

Programmer avec R

Vincent Goulet



UNIVERSITÉ
LAVAL

Programmer avec R

Vincent Goulet

Professeur titulaire
École d'actuariat, Université Laval

© 2017-2024 par Vincent Goulet. *Programmer avec R* est mis à disposition sous licence **Attribution-Partage dans les mêmes conditions 4.0 International** de Creative Commons. En vertu de cette licence, vous êtes autorisé à :

- ▶ **partager** — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats ;
- ▶ **adapter** — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.

L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :



Attribution — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois 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 effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous devez diffuser l'œuvre modifiée dans les mêmes conditions, c'est-à-dire avec la même licence avec laquelle l'œuvre originale a été diffusée.

Code source

 [Voir sur GitLab](#)

Crédits

La citation de Donald E. Knuth en exergue est une traduction libre d'une citation tirée de : D'Agostino, S., « The Computer Scientist Who Can't Stop Telling Stories », *Quanta Magazine*.

Image de peintre utilisée aux figures 2.1 et 2.2 créée par [iconicbestiary/Freepik](#).

Couverture

Groupe d'écureuils de terre du Cap (*Xerus inauris*) sortant de leur terrier près de Solitaire, dans le désert du Namib, en Namibie. Crédit photo : © Hans Hillewaert, [CC BY-SA 3.0](#), via [Wikimedia Commons](#).

Introduction

Cet ouvrage traite de programmation informatique. Au cœur de l'actuelle révolution numérique, la programmation devient une compétence essentielle chez les étudiants comme chez les travailleurs. De même, la place sans cesse grandissante accordée à la science des données — analyse prédictive, apprentissage profond, intelligence artificielle — dans la plupart des disciplines scientifiques ne fait qu'accentuer l'importance de disposer d'une solide formation en programmation.

Les principaux buts du présent ouvrage sont donc : d'abord de développer une culture de l'informatique ; ensuite d'acquérir la capacité à résoudre des problèmes concrets à l'aide de l'algorithmique et de la programmation ; enfin de se familiariser avec les bonnes pratiques reconnues en contexte de travail collaboratif. En effet, la programmation se pratique beaucoup plus souvent en groupe que seul.

Vous apprendrez à programmer en R, le langage au cœur de l'environnement statistique du même nom et l'un des outils les plus utilisés dans le monde pour l'analyse de données. 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 sur l'utilisation de R en sciences naturelles, en sciences sociales, en finance, etc., ne manque pas. Cependant, peu d'ouvrages se concentrent sur l'apprentissage du langage de programmation sous-jacent aux fonctionnalités statistiques. C'est la niche que je tâche d'occuper avec « Programmer avec R ».

L'analyse de données requiert souvent de manipuler ou de traiter du texte. Par conséquent, l'ouvrage aborde également les expressions régulières et divers utilitaires d'analyse et de contrôle de texte.

Enfin, vous trouverez en annexe de brèves introductions à l'environnement de développement intégré RStudio et à l'éditeur de texte pour programmeurs GNU Emacs, de même que les solutions complètes des exercices.

Formations concomitantes

Certaines parties de l'ouvrage tablent sur des connaissances de base dans l'utilisation d'une ligne de commande Unix et d'un système de gestion de versions. Mes

formations concomitantes *Ligne de commande Unix* (Goulet, 2024b) et *Gestion de versions avec Git* (Goulet, 2024a) permettent d'acquérir ces connaissances.

La ligne de commande est la plus ancienne des interfaces avec les ordinateurs. Si les interfaces graphiques ont à plusieurs égards grandement facilité l'interaction homme-machine, elles n'ont pas pour autant fait disparaître ou rendu obsolète la ligne de commande, tout particulièrement dans la pratique de la programmation.

Outil essentiel en contexte de travail collaboratif, le système de gestion de versions facilite le suivi et le partage de fichiers, ainsi que la mise en commun des contributions. Développé par Linus Torvalds pour administrer le code source du noyau du système d'exploitation Linux, Git est aujourd'hui le système de gestion de versions le plus utilisé dans le monde.

Utilisation de l'ouvrage

L'ouvrage vise d'abord à vous exposer à un maximum de code, puis à vous faire pratiquer. C'est pourquoi plusieurs chapitres sont rédigés de manière synthétique et qu'ils comportent peu d'exemples au fil du texte.



En revanche, vous devrez lire et évaluer le code informatique se trouvant dans les sections d'exemples à la fin de la plupart des chapitres. Ce code et les commentaires qui l'accompagnent reviennent sur l'essentiel des concepts du chapitre et les complètent souvent. Je considère l'exercice d'« étude active » consistant à évaluer du code et à voir ses effets comme essentiel à l'apprentissage de la programmation. Vous pouvez aussi inverser la proposition et étudier le code informatique avant le texte du chapitre correspondant si ce mode d'apprentissage vous convient mieux.

Le code des sections d'exemples est distribué avec le document sous forme de fichiers de script. De plus, à chaque fichier `.R` correspond un fichier `.Rout` contenant les résultats de son évaluation non interactive.


Fonctionnalités interactives

En consultation électronique, ce document se trouve enrichi de plusieurs fonctionnalités interactives.

- ▶ Intraliens du texte vers une ligne précise d'une section de code informatique et, en sens inverse, du numéro de la ligne vers le point de la référence dans le texte. Ces intraliens sont marqués par la couleur ■.
- ▶ Intraliens entre le numéro d'un exercice et sa solution, et vice versa. Ces intraliens sont aussi marqués par la couleur ■.
- ▶ Intraliens entre les citations dans le texte et leur entrée dans la bibliographie. Ces intraliens sont marqués par la couleur ■.

