer le flux d'exécution d'un programme : choix entre des blocs de code, répétition de commandes ou sortie forcée.

On se contente, ici, de mentionner les structures de contrôle disponibles en R. La section 3.7 fournit des exemples d'utilisation.

3.5.1 Exécution conditionnelle

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

Si *condition* est vraie, *branche.vrai* est exécutée, sinon ce sera *branche.faux*. Dans le cas où l'une ou l'autre de *branche.vrai* ou *branche.faux* comporte plus d'une expression, regrouper celles-ci dans des accolades `{ }`.

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

Fonction vectorielle qui retourne un vecteur de la même longueur que *condition* formé ainsi : pour chaque élément `TRUE` de *condition* on choisit l'élément correspondant de *expression.vrai* et pour chaque élément `FALSE` on choisit l'élément correspondant de *expression.faux*. L'utilisation n'est pas très intuitive, alors examiner attentivement les exemples de la rubrique d'aide.

`switch(test, cas.1 = action.1, cas.2 = action.2, ...)`

Structure utilisée plutôt rarement. Consulter la rubrique d'aide au besoin.

3.5.2 Boucles

Les boucles sont et doivent être utilisées avec parcimonie en R, car elles sont généralement inefficaces. Dans la majeure partie des cas, il est possible de vectoriser les calculs pour éviter les boucles explicites, ou encore de s'en remettre aux fonctions `outer`, `apply`, `lapply` `sapply` et `mapply` (section 6.2) pour réaliser les boucles de manière plus efficace.

`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 de nombres, en fait.

`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.

`break`

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

`next`

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

3.6 Fonctions additionnelles

La bibliothèque des fonctions internes de R est divisée en ensembles de fonctions et de jeux de données apparentés nommés *packages* (terme que l'équipe de traduction française de R a choisi de conserver tel quel). On démarrage, R charge automatiquement quelques packages de la bibliothèque, ceux contenant les fonctions les plus fréquemment utilisées. On peut voir la liste des packages déjà en mémoire avec la fonction `search` et le contenu de toute la bibliothèque avec la fonction `library` (résultat non montré ici) :

```
> search()

[1] ".GlobalEnv"        "package:stats"
[3] "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"
[9] "package:base"
```

Une des grandes forces de R est la facilité avec laquelle on peut ajouter des fonctionnalités au système par le biais de packages externes. Dès les débuts de R, les développeurs et utilisateurs ont mis sur pied le dépôt central de packages *Comprehensive R Archive Network* (CRAN ; http://cran.r-project.org). Ce site compte aujourd'hui des milliers d'extensions et le nombre ne cesse de croître.

Le système R rend simple le téléchargement et l'installation de nouveaux packages avec la fonction `install.packages`. L'annexe D explique plus en détails comment gérer sa bibliothèque personnelle et installer des packages externes.

Solution du problème

Nous savons déjà que l'expression

```
> m <- outer(x, y, "+")
```

calcule tous les résultats de la somme du lancer des deux dés. Il faut maintenant extraire de cette matrice les résultats supérieurs à 7 et en faire la moyenne.
Par la propriété d'opération élément par élément, la comparaison

```
> m > 7

      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] FALSE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE  TRUE
[3,] FALSE FALSE FALSE FALSE  TRUE  TRUE
[4,] FALSE FALSE FALSE  TRUE  TRUE  TRUE
[5,] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[6,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
[7,]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[8,]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

retourne une matrice booléenne de la même dimension que `m`. Pour extraire les résultats de `m` supérieurs à 7, il suffit d'indicer `m` avec la matrice booléenne ci-dessus :

```
> m[m > 7]

 [1]  8  9  8  9 10  8  9 10 11  8  9 10 11 12  8  9
[17] 10 11 12 13  8  9 10 11 12 13 14
```

Le résultat recherché est donc

Solution du problème (suite)

```
> mean(m[m > 7])

[1] 10.07407
```

Le tout peut s'écrire en une seule expression — quelque peu alambiquée — ainsi :

```
> mean((m <- outer(x, y, "+"))[m > 7])

[1] 10.07407
```

3.7 Exemples

```
###
### OPÉRATIONS ARITHMÉTIQUES
###

## L'arithmétique vectorielle caractéristique du langage R
## rend très simple et intuitif de faire des opérations
## mathématiques courantes. Là où plusieurs langages de
## programmation exigent des boucles, R fait le calcul
## directement. En effet, les règles de l'arithmétique en R
## sont globalement les mêmes qu'en algèbre vectorielle et
## matricielle.
5 * c(2, 3, 8, 10)          # multiplication par une constante
c(2, 6, 8) + c(1, 4, 9)     # addition de deux vecteurs
c(0, 3, -1, 4)^2            # élévation à une puissance

## Dans les règles de l'arithmétique vectorielle, les
## longueurs des vecteurs doivent toujours concorder. R permet
## plus de flexibilité en recyclant les vecteurs les plus
## courts dans une opération. Il n'y a donc à peu près jamais
## d'erreurs de longueur en R! C'est une arme à deux
## tranchants: le recyclage des vecteurs facilite le codage,
## mais peut aussi résulter en des réponses complètement
## erronées sans que le système ne détecte d'erreur.
8 + 1:10                    # 8 est recyclé 10 fois
c(2, 5) * 1:10              # c(2, 5) est recyclé 5 fois
```

```
c(-2, 3, -1, 4)^(1:4)      # quatre puissances différentes

## On se rappelle que les matrices (et les tableaux) sont des
## vecteurs. Les règles ci-dessus s'appliquent donc aussi aux
## matrices, ce qui résulte en des opérateurs qui ne sont pas
## définis en algèbre linéaire usuelle.
(x <- matrix(1:4, 2))       # matrice 2 x 2
(y <- matrix(3:6, 2))       # autre matrice 2 x 2
5 * x                       # multiplication par une constante
x + y                       # addition matricielle
x * y                       # produit *élément par élément*
x %*% y                     # produit matriciel
x / y                       # division *élément par élément*
x * c(2, 3)                 # produit par colonne

###
### OPÉRATEURS
###

## Seuls les opérateurs %%, %/% et logiques sont illustrés
## ici. Premièrement, l'opérateur modulo retourne le reste
## d'une division.
5 %% 2                     # 5/2 = 2 reste 1
5 %% 1:5                   # remarquer la périodicité
10 %% 1:15                 # x %% y = x si x < y

## Le modulo est pratique dans les boucles, par exemple pour
## afficher un résultat à toutes les n itérations seulement.
for (i in 1:50)
{
    ## Affiche la valeur du compteur toutes les 5 itérations.
    if (0 == i %% 5)
        print(i)
}

## La division entière retourne la partie entière de la
## division d'un nombre par un autre.
5 %/% 1:5
10 %/% 1:15

## Le ET logique est vrai seulement lorsque les deux
## expressions sont vraies.
c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)

## Le OU logique est faux seulement lorsque les deux
```

```
## expressions sont fausses.
c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)

## La négation logique transforme les vrais en faux et vice
## versa.
! c(TRUE, FALSE, FALSE, TRUE)

## On peut utiliser les opérateurs logiques &, | et !
## directement avec des nombres. Dans ce cas, le nombre zéro
## est traité comme FALSE et tous les autres nombres comme
## TRUE.
0:5 & 5:0
0:5 | 5:0
!0:5

## Ainsi, dans une expression conditionnelle, inutile de
## vérifier si, par exemple, un nombre est égal à zéro. On
## peut utiliser le nombre directement et sauver des
## opérations de comparaison qui peuvent devenir coûteuses en
## temps de calcul.
x <- 1                      # valeur quelconque
if (x != 0) x + 1           # TRUE pour tout x != 0
if (x) x + 1                # tout à fait équivalent!

## L'exemple de boucle ci-dessus peut donc être légèrement
## modifié.
for (i in 1:50)
{
    ## Affiche la valeur du compteur toutes les 5 itérations.
    if (!i %% 5)
        print (i)
}

## Dans les calculs numériques, TRUE vaut 1 et FALSE vaut 0.
a <- c("Impair", "Pair")
x <- c(2, 3, 6, 8, 9, 11, 12)
x %% 2
(!x %% 2) + 1
a[(!x %% 2) + 1]

## Un mot en terminant sur l'opérateur '=='. C'est l'opérateur
## à utiliser pour vérifier si deux valeurs sont égales, et
## non '='. C'est là une erreur commune --- et qui peut être
## difficile à détecter --- lorsque l'on programme en R.
5 = 2                       # erreur de syntaxe
```

```
5 == 2                      # comparaison

###
### APPELS DE FONCTIONS
###

## Les invocations de la fonction 'matrix' ci-dessous sont
## toutes équivalentes. On remarquera, entre autres, comment
## les arguments sont spécifiés (par nom ou par position).
matrix(1:12, 3, 4)
matrix(1:12, ncol = 4, nrow = 3)
matrix(nrow = 3, ncol = 4, data = 1:12)
matrix(nrow = 3, ncol = 4, byrow = FALSE, 1:12)
matrix(nrow = 3, ncol = 4, 1:12, FALSE)

###
### QUELQUES FONCTIONS UTILES
###

## MANIPULATION DE VECTEURS
x <- c(50, 30, 10, 20, 60, 30, 20, 40)  # vecteur non ordonné

## Séquences de nombres.
seq(from = 1, to = 10)       # équivalent à 1:10
seq_len(10)                  # plus rapide que 'seq'
seq(-10, 10, length = 50)    # incrément automatique
seq(-2, by = 0.5, along = x) # même longueur que 'x'
seq_along(x)                 # plus rapide que 'seq'

## Répétition de nombres ou de vecteurs complets.
rep(1, 10)                   # utilisation de base
rep(x, 2)                    # répéter un vecteur
rep(x, times = 2, each = 4) # combinaison des arguments
rep(x, times = 1:8)          # nombre de répétitions différent
                             # pour chaque élément de 'x'

## Classement en ordre croissant ou décroissant.
sort(x)                     # classement en ordre croissant
sort(x, decr = TRUE)        # classement en ordre décroissant
sort(c("abc", "B", "Aunt", "Jemima")) # chaînes de caractères
sort(c(TRUE, FALSE))        # FALSE vient avant TRUE

## La fonction 'order' retourne la position, dans le vecteur
## donné en argument, du premier élément selon l'ordre
## croissant, puis du deuxième, etc. Autrement dit, on obtient
```

```r
## l'ordre dans lequel il faut extraire les données du vecteur
## pour les obtenir en ordre croissant.
order(x)                    # regarder dans le blanc des yeux
x[order(x)]                 # équivalent à 'sort(x)'

## Rang des éléments d'un vecteur dans l'ordre croissant.
rank(x)                     # rang des élément de 'x'

## Renverser l'ordre d'un vecteur.
rev(x)

## Extraction ou suppression en tête ou en queue de vecteur.
head(x, 3)                  # trois premiers éléments
head(x, -2)                 # tous sauf les deux derniers
tail(x, 3)                  # trois derniers éléments
tail(x, -2)                 # tous sauf les deux premiers

## Expressions équivalentes sans 'head' et 'tail'
x[1:3]                      # trois premiers éléments
x[1:(length(x) - 2)]        # tous sauf les deux derniers
x[(length(x)-2):length(x)] # trois derniers éléments
rev(rev(x)[1:3])            # avec petits vecteurs seulement
x[c(-1, -2)]                # tous sauf les deux premiers

## Seulement les éléments différents d'un vecteur.
unique(x)

## RECHERCHE D'ÉLÉMENTS DANS UN VECTEUR
which(x >= 30)              # positions des éléments >= 30
which.min(x)                # position du minimum
which.max(x)                # position du maximum
match(20, x)                # position du premier 20 dans 'x'
match(c(20, 30), x)         # aussi pour plusieurs valeurs
60 %in% x                   # 60 appartient à 'x'
70 %in% x                   # 70 n'appartient pas à 'x'

## ARRONDI
(x <- c(-21.2, -pi, -1.5, -0.2, 0, 0.2, 1.7823, 315))
round(x)                    # arrondi à l'entier
round(x, 2)                 # arrondi à la seconde décimale
round(x, -1)                # arrondi aux dizaines
ceiling(x)                  # plus petit entier supérieur
floor(x)                    # plus grand entier inférieur
trunc(x)                    # troncature des décimales
```

```
## SOMMAIRES ET STATISTIQUES DESCRIPTIVES
sum(x)                      # somme des éléments
prod(x)                     # produit des éléments
diff(x)                     # x[2] - x[1], x[3] - x[2], etc.
mean(x)                     # moyenne des éléments
mean(x, trim = 0.125)       # moyenne sans minimum et maximum
var(x)                      # variance (sans biais)
(length(x) - 1)/length(x) * var(x) # variance biaisée
sd(x)                       # écart type
max(x)                      # maximum
min(x)                      # minimum
range(x)                    # c(min(x), max(x))
diff(range(x))              # étendue de 'x'
median(x)                   # médiane (50e quantile) empirique
quantile(x)                 # quantiles empiriques
quantile(x, 1:10/10)        # on peut spécifier les quantiles
summary(x)                  # plusieurs des résultats ci-dessus

## SOMMAIRES CUMULATIFS ET COMPARAISONS ÉLÉMENT PAR ÉLÉMENT
(x <- sample(1:20, 6))
(y <- sample(1:20, 6))
cumsum(x)                   # somme cumulative de 'x'
cumprod(y)                  # produit cumulatif de 'y'
rev(cumprod(rev(y)))        # produit cumulatif renversé
cummin(x)                   # minimum cumulatif
cummax(y)                   # maximum cumulatif
pmin(x, y)                  # minimum élément par élément
pmax(x, y)                  # maximum élément par élément

## OPÉRATIONS SUR LES MATRICES
(A <- sample(1:10, 16, replace = TRUE)) # avec remise
dim(A) <- c(4, 4)           # conversion en une matrice 4 x 4
b <- c(10, 5, 3, 1)         # un vecteur quelconque
A                           # la matrice 'A'
t(A)                        # sa transposée
solve(A)                    # son inverse
solve(A, b)                 # la solution de Ax = b
A %*% solve(A, b)           # vérification de la réponse
diag(A)                     # extraction de la diagonale de 'A'
diag(b)                     # matrice diagonale formée avec 'b'
diag(4)                     # matrice identité 4 x 4
(A <- cbind(A, b))          # matrice 4 x 5
nrow(A)                     # nombre de lignes de 'A'
ncol(A)                     # nombre de colonnes de 'A'
rowSums(A)                  # sommes par ligne
```

```
colSums(A)                  # sommes par colonne
apply(A, 1, sum)            # équivalent à 'rowSums(A)'
apply(A, 2, sum)            # équivalent à 'colSums(A)'
apply(A, 1, prod)           # produit par ligne avec 'apply'

## PRODUIT EXTÉRIEUR
x <- c(1, 2, 4, 7, 10, 12)
y <- c(2, 3, 6, 7, 9, 11)
outer(x, y)                 # produit extérieur
x %o% y                     # équivalent plus court
outer(x, y, "+")            # «somme extérieure»
outer(x, y, "<=")           # toutes les comparaisons possibles
outer(x, y, pmax)           # idem

###
### STRUCTURES DE CONTRÔLE
###

## Pour illustrer les structures de contrôle, on a recours à
## un petit exemple tout à fait artificiel: un vecteur est
## rempli des nombres de 1 à 100, à l'exception des multiples
## de 10. Ces derniers sont affichés à l'écran.
##
## À noter qu'il est possible --- et plus efficace --- de
## créer le vecteur sans avoir recours à des boucles.
(1:100)[-((1:10) * 10)]             # sans boucle!
rep(1:9, 10) + rep(0:9*10, each = 9) # une autre façon!

## Bon, l'exemple proprement dit...
x <- numeric(0)             # initialisation du contenant 'x'
j <- 0                      # compteur pour la boucle
for (i in 1:100)
{
    if (i %% 10)            # si i n'est pas un multiple de 10
        x[j <- j + 1] <- i # stocker sa valeur dans 'x'
    else                    # sinon
        print(i)            # afficher la valeur à l'écran
}
x                           # vérification

## Même chose que ci-dessus, mais sans le compteur 'j' et les
## valeurs manquantes aux positions 10, 20, ..., 100 sont
## éliminées à la sortie de la boucle.
x <- numeric(0)
```

```
for (i in 1:100)
{
    if (i %% 10)
        x[i] <- i
    else
        print(i)
}
x <- x[!is.na(x)]
x

## On peut refaire l'exemple avec une boucle 'while', mais
## cette structure n'est pas naturelle ici puisque l'on sait
## d'avance qu'il faudra faire la boucle exactement 100
## fois. Le 'while' est plutôt utilisé lorsque le nombre de
## répétitions est inconnu. De plus, une boucle 'while' n'est
## pas nécessairement exécutée puisque le critère d'arrêt est
## évalué dès l'entrée dans la boucle.
x <- numeric(0)
j <- 0
i <- 1                      # pour entrer dans la boucle [*]
while (i <= 100)
{
    if (i %% 10)
        x[j <- j + 1] <- i
    else
        print(i)
    i <- i + 1              # incrémenter le compteur!
}
x

## La remarque faite au sujet de la boucle 'while' s'applique
## aussi à la boucle 'repeat'. Par contre, le critère d'arrêt
## de la boucle 'repeat' étant évalué à la toute fin, la
## boucle est exécutée au moins une fois. S'il faut faire la
## manoeuvre marquée [*] ci-dessus pour s'assurer qu'une
## boucle 'while' est exécutée au moins une fois... c'est
## qu'il faut utiliser 'repeat'.
x <- numeric(0)
j <- 0
i <- 1
repeat
{
    if (i %% 10)
        x[j <- j + 1] <- i
    else
```

```
        print(i)
    if (100 < (i <- i + 1)) # incrément et critère d'arrêt
        break
}
x

###
### FONCTIONS ADDITIONNELLES
###

## La fonction 'search' retourne la liste des environnements
## dans lesquels R va chercher un objet (en particulier une
## fonction). '.GlobalEnv' est l'environnement de travail.
search()

## Liste de tous les packages installés sur votre système.
library()

## Chargement du package 'MASS', qui contient plusieurs
## fonctions statistiques très utiles.
library("MASS")
```

3.8 Exercices

3.1 À l'aide des fonctions `rep`, `seq` et `c` seulement, générer les séquences suivantes.

a) `0 6 0 6 0 6`

b) `1 4 7 10`

c) `1 2 3 1 2 3 1 2 3 1 2 3`

d) `1 2 2 3 3 3`

e) `1 1 1 2 2 3`

f) `1 5.5 10`

g) `1 1 1 1 2 2 2 2 3 3 3 3`

3.2 Générer les suites de nombres suivantes à l'aide des fonctions `:` et `rep` seulement, donc sans utiliser la fonction `seq`.

a) `1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2`

b) `1 3 5 7 9 11 13 15 17 19`

c) `-2 -1 0 1 2 -2 -1 0 1 2`

d) `-2 -2 -1 -1 0 0 1 1 2 2`

e) `10 20 30 40 50 60 70 80 90 100`

3.3 À l'aide de la commande `apply`, écrire des expressions R qui remplaceraient les fonctions suivantes.

a) `rowSums`

b) `colSums`

c) `rowMeans`

d) `colMeans`

3.4 Sans utiliser les fonctions `factorial`, `lfactorial`, `gamma` ou `lgamma`, générer la séquence $1!, 2!, \ldots, 10!$.

3.5 Trouver une relation entre `x`, `y`, `x %% y` (modulo) et `x %/% y` (division entière), où `y != 0`.

3.6 Simuler un échantillon $\mathbf{x} = (x_1, x_2, x_3, ..., x_{20})$ avec la fonction `sample`. Écrire une expression R permettant d'obtenir ou de calculer chacun des résultats demandés ci-dessous.

a) Les cinq premiers éléments de l'échantillon.

b) La valeur maximale de l'échantillon.

c) La moyenne des cinq premiers éléments de l'échantillon.

d) La moyenne des cinq derniers éléments de l'échantillon.

3.7 a) Trouver une formule pour calculer la position, dans le vecteur sous-jacent, de l'élément (i, j) d'une matrice $I \times J$ remplie par colonne.

b) Répéter la partie a) pour l'élément (i, j, k) d'un tableau $I \times J \times K$.

3.8 Simuler une matrice `mat` 10×7, puis écrire des expressions R permettant d'effectuer les tâches demandées ci-dessous.

a) Calculer la somme des éléments de chacunes des lignes de la matrice.

b) Calculer la moyenne des éléments de chacunes des colonnes de la matrice.

c) Calculer la valeur maximale de la sous-matrice formée par les trois premières lignes et les trois premières colonnes.

d) Extraire toutes les lignes de la matrice dont la moyenne des éléments est supérieure à 7.

3.9 On vous donne la liste et la date des 31 meilleurs temps enregistrés au 100 mètres homme entre 1964 et 2005 :

```
> temps <- c(10.06, 10.03, 10.02,  9.95, 10.04, 10.07,
+            10.08, 10.05,  9.98, 10.09, 10.01, 10.00,
+             9.97,  9.93,  9.96,  9.99,  9.92,  9.94,
+             9.90,  9.86,  9.88,  9.87,  9.85,  9.91,
+             9.84,  9.89,  9.79,  9.80,  9.82,  9.78,
+             9.77)
> names(temps) <- c("1964-10-15", "1968-06-20",
+     "1968-10-13", "1968-10-14", "1968-10-14",
+     "1968-10-14", "1968-10-14", "1975-08-20",
+     "1977-08-11", "1978-07-30", "1979-09-04",
+     "1981-05-16", "1983-05-14", "1983-07-03",
+     "1984-05-05", "1984-05-06", "1988-09-24",
+     "1989-06-16", "1991-06-14", "1991-08-25",
+     "1991-08-25", "1993-08-15", "1994-07-06",
+     "1994-08-23", "1996-07-27", "1996-07-27",
+     "1999-06-16", "1999-08-22", "2001-08-05",
+     "2002-09-14", "2005-06-14")
```

Extraire de ce vecteur les records du monde seulement, c'est-à-dire la première fois que chaque temps a été réalisé.

4 Exemples résolus

Objectifs du chapitre

- Mettre en pratique les connaissances acquises dans les chapitres précédents.
- Tirer profit de l'arithmétique vectorielle de R pour effectuer des calculs complexes sans boucles.
- Utiliser l'initialisation de vecteurs et leur indiçage de manière à réduire le temps de calcul.

Ce chapitre propose de faire le point sur les concepts étudiés jusqu'à maintenant par le biais de quelques exemples résolus. On y met particulièrement en évidence les avantages de l'approche vectorielle du langage R.

Les exemples font appel à quelques connaissances de base en mathématiques financières et en théorie des probabilités.

Énoncé du problème

On modélise la distance atteinte par le lancer du poids d'un adulte à l'aide d'une variable aléatoire $X \sim \text{Pareto}(\alpha = 20, \lambda = 75)$. On rappelle que la fonction de répartition de cette loi est :

$$F_X(x) = 1 - \left(\frac{\lambda}{\lambda + x}\right)^{\alpha}.$$

En compétition, les juges inscrivent un résultat seulement si le lancer dépasse la distance d. La probabilité qu'un lancer dépasse la distance e^d sachant que ce lancer a dépassé la distance d est de 0,75. Calculer la constante d pour laquelle la relation précédente est vraie, autrement dit la valeur de d tel que

$$\Pr[X > e^d | X > d] = 0{,}75. \tag{4.1}$$

4.1 Calcul de valeurs actuelles

La valeur actuelle d'une série de paiements $P_1, P_2, \ldots, P_n$ à la fin des années $1, 2, \ldots, n$ est

$$\sum_{j=1}^{n} \prod_{k=1}^{j} (1 + i_k)^{-1} P_j, \tag{4.2}$$

où i_k est le taux d'intérêt effectif annuellement durant l'année k. Lorsque le taux d'intérêt est constant au cours des n années, cette formule se simplifie en

$$\sum_{j=1}^{n} (1 + i)^{-j} P_j. \tag{4.3}$$

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

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

   Avec un paiement annuel et un taux d'intérêt constants, on utilise la formule (4.3) avec $P_j = P = 1\,000$:

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

[1] 4212.364
```

   Remarquer comme l'expression R se lit exactement comme la formule mathématique. De plus, le terme constant 1 000 est sorti de la somme pour réduire le nombre de multiplications de cinq à une seule.