- ▶ Hyperliens vers des ressources externes marqués par le symbole  et la couleur .
- ▶ Table des matières, liste des tableaux, liste des figures et liste des vidéos permettant d'accéder rapidement à des ressources du document.
- ▶ Index comprenant, entre autres, les principaux mots clés du langage R et toutes leurs occurrences dans le texte ainsi que dans le code informatique.

Blocs signalétiques

Le document est parsemé de divers types de blocs signalétiques inspirés de *AsciiDoc*  qui visent à attirer votre attention sur une notion.



Astuce! Ces blocs contiennent un truc, une astuce, ou tout autre type d'information utile.



Avertissement! Ces blocs mettent l'accent sur une notion ou fournissent une information importante pour la suite.



Attention! Vous risquez de vous bruler — métaphoriquement, s'entend — si vous ne suivez pas les recommandations de ces blocs.




Important! Ces blocs contiennent les remarques les plus importantes. Veuillez à en tenir compte.



Fait amusant! Ces blocs contiennent des informations amusantes, mais non essentielles pour la compréhension de l'ouvrage.



Ces blocs contiennent des liens vers des vidéos dans ma **chaîne YouTube** . Les vidéos sont répertoriées dans la liste des vidéos.



Ces blocs vous invitent à interrompre la lecture du texte pour passer à l'étude du code R des sections d'exemples.



Remarques spécifiques à macOS.



Remarques spécifiques à Windows.

Document libre

Tout comme R et l'ensemble des outils présentés dans ce document, le projet « Programmer avec R » s'inscrit dans le mouvement de l'**informatique libre** [🔗](#). Vous pouvez accéder à l'ensemble du code source en format \LaTeX en suivant le lien dans la page de copyright. Vous trouverez dans le fichier `README.md` toutes les informations utiles pour composer le document.

Votre contribution à l'amélioration du document est également la bienvenue ; consultez le fichier `CONTRIBUTING.md` fourni avec ce document et voyez votre nom ajouté au fichier `COLLABORATEURS`.

Bonne lecture !

Table des matières

Introduction **v**

Table des matières **ix**

Liste des tableaux **xiii**

Liste des figures **xv**

Liste des vidéos **xvii**

1	Éléments d'informatique pour programmeurs	1
1.1	Qu'est-ce que programmer ?	2
1.2	Quelques concepts fondamentaux	2
1.3	Bref historique des langages de programmation	5
1.4	Systèmes d'exploitation	10
1.5	Systèmes de fichiers	13
1.6	Exercices	16
2	Algorithmes et algorithmique	19
2.1	Définition et analogie	20
2.2	Pseudocode	22
2.3	Évaluation conditionnelle	23
2.4	Itération et récursion	26
2.5	Nombre d'opérations	31
2.6	Exercices	34
3	Présentation de R	39
3.1	Bref historique	39
3.2	Description sommaire de R	40
3.3	Interfaces	42
3.4	Stratégies de travail	43
3.5	Éditeurs de texte et environnements intégrés	44
3.6	Répertoire de travail	47
3.7	Organisation des projets et des fichiers	49

3.8	Programmer pour collaborer	49
3.9	Anatomie d'une session de travail	50
3.10	Obtenir de l'aide	51
3.11	Exemples	52
3.12	Exercices	54
4	Bases de la programmation	57
4.1	Données et procédures fondamentales	58
4.2	Commandes R	59
4.3	Vecteurs et arithmétique vectorielle	63
4.4	Fonctions	69
4.5	Expressions conditionnelles	73
4.6	Récursion	75
4.7	Fonctions internes utiles	76
4.8	Exemples	83
4.9	Exercices	103
5	Structures de données de R et fonctions d'application	107
5.1	Objets R	108
5.2	Matrice et tableau	113
5.3	Application pour les matrices et les tableaux	118
5.4	Liste	123
5.5	Application pour les listes et les vecteurs	125
5.6	Tableau de données	129
5.7	Facteur	130
5.8	Application pour les groupes de données	131
5.9	Date	131
5.10	Fonctions internes utiles	134
5.11	Produit extérieur	136
5.12	Exemples	137
5.13	Exercices	153
6	Bonnes pratiques de la programmation	159
6.1	Style	159
6.2	Présentation du code	164
6.3	Commentaires et documentation	166
6.4	Tests unitaires	168
6.5	Exemples	171
6.6	Exercices	180
7	Tri et recherche	181
7.1	Définition du problème et notation	182
7.2	Tri par insertion	184
7.3	Tri par sélection	186

7.4	Tri par échange	187
7.5	Tri par dénombrement	189
7.6	Recherche	192
7.7	Boucles itératives	194
7.8	Fonctions internes utiles	198
7.9	Exemples	200
7.10	Exercices	207
8	Débogage	211
8.1	Approche naïve	213
8.2	Évaluation pas à pas	216
8.3	Navigateur d'environnements	218
8.4	Méthode du canard en plastique	220
8.5	Exemples	221
8.6	Exercices	228
9	Importation et exportation de données	231
9.1	Importation de vecteurs de données	232
9.2	Importation de tableaux de données	233
9.3	Exportation de données	235
9.4	Importation et exportation d'objets R	236
9.5	Exemples	237
9.6	Exercices	239
10	Bibliothèques et paquetages	241
10.1	Système de base	241
10.2	Utilisation d'un paquetage	242
10.3	Création d'une bibliothèque personnelle	242
10.4	Installation de paquetages additionnels	244
10.5	Mise à jour des bibliothèques de paquetages	244
10.6	Exemples	244
10.7	Exercices	246
11	Analyse et contrôle de texte	247
11.1	Conventions typographiques et texte des exemples	248
11.2	Outils d'analyse et de contrôle du texte	251
11.3	Expressions régulières	256
11.4	Traitement de texte divisé en champs avec awk	267
11.5	Fonctions internes utiles	278
11.6	Exemples pour la ligne de commande Unix	281
11.7	Exemples pour les fonctions R	293
11.8	Exercices	296