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

   Les paiements annuels sont différents, mais le taux d'intérêt est toujours le même. La formule (4.3) s'applique donc directement :

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

[1] 3280.681
```

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.

   Avec différents paiements annuels et des taux d'intérêt différents, il faut employer la formule (4.2). On obtient le résultat voulu sans aucune boucle en réalisant le produit cumulatif des taux d'intérêt avec la fonction `cumprod` :

```
> sum(c(500, 800, 900, 750, 1000) /
+     cumprod(1 + c(0.05, 0.06, 0.055, 0.065, 0.07)))

[1] 3308.521
```

4.2 Fonctions de masse de probabilité

On souhaite calculer toutes ou la majeure partie des probabilités de deux lois de probabilité, puis vérifier que la somme des probabilités est bien égale à 1.

Cet exemple est quelque peu artificiel dans la mesure où il existe dans R des fonctions internes pour calculer les principales caractéristiques des lois de probabilité les plus usuelles. Nous utiliserons d'ailleurs ces fonctions pour vérifier nos calculs.

a) Calculer toutes les masses de probabilité de la distribution binomiale pour des valeurs des paramètres n et p quelconques. La fonction de masse de probabilité de la binomiale est

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

Soit $n = 10$ et $p = 0{,}8$. Les coefficients binomiaux sont calculés avec la fonction `choose` :

```
> 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
```

On vérifie les réponses obtenues avec la fonction interne `dbinom` :

```
> dbinom(x, n, prob = 0.8)

 [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
```

On vérifie enfin que les probabilités somment à 1 :

```
> sum(choose(n, x) * p^x * (1 - p)^rev(x))

[1] 1
```

b) Calculer la majeure partie des masses de probabilité de la distribution de Poisson, dont la fonction de masse de probabilité est

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

où $x! = x(x-1)\cdots 2 \cdot 1$.

La loi de Poisson ayant un support infini, on calcule les probabilités en $x = 0, 1, \ldots, 10$ seulement avec $\lambda = 5$. On calcule les factorielles avec la fonction `factorial`. On notera au passage que `factorial(x) == gamma(x + 1)`, où la fonction R `gamma` calcule les valeurs de la fonction mathématique du même nom

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

avec $\Gamma(0) = 1$. Pour n entier, on a donc $\Gamma(n) = (n-1)!$.

```
> 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
```

Vérification avec la fonction interne `dpois` :

```
> dpois(x, lambda)

 [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
```

Pour vérifier que les probabilités somment à 1, il faudra d'abord tronquer le support infini de la Poisson à une « grande » valeur. Ici, 170 est suffisamment éloigné de la moyenne de la distribution, 5. Remarquer que le

produit par $e^{-\lambda}$ est placé à l'extérieur de la somme pour ainsi faire un seul produit plutôt que 171.

```
> x <- 0:170
> exp(-lambda) * sum((lambda^x / factorial(x)))

[1] 1
```

4.3 Fonction de répartition de la loi gamma

La loi gamma est fréquemment employée pour modéliser des événements ne pouvant prendre que des valeurs positives et pour lesquels les petites valeurs sont plus fréquentes que les grandes. Par exemple, on utilise parfois la loi gamma en sciences actuarielles pour la modélisation des montants de sinistres. Nous utiliserons la paramétrisation où la fonction de densité de probabilité est

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}, \quad x > 0, \tag{4.4}$$

où $\Gamma(\cdot)$ est la fonction gamma définie dans l'exemple précédent.

Il n'existe pas de formule explicite de la fonction de répartition de la loi gamma. Néanmoins, la valeur de la fonction de répartition d'une loi gamma de paramètre α entier et $\lambda = 1$ peut être obtenue à partir de la formule

$$F(x; \alpha, 1) = 1 - e^{-x} \sum_{j=0}^{\alpha-1} \frac{x^j}{j!}. \tag{4.5}$$

a) Évaluer $F(4; 5, 1)$.

Cet exercice est simple puisqu'il s'agit de calculer une seule valeur de la fonction de répartition avec un paramètre α fixe. Par une application directe de (4.5), on a :

```
> 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
```

On peut éviter de générer essentiellement la même suite de nombres à deux reprises en ayant recours à une variable intermédiaire. Au risque de rendre le code un peu moins lisible (mais plus compact !), l'affectation et le calcul final peuvent même se faire dans une seule expression.

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

[1] 0.3711631
```

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

Cet exercice est beaucoup plus compliqué qu'il n'y paraît au premier abord. Ici, la valeur de α demeure fixe, mais on doit calculer, en une seule expression, la valeur de la fonction de répartition en plusieurs points. Or, cela exige de faire d'un coup le calcul x^j pour plusieurs valeur de x et plusieurs valeurs de j. 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 0.55950671
[5] 0.71494350 0.82700839 0.90036760 0.94503636
[9] 0.97074731
```

Vérification avec la fonction interne `pgamma` :

```
> pgamma(x, alpha)

[1] 0.05265302 0.18473676 0.37116306 0.55950671
[5] 0.71494350 0.82700839 0.90036760 0.94503636
[9] 0.97074731
```

Il est laissé en exercice de déterminer pourquoi la transposée est nécessaire dans l'expression ci-dessus. Exécuter l'expression étape par étape, de l'intérieur vers l'extérieur, pour mieux comprendre comment on arrive à faire le calcul en (4.5).

Astuce

Simplifions déjà l'équation à résoudre. On a

$$\begin{aligned}\Pr[X > e^d | X > d] &= \frac{\Pr[X > e^d \cap X > d]}{\Pr[X > d]} \\ &= \frac{\Pr[X > e^d]}{\Pr[X > d]} \\ &= \frac{[\lambda/(\lambda + e^d]^\alpha}{[\lambda/(\lambda + d)]^\alpha} \\ &= \left(\frac{\lambda + d}{\lambda + e^d}\right)^\alpha.\end{aligned}$$

On cherche donc à résoudre pour d l'équation

$$\left(\frac{\lambda + d}{\lambda + e^d}\right)^\alpha = 0{,}75.$$

Celle-ci n'admet pas de solution explicite.

4.4 Algorithme du point fixe

Trouver la racine d'une fonction g — c'est-à-dire le point x où $g(x) = 0$ — est un problème classique en mathématiques. Très souvent, il est possible de reformuler le problème de façon à plutôt chercher le point x où $f(x) = x$. La solution d'un tel problème est 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 $x_n = f(x_{n-1})$ pour $n = 1, 2, \ldots$;
3. répéter l'étape 2 jusqu'à ce que $|x_n - x_{n-1}| < \epsilon$ ou $|x_n - x_{n-1}|/|x_{n-1}| < \epsilon$.

On doit trouver, à l'aide de la méthode du point fixe, la valeur de i telle que

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

c'est à dire le taux de rendement d'une série de 10 versements de 1 pour laquelle on a payé un montant de 8,21.

Puisque, d'une part, nous ignorons combien de fois la procédure itérative devra être répétée et que, d'autre part, il faut exécuter la procédure au moins

une fois, le choix logique pour la structure de contrôle à utiliser dans cette procédure itérative est `repeat`. De plus, il faut comparer deux valeurs successives du taux d'intérêt, nous devrons donc avoir recours à deux variables. On a :

```
> 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
```

Vérification :

```
> (1 - (1 + i)^(-10))/i

[1] 8.21
```

Nous verrons au chapitre 5 comment créer une fonction à partir de ce code.

Astuce

L'algorithme du point fixe peut nous permettre de résoudre numériquement l'équation

$$\left(\frac{\lambda + d}{\lambda + e^d}\right)^{\alpha} = 0{,}75.$$

Pour utiliser l'algorithme, il faut réécrire l'équation sous une forme où l'inconnue d se trouve de part et d'autre de l'égalité. En isolant, pour des raisons numériques, le d se trouvant au dénominateur de l'équation ci-dessus, on obtient :

$$d = \log\left[\lambda(1 - 0{,}75^{1/\alpha}) + d\right] - \frac{\log 0{,}75}{\alpha}.$$

4.5 Suite de Fibonacci

La suite de Fibonacci est une suite de nombres entiers très connue. Les deux premiers termes de la suite sont 0 et 1 et tous les autres sont la somme des deux termes précédents. Mathématiquement, les valeurs de la suite de Fibonacci sont données par la fonction

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2), \quad n \geq 2. \end{aligned}$$

Le quotient de deux termes successifs converge vers $(1+\sqrt{5})/2$, le nombre d'or.

On veut calculer les $n > 2$ premiers termes de la suite de Fibonacci. Ce problème étant intrinsèquement récursif, nous devons utiliser une boucle.

Voici une première solution pour $n = 10$:

```
> n <- 10
> x <- c(0, 1)
> for (i in 3:n) x[i] <- x[i - 1] + x[i - 2]
> x

 [1]  0  1  1  2  3  5  8 13 21 34
```

La procédure ci-dessus a un gros défaut : la taille de l'objet `x` est constamment augmentée pour stocker une nouvelle valeur de la suite. Tentons une analogie alimentaire pour cette manière de procéder. Pour ranger des biscuits frais sortis du four, on prend un premier biscuit et on le range dans un plat ne pouvant contenir qu'un seul biscuit. Arrivé au second biscuit, on constate que le contenant n'est pas assez grand, alors on sort un plat pouvant contenir deux biscuits, on change le premier biscuit de plat et on y range aussi le second biscuit. Arrivé au troisième biscuit, le petit manège recommence, et ainsi de suite jusqu'à ce que le plateau de biscuits soit épuisé. C'est ce que nous nommerons, non sans un sourire en coin, le Syndrôme de la plaque à biscuits™.

Le manège décrit ci-dessus se reproduit à l'identique dans la mémoire de l'ordinateur, l'odeur des bons biscuits chauds en moins. En effet, l'ordinateur doit constamment allouer de la nouvelle mémoire et déplacer les termes déjà sauvegardés au fur et à mesure que le vecteur `x` grandit. On aura compris qu'une telle façon de faire est à éviter absolument lorsque c'est possible — et ça l'est la plupart du temps.

Quand on sait quelle sera la longueur d'un objet, comme c'est le cas dans cet exemple, il vaut mieux créer un contenant vide de la bonne longueur et le remplir par la suite. Cela nous donne une autre façon de calculer la suite de Fibonacci :

```
> n <- 10
> x <- numeric(n)      # création du contenant
> x[2] <- 1            # x[1] vaut déjà 0
> for (i in 3:n) x[i] <- x[i - 1] + x[i - 2]
> x

 [1]  0  1  1  2  3  5  8 13 21 34
```

Dans le code informatique du chapitre 5, nous composerons des fonctions avec ces deux solutions et nous comparerons les temps de calcul pour n grand.

Solution du problème

On résoud le problème par la méthode du point fixe en procédant de la même façon qu'à la section 4.4. On prend toutefois soin de modifier la deuxième ligne à l'intérieur de la boucle `repeat` :

```
> p <- 0.75
> alpha <- 20
> lambda <- 75
> d <- 2
> repeat
+ {
+     dt <- d
+     d <- log(lambda * (1 - p^(1/alpha)) + d) -
+         log(p)/alpha
+     if (abs(d - dt)/dt < 1E-10)
+         break
+ }
> d

[1] 0.4012012
```

Vérifions que l'équation de départ (4.1) est vraie :

Solution du problème (suite)

```
> ((lambda + d) / (lambda + exp(d)))^alpha

[1] 0.75
```

Le package **actuar** fournit une fonction `ppareto` pour calculer la fonction de répartition de la loi de Pareto. (Consulter l'annexe D pour installer le package à partir du site CRAN.) La fonction permet de vérifier directement l'équation de départ :

```
> library(actuar)
> ppareto(exp(d), alpha, lambda, lower.tail = FALSE) /
+     ppareto(d, alpha, lambda, lower.tail = FALSE)

[1] 0.75
```

4.6 Exercices

Dans chacun des exercices ci-dessous, écrire une expression R pour faire le calcul demandé. Parce qu'elles ne sont pas nécessaires, il est interdit d'utiliser des boucles.

4.1 Calculer la valeur actuelle d'une série de paiements fournie dans un vecteur `P` en utilisant les taux d'intérêt annuels d'un vecteur `i`.

4.2 Étant donné un vecteur d'observations $\mathbf{x} = (x_1, \ldots, x_n)$ et un vecteur de poids correspondants $\mathbf{w} = (w_1, \ldots, w_n)$, calculer la moyenne pondérée des observations,

$$\sum_{i=1}^{n} \frac{w_i}{w_\Sigma} x_i,$$

où $w_\Sigma = \sum_{i=1}^{n} w_i$. Tester l'expression avec les vecteurs de données

$$\mathbf{x} = (7, 13, 3, 8, 12, 12, 20, 11)$$

et

$$\mathbf{w} = (0{,}15,\ 0{,}04,\ 0{,}05,\ 0{,}06,\ 0{,}17,\ 0{,}16,\ 0{,}11,\ 0{,}09).$$

4.3 Soit un vecteur d'observations $\mathbf{x} = (x_1, \ldots, x_n)$. Calculer la moyenne harmonique de ce vecteur, définie comme

$$\frac{n}{\frac{1}{x_1} + \cdots + \frac{1}{x_n}}.$$

Tester l'expression avec les valeurs de l'exercice 4.2.

4.4 Calculer la fonction de répartition en $x = 5$ d'une loi de Poisson avec paramètre $\lambda = 2$, qui est donnée par

$$\sum_{k=0}^{5} \frac{2^k e^{-2}}{k!},$$

où $k! = 1 \cdot 2 \cdots k$.

4.5 a) Calculer l'espérance d'une variable aléatoire X dont le support est $x = 1, 10, 100, \ldots, 1\,000\,000$ et les probabilités correspondantes sont $\frac{1}{28}, \frac{2}{28}, \ldots, \frac{7}{28}$, dans l'ordre.

b) Calculer la variance de la variable aléatoire X définie en a).

4.6 Calculer le taux d'intérêt nominal composé quatre fois par année, $i^{(4)}$, équivalent à un taux de $i = 6\,\%$ effectif annuellement.

4.7 La valeur actuelle d'une série de n paiements de fin d'année à un taux d'intérêt i effectif annuellement est

$$a_{\overline{n}|} = v + v^2 + \cdots + v^n = \frac{1 - v^n}{i},$$

où $v = (1+i)^{-1}$. Calculer en une seule expression, toujours sans boucle, un tableau des valeurs actuelles de séries de $n = 1, 2, \ldots, 10$ paiements à chacun des taux d'intérêt effectifs annuellement $i = 0{,}05, 0{,}06, \ldots, 0{,}10$.

4.8 Calculer la valeur actuelle d'une annuité croissante de 1 \$ payable annuellement en début d'année pendant dix ans si le taux d'actualisation est de 6 %. Cette valeur actuelle est donnée par

$$I\ddot{a}_{\overline{10}|} = \sum_{k=1}^{10} k v^{k-1},$$

toujours avec $v = (1 + i)^{-1}$.

4.9 Calculer la valeur actuelle de la suite de paiements 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 si les paiements sont effectués en fin d'année et que le taux d'actualisation est de 7 %.

4.10 Calculer la valeur actuelle de la suite de paiements de l'exercice 4.9 en supposant que le taux d'intérêt d'actualisation alterne successivement entre 5 % et 8 % chaque année, c'est-à-dire que le taux d'intérêt est de 5 %, 8 %, 5 %, 8 %, etc.

5 Fonctions définies par l'usager

Objectifs du chapitre

- Définir une fonction R, ses divers arguments et, le cas échéant, les valeurs par défaut de ceux-ci.
- Déboguer une fonction R.
- Adopter un style de codage correspondant à la pratique reconnue en R.

La possibilité pour l'usager de définir facilement et rapidement de nouvelles fonctions — et donc des extensions au langage — est une des grandes forces de R. Les fonctions personnelles définies dans l'espace de travail ou dans un package sont traitées par le système exactement comme les fonctions internes.

Ce court chapitre passe en revue la syntaxe et les règles pour créer des fonctions dans R. On discute également brièvement de débogage et de style de codage.

Énoncé du problème

Stock Ticker est un jeu canadien datant de 1937 dans lequel on brasse une série de dés pour simuler les mouvements boursiers et les dividendes de six titres financiers.
Pour les fins de cet exercice, nous ne nous intéresserons qu'aux mouvements boursiers, et ce, pour un seul titre — nous ignorerons donc le versement des dividendes.
La valeur du titre est établie à partir de la valeur au tour de jeu précédent et du résultat du lancer de deux dés.

1. Un premier dé à trois faces détermine la direction du mouvement boursier : hausse, baisse ou nul.

Énoncé du problème (suite)

2. Un second dé à trois faces détermine l'amplitude de la hausse ou de la baisse, le cas échéant. Les valeurs possibles sont 5, 10 et 20.

La valeur de départ du titre est 100.
Par exemple, si les résultats des dés au premier tour de jeu sont « baisse » et 20, la valeur du titre après ce tour sera 80. Si, au tour suivant, les résultats des dés sont « nul » et « 5 », la valeur du titre demeurera de 80.
On fournit les résultats de 10 lancers des dés sous forme de *data frame* :

```
> x

   direction amplitude
1     baisse        20
2     baisse        10
3        nul        20
4        nul        20
5        nul        20
6     hausse        20
7        nul        20
8     hausse         5
9     baisse        10
10    hausse        20
```

Écrire une fonction `valeurs()` servant à calculer les valeurs successives du titre.
La fonction prend en arguments `x`, un *data frame* contenant les résultats des lancers des dés, ainsi que `start`, la valeur de départ du titre (100 par défaut). La fonction retourne un vecteur contenant les valeurs successives du titre.
Par exemple, avec les données ci-dessus, on obtiendrait :

```
> valeurs(x)              # start = 100 par défaut

 [1]  80  70  70  70  70  90  90  95  85 105
```

Astuce

Le calcul de la valeur du titre étant récursif, l'utilisation d'une boucle est ici inévitable. Comme le nombre de répétition est connu d'avance (le nombre de valeurs à calculer correspond au nombre de lancers de dés), une boucle `for()` serait le choix approprié.

5.1 Définition d'une fonction

On définit une nouvelle fonction avec la syntaxe suivante :

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

où

- `fun` est le nom de la fonction (les règles pour les noms de fonctions étant les mêmes que celles présentées à la section 2.2 pour tout autre objet) ;
- *`arguments`* est la liste des arguments, séparés par des virgules ;
- *`expression`* constitue le corps de la fonction, soit une expression ou un groupe d'expressions réunies par des accolades.

5.2 Retourner des résultats

La plupart des fonctions sont écrites dans le but de retourner un résultat. Or, les règles d'interprétation d'un groupe d'expressions présentées à la section 2.1 s'appliquent ici au corps de la fonction.

- Une fonction retourne tout simplement le résultat de la *dernière expression* du corps de la fonction.
- On évitera donc que la dernière expression soit une affectation, car la fonction ne retournera alors rien et on ne pourra utiliser une construction de la forme `x <- f()` pour affecter le résultat de la fonction à une variable.
- Si on doit retourner un résultat sans être à la dernière ligne de la fonction (à l'intérieur d'un bloc conditionnel, par exemple), on utilise la fonction `return`. *L'utilisation de* `return` *à la toute fin d'une fonction est tout à fait inutile et considérée comme du mauvais style en R.*
- Lorsqu'une fonction doit retourner plusieurs résultats, il est en général préférable d'avoir recours à une liste nommée.

5.3 Variables locales et globales

Comme la majorité des langages de programmation, R comporte des concepts de variable locale et de variable globale.

- 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 ;
  - n'écrase pas une variable du même nom dans l'espace de travail.
- Il est possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur d'affectation `<<-`. Il est très rare — et généralement non recommandé — de devoir recourir à de telles variables globales.
- On peut définir une fonction à l'intérieur d'une autre fonction. Cette fonction sera locale à la fonction dans laquelle elle est définie.

Le lecteur intéressé à en savoir plus pourra consulter les sections de la documentation de R portant sur la portée lexicale (*lexical scoping*). C'est un sujet important et intéressant, mais malheureusement trop avancé pour ce document d'introduction à la programmation en R.

5.4 Exemple de fonction

Le code développé pour l'exemple de point fixe de la section 4.4 peut être intégré dans une fonction ; voir la figure 5.1.

- Le nom de la fonction est `fp`.
- La fonction compte quatre arguments : `k`, `n`, `start` et `TOL`.
- Les deux derniers arguments ont respectivement des valeurs par défaut de 0,05 et 10^{-10}.
- La fonction retourne la valeur de la variable `i` puisque l'on évalue celle-ci à la dernière ligne (ou expression) de la fonction.

Astuce

La définition de la fonction devra inclure `start = 100` comme deuxième argument afin de spécifier que :

i) `start` est un argument de la fonction ;

ii) la valeur par défaut de l'argument est 100.

```
fp <- function(k, n, start = 0.05, TOL = 1E-10)
{
    ## Fonction pour trouver par la méthode du point
    ## fixe le taux d'intérêt pour lequel une série de
    ## 'n' paiements vaut 'k'.
    ##
    ## ARGUMENTS
    ##
    ##     k: la valeur présente des paiements
    ##     n: le nombre de paiements
    ## start: point de départ des itérations
    ##   TOL: niveau de précision souhaité
    ##
    ## RETOURNE
    ##
    ## Le taux d'intérêt

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

FIG. 5.1 – Exemple de fonction de point fixe

5.5 Fonctions anonymes

Il est parfois utile de définir une fonction sans lui attribuer un nom — d'où la notion de *fonction anonyme*. Il s'agira en général de fonctions courtes utilisées dans une autre fonction. Par exemple, pour calculer la valeur de xy^2 pour toutes les combinaisons de x et y stockées dans des vecteurs du même

nom, on pourrait utiliser la fonction `outer` ainsi :

```
> 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
```

Cependant, si la fonction `f` ne sert à rien ultérieurement, on peut se contenter de passer l'objet fonction à `outer` sans jamais lui attribuer un nom :

```
> outer(x, y, function(x, y) x * y^2)

     [,1] [,2] [,3]
[1,]   16   25   36
[2,]   32   50   72
[3,]   48   75  108
```

On a alors utilisé dans `outer` une fonction anonyme.

5.6 Débogage de fonctions

Il est assez rare d'arriver à écrire un bout de code sans bogue du premier coup. Par conséquent, qui dit programmation dit séances de débogage.

Les techniques de débogages les plus simples et naïves sont parfois les plus efficaces et certainement les plus faciles à apprendre. Loin d'un traité sur le débogage de code R, nous offrons seulement ici quelques trucs que nous utilisons régulièrement.

- Les erreurs de syntaxe sont les plus fréquentes (en particulier l'oubli de virgules). Lors de la définition d'une fonction, une vérification de la syntaxe est effectuée par l'interprète R. Attention, cependant : une erreur peut prendre sa source plusieurs lignes avant celle que l'interprète pointe comme causant problème.
- Les messages d'erreur de l'interprète ne sont pas toujours d'un grand secours… tant que l'on n'a pas appris à les reconnaître. Un exemple de message d'erreur fréquemment rencontré :

  ```
  valeur manquante là où TRUE / FALSE est requis
  ```

Cette erreur provient généralement d'une commande `if` dont l'argument vaut `NA` plutôt que `TRUE` ou `FALSE`. La raison : des valeurs manquantes se sont faufilées dans les calculs à notre insu jusqu'à l'instruction `if`, faisant en sorte que l'argument de `if` vaut `NA` alors qu'il ne peut être que booléen.

- Lorsqu'une fonction ne retourne pas le résultat attendu, placer des commandes `print` à l'intérieur de la fonction, de façon à pouvoir suivre les valeurs prises par les différentes variables.

  Par exemple, la modification suivante à la boucle de la fonction `fp` permet d'afficher les valeurs successives de la variable `i` et de détecter, par exemple, une procédure itérative divergente :

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

- Quand ce qui précède ne fonctionne pas, ne reste souvent qu'à exécuter manuellement la fonction. Pour ce faire, définir dans l'espace de travail tous les arguments de la fonction, puis exécuter le corps de la fonction ligne par ligne. La vérification du résultat de chaque ligne permet généralement de retrouver la ou les expressions qui causent problème.

5.7 Styles de codage

Si tous conviennent que l'adoption d'un style propre et uniforme favorise le développement et la lecture de code, il existe plusieurs chapelles dans le monde des programmeurs quant à la « bonne façon » de présenter et, surtout, d'indenter le code informatique.

Par exemple, Emacs reconnaît et supporte les styles de codage suivants, entre autres :

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
  }
```

- Pour des raisons générales de lisibilité et de popularité, le style C++, avec les accolades sur leurs propres lignes et une indentation de quatre (4) espaces est considéré comme standard pour la programmation en R.
- Consulter la documentation de votre éditeur de texte pour savoir s'il est possible de configurer le niveau d'indentation. La plupart des bons éditeurs pour programmeurs le permettent.
- Surtout, éviter de ne pas du tout indenter le code.

Solution du problème

Il existe bien évidemment une multitude de solutions valides. Celle que nous proposons à la figure 5.2 repose sur deux idées principales :

1. le vecteur créé pour accueillir les résultats contient la valeur de départ en première position afin d'éviter de traiter la première boucle comme une exception ; cette valeur de départ est supprimée du vecteur au moment de retourner les résultats ;
2. les étiquettes de mouvement du titre sont rapidement converties en valeur numériques qui permettent de calculer les valeurs successives du titre.

Ensemble, ces deux stratégies permettent d'en arriver à une fonction compacte et efficace.
On remarquera également que la création d'un contenant pour les résultats permet d'éviter le Syndrôme de la plaque à biscuits™.