12	Environnement et règles d'évaluation	299
12.1	Environnement	300
12.2	Environnement d'évaluation	302
12.3	Portée lexicale	304
12.4	Évaluation paresseuse	306
12.5	Exemples	307
12.6	Exercices	311
A	RStudio : une introduction	315
A.1	Installation	315
A.2	Description sommaire	315
A.3	Projets	316
A.4	Commandes de base	317
A.5	Anatomie d'une session de travail (bis)	318
A.6	Terminal RStudio	319
A.7	Configuration de l'éditeur	319
A.8	Aide et documentation	321
B	GNU Emacs et ESS : la base	323
B.1	Mise en contexte	323
B.2	Installation	324
B.3	Description sommaire	324
B.4	<i>Emacs-ismes</i> et <i>Unix-ismes</i>	325
B.5	Commandes de base	326
B.6	Anatomie d'une session de travail (ter)	330
B.7	Configuration de l'éditeur	331
B.8	Aide et documentation	331
C	Solutions des exercices	333
	Chapitre 1	333
	Chapitre 2	333
	Chapitre 4	340
	Chapitre 5	346
	Chapitre 6	352
	Chapitre 7	353
	Chapitre 8	360
	Chapitre 9	361
	Chapitre 11	362
	Chapitre 12	365
	Bibliographie	367
	Index	373

Liste des tableaux

2.1	Calcul du plus grand commun diviseur de 544 et 119 avec l'algorithme d'Euclide	21
4.1	Principaux opérateurs du langage R	59
5.1	Modes disponibles et contenus correspondants	110
5.2	Attributs les plus usuels d'un objet	112
7.1	Exemple de tri par insertion	185
7.2	Exemple de tri par sélection	187
7.3	Exemple de tri à bulles	188
7.4	Exemple de tri par sélection	191
7.5	Meilleurs temps au 100 mètres homme entre 1964 et 2005	209
9.1	Caractéristiques des fonctions d'importation de données de la famille <code>read.table</code>	234
11.1	Principaux opérateurs des expressions régulières étendues	258
11.2	Comparaison des trois variantes des quantificateurs bornés	260
11.3	Classes de caractères prédéfinies de la norme POSIX	264
C.1	Déroulement de l'algorithme de tri à dénombrement rapide pour la partie a) de l'exercice 7.5	357
C.2	Déroulement de l'algorithme de tri à dénombrement rapide pour la partie b) de l'exercice 7.5	358
C.3	Déroulement de l'algorithme de recherche séquentielle pour données triées pour l'exercice 7.7	359

Liste des figures

1.1	Programme en assembleur pour un microprocesseur 8 bits Motorola 6800	6
1.2	Opérations simples en APL	9
1.3	Extraits de la hiérarchie des systèmes de fichiers	13
1.4	Interfaces graphiques de navigation dans le système de fichiers de Windows et de macOS	14
1.5	Réglage du Finder de macOS pour afficher le raccourci vers le répertoire personnel	18
2.1	Processus itératif de peinture d'une clôture	27
2.2	Processus récursif de peinture d'une clôture	29
2.3	Processus récursif du calcul de 6!	30
2.4	Processus itératif du calcul de 6!	31
2.5	Illustrations pour des algorithmes qui génèrent des valeurs sur la surface et sur le pourtour d'un cube	38
2.6	Illustrations pour un algorithme qui génère des valeurs pour une pyramide comme dans le jeu Q*bert	38
3.1	Mises en œuvre en Scheme et en S de la fonction factorielle	40
3.2	Fenêtre de la console sous macOS au démarrage de R	42
3.3	Fichier de script et ligne de commande R	45
3.4	RStudio sous macOS dans sa configuration par défaut	46
3.5	GNU Emacs sous macOS en mode d'édition de code R	48
4.1	Représentation en arbre de l'évaluation de l'expression composée $(2 + ((2 + (4 * 6)) * (3 + 5)))/2$	61
4.2	Représentation d'un vecteur et de l'opération d'indexage	64
5.1	Représentations d'une matrice et de tableaux à trois et à quatre dimensions	115
5.2	Représentations d'opérations d'indexage sur une matrice	117
5.3	Représentations d'opérations d'indexage sur un tableau à trois dimensions	117


5.4	Représentations de la fonction d'application <code>apply</code> avec une matrice	120
5.5	Représentations de la fonction d'application <code>apply</code> avec un tableau à trois dimensions	121
5.6	Représentation d'une liste	123
5.7	Représentations d'opérations d'indexage sur une liste	125
5.8	Représentations des fonctions d'application <code>lapply</code> et <code>sapply</code> avec une liste	128
5.9	Représentation de la fonction d'application <code>mapply</code>	129
5.10	Représentation de la fonction d'application <code>tapply</code>	131
6.1	Blocs de code sans et avec les espaces appropriées	165
6.2	Exemple de documentation pour une fonction simple	168
7.1	Illustrations de quatre méthodes de tri pour une main de cinq cartes à jouer	183
7.2	Illustration du positionnement d'un enregistrement non trié parmi les enregistrements triés à l'aide d'une variable tampon	186
7.3	Illustration de <i>quicksort</i>	190
8.1	Code d'une fonction à déboguer	212
9.1	Exemples de fichiers contenant des vecteurs de données en texte brut	232
9.2	Exemples de tableaux de données en texte brut	233
11.1	Texte de la chanson <i>La journée qui s'en vient est flambant neuve</i> de Avec pas d'casque	250
11.2	Flux des données à la ligne de commande Unix	251
12.1	Représentation schématique d'un environnement dans R	301
12.2	Environnement de la fonction <code>square</code>	302
12.3	Environnement d'évaluation de la fonction <code>square</code>	303
12.4	Arbre des environnements	306
A.1	Fenêtre de RStudio sous macOS	316
A.2	Réglage de RStudio pour ne pas sauvegarder l'espace de travail de R au moment de quitter l'application	320
A.3	Réglage de RStudio pour une indentation du code de quatre caractères	321
A.4	Réglage de RStudio pour la sauvegarde des fichiers dans un format portable et sans espaces en fin de ligne	322
B.1	Fenêtre de GNU Emacs sous macOS	325

Liste des vidéos

Le numéro indiqué à gauche est celui de la section dans laquelle se trouve le bloc signalétique.

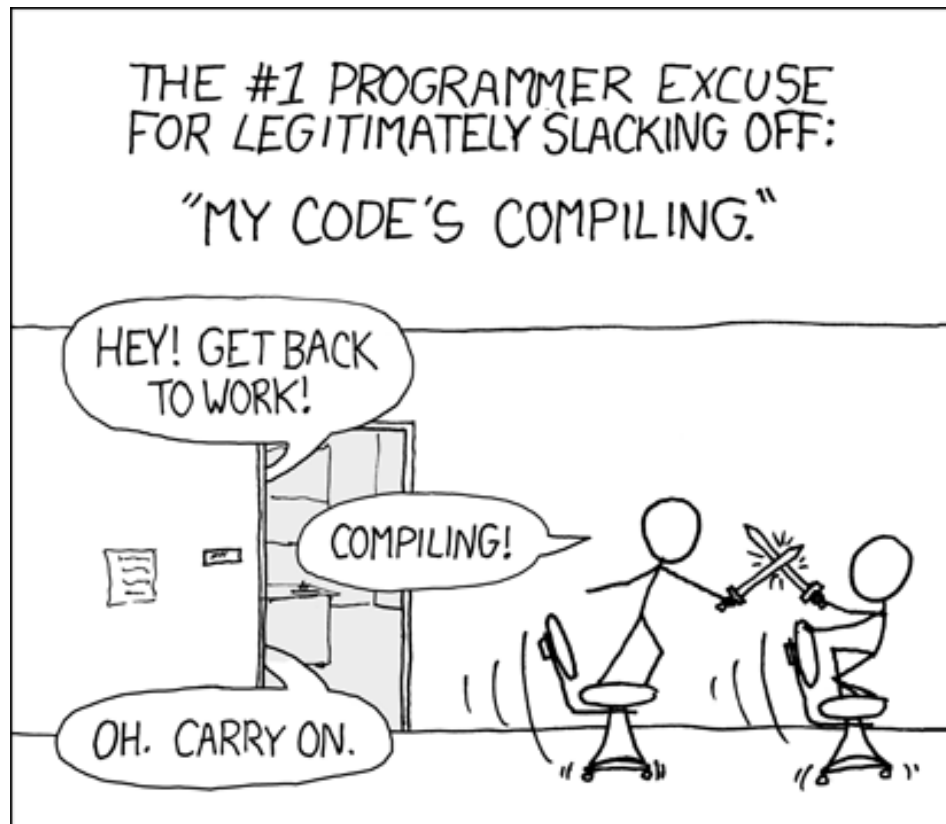
3.2	Présentation de R	42
5.2	Matrices et tableaux	117
5.3	Fonction <code>apply</code>	122
5.11	Fonction <code>outer</code>	137
6.1	Le guide de style R du <i>geek</i> de placard	163
7.0	Illustration des algorithmes de tri	182
7.8	Fonction <code>order</code>	199
A.5	Anatomie d'une session de travail avec RStudio	319
B.6	Anatomie d'une session de travail avec GNU Emacs	331
B.7	Création de fichiers de configuration pour Emacs et R	331

Comment (vraiment) bien utiliser ce document

1. Lire l'introduction.
2. Consulter le document dans une visionneuse PDF (Aperçu, Acrobat Reader) plutôt que dans un navigateur web.
3. Suivre le guide dans la lecture des chapitres. Aux avis signalés par le symbole , passer à l'étude du code R des sections d'exemples pour revenir ensuite au texte.
4. Lire les commentaires dans les fichiers de script R et prendre le temps d'analyser et de bien comprendre les résultats.
5. Travailler sur les exercices seulement après avoir complété les étapes précédentes.

L'ultime test pour vérifier si je comprends quelque chose consiste à l'expliquer à un ordinateur. Je peux vous dire quelque chose et vous hocherez la tête, mais sans que je sois certain d'avoir bien expliqué. L'ordinateur, lui, ne hoche pas la tête. Il répète exactement ce que je lui ai dit. Vous pouvez faire illusion dans plusieurs aspects de la vie, mais pas avec les ordinateurs.

Donald E. Knuth



Tiré de [XKCD.com](http://xkcd.com)

1 Éléments d'informatique pour programmeurs

Objectifs du chapitre

- ▶ Connaître les grands jalons de l'histoire des langages de programmation.
- ▶ Identifier les éléments distinctifs des langages de programmation.
- ▶ Utiliser le système de fichiers d'un ordinateur.

Vous savez programmer.