```
valeurs <- function(x, start = 100)
{
    ## Création d'un vecteur pour les résultats. La valeur
    ## de départ est placée au début du vecteur pour faire
    ## la boucle. Elle sera supprimée à la fin.
    res <- c(start, numeric(nrow(x)))

    ## Conversion des étiquettes ("hausse", "nul",
    ## "baisse") de la première colonne des données en
    ## valeurs numériques (1, 0, -1).
    d <- (x[, 1] == "hausse") - (x[, 1] == "baisse")

    ## Calcul des valeurs successives du titre.
    for(i in seq(length(res) - 1))
        res[i + 1] <- res[i] + d[i] * x[i, 2]

    ## Résultats sans la valeur de départ
    res[-1]
}
```

FIG. 5.2 – Fonction `valeurs` solution du problème du chapitre

5.8 Exemples

```
### POINT FIXE

## Comme premier exemple de fonction, on réalise une mise en
## oeuvre de l'algorithme du point fixe pour trouver le taux
## d'intérêt tel que a_angle{n} = k pour 'n' et 'k' donnés.
## Cette mise en oeuvre est peu générale puisqu'il faudrait
## modifier la fonction chaque fois que l'on change la
## fonction f(x) dont on cherche le point fixe.
fp1 <- 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
}

fp1(7.2, 10)              # valeur de départ par défaut
fp1(7.2, 10, 0.06)        # valeur de départ spécifiée
i                         # les variables n'existent pas...
start                     # ... dans l'espace de travail

## Généralisation de la fonction 'fp1': la fonction f(x) dont
## on cherche le point fixe (c'est-à-dire la valeur de 'x'
## tel que f(x) = x) est passée en argument. On peut faire
## ça? Bien sûr, puisqu'une fonction est un objet comme un
## autre en R. On ajoute également à la fonction un argument
## 'echo' qui, lorsque TRUE, fera en sorte d'afficher à
## l'écran les valeurs successives de 'x'.
##
## Ci-dessous, il est implicite que le premier argument, FUN,
## est une fonction.
fp2 <- function(FUN, start, echo = FALSE, TOL = 1E-10)
{
    x <- start
    repeat
    {
        xt <- x

        if (echo)       # inutile de faire 'if (echo == TRUE)'
            print(xt)

        x <- FUN(xt)    # appel de la fonction

        if (abs(x - xt)/xt < TOL)
            break
    }
    x
}

f <- function(i) (1 - (1+i)^(-10))/7.2 # définition de f(x)
fp2(f, 0.05)              # solution
fp2(f, 0.05, echo = TRUE)  # avec résultats intermédiaires
fp2(function(x) 3^(-x), start = 0.5) # avec fonction anonyme
```

```
## Amélioration mineure à la fonction 'fp2': puisque la
## valeur de 'echo' ne change pas pendant l'exécution de la
## fonction, on peut éviter de refaire le test à chaque
## itération de la boucle. Une solution élégante consiste à
## utiliser un outil avancé du langage R: les expressions.
##
## L'objet créé par la fonction 'expression' est une
## expression non encore évaluée (comme si on n'avait pas
## appuyé sur Entrée à la fin de la ligne). On peut ensuite
## évaluer l'expression (appuyer sur Entrée) avec 'exec'.
fp3 <- function(FUN, start, echo = FALSE, TOL = 1E-10)
{
    x <- start

    ## Choisir l'expression à exécuter plus loin
    if (echo)
        expr <- expression(print(xt <- x))
    else
        expr <- expression(xt <- x)

    repeat
    {
        eval(expr)          # évaluer l'expression

        x <- FUN(xt)        # appel de la fonction

        if (abs(x - xt)/xt < TOL)
            break
    }
    x
}

fp3(f, 0.05, echo = TRUE)  # avec résultats intermédiaires
fp3(function(x) 3^(-x), start = 0.5) # avec une fonction anonyme

### SUITE DE FIBONACCI

## On a présenté au chapitre 4 deux manières différentes de
## pour calculer les 'n' premières valeurs de la suite de
## Fibonacci. On crée d'abord des fonctions à partir de ce
## code. Avantage d'avoir des fonctions: elles sont valides
## pour tout 'n' > 2.
##
## D'abord la version inefficace parce qu'elle souffre du
## Syndrôme de la plaque à biscuits décrit au chapitre 4.
```

```
fib1 <- function(n)
{
    res <- c(0, 1)
    for (i in 3:n)
        res[i] <- res[i - 1] + res[i - 2]
    res
}
fib1(10)
fib1(20)

## Puis la version qui devrait s'avérer plus efficace parce
## que l'on initialise d'entrée de jeu un contenant de la
## bonne longueur qu'on remplit par la suite.
fib2 <- function(n)
{
    res <- numeric(n)      # contenant créé
    res[2] <- 1            # res[1] vaut déjà 0
    for (i in 3:n)
        res[i] <- res[i - 1] + res[i - 2]
    res
}
fib2(5)
fib2(20)

## A-t-on vraiment gagné en efficacité? Comparons le temps
## requis pour générer une longue suite de Fibonacci avec les
## deux fonctions.
system.time(fib1(10000))   # version inefficace
system.time(fib2(10000))   # version efficace, ~5x plus rapide

## Variation sur un même thème: une fonction pour calculer non
## pas les 'n' premières valeurs de la suite de Fibonacci,
## mais uniquement la 'n'ième valeur.
##
## Mais il y a un mais: la fonction 'fib3' est truffée
## d'erreurs (de syntaxe, d'algorithmique, de conception). À
## vous de trouver les bogues. (Afin de préserver cet
## exemple, copier le code erroné plus bas ou dans un autre
## fichier avant d'y faire les corrections.)
fib3 <- function(nb)
{
    x <- 0
    x1 _ 0
    x2 <- 1
    while (n > 0))
```

```
x <- x1 + x2
x2 <- x1
x1 <- x
n <- n - 1
}
fib3(1)                    # devrait donner 0
fib3(2)                    # devrait donner 1
fib3(5)                    # devrait donner 3
fib3(10)                   # devrait donner 34
fib3(20)                   # devrait donner 4181
```

5.9 Exercices

5.1 La fonctions `var` calcule l'estimateur sans biais de la variance d'une population à partir de l'échantillon donné en argument. Écrire une fonction `variance` qui calculera l'estimateur biaisé ou sans biais selon que l'argument `biased` sera `TRUE` ou `FALSE`, respectivement. Le comportement par défaut de `variance` devrait être le même que celui de `var`. L'estimateur sans biais de la variance à partir d'un échantillon $X_1, \ldots, X_n$ est

$$S_{n-1}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2,$$

alors que l'estimateur biaisé est

$$S_n^2 = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})^2,$$

où $\bar{X} = n^{-1}(X_1 + \cdots + X_n)$.

5.2 Écrire une fonction `matrix2` qui, contrairement à la fonction `matrix`, remplira par défaut la matrice par ligne. La fonction *ne doit pas* utiliser `matrix`. Les arguments de la fonction `matrix2` seront les mêmes que ceux de `matrix`, sauf que l'argument `byrow` sera remplacé par `bycol`.

5.3 Écrire une fonction `phi` servant à calculer la fonction de densité de probabilité d'une loi normale centrée réduite, soit

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \quad -\infty < x < \infty.$$

La fonction devrait prendre en argument un vecteur de valeurs de x. Comparer les résultats avec ceux de la fonction `dnorm`.

5.4 Écrire une fonction `Phi` servant à calculer la fonction de répartition d'une loi normale centrée réduite, soit

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-y^2/2}\, dy, \quad -\infty < x < \infty.$$

Supposer, pour le moment, que $x \geq 0$. L'évaluation numérique de l'intégrale ci-dessus peut se faire avec l'identité

$$\Phi(x) = \frac{1}{2} + \phi(x) \sum_{n=0}^{\infty} \frac{x^{2n+1}}{1 \cdot 3 \cdot 5 \cdots (2n+1)}, \quad x \geq 0.$$

Utiliser la fonction `phi` de l'exercice 5.3 et tronquer la somme infinie à une grande valeur, 50 par exemple. La fonction ne doit pas utiliser de boucles, mais peut ne prendre qu'une seule valeur de x à la fois. Comparer les résultats avec ceux de la fonction `pnorm`.

5.5 Modifier la fonction `Phi` de l'exercice 5.4 afin qu'elle admette des valeurs de x négatives. Lorsque $x < 0$, $\Phi(x) = 1 - \Phi(-x)$. La solution simple consiste à utiliser une structure de contrôle `if ... else`, mais les curieux chercheront à s'en passer.

5.6 Généraliser maintenant la fonction de l'exercice 5.5 pour qu'elle prenne en argument un vecteur de valeurs de x. Ne pas utiliser de boucle. Comparer les résultats avec ceux de la fonction `pnorm`.

5.7 Sans utiliser l'opérateur `%*%`, écrire une fonction `prod.mat` qui effectuera le produit matriciel de deux matrices seulement si les dimensions de celles-ci le permettent. Cette fonction aura deux arguments (`mat1` et `mat2`) et devra tout d'abord vérifier si le produit matriciel est possible. Si celui-ci est impossible, la fonction retourne un message d'erreur.

a) Utiliser une structure de contrôle `if ... else` et deux boucles.

b) Utiliser une structure de contrôle `if ... else` et une seule boucle.

Dans chaque cas, comparer le résultat avec l'opérateur `%*%`.

5.8 Vous devez calculer la note finale d'un groupe d'étudiants à partir de deux informations : 1) une matrice contenant la note sur 100 des étudiants à chacune des évaluations, et 2) un vecteur contenant la pondération des évaluations. Un de vos collègues a composé la fonction `notes.finales` ci-dessous afin de faire le calcul de la note finale pour chacun de ses étudiants. Votre collègue vous mentionne toutefois que sa fonction est plutôt lente et inefficace pour de grands groupes d'étudiants. Modifiez la fonction afin d'en réduire le nombre d'opérations et faire en sorte qu'elle n'utilise aucune boucle.

```
notes.finales <- function(notes, p)
{
    netud <- nrow(notes)
    neval <- ncol(notes)
    final <- (1:netud) * 0
    for(i in 1:netud)
    {
        for(j in 1:neval)
        {
            final[i] <- final[i] + notes[i, j] * p[j]
        }
    }
    final
}
```

5.9 Trouver les erreurs qui empêchent la définition de la fonction ci-dessous.

```
AnnuiteFinPeriode <- function(n, i)
{{
    v <- 1/1 + i)
    ValPresChaquePmt <- v^(1:n)
    sum(ValPresChaquepmt)
}
```

5.10 La fonction ci-dessous calcule la valeur des paramètres d'une loi normale, gamma ou Pareto à partir de la moyenne et de la variance, qui sont connues par l'utilisateur.

```
param <- function(moyenne, variance, loi)
{
    loi <- tolower(loi)
    if (loi == "normale")
        param1 <- moyenne
        param2 <- sqrt(variance)
        return(list(mean = param1, sd = param2))
    if (loi == "gamma")
        param2 <- moyenne/variance
        param1 <- moyenne * param2
        return(list(shape = param1, scale = param2))
    if (loi == "pareto")
```

```
        cte <- variance/moyenne^2
        param1 <- 2 * cte/(cte-1)
        param2 <- moyenne * (param1 - 1)
        return(list(alpha = param1, lambda = param2))
    stop("La loi doit etre une de \"normale\",
\"gamma\" ou \"pareto\"")
}
```

L'utilisation de la fonction pour diverses lois donne les résultats suivants :

```
> param(2, 4, "normale")

$mean
[1] 2

$sd
[1] 2
```

```
> param(50, 7500, "gamma")

Erreur dans param(50, 7500, "gamma") : Objet "param1"
introuvable
```

```
> param(50, 7500, "pareto")

Erreur dans param(50, 7500, "pareto") : Objet "param1"
introuvable
```

a) Expliquer pour quelle raison la fonction se comporte ainsi.

b) Appliquer les correctifs nécessaires à la fonction pour que celle-ci puisse calculer les bonnes valeurs. (Les erreurs ne se trouvent pas dans les mathématiques de la fonction.) *Astuce* : tirer profit du moteur d'indentation de votre éditeur de texte pour programmeur.

6 Concepts avancés

Objectifs du chapitre

- Passer des valeurs à une fonction via l'argument '`...`'.
- Effectuer des sommaires sur des tableaux à l'aide de la fonction `apply`.
- Réduire des listes avec les fonctions `lapply`, `sapply` et `mapply`; comparer l'effet de ces fonctions.
- Concevoir comment la classe d'un objet peut modifier le traitement qu'en feront les fonctions génériques.

Ce chapitre traite de divers concepts et fonctions un peu plus avancés du langage R, dont les fonctions de la famille `apply` auxquelles nous avons fait référence à quelques reprises dans les chapitres précédents. Ce sont des fonctions d'une grande importance en R.

Énoncé du problème

Soit la fonction

$$g(x) = \begin{cases} 2x, & x \leq 5 \\ 0, & \text{ailleurs.} \end{cases}$$

On essaie d'en faire une mise en œuvre en R :

```
> g <- function(x)
+ {
+     if (x <= 5)
+         2 * x
+     else
+         0
+ }
```

Énoncé du problème (suite)

On évalue $g(1)$. On devrait obtenir $2 \times 1 = 2$ (cas $x \leq 5$).

```
> g(1)

[1] 2
```

On évalue $g(10)$. On devrait obtenir 0 (cas $x > 5$).

```
> g(10)

[1] 0
```

On désire maintenant évaluer la fonction à tous les entiers de 1 à 10, inclusivement. On devrait obtenir $(2, 4, 6, 8, 10, 0, 0, 0, 0, 0)$.

```
> g(1:10)

 [1]  2  4  6  8 10 12 14 16 18 20
Warning message:
In if (x <= 5) 2 * x else 0 :
  la condition a une longueur > 1 et seul le premier
  élément est utilisé
```

Que se passe-t-il ? Les résultats sont erronés et le message d'avertissement nous indique que seulement la première valeur a été utilisée pour évaluer la condition. La fonction `g` évalue donc correctement son argument seulement lorsqu'il s'agit d'un vecteur de longueur 1. En d'autres termes, la fonction n'est pas *vectorielle*.

6.1 Argument '. . .'

La mention '`...`' apparaît dans la définition de plusieurs fonctions en R. Il ne faut pas voir là de la paresse de la part des rédacteurs des rubriques d'aide, mais bel et bien un argument formel dont '`...`' est le nom.

- Cet argument signifie qu'une fonction peut accepter un ou plusieurs arguments autres que ceux faisant partie de sa définition.

- Le contenu de l'argument '...' n'est ni pris en compte, ni modifié par la fonction. Il est généralement simplement passé tel quel à une autre fonction qui, elle, saura traiter les arguments qui lui sont ainsi passés.
- Pour des exemples, voir les définitions des fonctions `apply`, `lapply` et `sapply`, ci-dessous.

6.2 Fonction apply

La fonction `apply` sert à appliquer une fonction quelconque sur une partie d'une matrice ou, plus généralement, d'un tableau. La syntaxe de la fonction est la suivante :

```
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`.

Lorsque `X` est une matrice, `apply` sert principalement à calculer des sommaires par ligne (dimension 1) ou par colonne (dimension 2) autres que la somme ou la moyenne (puisque les fonctions `rowSums`, `colSums`, `rowMeans` et `colMeans` existent pour ce faire).

- Utiliser la fonction `apply` plutôt que des boucles puisque celle-ci est plus efficace.
- Considérer les exemples suivants :

```
> (x <- matrix(sample(1:100, 20, rep = TRUE), 5, 4))

     [,1] [,2] [,3] [,4]
[1,]   27   90   21   50
[2,]   38   95   18   72
[3,]   58   67   69  100
[4,]   91   63   39   39
[5,]   21    7   77   78

> apply(x, 1, var)              # variance par ligne
```

```
[1]  978.000 1181.583  335.000  612.000 1376.917

> apply(x, 2, min)              # minimum par colonne

[1] 21  7 18 39

> apply(x, 1, mean, trim = 0.2) # moy. tronquée par ligne

[1] 47.00 55.75 73.50 58.00 45.75
```

Puisqu'il n'existe pas de fonctions internes pour effectuer des sommaires sur des tableaux, il faut toujours utiliser la fonction `apply`.

- Si `X` est un tableau de plus de deux dimensions, alors l'argument passé à `FUN` peut être une matrice ou un tableau.
- Lorsque `X` est un tableau à trois dimensions et que `MARGIN` est de longueur 1, cela équivaut à appliquer la fonction `FUN` sur des « tranches » (des matrices) de `X`. Si `MARGIN` est de longueur 2, on applique `FUN` sur des « carottes » (des vecteurs) tirées de `X`.
- Truc mnémotechnique : la ou les dimensions absentes de `MARGIN` sont celles qui disparaissent après le passage de `apply`.
- Considérer les exemples suivants :

```
> (x <- array(sample(1:10, 80, rep = TRUE), c(3, 3, 4)))

, , 1

     [,1] [,2] [,3]
[1,]   10    2    1
[2,]    3    3    4
[3,]    7    4    9

, , 2

     [,1] [,2] [,3]
[1,]    4    5    7
[2,]    5    2    8
[3,]    6    9    2

, , 3
```

```
     [,1] [,2] [,3]
[1,]    8    7    6
[2,]    5    8    8
[3,]    9    6    1

, , 4

     [,1] [,2] [,3]
[1,]    5    5    3
[2,]    8    9    1
[3,]    7    5    1

> apply(x, 3, det)       # déterminants matrices 3 x 3

[1]  103  149 -103  -54

> apply(x, 1, sum)       # sommes tranches horizontales

[1] 63 64 66

> apply(x, c(1, 2), sum) # sommes carottes horizontales

     [,1] [,2] [,3]
[1,]   27   19   17
[2,]   21   22   21
[3,]   29   24   13

> apply(x, c(1, 3), sum) # sommes carottes transversales

     [,1] [,2] [,3] [,4]
[1,]   13   16   21   13
[2,]   10   15   21   18
[3,]   20   17   16   13

> apply(x, c(2, 3), sum) # sommes carottes verticales

     [,1] [,2] [,3] [,4]
[1,]   20   15   22   20
[2,]    9   16   21   19
[3,]   14   17   15    5
```

6.3 Fonctions `lapply` et `sapply`

Les fonctions `lapply` et `sapply` sont similaires à la fonction `apply` en ce qu'elles permettent d'appliquer une fonction aux éléments d'une structure — le vecteur ou la liste en l'occurrence. Leur syntaxe est similaire :

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

- La fonction `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. Le résultat est donc :

```
list(FUN(X[[1]], ...),
     FUN(X[[2]], ...),
     FUN(X[[3]], ...),
     ...)
```

- Les éléments de `X` sont passés comme à la fonction `FUN` sans être nommés. Les règles habituelles d'évaluation d'un appel de fonction s'appliquent. Par conséquent, les éléments de `X` seront considérés comme les premiers arguments de `FUN` à moins que des arguments nommés dans '`...`' aient préséance.
- Par exemple, on crée une liste formée de quatre vecteurs aléatoires de taille 5, 6, 7 et 8 :

```
> (x <- lapply(5:8, sample, x = 1:10))

[[1]]
[1]  7  4  3 10  9

[[2]]
[1] 3 2 4 7 8 5

[[3]]
[1]  8  4  9  2  1 10  6

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

Le premier argument de la fonction `sample` est `x`. Dans l'expression ci-dessus, cet argument est passé à la fonction via l'argument '`...`' de `lapply`. Par conséquent, les valeurs successives de `5:8` servent comme deuxième argument à la fonction `sample`, soit la taille de l'échantillon.

- On peut ensuite calculer la moyenne de chacun des vecteurs obtenus ci-dessus, toujours sans faire de boucle :

```
> lapply(x, mean)

[[1]]
[1] 6.6

[[2]]
[1] 4.833333

[[3]]
[1] 5.714286

[[4]]
[1] 4.5
```

La fonction sapply est similaire à lapply, sauf que le résultat est retourné sous forme de vecteur, si possible. Le résultat est donc *simplifié* par rapport à celui de lapply, d'où le nom de la fonction.

- Dans l'exemple ci-dessus, il est souvent plus utile d'obtenir les résultats sous la forme d'un vecteur :

```
> sapply(x, mean)

[1] 6.600000 4.833333 5.714286 4.500000
```

- Si le résultat de chaque application de la fonction est un vecteur et que les vecteurs sont tous de la même longueur, alors sapply retourne une matrice, remplie comme toujours par colonne :

```
> (x <- lapply(rep(5, 3), sample, x = 1:10))

[[1]]
[1] 6 2 9 5 4

[[2]]
[1]  1 10  6  7  4

[[3]]
[1] 6 5 8 4 9

> sapply(x, sort)
```

```
     [,1] [,2] [,3]
[1,]    2    1    4
[2,]    4    4    5
[3,]    5    6    6
[4,]    6    7    8
[5,]    9   10    9
```

Dans un grand nombre de cas, il est possible de remplacer les boucles `for` par l'utilisation de `lapply` ou `sapply`. On ne saurait donc trop insister sur l'importance de ces fonctions.

Astuce

La fonction `sapply` permet de vectoriser une fonction R qui n'est pas vectorielle. On peut procéder de deux façons : en utilisant la fonction non vectorielle dans une application de `sapply()` ou en vectorisant la fonction à y ajoutant un appel à `sapply`.

6.4 Fonction `mapply`

La fonction `mapply` est une version multidimensionnelle de `sapply`. Sa syntaxe est, essentiellement,

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

- Le résultat de `mapply` est l'application de la fonction `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 sous forme de liste, de vecteur ou de matrice, selon le cas, `FUN(v[1], w[1])`, `FUN(v[2], w[2])`, etc.
- Par 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

[[6]]
[1] 6 7 8
```

6.5 Fonction `replicate`

La fonction `replicate` est une fonction enveloppante de `sapply` simplifiant la syntaxe pour l'exécution répétée d'une expression.

- Son usage est particulièrement indiqué pour les simulations. Ainsi, on peut construire une fonction `fun` qui fait tous les calculs d'une simulation, puis obtenir les résultats pour, disons, 10 000 simulations avec

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

- L'annexe C présente en détail différentes stratégies — dont l'utilisation de la fonction `replicate` — pour la réalisation d'études de simulation en R.

6.6 Classes et fonctions génériques

Dans le langage R, tous les objets 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, dites *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 et, généralement, une méthode `default` pour les autres objets. La liste des méthodes existant pour une fonction générique s'obtient avec la fonction `methods` :

```
> methods(plot)

 [1] plot.acf*           plot.data.frame*
 [3] plot.decomposed.ts* plot.default
 [5] plot.dendrogram*    plot.density*
 [7] plot.ecdf           plot.factor*
 [9] plot.formula*       plot.function
[11] plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*
[15] plot.lm*            plot.medpolish*
[17] plot.mlm*           plot.ppr*
[19] plot.prcomp*        plot.princomp*
[21] plot.profile.nls*   plot.raster*
[23] plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*
[27] plot.ts             plot.tskernel*
[29] plot.TukeyHSD*
see '?methods' for accessing help and source code
```

- À chaque méthode `methode` d'une fonction générique `fun` correspond une fonction `fun.methode`. C'est donc la rubrique d'aide de cette dernière

fonction qu'il faut consulter au besoin, et non celle de la fonction générique, qui contient en général peu d'informations.

- Il est intéressant de savoir que 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 en créant une nouvelle classe et une nouvelle méthode pour la fonction `print`.

Solution du problème

Débutons par expliquer ce qui cloche avec notre fonction `g` de départ :

```
> g

function(x)
{
    if (x <= 5)
        2 * x
    else
        0
}
```

Il s'avère que la structure de contrôle `if` n'est *pas* une fonction vectorielle. Elle n'accepte qu'une seule valeur. De la rubrique d'aide :

```
cond: A length-one logical vector that is not 'NA'.
      Conditions of length greater than one are accepted
      with a warning, but only the first element is used.
```

Par conséquent, lorsque l'argument de la fonction `g` est `1:10`, l'exécution de la fonction est la suivante :

i) l'argument `x` vaut `1:10`;

ii) la condition `if (x <= 5)` n'est évaluée que pour la première valeur de `x`;

iii) puisque `x[1] <= 5`, c'est l'expression vectorielle `2 * x` qui est évaluée;

iv) le résultat de la fonction est `2 * x`.