En effet, l'omniprésence des outils numériques fait en sorte que nous automatisons tous certaines tâches de notre vie quotidienne : demander une alarme pour le lendemain matin ; enregistrer une émission de télé qui sera diffusée dans trois jours ; planifier des virements chez notre institution financière ; signaler du contenu ou interpellier quelqu'un sur les médias sociaux à l'aide de symboles spéciaux (comme # et @) ; entrer un code de triche dans un jeu vidéo ; calculer la moyenne d'une colonne dans un tableur ; guider un robot vers la sortie d'un labyrinthe. Vous pouvez sans doute ajouter vos propres exemples à cette liste.

Dans chacun des exemples précités, vous dictez à une machine des tâches à effectuer en utilisant un langage qu'elle est à même de comprendre. Que le langage soit excessivement simple ou qu'une interface vous guide pas à pas dans la programmation n'y change fondamentalement rien : vous programmez. Cet ouvrage vise simplement à vous faire atteindre un niveau de sophistication bien plus grand, un niveau qui fera réellement de vous des programmeuses et des programmeurs informatiques.

Larry Wall est célèbre pour avoir inventé le langage Perl, mais également pour avoir formulé que la première des trois plus grandes vertus du programmeur est... la paresse. En effet, c'est grâce à cette qualité — car ça en devient une dans le contexte — que nous travaillons très fort pour dépenser le moins d'énergie possible. Nous concevons donc des programmes qui nous feront économiser du temps et que d'autres trouveront utiles. Nous les documentons aussi de manière

adéquate de telle sorte que nous n'aurons pas à répondre à des questions à leur sujet.¹

1.1 Qu'est-ce que programmer ?

Programmer est une activité scientifique consistant d'abord et avant tout à *résoudre des problèmes*. Plutôt que d'avoir recours aux mathématiques ou à l'expérimentation, les programmeurs demandent à une machine de résoudre le problème pour eux. Comme le souligne Swinnen (2012) — dont je vous recommande chaudement la lecture du chapitre 1 — programmer est « une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète. »

La première difficulté réside toutefois dans le fait de « demander » à une machine de résoudre un problème : il faut établir un mode de communication pour que l'humain puisse dicter des instructions à une machine qui, en définitive, ne comprend que deux instructions : vrai et faux, ouvert ou fermé, 0 ou 1. C'est là le rôle du *langage de programmation*.

Tout au long de leur histoire, les humains ont inventé une multitude de langages pour échanger entre eux : langues parlées, écritures, langues des signes, peinture, musique, etc. Certains langages parviennent à mieux exprimer certaines réalités ou certains sentiments que d'autres. Or, il en va de même des langages de programmation : non seulement sont-ils nombreux, mais certains conviennent mieux à certains types de tâches que d'autres. Vous serez à même d'apprécier la diversité des langages de programmation à la lecture de la [section 1.3](#) qui retrace les grands jalons de leur histoire.

1.2 Quelques concepts fondamentaux

Cette section introduit quelques concepts avec lesquels il s'avère utile de vous familiariser avant de vous lancer dans l'étude de la programmation. Nous aurons l'occasion de revenir sur ceux-ci plus loin dans l'ouvrage.

1.2.1 Algorithme

Vous avez probablement déjà trié une liste de mots ou de noms à l'aide d'un ordinateur. Pourtant, l'ordinateur n'a aucune connaissance du concept d'ordre alphabétique. Et même s'il « connaissait » cet ordre, comment au juste réalise-t-il l'opération de tri ? Si vous doutez qu'il existe plusieurs manières de le faire,

1. Les deux autres vertus sont l'impatience — ne pas tolérer d'attendre après l'ordinateur — et l'orgueil — concevoir des programmes de bonne qualité (Wall et collab., 1996).

comparez votre technique pour trier une main de cartes à celle de quelques autres personnes. Le résultat pourrait vous surprendre !

Les techniques de tri de données dans un ordinateur relèvent d'une pierre d'assise de l'informatique : l'algorithmique, la science qui étudie les algorithmes et les structures de données. Un *algorithme* est une procédure de calcul permettant de résoudre un problème bien spécifié, un peu comme une recette de cuisine explique comment produire un mets à partir d'une liste d'ingrédients.

L'apprentissage de la programmation va de pair avec l'étude des principes de base de l'algorithmique et des algorithmes classiques, notamment ceux de tri et de recherche. C'est ce que nous ferons aux chapitres 2 et 7.

1.2.2 Sémantique et syntaxe

Il y a plusieurs parallèles à dresser entre les langages de programmation et les langues parlées ou écrites : leur apprentissage requiert de la pratique ; on en connaît jamais un trop grand nombre ; le premier est plus difficile à apprendre que les suivants. L'informatique a également emprunté à la linguistique les notions de sémantique et de syntaxe.

La sémantique est l'étude de ce que signifie un message ou un programme informatique, c'est-à-dire de ce qu'il transmet ou exécute. La syntaxe, quant à elle, étudie la structure du message ou du programme. En simplifiant, disons que, pour une langue donnée, un dictionnaire permet de connaître la sémantique, alors qu'une grammaire décrit la syntaxe (Hebenstreit, 2024).

Par exemple, exprimons le message « J'ai soif » en anglais. Dire « *I am hungry* » relèverait d'une erreur de sémantique, puisque la signification du message s'en trouve changée. En revanche, avec « *I have thirsty* », le bon message se rendra, mais peut-être pas sans que l'erreur de syntaxe n'ait d'abord suscité une hésitation chez l'interlocuteur ².

Comme pour une langue, la maîtrise d'un langage de programmation exige de respecter à la fois les règles de sémantique et les règles de syntaxe du langage. Le second volet est relativement facile : si le programme compile ou s'exécute, c'est généralement que la syntaxe est respectée. Le respect de la sémantique, ou de la manière propre d'un langage d'exprimer des idées, demande plus d'efforts et de pratique.