Solution du problème (suite)

Tel qu'indiqué précédemment, la fonction `sapply` peut rendre vectorielle une fonction ou une expression qui ne le sont pas. Ici, la solution la plus simple à court terme serait :

```
> sapply(1:10, g)

 [1]  2  4  6  8 10  0  0  0  0  0
```

Une solution plus complète consisterait à transformer la fonction `g` pour la rendre vectorielle avec la fonction `sapply` :

```
> g2 <- function(x)
+ {
+     h <- function(x)
+     {
+         if (x <= 5)
+             2 * x
+         else
+             0
+     }
+     sapply(x, h)
+ }
> g2(1:10)

 [1]  2  4  6  8 10  0  0  0  0  0
```

Attention, toutefois, cette solution n'est pas une panacée.
Par exemple, dans ce cas bien précis où la fonction g est une fonction en branches, une autre solution fait appel à la fonction `ifelse` :

```
> g3 <- function(x)
+     ifelse (x <= 5, 2 * x, 0)
> g3(1:10)

 [1]  2  4  6  8 10  0  0  0  0  0
```

Pour le problème sous étude, on peut faire encore beaucoup mieux en tirant directement profit de l'approche vectorielle de R. Par une judicieuse

Solution du problème (suite)

utilisation des valeurs booléennes, il est possible d'éliminer complètement la condition `if` :

```
> g4 <- function(x)
+     2 * x * (x <= 5)
> g4(1:10)

 [1]  2  4  6  8 10  0  0  0  0  0
```

On préférera, dans l'ordre, les fonctions `g4`, `g3` (la fonction `ifelse` est lente, mais néanmoins plus rapide que `sapply`), `g2` et l'approche combinant `sapply` et la fonction `g` de départ.

6.7 Exemples

```
###
### FONCTION 'apply'
###

## Création d'une matrice et d'un tableau à trois dimensions
## pour les exemples.
m <- matrix(sample(1:100, 20), nrow = 4, ncol = 5)
a <- array(sample(1:100, 60), dim = 3:5)

## Les fonctions 'rowSums', 'colSums', 'rowMeans' et
## 'colMeans' sont des raccourcis pour des utilisations
## fréquentes de 'apply'.
rowSums(m)                  # somme par ligne
apply(m, 1, sum)            # idem, mais moins lisible
colMeans(m)                 # somme par colonne
apply(m, 2, mean)           # idem, mais moins lisible

## Puisqu'il n'existe pas de fonctions comme 'rowMax' ou
## 'colProds', il faut utiliser 'apply'.
apply(m, 1, max)            # maximum par ligne
apply(m, 2, prod)           # produit par colonne

## L'argument '...' de 'apply' permet de passer des arguments
## à la fonction FUN.
```

```
m[sample(1:20, 5)] <- NA   # ajout de données manquantes
apply(m, 1, var, na.rm = TRUE) # variance par ligne sans NA

## Lorsque 'apply' est utilisée sur un tableau, son résultat
## est de dimensions dim(X)[MARGIN], d'où le truc
## mnémotechnique donné dans le texte du chapitre.
apply(a, c(2, 3), sum)     # le résultat est une matrice
apply(a, 1, prod)          # le résultat est un vecteur

## L'utilisation de 'apply' avec les tableaux peut rapidement
## devenir confondante si l'on ne visualise pas les calculs
## qui sont réalisés. On reprend ici les exemples du chapitre
## en montrant comment l'on calculerait le premier élément de
## chaque utilisation de 'apply'. Au besoin, retourner à
## l'indiçage des tableaux au chapitre 2.
(x <- array(sample(1:10, 80, rep = TRUE), c(3, 3, 4)))
apply(x, 3, det)      # déterminants des quatre matrices 3 x 3
det(x[, , 1])         # équivalent pour le premier déterminant

apply(x, 1, sum)      # sommes des trois tranches horizontales
sum(x[1, , ])         # équivalent pour la première somme

apply(x, c(1, 2), sum) # sommes des neuf carottes horizontales
sum(x[1, 1, ])         # équivalent pour la première somme

apply(x, c(1, 3), sum) # sommes des 12 carottes transversales
sum(x[1, , 1])         # équivalent pour la première somme

apply(x, c(2, 3), sum) # sommes des 12 carottes verticales
sum(x[, 1, 1])         # équivalent pour la première somme

###
### FONCTIONS 'lapply' ET 'sapply'
###

## La fonction 'lapply' applique une fonction à tous les
## éléments d'un vecteur ou d'une liste et retourne une liste,
## peu importe les dimensions des résultats. La fonction
## 'sapply' retourne un vecteur ou une matrice, si possible.
##
## Somme «interne» des éléments d'une liste.
(x <- list(1:10, c(-2, 5, 6), matrix(3, 4, 5)))
sum(x)                    # erreur
lapply(x, sum)            # sommes internes (liste)
sapply(x, sum)            # sommes internes (vecteur)
```

```
## Création de la suite 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, ..., 1,
## 2, ..., 9, 10.
lapply(1:10, seq)          # le résultat est une liste
unlist(lapply(1:10, seq))  # le résultat est un vecteur

## Soit une fonction calculant la moyenne pondérée d'un
## vecteur. Cette fonction prend en argument une liste de deux
## éléments: 'donnees' et 'poids'.
fun <- function(x)
    sum(x$donnees * x$poids)/sum(x$poids)

## On peut maintenant calculer la moyenne pondérée de
## plusieurs ensembles de données réunis dans une liste
## itérée.
(x <- list(list(donnees = 1:7,
                poids = (5:11)/56),
           list(donnees = sample(1:100, 12),
                poids = 1:12),
           list(donnees = c(1, 4, 0, 2, 2),
                poids = c(12, 3, 17, 6, 2))))
sapply(x, fun)             # aucune boucle explicite!

###
### EXEMPLES ADDITIONNELS SUR L'UTILISATION DE L'ARGUMENT
### '...' AVEC 'lapply' ET 'sapply'
###

## Aux fins des exemples ci-dessous, on crée d'abord une liste
## formée de nombres aléatoires. Cette expression fait usage
## de l'argument '...' de 'lapply'. Pouvez-vous la décoder?
## Nous y reviendrons plus loin, ce qui compte pour le moment
## c'est simplement de l'exécuter.
x <- lapply(c(8, 12, 10, 9), sample, x = 1:10, replace = TRUE)

## Soit maintenant une fonction qui calcule la moyenne
## arithmétique des données d'un vecteur 'x' supérieures à une
## valeur 'y'. On remarquera que cette fonction n'est pas
## vectorielle pour 'y', c'est-à-dire qu'elle n'est valide que
## lorsque 'y' est un vecteur de longueur 1.
fun <- function(x, y) mean(x[x > y])

## Pour effectuer ce calcul sur chaque élément de la liste
## 'x', nous pouvons utiliser 'sapply' plutôt que 'lapply',
## car chaque résultat est de longueur 1. Cependant, il faut
```

```
## passer la valeur de 'y' à la fonction 'fun'. C'est là
## qu'entre en jeu l'argument '...' de 'sapply'.
sapply(x, fun, 7)          # moyennes des données > 7

## Les fonctions 'lapply' et 'sapply' passent tout à tour les
## éléments de leur premier argument comme *premier* argument
## à la fonction, sans le nommer explicitement. L'expression
## ci-dessus est donc équivalente à
##
##   c(fun(x[[1]], 7), ..., fun(x[[4]], 7)
##
## Que se passe-t-il si l'on souhaite passer les valeurs à un
## argument de la fonction autre que le premier? Par exemple,
## supposons que l'ordre des arguments de la fonction 'fun'
## ci-dessus est inversé.
fun <- function(y, x) mean(x[x > y])

## Les règles d'appariement des arguments des fonctions en R
## font en sorte que lorsque les arguments sont nommés dans
## l'appel de fonction, leur ordre n'a pas d'importance. Par
## conséquent, un appel de la forme
##
##   fun(x, y = 7)
##
## est tout à fait équivalent à fun(7, x). Pour effectuer les calculs
##
##   c(fun(x[[1]], y = 7), ..., fun(x[[4]], y = 7)
##
## avec la liste définie plus haut, il s'agit de nommer
## l'argument 'y' dans '...' de 'sapply'.
sapply(x, y = 7)

## Décodons maintenant l'expression
##
##   lapply(c(8, 12, 10, 9), sample, x = 1:10, replace = TRUE)
##
## qui a servi à créer la liste. La définition de la fonction
## 'sample' est la suivante:
##
##   sample(x, size, replace = FALSE, prob = NULL)
##
## L'appel à 'lapply' est équivalent à
##
##   list(sample(8, x = 1:10, replace = TRUE),
##        ...,
```

```
##        sample(9, x = 1:10, replace = TRUE))
##
## Toujours selon les règles d'appariement des arguments, on
## voit que les valeurs 8, 12, 10, 9 seront attribuées à
## l'argument 'size', soit la taille de l'échantillon.
## L'expression crée donc une liste comprenant quatre
## échantillons aléatoires de tailles différentes des nombres
## de 1 à 10 pigés avec remise.
##
## Une expression équivalente, quoique moins élégante, aurait
## recours à une fonction anonyme pour replacer les arguments
## de 'sample' dans l'ordre voulu.
lapply(c(8, 12, 10, 9),
       function(x) sample(1:10, x, replace = TRUE))

## La fonction 'sapply' est aussi très utile pour vectoriser
## une fonction qui n'est pas vectorielle. Supposons que l'on
## veut généraliser la fonction 'fun' pour qu'elle accepte un
## vecteur de seuils 'y'.
fun <- function(x, y)
    sapply(y, function(y) mean(x[x > y]))

## Utilisation sur la liste 'x' avec trois seuils.
sapply(x, fun, y = c(3, 5, 7))

###
### FONCTION 'mapply'
###

## Création de quatre échantillons aléatoires de taille 12.
x <- lapply(rep(12, 4), sample, x = 1:100)

## Moyennes tronquées à 0, 10, 20 et 30%, respectivement, de
## ces quatre échantillons aléatoires.
mapply(mean, x, 0:3/10)

###
### FONCTION 'replicate'
###

## La fonction 'replicate' va répéter un certain nombre de
## fois une expression quelconque. Le principal avantage de
## 'replicate' sur 'sapply' est qu'on n'a pas à se soucier des
## arguments à passer à une fonction.
##
```

```
## Par exemple, on veut simuler dix échantillons aléatoires
## indépendants de longueur 12. On peut utiliser 'sapply',
## mais la syntaxe n'est ni élégante, ni facile à lire
## (l'argument 'i' ne sert à rien).
sapply(rep(1, 10), function(i) sample(1:100, 12))

## En utilisant 'replicate', on sait tout de suite de quoi il
## s'agit. À noter que les échantillons se trouvent dans les
## colonnes de la matrice résultante.
replicate(10, sample(1:100, 12))

## Vérification que la moyenne arithmétique (bar{X}) est un
## estimateur sans biais de la moyenne de la loi normale. On
## doit calculer la moyenne de plusieurs échantillons
## aléatoires, puis la moyenne de toutes ces moyennes.
##
## On définit d'abord une fonction pour faire une simulation.
## Remarquer que dans la fonction ci-dessous, 'mean' est tour
## à tour le nom d'un argument (qui pourrait aussi bien être
## «toto») et la fonction pour calculer une moyenne.
fun <- function(n, mean, sd)
    mean(rnorm(n, mean = mean, sd = sd))

## Avec 'replicate', on fait un grand nombre de simulations.
x <- replicate(10000, fun(100, 0, 1)) # 10000 simulations
hist(x)                    # distribution de bar{X}
mean(x)                    # moyenne de bar{X}

###
### CLASSES ET FONCTIONS GÉNÉRIQUES
###

## Pour illustrer les classes et fonctions génériques, on
## reprend la fonction de point fixe 'fp3' des exemples du
## chapitre 5 en y faisant deux modifications:
##
##   1. ajout d'un compteur pour le nombre d'itérations;
##   2. la fonction retourne une liste de classe 'fp'
##      contenant diverses informations relatives à la
##      procédure de point fixe.
##
## Ainsi, la fonction 'fp4' retourne un objet qui peut ensuite
## être manipulé par des méthodes de fonctions génériques.
## C'est l'approche de programmation objet favorisée dans le
## langage R.
```

```
fp4 <- function(FUN, start, echo = FALSE, TOL = 1E-10)
{
    x <- start              # valeur de départ
    i <- 0                  # compteur des itérations

    if (echo)
        expr <- expression(print(xt <- x))
    else
        expr <- expression(xt <- x)

    repeat
    {
        eval(expr)

        x <- FUN(xt)        # nouvelle valeur
        i <- i + 1          # incrémenter le compteur

        if (abs(x - xt)/xt < TOL)
            break
    }

    structure(list(fixed.point = x, # point fixe
                   nb.iter = i,     # nombre d'itérations
                   fun = FUN,       # fonction f(x)
                   x0 = start,      # valeur de départ
                   TOL = TOL),      # précision relative
              class = "fp")
}

## On crée maintenant des méthodes pour la classe 'fp' pour
## les fonctions génériques les plus courantes, soit 'print',
## 'summary' et 'plot'.
##
## La méthode de 'print' sera utilisée pour afficher seulement
## la valeur du point fixe. C'est en quelque sorte
## l'utilisation la plus simple de la fonction 'fp4'.
##
## La méthode de 'summary' fournira un peu plus d'informations
## sur la procédure de point fixe.
##
## Enfin, la méthode de 'plot' fera un graphique de la
## fonction f(x) et son intersection avec la droite y = x.
print.fp <- function(x)
    print(x$fixed.point)
```

```
summary.fp <- function(x)
{
    if (class(x) != "fp")
        stop("object is not of class 'fp'")
    cat("Function:\n ")
    print(x$fun)
    cat("\n")
    cat("Fixed point:\n ", x$fixed.point, fill = TRUE)
    cat("\n")
    cat("Number of iterations:\n ", x$nb.iter, fill = TRUE)
    cat("\n")
    cat("Precision:\n ", x$TOL, fill = TRUE)
}

plot.fp <- function(x, ...)
{
    ## Valeur du point fixe
    fp <- x$fixed.point

    ## Il faut déterminer un intervalle pour lequel tracer la
    ## fonction. Celui-ci est déterminé de façon arbitraire
    ## comme un multiple de la distance entre la valeur de
    ## départ et la valeur du point fixe.
    r <- abs(x$x0 - fp)

    ## Fonction à tracer
    FUN <- x$fun

    ## Fonction y = x. 'FUN2' est nécessaire parce que 'curve'
    ## n'admet pas de fonctions anonymes en argument.
    FUN2 <- function(x) x

    ## Graphique de la fonction 'FUN'
    curve(FUN, from = fp - 3 * r, to = fp + 3 * r,
          xlab = "x", ylab = "f(x)", lwd = 2)

    ## Ajout de la droite 'FUN2' au graphique
    curve(FUN2, add = TRUE, lwd = 1)

    ## Ajout d'un point sur le point fixe
    points(fp, FUN(fp), ...)
}

## Exemples d'utilisation
x <- fp4(function(x) 3^(-x), start = 0.5)
```

```
x                           # affichage de 'print.fp'
summary(x)                  # plus d'information
plot(x)                     # graphique de base
plot(x, pch = 21,           # graphique plus élaboré...
     bg = "orange",         # ... consulter la rubrique
     cex = 2, lwd = 2)      # ... d'aide de 'par'

###
### OPÉRATEURS EN TANT QUE FONCTIONS
###

## Les opérateurs représentés par des caractères spéciaux sont
## des fonctions comme les autres. On peut donc les appeler
## comme toute autre fonction. (En fait, l'interprète R fait
## cette traduction à l'interne.)
x <- sample(1:100, 12)      # un vecteur
x + 2                       # appel usuel
"+"(x, 2)                   # équivalent
x[c(3, 5)]                  # extraction usuelle
"["(x, c(3, 5))             # équivalent
x[1] <- 0; x                # assignation usuelle
"[<-"(x, 2, 0)              # équivalent (à x[2] <- 0)

## D'une part, cela explique pourquoi il faut placer les
## opérateurs entre guillemets (" ") lorsqu'on les utilise
## dans les fonctions comme 'outer', 'lapply', etc.
outer(x, x, +)              # erreur de syntaxe
outer(x, x, "+")            # correct

## D'autre part, cela permet d'utiliser les opérateurs
## d'extraction "[" et "[[" dans de telles fonctions. Par
## exemple, voici comment extraire le deuxième élément de
## chaque élément d'une liste.
(x <- list(1:4, 8:2, 6:12, -2:2)) # liste quelconque
x[[1]][2]                   # 2e élément du 1er élément
x[[2]][2]                   # 2e élément du 2e élément
x[[3]][2]                   # 2e élément du 3e élément
x[[4]][2]                   # 2e élément du 4e élément
lapply(x, "[", 2)           # même chose en une ligne
sapply(x, "[", 2)           # résultat sous forme de vecteur

###
### COMMENT JOUER DES TOURS AVEC R
###
```

```
## Redéfinir un opérateur dans l'espace de travail de
## quelqu'un...
"+" <- function(x, y) x * y # redéfinition de "+"
5 + 2                       # ouch!
ls()                        # traîtrise dévoilée...
rm("+")                     # ... puis éliminée
5 + 2                       # c'est mieux

## Faire croire qu'une fonction fait autre chose que ce
## qu'elle fait en réalité. Si l'attribut "source" d'une
## fonction existe, c'est son contenu qui est affiché lorsque
## l'on examine une fonction.
f <- function(x, y) x + y  # vraie fonction
attr(f, "source") <- "function(x, y) x * y" # ce qui est affiché
f                           # une fonction pour faire le produit?
f(2, 3)                     # non!
str(f)                      # structure de l'objet
attr(f, "source") <- NULL   # attribut "source" effacé
f                           # c'est mieux

## Redéfinir la méthode de 'print' pour une classe d'objet...
## Ici, l'affichage d'un objet de classe "lm" cause la
## fermeture de R!
print.lm <- function(x) q("ask")

x <- rnorm(10)              # échantillon aléatoire
y <- x + 2 + rnorm(10)      # modèle de régression linéaire
lm(y ~ x)                   # répondre "c"!
```

6.8 Exercices

6.1 À l'exercice 4.2, on a calculé la moyenne pondérée d'un vecteur d'observations

$$X_w = \sum_{i=1}^{n} \frac{w_i}{w_\Sigma} X_i,$$

où $w_\Sigma = \sum_{i=1}^{n} w_i$. Si l'on a plutôt une matrice $n \times p$ d'observations X_{ij}, on peut définir les moyennes pondérées

$$X_{iw} = \sum_{j=1}^{p} \frac{w_{ij}}{w_{i\Sigma}} X_{ij}, \quad w_{i\Sigma} = \sum_{j=1}^{p} w_{ij}$$

$$X_{wj} = \sum_{i=1}^{n} \frac{w_{ij}}{w_{\Sigma j}} X_{ij}, \quad w_{\Sigma j} = \sum_{i=1}^{n} w_{ij}$$

et

$$X_{ww} = \sum_{i=1}^{n}\sum_{j=1}^{p}\frac{w_{ij}}{w_{\Sigma\Sigma}}X_{ij}, \quad w_{\Sigma\Sigma} = \sum_{i=1}^{n}\sum_{j=1}^{p} w_{ij}.$$

De même, on peut définir des moyennes pondérées calculées à partir d'un tableau de données X_{ijk} de dimensions $n \times p \times r$ dont la notation suit la même logique que ci-dessus. Écrire des expressions R pour calculer, sans boucle, les moyennes pondérées suivantes.

a) X_{iw} en supposant une matrice de données $n \times p$.

b) X_{wj} en supposant une matrice de données $n \times p$.

c) X_{ww} en supposant une matrice de données $n \times p$.

d) X_{ijw} en supposant un tableau de données $n \times p \times r$.

e) X_{iww} en supposant un tableau de données $n \times p \times r$.

f) X_{wjw} en supposant un tableau de données $n \times p \times r$.

g) X_{www} en supposant un tableau de données $n \times p \times r$.

6.2 Générer les suites de nombres suivantes à l'aide d'expressions R. (Évidemment, il faut trouver un moyen de générer les suites sans simplement concaténer les différentes sous-suites.)

a) $0, 0, 1, 0, 1, 2, \ldots, 0, 1, 2, 3, \ldots, 10$.

b) $10, 9, 8, \ldots, 2, 1, 10, 9, 8, \ldots 3, 2, \ldots, 10, 9, 10$.

c) $10, 9, 8, \ldots, 2, 1, 9, 8, \ldots, 2, 1, \ldots, 2, 1, 1$.

6.3 La fonction de densité de probabilité et la fonction de répartition de la loi de Pareto de paramètres α et λ sont, respectivement,

$$f(x) = \frac{\alpha\lambda^{\alpha}}{(x+\lambda)^{\alpha+1}}$$

et

$$F(x) = 1 - \left(\frac{\lambda}{x+\lambda}\right)^{\alpha}.$$

La fonction suivante simule un échantillon aléatoire de taille n issu d'une distribution de Pareto de paramètres $\alpha =$ `shape` et $\lambda =$ `scale` :

```
rpareto <- function(n, shape, scale)
    scale * (runif(n)^(-1/shape) - 1)
```

a) Écrire une expression R utilisant la fonction `rpareto` ci-dessus qui permet de simuler cinq échantillons aléatoires de tailles 100, 150, 200, 250 et 300 d'une loi de Pareto avec $\alpha = 2$ et $\lambda = 5\,000$. Les échantillons aléatoires devraient être stockés dans une liste.

b) On vous donne l'exemple suivant d'utilisation de la fonction `paste` :

```
> paste("a", 1:5, sep = "")

[1] "a1" "a2" "a3" "a4" "a5"
```

Nommer les éléments de la liste créée en a) `sample1`, …, `sample5`.

c) Calculer la moyenne de chacun des échantillons aléatoires obtenus en a). Retourner le résultat dans un vecteur.

d) Évaluer la fonction de répartition de la loi de Pareto$(2, 5\,000)$ en chacune des valeurs de chacun des échantillons aléatoires obtenus en a). Retourner les valeurs de la fonction de répartition en ordre croissant.

e) Faire l'histogramme des données du cinquième échantillon aléatoire avec la fonction `hist`.

f) Ajouter 1 000 à toutes les valeurs de tous les échantillons simulés en a), ceci afin d'obtenir des observations d'une distribution de Pareto *translatée*.

6.4 Une base de données contenant toutes les informations sur les assurés est stockée dans une liste de la façon suivante :

```
> x[[1]]

$num.police
[1] 1001

$franchise
[1] 500

$nb.acc
[1] 0 1 1 0

$montants
[1] 3186.864 3758.389

> x[[2]]
```

```
$num.police
[1] 1002

$franchise
[1] 250

$nb.acc
[1] 4 0 0 4 1 1 0

$montants
 [1]  16728.7354   1414.7264   1825.7495    282.5609
 [5]   1684.6686  14869.1731   7668.4196   2501.7257
 [9] 108979.3725   2775.3161
```

Ainsi, `x[[i]]` contient les informations relatives à l'assuré i. Sans utiliser de boucles, écrire des expressions ou des fonctions R qui permettront de calculer les quantités suivantes.

a) La franchise moyenne dans le portefeuille.

b) Le nombre annuel moyen de réclamations par assuré.

c) Le nombre total de réclamations dans le portefeuille.

d) Le montant moyen par accident dans le portefeuille.

e) Le nombre d'assurés n'ayant eu aucune réclamation.

f) Le nombre d'assurés ayant eu une seule réclamation dans leur première année.

g) La variance du nombre total de sinistres.

h) La variance du nombre de sinistres pour chaque assuré.

i) La probabilité empirique qu'une réclamation soit inférieure à x (un scalaire) dans le portefeuille.

j) La probabilité empirique qu'une réclamation soit inférieure à $\mathbf{x}$ (un vecteur) dans le portefeuille.

7 Fonctions d'optimisation

Objectifs du chapitre

- Connaître et savoir utiliser les différentes fonctions de calcul de racines de R.
- Connaître et savoir utiliser les différentes fonctions d'optimisation de R.
- Savoir reformuler un problème d'optimisation en base logarithmique pour éviter les difficultés numériques.

Les méthodes de bissection, du point fixe, de Newton–Raphson et consorts permettent de résoudre des équations à une variable de la forme $f(x) = 0$ ou $g(x) = x$. Il existe également des versions de ces méthodes pour les systèmes à plusieurs variables de la forme

$$\begin{aligned} f_1(x_1, x_2, x_3) &= 0 \\ f_2(x_1, x_2, x_3) &= 0 \\ f_3(x_1, x_2, x_3) &= 0. \end{aligned}$$

De tels systèmes d'équations surviennent plus souvent qu'autrement lors de l'optimisation d'une fonction. Par exemple, en recherchant le maximum ou le minimum d'une fonction $f(x, y)$, on souhaitera résoudre le système d'équations

$$\begin{aligned} \frac{\partial}{\partial x} f(x, y) &= 0 \\ \frac{\partial}{\partial y} f(x, y) &= 0. \end{aligned}$$

En inférence statistique, les fonctions d'optimisation sont fréquemment employées pour calculer numériquement des estimateurs du maximum de vraisemblance.

La grande majorité des suites logicielles de calcul comportent des outils d'optimisation de fonctions. Ce chapitre passe en revue les fonctions disponibles dans R. Comme pour les chapitres précédents, des exemples d'utilisation de chacune des fonctions se trouvent dans le code informatique de la section 7.4.

7.1 Fonctions d'optimisation et de calcul de racines

Le système R compte un certain nombre de fonctions pour trouver numériquement le minimum ou le maximum d'une fonction ainsi que pour calculer la racine d'une fonction dans un intervalle ou toutes les racines d'un polynôme. Ces fonctions diffèrent par certaines de leurs fonctionnalités et leur interface, mais aussi par les algorithmes utilisés. Consulter les rubriques d'aide pour les détails.

7.1.1 Fonction `uniroot`

La fonction `uniroot` recherche la racine d'une fonction dans un intervalle. C'est donc la fonction de base pour trouver la solution (unique) de l'équation $f(x) = 0$ dans un intervalle déterminé.

7.1.2 Fonction `optimize`

La fonction `optimize` recherche le minimum local (par défaut) ou le maximum local d'une fonction dans un intervalle donné.

7.1.3 Fonction `nlm`

La fonction `nlm` minimise une fonction non linéaire sur un nombre arbitraire de paramètres.

7.1.4 Fonction `nlminb`

La fonction `nlminb` est similaire à `nlm`, sauf qu'elle permet de spécifier des bornes inférieure ou supérieure pour les paramètres. Attention, toutefois : les arguments de la fonction ne sont ni les mêmes, ni dans le même ordre que ceux de `nlm`.

7.1.5 Fonction `optim`

La fonction `optim` est l'outil d'optimisation tout usage de R. À ce titre, la fonction est souvent utilisée par d'autres fonctions. Elle permet de choisir parmi plusieurs algorithmes d'optimisation différents et, selon l'algorithme choisi, de fixer des seuils minimum ou maximum aux paramètres à optimiser.

7.1.6 `polyroot`

En terminant, un mot sur `polyroot()`, qui n'est pas à proprement parler une fonction d'optimisation, mais qui pourrait être utilisée dans ce contexte. La fonction `polyroot` calcule toutes les racines (complexes) du polynôme $\sum_{i=0}^{n} a_i x^i$. Le premier argument est le vecteur des coefficients $a_0, a_1, \ldots, a_n$, dans cet ordre.

7.2 Astuce Ripley

Brian Ripley — un important développeur de R — a publié le truc suivant dans les forums de discussion de R. Puisqu'il est très utile, nous nous permettons de le disséminer.

Une application statistique fréquente de l'optimisation est la maximisation numérique d'une fonction de vraisemblance ou, plus communément, la minimisation de la log-vraisemblance négative

$$-l(\theta) = -\sum_{i=1}^{n} \ln f(x_i; \theta).$$

Les fonctions d'optimisation sont d'ailleurs illustrées dans ce contexte dans le code informatique de la section 7.4.

Plusieurs lois de probabilité ont des paramètres strictement positifs. Or, en pratique, il n'est pas rare que les fonctions d'optimisation s'égarent dans les valeurs négatives des paramètres. La fonction de densité n'étant pas définie, la log-vraisemblance vaut alors `NaN` et cela peut faire complètement dérailler la procédure d'optimisation ou, à tout le moins, susciter des doutes sur la validité de la réponse.

Afin de pallier à ce problème, l'Astuce Ripley™ propose d'estimer non pas les paramètres de la loi eux-mêmes, mais plutôt leurs logarithmes. Si l'on définit $\tilde{\theta} = \ln \theta$, alors on peut écrire la fonction de log-vraisemblance

ci-dessus sous la forme

$$-l(\tilde{\theta}) = -\sum_{i=1}^{n} \ln f(x_i; e^{\tilde{\theta}}).$$

Dès lors, $\tilde{\theta}$ (qui peut représenter un ou plusieurs paramètres) demeure valide sur tout l'axe des réels, ce qui permet d'éviter bien des soucis de nature numérique lors de la minimisation de $-l(\tilde{\theta})$.

Évidemment, le résultat de l'optimisation est l'estimateur du maximum de vraisemblance de $\tilde{\theta}$. Il faudra donc veiller à faire la transformation inverse pour retrouver l'estimateur de θ.

L'utilisation de l'astuce est illustrée à la section 7.4.

7.3 Pour en savoir plus

Les packages disponible sur CRAN fournissent plusieurs autres outils d'optimisation pour R. Pour un bon résumé des options disponibles, consulter la *CRAN Task View* consacrée à l'optimisation :

http://cran.r-project.org/web/views/Optimization.html

7.4 Exemples

```
###
### FONCTION 'uniroot'
###

## La fonction 'uniroot' recherche la racine d'une fonction
## 'f' dans un intervalle spécifié soit comme une paire de
## valeurs dans un argument 'interval', soit via des arguments
## 'lower' et 'upper'.
##
## On calcule la solution de l'équation x - 2^(-x) = 0 dans
## l'intervalle [0, 1].
f <- function(x) x - 2^(-x)      # fonction
uniroot(f, c(0, 1))              # appel simple
uniroot(f, lower = 0, upper = 1) # équivalent

## On peut aussi utiliser 'uniroot' avec une fonction anonyme.
uniroot(function(x) x - 2^(-x), lower = 0, upper = 1)

###
```

```
### FONCTION 'optimize'
###

## On cherche le maximum local de la densité d'une loi bêta
## dans l'intervalle (0, 1), son domaine de définition. (Ce
## problème est facile à résoudre explicitement.)
##
## Les arguments de 'optimize' sont essentiellement les mêmes
## que ceux de 'uniroot'. Ici, on utilise aussi l'argument
## '...' pour passer les paramètres de la loi bêta à 'dbeta'.
##
## Par défaut, la fonction recherche un minimum. Il faut donc
## lui indiquer de rechercher plutôt un maximum.
optimize(dbeta, interval = c(0, 1), maximum = TRUE,
         shape1 = 3, shape2 = 2)

## On pourrait aussi avoir recours à une fonction auxiliaire.
## Moins élégant et moins flexible.
f <- function(x) dbeta(x, 3, 2)
optimize(f, lower = 0, upper = 1, maximum = TRUE)

###
### FONCTION 'nlm'
###

## Pour la suite, nous allons donner des exemples
## d'utilisation des fonctions d'optimisation dans un contexte
## d'estimation des paramètres d'une loi gamma par la méthode
## du maximum de vraisemblance.
##
## On commence par se donner un échantillon aléatoire de la
## loi. Évidemment, pour ce faire nous devons connaître les
## paramètres de la loi. C'est un exemple fictif.
set.seed(1)                # toujours le même échantillon
x <- rgamma(10, 5, 2)

## Les estimateurs du maximum de vraisemblance des paramètres
## 'shape' et 'rate' de la loi gamma sont les valeurs qui
## maximisent la fonction de vraisemblance
##
##     prod(dgamma(x, shape, rate))
##
## ou, de manière équivalente, qui minimisent la fonction de
## log-vraisemblance négative
##
```

```
##     -sum(log(dgamma(x, shape, rate))).
##
## On remarquera au passage que les fonctions de calcul de
## densités de lois de probabilité dans R ont un argument
## 'log' qui, lorsque TRUE, retourne la valeur du logarithme
## (naturel) de la densité de manière plus précise qu'en
## prenant le logarithme après coup. Ainsi, pour faire le
## calcul ci-dessus, on optera plutôt, pour l'expression
##
##     -sum(dgamma(x, shape, rate, log = TRUE))
##
## La fonction 'nlm' suppose que la fonction à optimiser
## passée en premier argument a elle-même comme premier
## argument le vecteur 'p' des paramètres à optimiser. Le
## second argument de 'nlm' est un vecteur de valeurs de
## départ, une pour chaque paramètre.
##
## Ainsi, pour trouver les estimateurs du maximum de
## vraisemblance avec la fonction 'nlm' pour l'échantillon
## ci-dessus, on doit d'abord définir une fonction auxiliaire
## conforme aux attentes de 'nlm' pour calculer la fonction de
## log-vraisemblance (à un signe près).
f <- function(p, x) -sum(dgamma(x, p[1], p[2], log = TRUE))

## L'appel de 'nlm' est ensuite tout simple. Remarquer comment
## on passe notre échantillon aléatoire (contenu dans l'objet
## 'x') comme second argument à 'f' via l'argument '...' de
## 'nlm'. Le fait que l'argument de 'f' et l'objet contenant
## les valeurs portent le même nom est sans importance. R sait
## faire la différence entre l'un et l'autre.
nlm(f, c(1, 1), x = x)

## === ASTUCE RIPLEY ===
## L'optimisation ci-dessus a généré des avertissements? C'est
## parce que la fonction d'optimisation s'est égarée dans les
## valeurs négatives, alors que les paramètres d'une gamma
## sont strictement positifs. Cela arrive souvent en pratique
## et cela peut faire complètement dérailler la procédure
## d'optimisation (c'est-à-dire: pas de convergence).
##
## L'Astuce Ripley consiste à pallier à ce problème en
## estimant plutôt les logarithmes des paramètres. Pour ce
## faire, il s'agit de réécrire la log-vraisemblance comme une
## fonction du logarithme des paramètres, mais de la calculer
## avec les véritables paramètres.
```

```
f2 <- function(logp, x)
{
    p <- exp(logp)          # retour aux paramètres originaux
    -sum(dgamma(x, p[1], p[2], log = TRUE))
}
nlm(f2, c(0, 0), x = x)

## Les valeurs obtenues ci-dessus sont toutefois les
## estimateurs des logarithmes des paramètres de la loi gamma.
## On retrouve les estiamteurs des paramètres en prenant
## l'exponentielle des réponses.
exp(nlm(f2, c(0, 0), x = x)$estimate)
## ====================

###
### FONCTION 'nlminb'
###

## L'utilisation de la fonction 'nlminb' peut s'avérer
## intéressante dans notre contexte puisque l'on sait que les
## paramètres d'une loi gamma sont strictement positifs.
nlminb(c(1, 1), f, x = x, lower = 0, upper = Inf)

###
### FONCTION 'optim'
###

## La fonction 'optim' est très puissante, mais requiert aussi
## une bonne dose de prudence. Ses principaux arguments sont:
##
##  par: un vecteur contenant les valeurs initiales des
##       paramètres;
##   fn: la fonction à minimiser. Le premier argument de fn
##       doit être le vecteur des paramètres.
##
## Comme pour les autres fonctions étudiées ci-dessus, on peut
## passer des arguments à 'fn' (les données, par exemple) par
## le biais de l'argument '...' de 'optim'.
optim(c(1, 1), f, x = x)

## L'estimation par le maximum de
##    vraisemblance\index{vraisemblance} est de beaucoup
##    simplifiée par l'utilisation de la fonction
##    \fonction{fitdistr} du package
##    \texttt{MASS}\index{package!MASS@\texttt{MASS}}.
```

```
###
### FONCTION 'polyroot'
###

## Racines du polynôme x^3 + 4 x^2 - 10. Les réponses sont
## données sous forme de nombre complexe. Utiliser les
## fonctions 'Re' et 'Im' pour extraire les parties réelles et
## imaginaires des nombres, respectivement.
polyroot(c(-10, 0, 4, 1))     # racines
Re(polyroot(c(-10, 0, 4, 1))) # parties réelles
Im(polyroot(c(-10, 0, 4, 1))) # parties imaginaires
```

7.5 Exercices

7.1 Trouver la solution des équations suivantes à l'aide des fonctions R appropriées.

a) $x^3 - 2x^2 - 5 = 0$ pour $1 \leq x \leq 4$

b) $x^3 + 3x^2 - 1 = 0$ pour $-4 \leq x \leq 0$

c) $x - 2^{-x} = 0$ pour $0 \leq x \leq 1$

d) $e^x + 2^{-x} + 2\cos x - 6 = 0$ pour $1 \leq x \leq 2$

e) $e^x - x^2 + 3x - 2 = 0$ pour $0 \leq x \leq 1$

7.2 En théorie de la crédibilité, l'estimateur d'un paramètre a est donné sous forme de point fixe

$$\hat{a} = \frac{1}{n-1} \sum_{i=1}^{n} z_i (X_i - \bar{X}_z)^2,$$

où

$$z_i = \frac{\hat{a} w_i}{\hat{a} w_i + s^2}$$

$$\bar{X}_z = \sum_{i=1}^{n} \frac{z_i}{z_\Sigma} X_i$$

et $X_1, \ldots, X_n$, $w_1, \ldots, w_n$ et s^2 sont des données. Calculer la valeur de $\hat{a}$ si $s^2 = 140\,000\,000$ et que les valeurs de X_i et w_i sont telles qu'elles apparaissent dans le tableau ci-dessous.

i	1	2	3	4	5
X_i	2 061	1 511	1 806	1 353	1 600
w_i	100 155	19 895	13 735	4 152	36 110

7.3 Les fonctions de densité de probabilité et de répartition de la distribution de Pareto sont données à l'exercice 6.3. Calculer les estimateurs du maximum de vraisemblance des paramètres de la Pareto à partir d'un échantillon aléatoire obtenu par simulation avec la commande

```
> x <- lambda * (runif(100)^(-1/alpha) - 1)
```

pour des valeurs de `alpha` et `lambda` choisies.

8 Génération de nombres aléatoires

Objectifs du chapitre

- Générer des nombres aléatoires uniformes avec la fonction `runif`.
- Générer des nombres aléatoires non uniformes provenant de lois de probabilité discrètes et continues.
- Générer des nombres aléatoires provenant d'une distribution discrète quelconque.
- Tirer profit de la nature vectorielle des fonctions de simulation de R.

Avant d'utiliser pour quelque tâche de simulation moindrement importante un générateur de nombres aléatoires inclus dans un logiciel, il importe de s'assurer de la qualité de cet outil. On trouvera en général relativement facilement de l'information dans Internet.

On présente ici, sans entrer dans les détails, les générateurs de nombres uniformes utilisés dans R ainsi que la liste des différentes fonctions de simulation de variables aléatoires.

8.1 Générateurs de nombres aléatoires uniformes

On obtient des nombres uniformes sur un intervalle quelconque avec la fonction `runif` dans R. La fonction `set.seed` permet de spécifier la valeur de l'amorce du générateur aléatoire, ce qui est utile si on veut répéter une simulation absolument à l'identique.

R offre la possibilité de choisir entre plusieurs générateurs de nombres aléatoires différents, ou encore de spécifier son propre générateur. Par défaut, R utilise le générateur Marsenne–Twister, considéré comme le plus avancé en ce moment. La période de ce générateur est $2^{19\,937} - 1$ (rien de

moins !) et la distribution des nombres est uniforme dans 623 dimensions consécutives sur toute la période.

Pour de plus amples détails et les dernières informations sur les générateurs disponibles et la procédure de réglage de l'amorce, consulter les rubriques d'aide des fonctions `.Random.seed` et `set.seed`.

8.2 Simulation de variables aléatoires non uniformes

Un large éventail de fonctions donne directement accès aux caractéristiques de plusieurs lois de probabilité dans R. Pour chaque racine *loi*, il existe quatre fonctions différentes :

1. `d`*loi* calcule la fonction de densité de probabilité (loi continue) ou la fonction de masse de probabilité (loi discrète) ;
2. `p`*loi* calcule la fonction de répartition ;
3. `q`*loi* calcule la fonction de quantile ;
4. `r`*loi* simule des observations de cette loi.

Les différentes lois de probabilité disponibles dans le système R de base, leur racine et le nom de leurs paramètres sont rassemblés au tableau 8.1. Des packages fournissent des fonctions pour d'autres lois dont, entre autres, **actuar** (Dutang et collab., 2008) et **SuppDists** (Wheeler, 2013).

Toutes les fonctions du tableau 8.1 sont vectorielles, c'est-à-dire qu'elles acceptent en argument un vecteur de points où la fonction (de densité, de répartition ou de quantile) doit être évaluée et même un vecteur de paramètres. Par exemple,

```
> dpois(c(3, 0, 8), lambda = c(1, 4, 10))

[1] 0.06131324 0.01831564 0.11259903
```

retourne la probabilité que des lois de Poisson de paramètre 1, 4 et 10 prennent les valeurs 3, 0 et 8, dans l'ordre.

Le premier argument de toutes les fonctions de simulation est la quantité de nombres aléatoires désirée. Ainsi,

```
> rpois(3, lambda = c(1, 4, 10))

[1] 0 2 4
```

retourne trois nombres aléatoires issus de distributions de Poisson de paramètre 1, 4 et 10, respectivement. Évidemment, passer un vecteur comme premier argument n'a pas tellement de sens, mais, si c'est fait, R retournera

Loi de probabilité	Racine dans R	Noms des paramètres
Bêta	`beta`	`shape1`, `shape2`
Binomiale	`binom`	`size`, `prob`
Binomiale négative	`nbinom`	`size`, `prob` ou `mu`
Cauchy	`cauchy`	`location`, `scale`
Exponentielle	`exp`	`rate`
F (Fisher)	`f`	`df1`, `df2`
Gamma	`gamma`	`shape`, `rate` ou `scale`
Géométrique	`geom`	`prob`
Hypergéométrique	`hyper`	`m`, `n`, `k`
Khi carré	`chisq`	`df`
Logistique	`logis`	`location`, `scale`
Log-normale	`lnorm`	`meanlog`, `sdlog`
Normale	`norm`	`mean`, `sd`
Poisson	`pois`	`lambda`
t (Student)	`t`	`df`
Uniforme	`unif`	`min`, `max`
Weibull	`weibull`	`shape`, `scale`
Wilcoxon	`wilcox`	`m`, `n`

TAB. 8.1 – Lois de probabilité pour lesquelles il existe des fonctions dans le système R de base

une quantité de nombres aléatoires égale à la *longueur* du vecteur (sans égard aux valeurs contenues dans le vecteur).

La fonction `sample` permet de simuler des nombres d'une distribution discrète quelconque. Sa syntaxe est

```
sample(x, size, replace = FALSE, prob = NULL),
```

où `x` est un vecteur des valeurs possibles de l'échantillon à simuler (le support de la distribution), `size` est la quantité de nombres à simuler et `prob` est un vecteur de probabilités associées à chaque valeur de `x` (`1/length(x)` par défaut). Enfin, si `replace` est `TRUE`, l'échantillonnage se fait avec remise.

8.3 Exemples

```
###
### GÉNÉRATEURS DE NOMBRES ALÉATOIRES
###

## La fonction de base pour simuler des nombres uniformes est
## 'runif'.
runif(10)                   # sur (0, 1) par défaut
runif(10, 2, 5)             # sur un autre intervalle
2 + 3 * runif(10)           # équivalent, moins lisible

## R est livré avec plusieurs générateurs de nombres
## aléatoires. On peut en changer avec la fonction 'RNGkind'.
RNGkind("Wichmann-Hill")    # générateur de Excel
runif(10)                   # rien de particulier à voir
RNGkind("default")          # retour au générateur par défaut

## La fonction 'set.seed' est très utile pour spécifier
## l'amorce d'un générateur. Si deux simulations sont
## effectuées avec la même amorce, on obtiendra exactement les
## mêmes nombres aléatoires et, donc, les mêmes résultats.
## Très utile pour répéter une simulation à l'identique.
set.seed(1)                 # valeur sans importance
runif(5)                    # 5 nombres aléatoires
runif(5)                    # 5 autres nombres
set.seed(1)                 # réinitialisation de l'amorce
runif(5)                    # les mêmes 5 nombres que ci-dessus

###
### FONCTIONS POUR LA SIMULATION DE VARIABLES ALÉATOIRES NON
### UNIFORMES
###

## Plutôt que de devoir utiliser la méthode de l'inverse ou un
## autre algorithme de simulation pour obtenir des nombres
## aléatoires d'une loi de probabilité non uniforme, R fournit
## des fonctions de simulation pour bon nombre de lois. Toutes
## ces fonctions sont vectorielles. Ci-dessous, P == Poisson
## et G == Gamma pour économiser sur la notation.
n <- 10                     # taille des échantillons
rbinom(n, 5, 0.3)           # Binomiale(5, 0,3)
rbinom(n, 1, 0.3)           # Bernoulli(0,3)
rnorm(n)                    # Normale(0, 1)
```

```
rnorm(n, 2, 5)              # Normale(2, 25)
rpois(n, c(2, 5))           # P(2), P(5), P(2), ..., P(5)
rgamma(n, 3, 2:11)          # G(3, 2), G(3, 3), ..., G(3, 11)
rgamma(n, 11:2, 2:11)       # G(11, 2), G(10, 3), ..., G(2, 11)

## La fonction 'sample' sert pour simuler d'une distribution
## discrète quelconque. Le premier argument est le support de
## la distribution et le second, la taille de l'échantillon
## désirée. Par défaut, l'échantillonnage se fait avec remise
## et avec des probabilités égales sur tout le support.
sample(1:49, 7)             # numéros pour le 7/49
sample(1:10, 10)            # mélange des nombres de 1 à 10

## On peut échantillonner avec remise.
sample(1:10, 10, replace = TRUE)

## On peut aussi spécifier une distribution de probabilités
## non uniforme.
x <- sample(c(0, 2, 5), 1000, replace = TRUE,
            prob = c(0.2, 0.5, 0.3))
table(x)                    # tableau de fréquences
```

8.4 Exercices

8.1 La loi log-normale est obtenue par transformation de la loi normale : si la distribution de la variable aléatoire X est une normale de paramètres μ et σ^2, alors la distribution de e^X est une log-normale. Simuler 1 000 observations d'une loi log-normale de paramètres $\mu = \ln 5000 - \frac{1}{2}$ et $\sigma^2 = 1$, puis tracer l'histogramme de l'échantillon aléatoire obtenu.

8.2 Simuler 10 000 observations d'un mélange continu Poisson/gamma où les paramètres de la loi gamma sont $\alpha = 5$ et $\lambda = 4$, puis tracer la distribution de fréquence de l'échantillon aléatoire obtenu à l'aide des fonctions `plot` et `table`. Superposer à ce graphique la fonction de probabilité d'une binomiale négative de paramètres $r = 5$ et $\theta = 0{,}8$.

8.3 Simuler 10 000 observations d'un mélange discret de deux distributions log-normales, l'une de paramètres $(\mu = 3{,}5, \sigma^2 = 0{,}6)$ et l'autre de paramètres $(\mu = 4{,}6, \sigma^2 = 0{,}3)$. Utiliser un paramètre de mélange $p = 0{,}55$. Tracer ensuite l'histogramme de l'échantillon aléatoire obtenu.

A GNU Emacs et ESS : la base

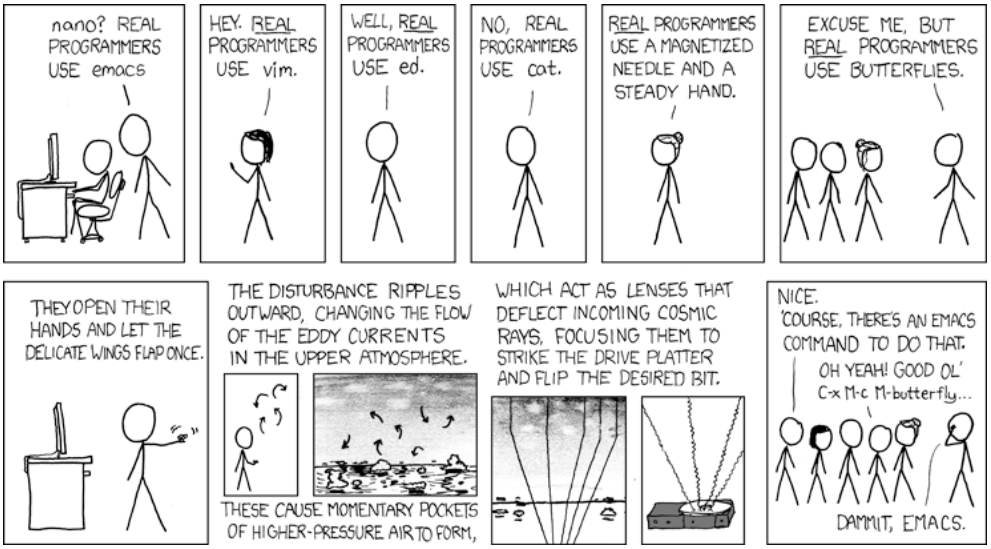
Emacs est l'Éditeur de texte des éditeurs de texte. À l'origine un éditeur pour les programmeurs (avec des modes spéciaux pour une multitude de langages différents), Emacs est devenu au fil du temps un environnement logiciel en soi dans lequel on peut réaliser une foule de tâches différentes : rédiger des documents LaTeX, interagir avec R, SAS ou un logiciel de base de données, consulter son courrier électronique, gérer son calendrier ou même jouer à Tetris !