1.2.3 Langages compilés et interprétés

Tous les langages de programmation ³ requièrent un traitement afin que les programmes écrits avec ceux-ci puissent être exploités par un ordinateur. Il existe

2. La formulation correcte est « *I am thirsty* », soit littéralement « Je suis soif » en français.

3. À l'exception du langage machine, mais rares sont les personnes qui ne souhaitent programmer qu'avec des 0 et des 1.

deux grandes façons d'effectuer ce traitement : par compilation et par interprétation.

Un compilateur est un programme informatique qui transforme en langage machine le code source rédigé dans un langage donné. On dit alors de ce langage qu'il est *compilé*. Le compilateur produit un fichier informatique que l'ordinateur peut ensuite exécuter directement. Sur la plateforme Windows, on reconnaît notamment ces fichiers par leurs extensions `.exe` et `.dll`. Les noyaux d'à peu près toutes les applications sur nos ordinateurs sont des fichiers compilés.

Un interpréteur, quant à lui, est un programme qui analyse, traduit et exécute le code source d'un programme informatique, et ce, à chaque fois que le programme doit être exécuté. Un langage qui nécessite l'intermédiaire d'un interpréteur est dit *interprété*. Un programme écrit dans un langage interprété ne peut être distribué sans son interpréteur.

Le traitement additionnel entre le code source et le langage machine fait généralement en sorte que les langages interprétés sont plus lents que les langages compilés à l'exécution. En revanche, une foule de détails de mise en œuvre sont pris en charge par l'interpréteur, ce qui réduit le temps de développement.

1.2.4 Paradigmes de programmation

Un programme informatique est toujours écrit pour résoudre un problème donné. Or, comme pour tout problème dans la vie, la solution à celui-ci peut être formulée de différentes manières. Le style fondamental avec lequel on exprime la solution est appelé le *paradigme* de programmation.

Il existe plusieurs paradigmes de programmation. Nous nous contenterons ici de nous pencher sur seulement quatre.

Impératif On indique à l'ordinateur les opérations à exécuter et l'ordre dans lequel les exécuter. C'est le paradigme le plus intuitif.

Déclaratif On indique cette fois à l'ordinateur ce que l'on souhaite obtenir comme résultat, mais sans préciser comment y parvenir. Il est laissé à la mise en œuvre du langage de déterminer la meilleure méthode de résolution du problème.

Par exemple, pour extraire d'une base de données `etudiants` le prénom de toutes les personnes de plus de 25 ans, on écrira dans un langage déclaratif comme le SQL :

```
SELECT prenom FROM etudiants WHERE age > 25
```

Nulle part n'est-il précisé comment le programme devrait procéder à l'extraction.

Fonctionnel Un programme est une suite d'appels de fonctions, comme en mathématiques. L'exécution d'une fonction n'a pas d'impact sur les autres fonc-

tions⁴. Les opérations complexes sont réalisées en combinant les fonctions, de manière analogue à la composition de fonctions $g \circ f$.

Orienté objet Un programme est conçu comme un ensemble de blocs logiciels (les objets) qui interagissent entre eux. Une *méthode* applique un traitement différent à un objet selon sa *classe*. Ce paradigme est particulièrement utilisé dans les grands et complexes projets informatiques.

La plupart des langages d'usage courant combinent d'office, ou du moins permettent de combiner, plusieurs paradigmes. Par exemple, le langage C++ est à la fois un langage impératif et orienté objet.

1.3 Bref historique des langages de programmation

Ada Lovelace (1815–1852) est généralement reconnue comme la première autrice d'un algorithme et de ce que l'on appelle aujourd'hui un programme informatique. C'est en son honneur qu'a été nommé le langage Ada conçu en réponse à un cahier de charges du département de la Défense des États-Unis au début des années 1980.

Franchissons d'un bond les premiers jours de l'informatique et de la programmation pour arriver aux ordinateurs électriques modernes, dans les années 1940. On programme alors ceux-ci en assembleur, un langage de très bas niveau conçu pour faciliter l'entrée de programmes dans un ordinateur (figure 1.1). Le langage demeure loin d'être facile à déchiffrer, mais c'est toujours préférable au langage machine, constitué uniquement de 0 et de 1. Le coût d'utilisation d'un ordinateur est à l'époque bien supérieur à celui d'une personne, aussi la traduction du langage assembleur vers le langage machine demeure-t-elle confiée à des humains. Au fil des années, le rapport de coût s'inversera et il deviendra financièrement plus avantageux de confier cette tâche cléricale à l'ordinateur lui-même, via un programme d'assembleur⁵.

Les premiers langages créés pour transmettre des instructions à un ordinateur tout en demeurant faciles à lire par les humains apparaissent dans les années 1950. Ils sont à l'origine intimement liés à l'architecture d'un ordinateur : à chaque type d'ordinateur son langage de programmation. Certaines contraintes — ou exigences — des langages de l'époque proviennent aussi du support physique alors utilisé pour stocker les programmes : les cartes perforées.

1.3.1 Fortran

En 1954, l'ingénieur de IBM John Backus publie les spécifications du langage FORTRAN (*FORmula TRANslating System*). Le premier compilateur voit le jour

4. Autrement on dit qu'une fonction a un effet de bord (*side effect*).

5. J'ai tiré cette manière de présenter les choses de Oualline (1997).

```

MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

C000                      ORG    ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START  LDS    #STACK

*****
* FUNCTION: INITA - Initialize ACIA
* INPUT: none
* OUTPUT: none
* CALLS: none
* DESTROYS: acc A

0013          RESETA EQU    %00010011
0011          CTLREG EQU    %00010001

C003 86 13          INITA  LDA A  #RESETA  RESET ACIA
C005 B7 80 04          STA A  ACIA
C008 86 11          LDA A  #CTLREG  SET 8 BITS AND 2 STOP
C00A B7 80 04          STA A  ACIA

C00D 7E C0 F1          JMP    SIGNON  GO TO START OF MONITOR

```

FIG. 1.1 – Extrait d'un programme en assembleur pour un microprocesseur 8 bits Motorola 6800. Source : [Wikimedia Commons](#) 

deux ans plus tard. Fortran (avec des minuscules à partir de 1977) deviendra rapidement le langage standard dans le calcul scientifique.