Cette annexe passe en revue les quelques commandes essentielles à connaître pour commencer à travailler avec GNU Emacs et le mode ESS. L'ouvrage de Cameron et collab. (2004) constitue une excellente référence pour l'apprentissage plus poussé de l'éditeur.

A.1 Mise en contexte

Emacs est le logiciel étendard du projet GNU (« *GNU is not Unix* »), dont le principal commanditaire est la *Free Software Foundation* (FSF) à l'origine de tout le mouvement du logiciel libre.

- Richard M. Stallman, président de la FSF et grand apôtre du libre, a écrit la première version de Emacs et il continue à ce jour à contribuer au projet.
- Les origines de Emacs remontent au début des années 1980, une époque où les interfaces graphiques n'existaient pas, le parc informatique était beaucoup plus hétérogène qu'aujourd'hui (les claviers n'étaient pas les mêmes d'une marque d'ordinateur à une autre) et les modes de communication entre les ordinateurs demeuraient rudimentaires.
- L'âge vénérable de Emacs transparaît à plusieurs endroits, notamment dans la terminologie inhabituelle, les raccourcis clavier non conformes aux standards d'aujourd'hui ou la manipulation des fenêtres qui ne se fait pas avec une souris.



Tiré de XKCD.com. La commande `M-X butterfly` existe vraiment dans Emacs... en référence à cette bande dessinée!

Emacs s'adapte à différentes tâches par l'entremise de *modes* qui modifient son comportement ou lui ajoutent des fonctionnalités. L'un de ces modes est ESS (*Emacs Speaks Statistics*).

- ESS permet d'interagir avec des logiciels statistiques (en particulier R, S+ et SAS) directement depuis Emacs.
- Quelques-uns des développeurs de ESS sont aussi des développeurs de R, d'où la grande compatibilité entre les deux logiciels.
- Lorsque ESS est installé, le mode est activé automatiquement en ouvrant dans Emacs un fichier dont le nom se termine par l'extension `.R`.

A.2 Installation

GNU Emacs et le mode ESS sont normalement livrés d'office avec toutes les distributions Linux. Pour les environnements Windows et macOS, le plus simple consiste à installer les distributions préparées par le présent auteur. Consulter le site

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

A.3 Description sommaire

Au lancement, Emacs affiche un écran d'information contenant des liens vers différentes ressources. Cet écran disparaît dès que l'on appuie sur une touche. La fenêtre Emacs se divise en quatre zone principales (voir la figure A.1) :

1. tout au haut de la fenêtre (ou de l'écran sous macOS), on trouve l'habituelle barre de menu dont le contenu change selon le mode dans lequel se trouve Emacs ;
2. l'essentiel de la fenêtre sert à afficher un *buffer*, soit le contenu d'un fichier ouvert ou l'invite de commande d'un programme externe ;
3. la ligne de mode est le séparateur horizontal contenant diverses informations sur le fichier ouvert et l'état de Emacs ;
4. le *minibuffer* est la région au bas de la fenêtre où l'on entre des commandes et reçoit de l'information de Emacs.

Il est possible de séparer la fenêtre Emacs en sous-fenêtres pour afficher plusieurs *buffers* à la fois. Il y a alors une ligne de mode pour chaque *buffer*.

A.4 *Emacs-ismes* et *Unix-ismes*

Emacs possède sa propre terminologie qu'il vaut mieux connaître lorsque l'on consulte la documentation. De plus, l'éditeur utilise des conventions du monde Unix qui sont moins usitées sur les plateformes Windows et macOS.

- Dans les définitions de raccourcis claviers :
  - `C` est la touche `Contrôle` (^) ;
  - `M` est la touche `Meta`, qui correspond à la touche `Alt` de gauche sur un PC ou la touche `Option` (⌥) sur un Mac (toutefois, voir l'encadré à la page 151) ;
  - `ESC` est la touche `Échap` (⎋) et est équivalente à `Meta` ;
  - `SPC` est la barre d'espacement ;
  - `DEL` est la touche `Retour arrière` (⌫) — *et non la touche* `Supprimer`.
  - `RET` est la touche `Entrée` (↩) ;
- Toutes les fonctionnalités de Emacs correspondent à une commande pouvant être tapée dans le *minibuffer*. `M-x` démarre l'invite de commande.

FIG. A.1 – Fenêtre GNU Emacs et ses différentes parties au lancement de l'application sous macOS. Sous Windows et Linux, la barre de menu se trouve à l'intérieur de la fenêtre.

- Le caractère ~ représente le dossier vers lequel pointe la variable d'environnement `$HOME` (Linux, macOS) ou `%HOME%` (Windows). C'est le dossier par défaut de Emacs.
- 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.

Par défaut sous macOS, la touche `Meta` est assignée à `Option` (⌥). Sur les claviers français, cela empêche d'accéder à certains caractères spéciaux tels que [,], { ou }.
Une solution à cette fâcheuse situation consiste à assigner la touche `Meta` à `Commande` (⌘). Cela bloque alors l'accès à certains raccourcis Mac, mais la situation est moins critique ainsi.
Pour assigner la touche `Meta` à `Commande` (⌘) et laisser la touche `Option` (⌥) jouer son rôle usuel, il suffit d'insérer les lignes suivantes dans son fichier de configuration `.emacs` (voir la section A.7) :

```
;;; ===================================
;;;  Assigner la touche Meta à Commande
;;;  et laisser Option être Option
;;; ===================================
(setq-default ns-command-modifier 'meta)
(setq-default ns-option-modifier 'none)
```

A.5 Commandes de base

Emacs comporte une pléthore de commandes, il serait donc futile de tenter d'en faire une liste exhaustive ici. Nous nous contenterons de mentionner les commandes les plus importantes regroupées par tâche.

A.5.1 Les essentielles

`M-x` démarrer l'invite de commande

`C-g` bouton de panique : annuler, quitter ! Presser plus d'une fois au besoin.

A.5.2 Manipulation de fichiers

Entre parenthèses, le nom de la commande Emacs correspondante. On peut entrer cette commande dans le *minibuffer* au lieu d'utiliser le raccourci clavier.

On remarquera qu'il n'existe pas de commande « nouveau fichier » dans Emacs. Pour créer un nouveau fichier, il suffit d'ouvrir un fichier n'existant pas.

`C-x C-f`	ouvrir un fichier (`find-file`)
`C-x C-s`	sauvegarder (`save-buffer`)
`C-x C-w`	sauvegarder sous (`write-file`)
`C-x k`	fermer un fichier (`kill-buffer`)
`C-_`	annuler (pratiquement illimité) ; aussi `C-x u` (`undo`)
`C-s`	recherche incrémentale avant (`isearch-forward`)
`C-r`	Recherche incrémentale arrière (`isearch-backward`)
`M-%`	rechercher et remplacer (`query-replace`)

A.5.3 Déplacements simples du curseur

`C-b` | `C-f`	déplacer d'un caractère vers l'arrière | l'avant (`backward-char` | `forward-char`)
`C-a` | `C-e`	aller au début | fin de la ligne (`move-beginning-of-line` | `move-end-of-line`)
`C-p` | `C-n`	aller à la ligne précédente | suivante (`previous-line` | `next-line`)
`M-<` | `M->`	aller au début | fin du fichier (`beginning-of-buffer` | `end-of-buffer`)
`DEL` | `C-d`	effacer le caractère à gauche | droite du curseur (`delete-backward-char` | `delete-char`)
`M-DEL` | `M-d`	effacer le mot à gauche | droite du curseur (`backward-kill-word` | `kill-word`)
`C-k`	supprimer jusqu'à la fin de la ligne (`kill-line`)

Plusieurs des raccourcis clavier de Emacs composés avec la touche `Contrôle` (^) sont valides sous macOS. Par exemple, `^A` et `^E` déplacent le curseur au début et à la fin de la ligne dans les champs texte.

A.5.4 Sélection de texte, copier, coller, couper

`C-SPC`	débute la sélection (`set-mark-command`)
`C-w`	couper la sélection (`kill-region`)
`M-w`	copier la sélection (`kill-ring-save`)
`C-y`	coller (`yank`)
`M-y`	remplacer le dernier texte collé par la sélection précédente (`yank-pop`)

- Il est possible d'utiliser les raccourcis clavier usuels de Windows (`C-c`, `C-x`, `C-v`) et macOS (`⌘C`, `⌘X`, `⌘V`) en activant le mode CUA dans le menu `Options`.
- On peut copier-coller directement avec la souris dans Windows en sélectionnant du texte puis en appuyant sur le bouton central (ou la molette) à l'endroit souhaité pour y copier le texte.

A.5.5 Manipulation de fenêtres

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

A.5.6 Manipulation de fichiers de script dans le mode ESS

Le mode ESS dispose de fonctions « intelligentes » qui facilitent grandement la manipulation des fichiers de script. Les deux principales commandes à connaître sont les suivantes :

`C-RET` évaluer dans le processus R la ligne sous le curseur ou la région sélectionnée, puis déplacer le curseur à la prochaine expression (`ess-eval-region-or-line-and-step`)

`C-c C-c` évaluer dans le processus R la région sélectionnée, la fonction ou le paragraphe (tout bloc entre deux lignes blanches) dans lequel se trouve le curseur, puis déplacer le curseur à la prochaine expression (`ess-eval-region-or-function-or-paragraph-and-step`)

Les quelques autres fonctions utiles sont :

`C-c C-z` déplacer le curseur vers le processus R (`ess-switch-to-inferior-or-script-buffer`)

`C-c C-f` évaluer le code de la fonction courante dans le processus R (`ess-eval-function`)

`C-c C-l` évaluer le code du fichier courant en entier dans le processus R (`ess-load-file`)

`C-c C-v` aide sur une commande R (`ess-display-help-on-object`)

A.5.7 Interaction avec l'invite de commande R

`M-p` | `M-n` commande précédente | suivante dans l'historique (`previous-matching-history-from-input`, `next-matching-history-from-input`)

`C-c C-z` déplacer le curseur vers le fichier de script courant (`ess-switch-to-inferior-or-script-buffer`)

`M-h` sélectionner le résultat de la dernière commande (`mark-paragraph`)

`C-c C-o` effacer le résultat de la dernière commande (`comint-delete-output`)

`C-c C-v` aide sur une commande R (`ess-display-help-on-object`)

`C-c C-q` terminer le processus R (`ess-quit`)

A.5.8 Consultation des rubriques d'aide de R

`p` | `n` aller à la section précédente | suivante de la rubrique (`ess-skip-to-previous-section` | `ess-skip-to-next-section`)

`s a` aller à la section de la liste des arguments *(Arguments)*

`s D`	aller à la section des détails sur la fonction *(Details)*
`s v`	aller à la section sur la valeur retournée par la fonction *(Value)*
`s s`	aller à la section des fonctions apparentée *(See Also)*
`s e`	aller à la section des exemples *(Examples)*
`l`	évaluer la ligne sous le curseur ; pratique pour exécuter les exemples (`ess-eval-line-and-step`)
`r`	évaluer la région sélectionnée (`ess-eval-region`)
`h`	ouvrir une nouvelle rubrique d'aide, par défaut pour le mot se trouvant sous le curseur (`ess-display-help-on-object`)
`q`	retourner au processus ESS en laissant la rubrique d'aide visible (`ess-switch-to-end-of-ESS`)
`x`	fermer la rubrique d'aide et retourner au processus ESS (`ess-kill-buffer-and-go`)

A.6 Anatomie d'une session de travail (bis)

On reprend ici les étapes d'une session de travail type présentées à la section 1.6, mais en expliquant comment compléter chacune dans Emacs avec le mode ESS.

1. Lancer Emacs et ouvrir un fichier de script avec

   `C-x C-f`

   ou avec le menu

   `File|Open File...`

   En spécifiant un nom de fichier qui n'existe pas déjà, on se trouve à créer un nouveau fichier de script. S'assurer de terminer le nom du nouveau fichier par `.R` pour que Emacs reconnaisse automatiquement qu'il s'agit d'un fichier de script R.

2. Démarrer un processus R à l'intérieur même de Emacs avec

   `M-x R ⏎`

   Emacs demandera alors de spécifier de répertoire de travail (*starting data directory*). Accepter la valeur par défaut, par exemple

   `~/ ⏎`

ou indiquer un autre dossier. Un éventuel message de Emacs à l'effet que le fichier `.Rhistory` n'a pas été trouvé est sans conséquence et peut être ignoré.

3. Composer le code. Lors de cette étape, on se déplacera souvent du fichier de script à la ligne de commande afin d'essayer diverses expressions. On exécutera également des parties seulement du code se trouvant dans le fichier de script. Les commandes les plus utilisées sont alors :

   `C-RET` pour exécuter une ligne du fichier de script ou la région sélectionnée ;

   `C-c C-c` pour exécuter un paragraphe du fichier de script ;

   `C-c C-z` pour se déplacer entre le fichier de script et la ligne de commande R.

4. Sauvegarder le fichier de script :

   `C-x C-s`

   Les quatrième et cinquième caractères de la ligne de mode changent de `**` à `--`.

5. Sauvegarder si désiré l'espace de travail de R avec `save.image()`. Cela n'est habituellement pas nécessaire à moins que l'espace de travail ne contienne des objets importants ou longs à recréer.

6. Quitter le processus R avec

   `C-c C-q`

   Cette commande ESS se chargera de fermer tous les fichiers associés au processus R. On peut ensuite quitter Emacs en fermant l'application de la manière usuelle.

A.7 Configuration de l'éditeur

Une des grandes forces de Emacs est qu'à peu près chacune de ses facettes est configurable : couleurs, polices de caractère, raccourcis clavier, etc.

- La configuration de Emacs se fait par le biais de commandes réunies dans un fichier de configuration nommé `.emacs` (le point est important !) que Emacs lit au démarrage.
- Le fichier `.emacs` doit se trouver dans le dossier `~/`, c'est-à-dire dans le dossier de départ de l'utilisateur sous Linux et macOS, et dans le dossier référencé par la variable d'environnement `%HOME%` sous Windows.

A.8 Aide et documentation

Emacs possède son propre système d'aide très exhaustif, mais dont la navigation est peu intuitive selon les standards d'aujourd'hui. Consulter le menu `Help`.

Autrement, on trouvera les manuels de Emacs et de ESS en divers formats dans les sites respectifs des deux projets :

```
http://www.gnu.org/software/emacs
http://ess.r-project.org
```

Enfin, si le désespoir vous prend au cours d'une séance de codage intensive, vous pouvez toujours consulter le psychothérapeute Emacs. On le trouve, bien entendu, dans le menu `Help`!

B RStudio : une introduction

Un environnement de développement intégré (*integrated development environment*, IDE) est un progiciel de productivité destiné au développement de logiciels ou, plus largement, à la programmation informatique. Il comprend toujours un éditeur de texte adapté au langage de programmation visé, un environnement de compilation ou d'exécution du code et, généralement, des outils de contrôle de versions, de gestion des projets et de navigation dans le code source. [1]

Offert au public depuis 2011, RStudio est un IDE convivial conçu spécifiquement pour l'analyse de données et le développement de packages avec R. Il est produit par RStudio Inc. et est offert en version libre ou commerciale, pour une exécution locale (*desktop*) ou pour une exécution sur un serveur via un navigateur web.

B.1 Installation

RStudio est disponible à l'identique pour les plateformes Windows, macOS et Linux. Pour une utilisation locale sur son poste de travail, on installera la version libre (*Open Source*) de RStudio Desktop depuis le site

`https://www.rstudio.com/products/rstudio/download/`

B.2 Description sommaire

La fenêtre de RStudio se divise toujours en quatre sous-fenêtres [2] — sauf au lancement, alors que la sous-fenêtre d'édition de code source n'est pas

1. À ce compte, GNU Emacs constitue un environnement de développement intégré. Seulement, nous avons davantage insisté sur ses fonctionnalités d'éditeur de texte dans le présent document.
2. Les sous-fenêtres sont appelées *panes* (en anglais) dans l'application.

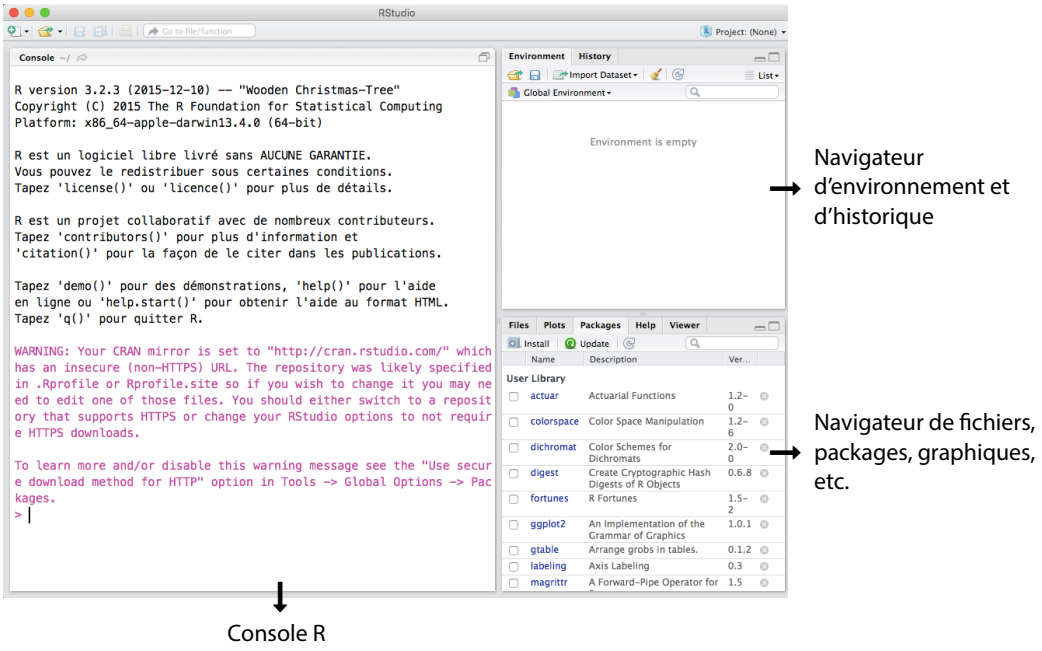


FIG. B.1 – Fenêtre de RStudio et trois de ses sous-fenêtres au lancement de l'application sous macOS. Sous Windows et Linux, la fenêtre comporte également une barre de menu.

visible ; voir la figure B.1. Dans le sens des aiguilles d'une montre en partant en haut à gauche, on trouve :

1. la sous-fenêtre d'édition de code source, avec un onglet par fichier de script ;
2. le navigateur d'environnement de travail ou d'historique des commandes, selon l'onglet sélectionné ;
3. le navigateur de fichiers du projet, de packages, de graphiques, etc., selon l'onglet sélectionné ;
4. la console — ou ligne de commande — R.

Au lancement de l'application, la console R occupe toute la gauche de la fenêtre jusqu'à ce qu'un fichier de script soit ouvert.

- Le navigateur d'environnement de travail est particulièrement utile pour voir le contenu, les attributs, le type et la taille de chaque objet sauvegardé dans la session R. Il permet également de visualiser le contenu des objets en cliquant sur leur nom ou sur l'icône de grille à droite de leur nom.

- Il ne peut y avoir qu'un seul processus R (affiché dans la console) actif par fenêtre RStudio. Pour utiliser plusieurs processus R simultanément, il faut démarrer autant de copies de RStudio.
- La position des sous-fenêtres dans la grille ne peut être modifiée. Par contre, chaque sous-fenêtre peut être redimensionnée.
- On peut modifier la liste des onglets affichés dans les deux navigateurs dans les préférences de l'application ; voir la section B.6.

B.3 Projets

Il est possible d'utiliser RStudio un peu comme un simple éditeur de texte.

- On ouvre les fichiers de scripts un à un, soit à partir du menu `File|Open file...`, soit à partir de l'onglet `Files` du navigateur de fichiers.
- Lorsque nécessaire, on change le répertoire de travail de R à partir du menu `Session`.

Pour faciliter l'organisation de son travail, l'ouverture des fichiers de script et le lancement d'un processus R dans le bon répertoire de travail, RStudio propose la notion de *projet*.

- Un projet RStudio est associé à un répertoire de travail de R (section 1.7).
- On crée un nouveau projet à partir du menu `Project` à l'extrémité droite de la barre d'outils ou à partir du menu `File|New Project...` On a alors l'option de créer un nouveau dossier sur notre poste de travail ou de créer un projet à partir d'un dossier existant.
- Lors de la création d'un projet, RStudio crée dans le dossier visé un fichier avec une extension `.Rproj` contenant diverses informations en lien avec le projet. De plus, le projet est immédiatement chargé dans RStudio.
- L'ouverture d'un projet entraîne : le lancement d'une session R ayant comme répertoire de travail le dossier du projet ; le chargement du fichier `.RData` (le cas échéant) ; l'ouverture de tous les fichiers de scripts qui étaient ouverts lors de la dernière séance de travail.
- Chaque projet dispose de ses propres réglages. On accède à ceux-ci via la commande `Project Options...` du menu `Project` de la barre d'outils.

On trouvera plus d'information sur les projets dans l'aide en ligne de RStudio.

B.4 Commandes de base

Comme l'interface de RStudio respecte les standards modernes, nous ne soulignons ici que les commandes particulièrement utiles pour la manipulation des fichiers de script. On accède rapidement à la liste des commandes les plus utiles via le menu `Help` de l'application.

Les raccourcis clavier sous, d'une part, Windows et Linux et sous, d'autre part, macOS légèrement différents. Nous fournissons les deux jeux ci-dessous, séparés par le symbole •.

`Alt+-` • `⌥-`	insérer le symbole d'assignation ␣<-␣
`Ctrl+Retour` • `⌘↩`	évaluer dans le processus R la ligne sous le curseur ou la région sélectionnée, puis déplacer le curseur à la prochaine expression
`Ctrl+Shift+S` • `⇧⌘S`	évaluer le code du fichier courant en entier dans le processus R
`Ctrl+Alt+B` • `⌥⌘B`	évaluer dans le processus R le code source du début du fichier jusqu'à la ligne sous le curseur
`Ctrl+Alt+E` • `⌥⌘E`	évaluer dans le processus R le code source de la ligne sous le curseur jusqu'à la fin du fichier
`Ctrl+Alt+F` • `⌥⌘F`	évaluer le code de la fonction courante dans le processus R

À la console — ou ligne de commande — R, les raccourcis suivants sont particulièrement utiles.

`↑` | `↓`	expression précédente | suivante dans l'historique des commandes
`Ctrl+↑` • `⌘↑`	afficher la fenêtre d'historique des commandes

B.5 Anatomie d'une session de travail (ter)

On reprend ici les étapes d'une session de travail type présentées à la section 1.6, mais en expliquant comment compléter chacune dans RStudio.

1. Lancer RStudio et ouvrir soit un nouveau fichier de script avec

   `Ctrl+Shift+N` • `⇧⌘N`

   ou avec le menu

   `File|New File|R Script...`,

Le très pratique raccourci clavier ⌥- servant à insérer le symbole d'assignation ne fonctionne pas avec le clavier canadien français puisque cette combinaison de touches sert déjà à insérer le symbole |.
Au moment d'écrire ces lignes, RStudio ne permet pas de réaffecter la commande d'insertion du symbole d'assignation à une autre combinaison de touches.
Une solution de rechange consiste à utiliser une disposition de clavier anglaise pour travailler dans RStudio.
Pour ce faire, accéder aux préférences système de macOS puis sélectionner l'option Clavier. Dans l'onglet Méthodes de saisie, installer un nouveau clavier Anglais. Cocher l'option « Afficher le menu de saisie dans la barre des menus » pour pouvoir rapidement et facilement passer d'un type de clavier à un autre.

soit un fichier de script existant avec

`Ctrl+O` • `⌘O`

ou

`File|Open File...`

2. C'est une bonne pratique de faire du dossier où se trouve son ou ses fichiers de scripts le répertoire de travail de R. Il suffit de sélectionner le menu

   `Session|Set Working Directory|To Source File Location`

3. Composer le code. Lors de cette étape de programmation, on se déplacera souvent du fichier de script à la ligne de commande afin d'essayer diverses expressions. On exécutera également des parties seulement du code se trouvant dans le fichier de script. Les commandes les plus utilisées sont alors :

   `Ctrl+Retour` • `⌘↩` pour exécuter une ligne ou une région sélectionnée du fichier de script ;

   `Ctrl+1` • `⌘1` pour déplacer le curseur vers la sous-fenêtre d'édition de script ;

   `Ctrl+2` • `⌘2` pour déplacer le curseur vers la console R.

4. Sauvegarder le fichier de script :

`Ctrl+S` • `⌘S`

(S'il s'agit d'un nouveau fichier, s'assurer de terminer son nom par `.R`.) Le nom du fichier dans l'onglet de la sous-fenêtre passe du rouge au noir.

5. Sauvegarder si désiré l'espace de travail de R avec `save.image()`. Cela n'est habituellement pas nécessaire à moins que l'espace de travail ne contienne des objets importants ou longs à recréer.
6. Quitter RStudio de la manière usuelle. Par défaut, RStudio devrait demander si l'on souhaite sauvegarder l'espace de travail de R. Nous suggérons de ne pas le faire.

   La section B.6 explique comment configurer RStudio afin d'éviter de se faire poser la question à chaque fermeture de l'application.

B.6 Configuration de l'éditeur

Il est possible de configurer plusieurs facettes de RStudio à partir d'une interface familière.

- On accède aux options de configuration par le menu

  `Tools|Global Options...`

  sous Windows et Linux et par le menu standard

  `RStudio|Preferences` (`⌘,`)

  sous macOS.
- Nous suggérons de régler l'option `Save workspace to .RData on exit` à `Never` dans les options de configuration générales ; voir la figure B.2. Avec ce réglage, l'espace de travail de R ne sera pas sauvegardé à la fermeture de RStudio.

B.7 Aide et documentation

La documentation de RStudio se trouve entièrement en ligne. On y accède par le menu `Help`. L'onglet `Help` du navigateur de fichiers (sous-fenêtre en bas à droite) offre également une interface unifiée pour accéder à l'aide de R et à celle de RStudio.

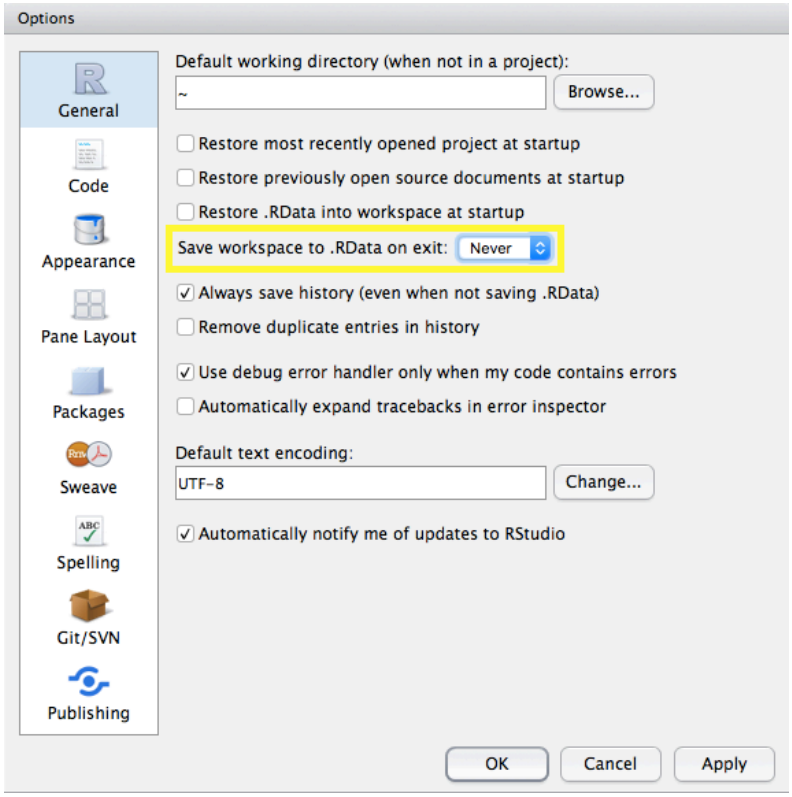


FIG. B.2 – Réglage suggéré de RStudio (encadré en jaune) faisant en sorte que l'espace de travail de R n'est jamais sauvegardé au moment de quitter l'application

C Planification d'une simulation en R

Il existe de multiples façons de réaliser la mise en œuvre informatique d'une simulation, mais certaines sont plus efficaces que d'autres. Cette annexe passe en revue diverses façons de faire des simulations avec R à l'aide d'un exemple simple de nature statistique.

C.1 Contexte

Soit $X_1, \ldots, X_n$ un échantillon aléatoire tiré d'une population distribuée selon une loi uniforme sur l'intervalle $(\theta - \frac{1}{2}, \theta + \frac{1}{2})$. On considère trois estimateurs sans biais du paramètre inconnu θ :

1. la moyenne arithmétique
$$\hat{\theta}_1 = \frac{1}{n} \sum_{i=1}^{n} X_i ;$$
2. la médiane empirique
$$\hat{\theta}_2 = \begin{cases} X_{(\frac{n+1}{2})}, & n \text{ impair} \\ \frac{1}{2}(X_{(\frac{n}{2})} + X_{(\frac{n}{2}+1)}), & n \text{ pair,} \end{cases}$$
où $X_{(k)}$ est la k^{e} statistique d'ordre de l'échantillon aléatoire ;
3. la mi-étendue
$$\hat{\theta}_3 = \frac{X_{(1)} + X_{(n)}}{2}.$$

À l'aide de la simulation on veut, d'une part, vérifier si les trois estimateurs sont bel et bien sans biais et, d'autre part, déterminer lequel a la plus faible variance.

Pour ce faire, on doit d'abord simuler un grand nombre N d'échantillons aléatoires de taille n d'une distribution $U(\theta - \frac{1}{2}, \theta + \frac{1}{2})$ pour une valeur de θ

choisie. Pour chaque échantillon, on calculera ensuite les trois estimateurs ci-dessus, puis la moyenne et la variance, par type d'estimateur, de tous les estimateurs obtenus. Si la moyenne des N estimateurs $\hat{\theta}_i$, $i = 1, 2, 3$ est près de θ, alors on pourra conclure que $\hat{\theta}_i$ est sans biais. De même, on déterminera lequel des trois estimateurs a la plus faible variance selon le classement des variances empiriques.

C.2 Première approche : avec une boucle

La façon la plus intuitive de mettre en œuvre cette étude de simulation en R consiste à utiliser une boucle `for`. Avec cette approche, il est nécessaire d'initialiser une matrice de 3 lignes et N colonnes (ou l'inverse) dans laquelle seront stockées les valeurs des trois estimateurs pour chaque simulation. Une fois la matrice remplie dans la boucle, il ne reste plus qu'à calculer la moyenne et la variance par ligne pour obtenir les résultats souhaités.

La figure C.1 présente un exemple de code adéquat pour réaliser la simulation à l'aide d'une boucle.

Si l'on souhaite pouvoir exécuter le code de la figure C.1 facilement à l'aide d'une seule expression, il suffit de placer l'ensemble du code dans une fonction. La fonction `simul1` de la figure C.2 reprend le code de la figure C.1, sans les commentaires. On a alors :

```
> simul1(10000, 100, 0)

$biais
      Moyenne       Mediane    Mi-etendue
-1.025386e-04 -4.106166e-04 -3.382494e-05

$variances
     Moyenne      Mediane   Mi-etendue
8.243552e-04 2.390898e-03 4.784212e-05
```

C.3 Seconde approche : avec `sapply`

On le sait, les boucles sont inefficaces en R. Il est en général plus efficace de déléguer les boucles aux fonctions `lapply` et `sapply` (section 6.3). On rappelle que la syntaxe de ces fonctions est

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

```
## Bonne habitude à prendre: stocker les constantes dans
## des variables faciles à modifier au lieu de les écrire
## explicitement dans le code.
size <- 100                # taille de chaque échantillon
nsimul <- 10000            # nombre de simulations
theta <- 0                 # la valeur du paramètre

## Les lignes ci-dessous éviteront de faire deux additions
## 'nsimul' fois.
a <- theta - 0.5           # borne inférieure de l'uniforme
b <- theta + 0.5           # borne supérieure de l'uniforme

## Initialisation de la matrice dans laquelle seront
## stockées les valeurs des estimateurs. On donne également
## des noms aux lignes de la matrice afin de facilement
## identifier les estimateurs.
x <- matrix(0, nrow = 3, ncol = nsimul)
rownames(x) <- c("Moyenne", "Mediane", "Mi-etendue")

## Simulation comme telle.
for (i in 1:nsimul)
{
    u <- runif(size, a, b)
    x[, i] <- c(mean(u),        # moyenne
                median(u)       # médiane
                mean(range(u))) # mi-étendue
}

## On peut maintenant calculer la moyenne et la variance
## par ligne.
rowMeans(x) - theta        # vérification du biais
apply(x, 1, var)           # comparaison des variances
```

FIG. C.1 – Code pour la simulation utilisant une boucle `for`

```
simul1 <- function(nsimul, size, theta)
{
    a <- theta - 0.5
    b <- theta + 0.5

    x <- matrix(0, nrow = 3, ncol = nsimul)
    rownames(x) <- c("Moyenne", "Mediane", "Mi-etendue")

    for (i in 1:nsimul)
    {
        u <- runif(size, a, b)
        x[, i] <- c(mean(u), median(u), mean(range(u)))
    }

    list(biais = rowMeans(x) - theta,
         variances = apply(x, 1, var))
}
```

FIG. C.2 – Définition de la fonction `simul1`

Ces fonctions appliquent la fonction `FUN` à tous les éléments de la liste ou du vecteur `x` et retournent les résultats sous forme de liste (`lapply`) ou, lorsque c'est possible, de vecteur ou de matrice (`sapply`). Il est important de noter que les valeurs successives de `x` seront passées comme *premier* argument à la fonction `FUN`. Le cas échéant, les autres arguments de `FUN` sont spécifiés dans le champ '`...`'.

Pour pouvoir utiliser ces fonctions dans le cadre d'une simulation comme celle dont il est question ici, il s'agit de définir une fonction qui fera tous les calculs pour une simulation, puis de la passer à `sapply` pour obtenir les résultats de N simulations. La figure C.3 présente une première version d'une telle fonction. On remarquera que l'argument `i` ne joue aucun rôle dans la fonction. Voici un exemple d'utilisation pour un petit nombre de simulations (4) :

```
> sapply(1:4, fun1, size = 10, a = -0.5, b = 0.5)

                [,1]      [,2]       [,3]      [,4]
Moyenne    0.1215437 0.1674711 0.07462030 0.1423579
Mediane    0.1303616 0.1466585 0.16737903 0.1850391
```

```
fun1 <- function(i, size, a, b)
{
    u <- runif(size, a, b)
    c(Moyenne = mean(u),
      Mediane = median(u),
      "Mi-etendue" = mean(range(u)))
}

simul2 <- function(nsimul, size, theta)
{
    a <- theta - 0.5
    b <- theta + 0.5

    x <- sapply(1:nsimul, fun1, size, a, b)

    list(biais = rowMeans(x) - theta,
         variances = apply(x, 1, var))
}
```

FIG. C.3 – Définitions des fonction `fun1` et `simul2`

```
Mi-etendue 0.1259605 0.1157009 0.06707446 0.0231649
```

On remarque donc que les résultats de chaque simulation se trouvent dans les colonnes de la matrice obtenue avec `sapply`.

Pour compléter l'analyse, on englobe le tout dans une fonction `simul2`, dont le code se trouve à la figure C.3 :

```
> simul2(10000, 100, 0)

$biais
      Moyenne       Mediane    Mi-etendue
-1.185861e-04 -4.300294e-05 -7.561430e-05

$variances
     Moyenne      Mediane   Mi-etendue
8.448004e-04 2.484321e-03 4.867494e-05
```

Il est généralement plus facile de déboguer le code avec cette approche puisque l'on peut rapidement circonscrire un éventuel problème à `fun1` ou

simul2.

C.4 Variante de la seconde approche

Une chose manque d'élégance dans la seconde approche : l'obligation d'inclure un argument factice dans la fonction fun1. La fonction replicate (section 6.5) permet toutefois de passer outre cette contrainte. En effet, cette fonction exécute un nombre donné de fois une expression quelconque.

Les fonctions fun2 et simul3 de la figure C.4 sont des versions légèrement modifiées de fun1 et simul2 pour utilisation avec replicate. On a alors

```
> simul3(10000, 100, 0)

$biais
      Moyenne       Mediane    Mi-etendue
 4.847782e-05 -5.033418e-05 -4.604339e-05

$variances
     Moyenne      Mediane   Mi-etendue
8.225929e-04 2.406800e-03 5.088608e-05
```

C.5 Gestion des fichiers

Pour un petit projet comme celui utilisé en exemple ici, il est simple et pratique de placer tout le code informatique dans un seul fichier de script. Pour un plus gros projet, cependant, il vaut souvent mieux avoir recours à plusieurs fichiers différents. Le présent auteur utilise pour sa part un fichier par fonction.

À des fins d'illustration, supposons que l'on utilise l'approche de la section C.4 avec la fonction replicate et que le code des fonctions fun2 et simul3 est sauvegardé dans des fichiers fun2.R et simul3.R, dans l'ordre. Si l'on crée un autre fichier, disons go.R, ne contenant que des expressions source pour lire les autres fichiers, il est alors possible de démarrer des simulations en exécutant ce seul fichier. Dans notre exemple, le fichier go.R contiendrait les lignes suivantes :

```
source("fun2.R")
source("simul3.R")
simul3(10000, 100, 0)
```

```
fun2 <- function(size, a, b)
{
    u <- runif(size, a, b)
    c(Moyenne = mean(u),
      Mediane = median(u),
      "Mi-etendue" = mean(range(u)))
}

simul3 <- function(nsimul, size, theta)
{
    a <- theta - 0.5
    b <- theta + 0.5

    x <- replicate(nsimul, fun2(size, a, b))

    list(biais = rowMeans(x) - theta,
         variances = apply(x, 1, var))
}
```

FIG. C.4 – Définitions des fonction `fun2` et `simul3`

Une simple commande

```
> source("go.R")
```

exécutera alors une simulation complète.

C.6 Exécution en lot

Les utilisateurs plus avancés pourront vouloir exécuter leur simulation R en lot (*batch*) pour en accélérer le traitement. Dans ce mode, aucune interface graphique n'est démarrée et tous les résultats sont redirigés vers un fichier pour consultation ultérieure. Pour les simulations demandant un long temps de calcul, c'est très pratique.

On exécute R en lot depuis la ligne de commande (Invite de commande sous Windows, Terminal sous macOS ou Linux). Une fois placé dans le répertoire contenant les fichiers de script, il suffit d'entrer à la ligne de commande

```
R CMD BATCH go.R
```

La sortie de cette commande (et donc tous les résultats des expressions R du fichier `go.R`) seront placés par défaut dans le fichier `go.Rout`. Sous Windows, le dossier d'installation de R peut ne pas se trouver dans la variable d'environnement `%PATH%`, auquel cas il faut spécifier le chemin d'accès complet de l'exécutable à la ligne de commande :

```
"c:\Program Files\R\R-x.y.z\bin\R" CMD BATCH go.R
```

Remplacer `R-x.y.z` par le numéro de version courant de R.

C.7 Conclusion

Le nombre de simulations, N, et la taille de l'échantillon, n, ont tous deux un impact sur la qualité des résultats, mais de manière différente. Quand n augmente, la précision des estimateurs augmente. Ainsi, dans l'exemple ci-dessus, le biais et la variance des estimateurs de θ seront plus faibles. D'autre part, l'augmentation du nombre de simulations diminue l'impact des échantillons aléatoires individuels et, de ce fait, améliore la fiabilité des conclusions de l'étude.

D'ailleurs, les conclusions de l'étude de simulation sur le biais et la variance des trois estimateurs de la moyenne d'une loi uniforme sont les suivantes : les trois estimateurs sont sans biais et la mi-étendue a la plus faible variance. En effet, on peut démontrer mathématiquement que, pour n impair,

$$\begin{aligned}
\mathrm{Var}[\hat{\theta}_1] &= \frac{1}{12n} \\
\mathrm{Var}[\hat{\theta}_2] &= \frac{1}{4n+2} \\
\mathrm{Var}[\hat{\theta}_3] &= \frac{1}{2(n+1)(n+2)}
\end{aligned}$$

et donc

$$\mathrm{Var}[\hat{\theta}_3] \leq \mathrm{Var}[\hat{\theta}_1] \leq \mathrm{Var}[\hat{\theta}_2]$$

pour tout $n \geq 2$.

D Installation de packages dans R

Un package R est un ensemble cohérent de fonctions, de jeux de données et de documentation permettant de compléter les fonctionnalités du système de base ou d'en ajouter de nouvelles. Les packages sont normalement installés depuis le site *Comprehensive R Archive Network* (CRAN ; http://cran.r-project.org).

Cette annexe explique comment configurer R pour faciliter l'installation et l'administration de packages externes.

Les instructions ci-dessous se basent sur la création d'une bibliothèque personnelle où seront installés les packages R téléchargés de CRAN. Il est fortement recommandé de créer une telle bibliothèque. Cela permet d'éviter les éventuels problèmes d'accès en écriture dans la bibliothèque principale et de conserver les packages intacts lors des mises à jour de R. On montre également comment spécifier le site miroir de CRAN pour éviter d'avoir à le répéter à chaque installation de package.

1. Identifier le dossier de départ de l'utilisateur. En cas d'incertitude, examiner la valeur de la variable d'environnement `HOME`[1], depuis R avec la commande

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

   ou, pour les utilisateurs de Emacs, directement depuis l'éditeur avec

   ```
   M-x getenv RET HOME RET
   ```

   Nous référerons à ce dossier par le symbole ~.

2. Créer un dossier qui servira de bibliothèque de packages personnelle. Dans la suite, nous utiliserons `~/R/library`.

1. Pour les utilisateurs de GNU Emacs sous Windows, la variable est créée par l'assistant d'installation de Emacs lorsqu'elle n'existe pas déjà.

3. La configuration de R se fait à l'aide simples fichiers texte, comme pour GNU Emacs ; voir la section A.7. Dans un fichier nommé `~/.Renviron` (donc situé dans le dossier de départ), enregistrer la ligne suivante :

   ```
   R_LIBS_USER="~/R/library"
   ```

   Au besoin, remplacer le chemin `~/R/library` par celui du dossier créé à l'étape précédente. Utiliser la barre oblique avant (/) dans le chemin pour séparer les dossiers.

Sous macOS, ajouter dans le fichier `~/.Renviron` la ligne suivante :

```
R_INTERACTIVE_DEVICE=quartz
```

Ainsi, R utilisera toujours l'interface Quartz native de macOS pour afficher les graphiques.

4. Dans un fichier nommé `~/.Rprofile`, enregistrer l'option suivante :

   ```
   options(repos = "http://cran.ca.r-project.org")
   ```

   Si désiré, remplacer la valeur de l'option `repos` par l'URL d'un autre site miroir de CRAN.

   Les utilisateurs de GNU Emacs voudront ajouter une option pour éviter que R ait recours aux menus graphiques Tcl/Tk. Le code à entrer dans le fichier `~/.Rprofile` sera plutôt

   ```
   options(repos = "http://cran.ca.r-project.org",
           menu.graphics = FALSE)
   ```

Consulter la rubriques d'aide de `Startup` pour les détails sur la syntaxe et l'emplacement des fichiers de configuration, celles de `library` et `.libPaths` pour la gestion des bibliothèques et celle de `options` pour les différentes options reconnues par R.

Après un redémarrage de R, la bibliothèque personnelle aura préséance sur la bibliothèque principale et il ne sera plus nécessaire de préciser le site miroir de CRAN lors de l'installation de packages. Ainsi, la simple commande

```
> install.packages("actuar")
```

téléchargera le package de fonctions actuarielles **actuar** depuis le miroir canadien de CRAN et l'installera dans le dossier `~/R/library`. Pour charger le package en mémoire, on fera

```
> library("actuar")
```

On peut arriver au même résultat sans utiliser les fichiers de configuration `.Renviron` et `.Rprofile`. Il faut cependant recourir aux arguments `lib` et `repos` de la fonction `install.packages` et à l'argument `lib.loc` de la fonction `library`. Consulter les rubriques d'aide de ces deux fonctions pour de plus amples informations.

Réponses des exercices

Chapitre 2

2.1 a) Il y a plusieurs façons de créer les troisième et quatrième éléments de la liste. Le plus simple consiste à utiliser `numeric()` et `logical()` :

```
> x <- list(1:5, data = matrix(1:6, 2, 3), numeric(3),
+           test = logical(4))
```

b)
```
> names(x)
```

c)
```
> mode(x$test)
> length(x$test)
```

d)
```
> dim(x$data)
```

e)
```
> x[[2]][c(2, 3)]
```

f)
```
> x[[3]] <- 3:8
```

2.2 a)
```
> x[2]
```

b)
```
> x[1:5]
```

c)
```
> x[x > 14]
```

d)
```
> x[-c(6, 10, 12)]
```

2.3 a)
```
> x[4, 3]
```

b)
```
> x[6, ]
```

c)
```
> x[, c(1, 4)]
```

d)
```
> x[x[, 1] > 50, ]
```

Chapitre 3

3.1 a)
```
> rep(c(0, 6), 3)
```

b)
```
> seq(1, 10, by = 3)
```

c)
```
> rep(1:3, 4)
```

d)
```
> rep(1:3, 1:3)
```

e)
```
> rep(1:3, 3:1)
```

f)
```
> seq(1, 10, length = 3)
```

g)
```
> rep(1:3, rep(4,3))
```

3.2 a)
```
> 11:20 / 10
```

b)
```
> 2 * 0:9 + 1
```

c)
```
> rep(-2:2, 2)
```

d)
```
> rep(-2:2, each = 2)
```

e)
```
> 10 * 1:10
```

3.3 Soit `mat` une matrice.

a)
```
> apply(mat, 1, sum)
```

b)
```
> apply(mat, 2, sum)
```

c)
```
> apply(mat, 1, mean)
```

d)
```
> apply(mat, 2, mean)
```

3.4
```
> cumprod(1:10)
```

3.5
```
x == (x %% y) + y * (x %/% y)
```

3.6 a)
```
> x[1:5]
> head(x, 5)
```

b)
```
> max(x)
```

c)
```
> mean(x[1:5])
> mean(head(x, 5))
```

d)
```
> mean(x[16:20])
> mean(x[(length(x) - 4):length(x)])  # plus général
> mean(tail(x, 5))                    # plus lisible!
```

3.7 a)
```
(j - 1)*I + i
```

b)
```
((k - 1)*J + j - 1)*I + i
```

3.8 a)
```
> rowSums(mat)
```

b)
```
> colMeans(mat)
```

c)
```
> max(mat[1:3, 1:3])
```

d)
```
> mat[rowMeans(mat) > 7,]
```

3.9
```
> temps[match(unique(cummin(tps)), temps)]
```

Chapitre 4

4.1
```
> sum(P / cumprod(1 + i))
```

4.2
```
> x <- c(7, 13, 3, 8, 12, 12, 20, 11)
> w <- c(0.15, 0.04, 0.05, 0.06, 0.17, 0.16, 0.11, 0.09)
> sum(x * w)/sum(w)
```

4.3
```
> 1/mean(1/x)
```

4.4
```
> lambda <- 2
> x <- 5
> exp(-lambda) * sum(lambda^(0:x)/gamma(1 + 0:x))
```

4.5 a)
```
> x <- 10^(0:6)
> probs <- (1:7)/28
```

b)
```
> sum(x^2 * probs) - (sum(x * probs))^2
```

4.6
```
> i <- 0.06
> 4 * ((1 + i)^0.25 - 1)
```

4.7
```
> n <- 1:10
> i <- seq(0.05, 0.1, by = 0.01)
> (1 - outer((1 + i), -n, "^"))/i
```

ou

```
> n <- 1:10
> i <- (5:10)/100
> apply(outer(1/(1+i), n, "^"), 1, cumsum)
```

4.8
```
> v <- 1/1.06
> k <- 1:10
> sum(k * v^(k - 1))
```

4.9
```
> pmts <- rep(1:4, 1:4)
> v <- 1/1.07
> k <- 1:10
> sum(pmts * v^k)
```

4.10
```
> v <- cumprod(1/(1 + rep(c(0.05, 0.08), 5)))
> pmts <- rep(1:4, 1:4)
> sum(pmts * v)
```

Chapitre 5

5.1
```
variance <- function(x, biased = FALSE)
{
    if (biased)
    {
        n <- length(x)
        (n - 1)/n * var(x)
    }
    else
        var(x)
}
```

5.2 Une première solution utilise la transposée. La première expression de la fonction s'assure que la longueur de `data` est compatible avec le nombre de lignes et de colonnes de la matrice demandée.

```
matrix2 <- function(data = NA, nrow = 1, ncol = 1,
                    bycol = FALSE, dimnames = NULL)
{
    data <- rep(data, length = nrow * ncol)

    if (bycol)
        dim(data) <- c(nrow, ncol)
    else
    {
        dim(data) <- c(ncol, nrow)
        data <- t(data)
    }

    dimnames(data) <- dimnames
    data
}
```