Plus d'un demi-siècle plus tard, l'empreinte de Fortran demeure importante, notamment grâce aux bibliothèques d'algèbre linéaire BLAS⁶ et LAPACK⁷ auxquelles ont recours la plupart des progiciels scientifiques, dont R. Le langage est toujours utilisé en calcul haute performance et pour mesurer le rendement des superordinateurs.



Dans le très recommandable film « Les figures de l'ombre » (*Hidden Figures*, 2016), une des héroïnes entreprend de s'attaquer à la programmation des nouveaux ordinateurs de la NASA. On peut alors voir qu'elle apprend le Fortran.

1.3.2 Lisp

Le deuxième langage le plus ancien toujours largement diffusé est Lisp (*LISt Processing*). Créé par John McCarthy en 1958 en tant que modèle pratique pour représenter des programmes, le Lisp est devenu le langage de choix pour la recherche et les applications en intelligence artificielle. Le terme Lisp désigne aujourd'hui une famille de langages comprenant de nombreux dialectes, dont Common Lisp, Scheme et Emacs Lisp.

6. *Basic Linear Algebra Subprograms*; <https://www.netlib.org/blas> .

7. *Linear Algebra PACKage*; <https://www.netlib.org/lapack> .

Le Lisp se distingue en outre par une syntaxe simple en notation préfixée (voir l'[exercice 1.2](#)), le support pour la programmation fonctionnelle ([section 1.2.4](#)) et la faculté de manipuler le code source en tant que structure de données. Autre trait distinctif : la syntaxe du Lisp fait un usage immodéré des parenthèses.

Le Lisp est entouré d'une aura de beauté et d'élégance dont peu d'autres langages peuvent se targuer. Citons Eric Raymond dans *How to Become a Hacker* [↗](#) :

Apprendre le Lisp en vaut le coup pour l'extraordinaire expérience d'éveil [*enlightenment experience*] que procure le fait de finalement le comprendre. Cette expérience fera de vous un meilleur programmeur pour toujours, même si vous n'avez plus vraiment où utiliser le Lisp.

Pour illustrer encore davantage la place toute particulière qu'occupe le Lisp en programmation, mentionnons également l'aphorisme selon lequel « ceux qui ne connaissent pas le Lisp sont condamnés à le réinventer », d'une certaine façon la version courte de la célèbre *Greenspun's tenth rule of programming* [↗](#) :

Tout programme suffisamment complexe en C ou en Fortran comporte une mise en œuvre ad hoc, mal spécifiée, pleine de bogues et lente de la moitié de Common Lisp.

(Fait amusant à noter : il n'y a pas d'autres lois que la dixième, de l'[aveu même de l'auteur](#) [↗](#).)

1.3.3 COBOL

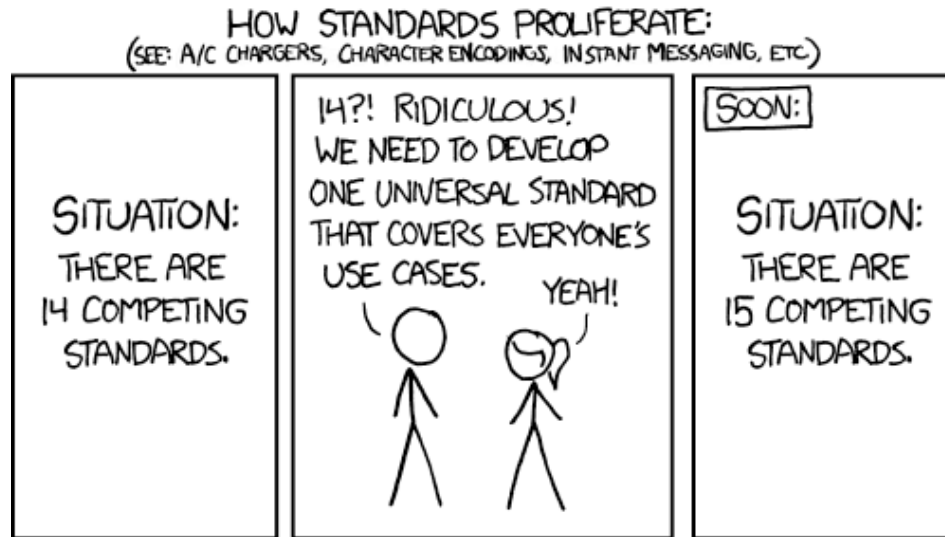
Le troisième langage développé dans les années 1950 et toujours en usage de nos jours est COBOL (*COmmon Business Oriented Language*). Ce langage spécialisé dans les applications de gestion a été créé en 1959 par un comité formé pour proposer un langage commun pour l'administration américaine.

Le COBOL reste très utilisé dans de grandes entreprises, notamment dans les institutions financières. La légende urbaine veut d'ailleurs que les programmeurs COBOL soient comparativement très bien rémunérés aujourd'hui sous l'effet combiné de leur rareté et de l'importance opérationnelle des applications qu'ils doivent maintenir.