La seconde solution n'a pas recours à la transposée. Pour remplir la matrice par ligne, il suffit de réordonner les éléments du vecteur `data` en utilisant la formule obtenue à l'exercice 3.7.

```
matrix2 <- function(data = NA, nrow = 1, ncol = 1,
                    bycol = FALSE, dimnames = NULL)
{
```

```
    data <- rep(data, length = nrow * ncol)

    if (!bycol)
    {
        i <- 1:nrow
        j <- rep(1:ncol, each = nrow)
        data <- data[(i - 1)*ncol + j]
    }
    dim(data) <- c(nrow, ncol)
    dimnames(data) <- dimnames
    data
}
```

5.3
```
phi <- function(x)
{
    exp(-x^2/2) / sqrt(2 * pi)
}
```

5.4
```
Phi <- function(x)
{
    n <- 1 + 2 * 0:50
    0.5 + phi(x) * sum(x^n / cumprod(n))
}
```

5.5 La première solution utilise une fonction interne et une structure de contrôle `if ... else`.

```
Phi <- function(x)
{
    fun <- function(x)
    {
        n <- 1 + 2 * 0:50
        0.5 + phi(x) * sum(x^n / cumprod(n))
    }

    if (x < 0)
        1 - fun(-x)
    else
        fun(x)
}
```

Seconde solution sans structure de contrôle `if ... else`. Rappelons que dans des calculs algébriques, `FALSE` vaut 0 et `TRUE` vaut 1.

```
Phi <- function(x)
{
    n <- 1 + 2 * 0:50
    neg <- x < 0
    x <- abs(x)
    neg + (-1)^neg * (0.5 + phi(x) *
                      sum(x^n / cumprod(n)))
}
```

5.6

```
Phi <- function(x)
{
    n <- 1 + 2 * 0:30
    0.5 + phi(x) * colSums(t(outer(x, n, "^")) /
                           cumprod(n))
}
```

5.7 a)

```
prod.mat <- function(mat1, mat2)
{
    if (ncol(mat1) == nrow(mat2))
    {
        res <- matrix(0, nrow = nrow(mat1),
                      ncol = ncol(mat2))
        for (i in 1:nrow(mat1))
        {
            for (j in 1:ncol(mat2))
            {
                res[i, j] <- sum(mat1[i,] * mat2[,j])
            }
        }
        res
    }
    else
        stop("Les dimensions des matrices ne
permettent pas le produit matriciel.")
}
```

b)
```
prod.mat<-function(mat1, mat2)
{
    if (ncol(mat1) == nrow(mat2))
    {
        res <- matrix(0, nrow = nrow(mat1),
                      ncol = ncol(mat2))
        for (i in 1:nrow(mat1))
            res[i,] <- colSums(mat1[i,] * mat2)
        res
    }
    else
        stop("Les dimensions des matrices ne
permettent pas le produit matriciel.")
}
```

Solutions bonus : deux façons de faire équivalentes qui cachent la boucle dans un `sapply`.

```
prod.mat<-function(mat1, mat2)
{
    if (ncol(mat1) == nrow(mat2))
        t(sapply(1:nrow(mat1),
                 function(i) colSums(mat1[i,] * mat2)))
    else
        stop("Les dimensions des matrices ne permettent
pas le produit matriciel.")
}

prod.mat<-function(mat1, mat2)
{
    if (ncol(mat1) == nrow(mat2))
        sapply(1:ncol(mat2),
               function(j) colSums(t(mat1) * mat2[,j]))
    else
        stop("Les dimensions des matrices ne permettent
pas le produit matriciel.")
}
```

5.8 Le calcul à faire n'est qu'un simple produit matriciel, donc :

```
notes.finales <- function(notes, p) notes %*% p
```

5.10
```
param <- function (moyenne, variance, loi)
{
    loi <- tolower(loi)
    if (loi == "normale")
    {
        param1 <- moyenne
        param2 <- sqrt(variance)
        return(list(mean = param1, sd = param2))
    }
    if (loi == "gamma")
    {
        param2 <- moyenne/variance
        param1 <- moyenne * param2
        return(list(shape = param1, scale = param2))
    }
    if (loi == "pareto")
    {
        cte <- variance/moyenne^2
        param1 <- 2 * cte/(cte-1)
        param2 <- moyenne * (param1 - 1)
        return(list(alpha = param1, lambda = param2))
    }
    stop("La loi doit etre une de \"normale\",
\"gamma\" ou \"pareto\"")
}
```

Chapitre 6

6.1 Soit `Xij` et `wij` des matrices, et `Xijk` et `wijk` des tableaux à trois dimensions.

a)
```
> rowSums(Xij * wij)/rowSums(wij)
```

b)
```
> colSums(Xij * wij)/colSums(wij)
```

c)
```
> sum(Xij * wij)/sum(wij)
```

d)
```
> apply(Xijk * wijk, c(1, 2), sum) /
+     apply(wijk, c(1, 2), sum)
```

e)
```
> apply(Xijk * wijk, 1, sum)/apply(wijk, 1, sum)
```

f)
```
> apply(Xijk * wijk, 2, sum)/apply(wijk, 2, sum)
```

g)
```
> sum(Xijk * wijk)/sum(wijk)
```

6.2 a)
```
> unlist(lapply(0:10, seq, from = 0))
```

b)
```
> unlist(lapply(1:10, seq, from = 10))
```

c)
```
> unlist(lapply(10:1, seq, to = 1))
```

6.3 a)
```
> x <- lapply(seq(100, 300, by = 50), rpareto,
+             shape = 2, scale = 5000)
```

b)
```
> names(x) <- paste("sample", 1:5, sep = "")
```

c)
```
> sapply(x, mean)
```

d)
```
> lapply(x, function(x) sort(ppareto(x, 2, 5000)))
> lapply(lapply(x, sort), ppareto,
+        shape = 2, scale = 5000)
```

e)
```
> hist(x$sample5)
```

f)
```
> lapply(x, "+", 1000)
```

6.4 a)
```
> mean(sapply(x, function(liste) liste$franchise))
```

Les crochets utilisés pour l'indiçage constituent en fait un opérateur dont le « nom » est `[[`. On peut donc utiliser cet opérateur dans la fonction `sapply` :

```
> mean(sapply(x, "[[", "franchise"))
```

b)
```
> sapply(x, function(x) mean(x$nb.acc))
```

c)
```
> sum(sapply(x, function(x) sum(x$nb.acc)))
```

ou

```
> sum(unlist(sapply(x, "[[", "nb.acc")))
```

d)
```
> mean(unlist(lapply(x, "[[", "montants")))
```

e)
```
> sum(sapply(x, function(x) sum(x$nb.acc) == 0))
```

f)
```
> sum(sapply(x, function(x) x$nb.acc[1] == 1))
```

g)
```
> var(unlist(lapply(x, function(x) sum(x$nb.acc))))
```

h)
```
> sapply(x, function(x) var(x$nb.acc))
```

i)
```
> y <- unlist(lapply(x, "[[", "montants"))
> sum(y <= x)/length(y)
```

La fonction `ecdf` retourne une fonction permettant de calculer la fonction de répartition empirique en tout point :

```
> ecdf(unlist(lapply(x, "[[", "montants")))(x)
```

j)
```
> y <- unlist(lapply(x, "[[", "montants"))
> colSums(outer(y, x, "<="))/length(y)
```

La fonction retournée par `ecdf` accepte un vecteur de points en argument :

```
> ecdf(unlist(lapply(x, "[[", "montants")))(x)
```

Chapitre 7

7.1 a)
```
> f <- function(x) x^3 - 2 * x^2 - 5
> uniroot(f, lower = 1, upper = 4)
```

b) Comme un simple graphique le démontre, il y a deux racines dans l'intervalle.

```
> f <- function(x) x^3 + 3 * x^2 - 1
> curve(f, xlim = c(-4, 0))
> uniroot(f, lower = -4, upper = -1)
> uniroot(f, lower = -1, upper = 0)
```

c)
```
> f <- function(x) x - 2^(-x)
> uniroot(f, lower = 0, upper = 1)
```

d)
```
> f <- function(x) exp(x) + 2^(-x) + 2 * cos(x) - 6
> uniroot(f, lower = 1, upper = 2)
```

e)
```
> f <- function(x) exp(x) - x^2 + 3 * x - 2
> uniroot(f, lower = 0, upper = 1)
```

7.2
```
> X <- c(2061, 1511, 1806, 1353, 1600)
> w <- c(100155, 19895, 13735, 4152, 36110)
> g <- function(a, X, w, s2)
+ {
+     z <- 1/(1 + s2/(a * w))
+     Xz <- sum(z * X)/sum(z)
+     sum(z * (X - Xz)^2)/(length(X) - 1)
+ }
> uniroot(function(x) g(x, X, w, 140E6) - x, c(50000, 80000))
```

7.3
```
> dpareto <- function(x, alpha, lambda)
+ {
+     (alpha * lambda^alpha)/(x + lambda)^(alpha+1)
+ }
> f <- function(par, x) -sum(log(dpareto(x, par[1], par[2])))
> optim(c(1, 1000), f, x = x)
```

ou, en utilisant le truc du logarithme des paramètres expliqué dans le code informatique de la section 7.4 pour éviter les soucis de convergence :

```
> dpareto <- function(x, logAlpha, logLambda)
+ {
+     alpha <- exp(logAlpha)
+     lambda <- exp(logLambda)
+     (alpha * lambda^alpha)/(x + lambda)^(alpha+1)
```

```
+ }
> optim(c(log(2), log(1000)), f, x = x)
> exp(optim(c(log(2), log(1000)), f, x = x)$par)
```

Chapitre 8

8.1
```
> x <- rlnorm(1000, meanlog = log(5000) - 0.5, sdlog = 1)
> hist(x)
```

8.2
```
> x <- rpois(10000, lambda = rgamma(10000, shape = 5, rate = 4))
> xx <- seq(min(x), max(x))
> px <- table(x)
> plot(xx, dnbinom(xx, size = 5, prob = 0.8),
+      type = "h", lwd = 5, col = "blue")
> points(xx, px/length(x), pch = 16)
```

8.3
```
> w <- rbinom(1, 10000, 0.55)
> x <- c(rlnorm(w, 3.5, 0.6), rlnorm(10000 - w, 4.6, 0.3))
> hist(x)
```

Bibliographie

Abelson, H., G. J. Sussman et J. Sussman. 1996, *Structure and Interpretation of Computer Programs*, 2e éd., MIT Press, ISBN 0-26201153-0.

Becker, R. A. 1994, «A brief history of S», cahier de recherche, AT&T Bell Laboratories. URL http://cm.bell-labs.com/cm/ms/departments/sia/doc/94.11.ps.

Becker, R. A. et J. M. Chambers. 1984, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, ISBN 0-53403313-X.

Becker, R. A., J. M. Chambers et A. R. Wilks. 1988, *The New S Language: A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole, ISBN 0-53409192-X.

Braun, W. J. et D. J. Murdoch. 2007, *A First Course in Statistical Programming with R*, Cambridge University Press, ISBN 978-0-52169424-7.

Cameron, D., J. Elliott, M. Loy, E. S. Raymond et B. Rosenblatt. 2004, *Leaning GNU Emacs*, 3e éd., O'Reilly, Sebastopol, CA, ISBN 0-59600648-9.

Chambers, J. M. 1998, *Programming with Data: A Guide to the S Language*, Springer, ISBN 0-38798503-4.

Chambers, J. M. 2000, «Stages in the evolution of S», URL http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html.

Chambers, J. M. 2008, *Software for Data Analysis: Programming with R*, Springer, ISBN 978-0-38775935-7.

Chambers, J. M. et T. J. Hastie. 1992, *Statistical Models in S*, Wadsworth & Brooks/Cole, ISBN 0-53416765-9.

Dutang, C., V. Goulet et M. Pigeon. 2008, «actuar: An R package for actuarial science», *Journal of Statistical Software*, vol. 25, nº 7. URL http://www.jstatsoft.org/v25/i07.

Hornik, K. 2013, «The R FAQ», URL http://cran.r-project.org/doc/FAQ/R-FAQ.html.

Iacus, S. M., S. Urbanek, R. J. Goedman et B. D. Ripley. 2016, «R for Mac OS X FAQ», URL http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html.

IEEE. 2003, *754-1985 IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, Piscataway, NJ.

Ihaka, R. et R. Gentleman. 1996, «R: A language for data analysis and graphics», *Journal of Computational and Graphical Statistics*, vol. 5, nº 3, p. 299–314.

Ligges, U. 2003, «R-winedt», dans *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, édité par K. Hornik, F. Leisch et A. Zeileis, TU Wien, Vienna, Austria, ISSN 1609-395X. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

Redd, A. 2010, «Introducing NppToR: R interaction for Notepad++», *R Journal*, vol. 2, nº 1, p. 62–63. URL http://journal.r-project.org/archive/2010-1/RJournal_2010-1.pdf.

Ripley, B. D. et D. J. Murdoch. 2013, «R for Windows FAQ», URL http://cran.r-project.org/bin/windows/base/rw-FAQ.html.

Venables, W. N. et B. D. Ripley. 2000, *S Programming*, Springer, New York, ISBN 0-38798966-8.

Venables, W. N. et B. D. Ripley. 2002, *Modern Applied Statistics with S*, 4e éd., Springer, New York, ISBN 0-38795457-0.

Venables, W. N., D. M. Smith et R Core Team. 2013, *An Introduction to R*, R Foundation for Statistical Computing. URL http://cran.r-project.org/doc/manuals/R-intro.html.

Wheeler, B. 2013, *SuppDists: Supplementary Distributions*. URL http://cran.r-project.org/package=SuppDists, R package version 1.1-9.

Index

Les numéros de page en caractères gras indiquent les pages où les concepts sont introduits, définis ou expliqués.

`!`, **51**
`!=`, **51**
`*`, **51**
`+`, **51**
`-`, **51**
`->`, 17, 51
`->>`, **51**
`-Inf`, **22**
`...`, **106**, 170
`/`, **51**
`:`, 51
`:`, 72
`;`, 17
`<`, **51**
`<-`, **17**, 51
`<<-`, **51**, 92
`<=`, **51**
`=`, 17
`==`, **51**
`>`, **51**
`>=`, **51**
`[`, **32**
`[<-`, **32**
`[[`, 188
`[[ ]]`, **29**, 29
`[ ]`, 24, 26, 29, **32**
`$`, **30**, 32, 51
`$<-`, 32
`%*%`, **51**, 102
`%/%`, **51**
`%%`, **51**
`%in%`, **55**, 68
`%o%`, **60**, 70
`&`, **51**
`&&`, **51**
`^`, **51**
`{ }`, **18**
`||`, **51**
`|`, **51**

`abs`, 98, 99, 123, 124
`add`, 124
affectation, 16
`apply`, 61, 70, 73, **107**, 107, 108, 110, 117, 118, 169–171, 173
`array`, **25**, 42, 117, 118
`array` (classe), 25
arrondi, 55
`as.data.frame`, **30**
`attach`, **30**, 44
`attr`, **23**, 39, 126
attribut, **23**
`attributes`, **23**, 39, 40

biscuits, *voir* Syndrôme de la plaque à biscuits
boucle, 61, 85, 168
`break`, **62**, 72, 98, 99, 123
`by`, 12, 67
`byrow`, 26, 52

`c`, **24**
`cat`, 124
`cbind`, **28**, 30, 41, 46, 69
`ceiling`, **56**, 68
`character`, **24**, 40
`character` (mode), **20**, 24
`choose`, 77
`class`, 39, 41, 42, 44, 114, 123, 124
`class` (attribut), **23**
`colMeans`, **58**, 73, 107, 117
`colSums`, **58**, 69, 73, 80, 107
compilé (langage), 2
`complex`, 40
`complex` (mode), **20**
`cos`, 36
`cummax`, **57**, 69
`cummin`, **57**, 69
`cumprod`, **57**, 69, 76
`cumsum`, **57**, 69
`curve`, 124

`data`, 39, 52, 67
`data.frame`, **30**
`data.frame` (classe), 30
`dbeta`, 135
`dbinom`, 77
`density`, 12
`det`, **59**
`detach`, **30**, 44
`dgamma`, 136, 137
`diag`, 46, **59**, 69
`diff`, **56**, 69
différences, 56
`dim`, 39, 41, 42, 44, 46, 69, 117
`dim` (attribut), **23**, 25
dimension, 23, 47
`dimnames`, 40, 52
`dimnames` (attribut), **23**
distribution
  binomiale, 77, 143
  binomiale négative, 143
  bêta, 143
  Cauchy, 143
  exponentielle, 143
  F, 143
  gamma, 79, 103, 143
  géométrique, 143
  hypergéométrique, 143
  khi carré, 143
  log-normale, 143, 145
  logistique, 143
  mélange discret, 145
  mélange Poisson/gamma, 145
  normale, 101–103, 143
  Pareto, 103, 127
  Poisson, 78, 86, 143
  t, 143
  uniforme, 143
  Weibull, 143
  Wilcoxon, 143
`dnorm`, 101
dossier de travail, *voir* répertoire de travail
`dpois`, 78

écart type, 56
`ecdf`, 189
`else`, **61**, 70, 71, 99, 123
Emacs, 7, 95, 147–157
  déplacement du curseur, 152
  mode ESS, 7, 153–156
  nouveau fichier, 152
  rechercher et remplacer, 152

sauvegarder, 152
sauvegarder sous, 152
sélection, 153
étiquette, 23, 47
`eval`, 99, 123
`exists`, 44
`exp`, 36, 78, 79, 137
expression, 16
`expression`, 38, 99, 123
`expression` (mode), **20**
extraction, *voir aussi* indiçage
derniers éléments, 54
éléments différents, 54
premiers éléments, 54

`F`, *voir* `FALSE`
`factorial`, 73, **78**
`FALSE`, 19, 95
`floor`, **55**, 68
fonction
anonyme, 93
appel, 52
débogage, 94
définie par l'usager, 91
générique, 114
maximum local, 132
minimum, 132
minimum local, 132
optimisation, 133
racine, 132
résultat, 91
`for`, **61**, 65, 66, 70, 100, 112, 168–170
`function`, **91**, 97–100, 119–124, 126, 134–137, 170, 171, 173
`function` (mode), **20**

`gamma`, 36, 73, **78**

`head`, **54**, 68
`hist`, 122, 128

`if`, **61**, 65, 66, 70–72, 95, 98, 99, 123, 124
`ifelse`, **61**, 116, 117
`Im`, 138
indiçage
liste, **29**, 47
matrice, 26, **32**, 47
vecteur, **32**, 47
`Inf`, **22**
`install.packages`, **63**
interprété (langage), 2
`is.finite`, **22**
`is.infinite`, **22**
`is.na`, **22**, 39, 45, 71
`is.nan`, **22**
`is.null`, **22**

`lapply`, 61, 107, **110**, 110–112, 118, 119, 121, 125, 168, 170
`length`, 12, **20**, 38, 40–43, 45, 67–69
`lfactorial`, 73
`lgamma`, 73
`library`, **62**, 72
`list`, **28**, 38, 40, 42, 43, 118, 119, 123, 125, 170, 171, 173
`list` (mode), **20**, 28
liste, 28
`lm`, 126
`log`, 136, 137
`logical`, **24**, 40
`logical` (mode), **20**, 22, 24
longueur, **21**, 47
`lower`, 134, 135, 137
`ls`, 13, 37, 126

`mapply`, 61, **112**, 121
`match`, **55**, 68
matrice, 73, 101, 102, 107
diagonale, 59

identité, 59
inverse, 59
moyennes par colonne, 58
moyennes par ligne, 58
somme par colonne, 58
sommes par ligne, 58
transposée, 58
matrix, 13, **25**, 36, 41, 43, 65, 67, 101, 117, 118, 169, 170
matrix (classe), 25
max, 12, **56**, 69, 117
maximum
cumulatif, 57
d'un vecteur, 56
local, 132
parallèle, 58
position dans un vecteur, 55
mean, 22, 39, **56**, 57, 69, 117, 119–122, 169–171, 173
median, **57**, 69, 169–171, 173
médiane, 57
methods, **114**
min, 12, **56**, 69
minimum
cumulatif, 57
d'un vecteur, 56
fonction non linéaire, 132
local, 132
parallèle, 58
position dans un vecteur, 55
mode, **20**, 47
mode, **20**, 37, 38, 42–44
moyenne
arithmétique, 56
harmonique, 86
pondérée, 85, 126
tronquée, 56

NA, **22**, 95
na.rm, **22**, 39, 118
names, 40, 44, 45
names (attribut), **23**
NaN, **22**
nchar, **21**, 38
ncol, 13, 41, 52, **58**, 67, 69, 117, 169, 170
next, **62**
nlm, **132**, 132, 136, 137
nlminb, **132**
noms d'objets
conventions, 18
réservés, 19
Notepad++, 9
nrow, 13, 41, 52, **58**, 67, 69, 117, 169, 170
NULL, **21**, 23
NULL (mode), **21**
numeric, **24**, 38, 40, 45, 70, 71, 100
numeric (mode), **20**, 24

optim, **133**, 137
optimize, **132**, 135
order, **54**, 68
ordre, 54
outer, **59**, 59–61, 70, 80, 94, 125

package, **62**
bibliothèque personnelle, 175
installation, 175–177
paste, 128
pgamma, 79, 80
plot, 12, 39, 114, 125, 145
pmax, **58**, 69, 70
pmin, **58**, 69
pnorm, 102
point fixe, 81, 92
points, 124
polyroot, **133**, 138
print, 65, 66, 70, 71, 95, 98, 99, 114, 115, 123, 124

prod, **56**, 69, 70, 117, 118
produit, 56
    cumulatif, 57
    extérieur, 59

q, 10, 126
quantile, 57
quantile, **57**, 69
Quartz, 176

racine
    d'un polynôme, 133
    d'une fonction, 132
rang, 53
range, **57**, 69, 169–171, 173
rank, **53**, 68
rbind, **28**, 30, 41, 46
rbinom, 144
Re, 138
renverser un vecteur, 54
rep, 12, **53**, 67, 70, 72, 118, 121, 122
repeat, **62**, 71, 82, 84, 97–99, 123
répertoire de travail, 10
répétition de valeurs, 53
replace, 45, 69, 119, 121, 145
replicate, **113**, 122, 172, 173
return, **91**
rev, **54**, 68, 69, 77
rgamma, 135, 145
rm, 13, 126
RNGkind, 144
rnorm, 12, 122, 126, 144
round, 12, 13, **55**, 68
row.names, 44
rowMeans, **58**, 73, 107, 169–171, 173
rownames, 169, 170
rowSums, **58**, 69, 73, 107, 117
rpois, 145
RStudio, 9, 159–164
    configuration, 164
    projets, 161
    sous-fenêtres, 160
    symbole d'assignation, 162, 163
runif, 12, 144, 169–171, 173

S, 1, 2
S+, 1
S-PLUS, 1
sample, 39, 45, 69, 73, 110, 117–119, 121, 122, 125, **143**, 145
sapply, 61, 107, **110**, 111–113, 117–122, 125, 168, 170, 171, 186
save.image, 4, 10, 156, 164
Scheme, 2
sd, **56**, 69, 122
search, **62**, 72
seq, 12, 38, 43, **53**, 67, 72, 119
seq_len, **53**
simulation
    nombres uniformes, 141
    planification, 167–174
    variables aléatoires, 142
sin, 36
solve, 13, **59**, 69
somme, 56
    cumulative, 57
sort, **53**, 67
source, 172
stop, 124
structure, 123
style, 95
suite de nombres, 53
suite de nombres, 53
sum, 22, **56**, 69, 70, 117–119, 136, 137
summary, **57**, 69, 114, 125
switch, **61**
Syndrôme de la plaque à biscuits, 83
system.time, 100

T, *voir* TRUE
t, 13, **58**, 69
table, 145
tableau, 73, 107
tail, **54**, 68
tri, 53
TRUE, 19, 95
trunc, **56**, 68
typeof, **21**

unique, **54**, 68
uniroot, **132**, 134
unlist, **30**, 43, 44, 119
upper, 134, 135, 137

valeur actuelle, 76, 85–87
var, **56**, 69, 101, 169–171, 173
variable
    globale, 92
    locale, 92
variance, 56
vecteur, 24, 50
vector, 40, 43
vide, *voir* NULL

which, **54**, 68
which.max, **55**, 68
which.min, **55**, 68
while, **62**, 71, 100
WinEdt, 9

Ce document a été produit avec le système de mise en page XeLaTeX. Le texte principal est en Lucida Bright OT 11 points, les mathématiques en Lucida Bright Math OT, le code informatique en Lucida Grande Mono DK et les titres en Adobe Myriad Pro. Des icônes proviennent de la police Font Awesome. Les graphiques ont été réalisés avec R.

ISBN 978-2-9811416-6-8