1.3.4 Algol

Dès la fin de la décennie 1950, un comité de scientifiques se réunit à Zurich pour concevoir ce que l'on voudrait voir devenir le langage de programmation standard. De ces rencontres naîtra Algol (*ALGorithmic Oriented Language*) en 1958. Comme la plupart des tentatives de définition d'un standard, c'est un échec : le langage est populaire dans les milieux académiques, mais restera peu utilisé dans les applications commerciales.

Cela dit, on doit à Algol plusieurs innovations importantes, de telle sorte qu'un grand nombre des langages qui verront le jour par la suite seront considérés



Tiré de XKCD.com

comme ses descendants ; le poster *History of Programming Languages* [produit](#) par O'Reilly Media illustre ceci à merveille. Hoare (1973) a d'ailleurs cette jolie formule :

Voici un langage très en avance sur son temps, il n'a pas seulement été une amélioration de ses prédécesseurs mais aussi une amélioration de presque tous ses successeurs.

1.3.5 APL

Notre historique ne serait pas complet sans un mot sur APL (*A Programming Language*, qui l'aurait cru). Même s'il n'a jamais connu une diffusion importante, ce langage conçu par Kenneth Iverson autour de 1962 n'en a pas moins eu une influence considérable sur la manière de penser et de représenter les opérations mathématiques sur les tableaux à plusieurs dimensions.

Doté d'une large gamme de symboles pour représenter des opérations et d'une syntaxe pour le moins particulière — les expressions sont exécutées de droite à gauche ! — l'APL est remarquablement concis et puissant ; voir la [figure 1.2](#) pour un aperçu. Le revers de la médaille et ce qui a assurément nui à son adoption à large échelle, c'est la difficulté que l'on éprouve à relire le code. Assez pour que d'aucuns qualifient l'APL de « langage à écriture seulement ».

APL a pendant longtemps été un langage très prisé par les actuaires, ce qui fait qu'il subsiste des applications dans ce langage dans certaines compagnies d'assurance. Le modèle de traitement des vecteurs, matrices et tableaux de l'APL a servi d'inspiration pour la conception du langage S — nous y reviendrons au [chapitre 3](#). Le langage continue sa vie aujourd'hui principalement sous forme de son successeur, J.

```

      x ← 1 5
      x
1 2 3 4 5
      + / x
15
      (+ / x) ÷ p x
3

```

FIG. 1.2 – Opérations simples en APL. De haut en bas : génération des nombres de 1 à 5 et stockage dans la variable *x* ; affichage du contenu de *x* ; somme des éléments de *x* ; moyenne des éléments de *x*. Image : © François-Dominique, [CC BY-SA 4.0](#), via [Wikimedia Commons](#)

1.3.6 C

Le langage C a été inventé en 1972 chez Bell Labs par Ken Thompson et Dennis Ritchie afin de réécrire le système d'exploitation UNIX ([section 1.4.2](#)). Il demeure beaucoup utilisé pour la programmation système : le noyau de systèmes d'exploitation comme Windows et Linux sont développés en grande partie en C.

Le C est un langage de programmation généraliste considéré, selon les standards actuels, comme de bas niveau. Pour illustrer, l'utilisateur doit programmer des traitements comme la libération de la mémoire, la vérification de la validité des indices sur les tableaux, l'ouverture et la fermeture des fichiers, etc.

Le langage demeure l'un des plus utilisés dans le monde et son influence est considérable. De nombreux langages plus modernes comme C++, C# et Java reprennent des aspects de C. Le C est également beaucoup utilisé pour le calcul numérique haute performance, où il s'est en quelque sorte substitué à Fortran. La plupart des progiciels scientifiques — dont R, encore une fois — offrent la possibilité d'appeler du code C lorsque la rapidité de calcul devient un enjeu. Le gain en temps d'exécution doit toutefois être suffisamment grand pour compenser le temps de développement plus long qu'exige le C par rapport à des langages de plus haut niveau.



Dans l'ouvrage classique de [Kernighan et Ritchie \(1978\)](#), le premier exemple d'un programme C affiche le message « *hello, world* » à l'écran. Ça deviendra ensuite une tradition de démontrer le fonctionnement ou la syntaxe d'un langage avec ce même exemple.

1.3.7 Autres jalons

Nous nous sommes attardés jusqu'ici à des langages de programmation vieux de plus de 40 ans à cause de leur importance historique et parce qu'ils sont toujours utilisés couramment. De nombreux autres langages ont vu le jour depuis, si

bien qu'ils se comptent aujourd'hui par milliers. En voici quelques autres ayant occupé une place prépondérante dans l'histoire.

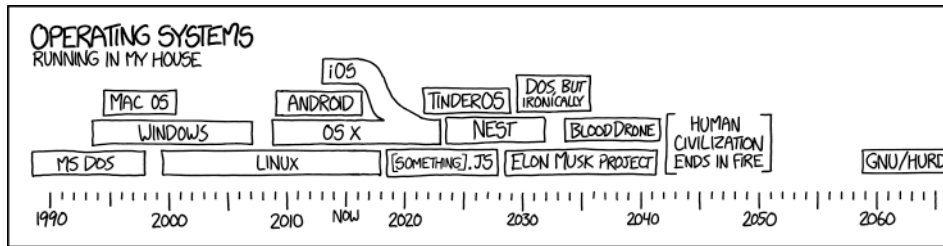
- ▶ Comme son nom l'indique, **C++** (Bjarne Stroustrup, 1980) est un dérivé du C qui lui ajoute, en autres choses, la programmation orientée objet. Certains considèrent que C++ devrait être le point d'entrée de toute personne voulant débiter à programmer avec des langages de la famille du C. Le code C est compatible avec le C++.
- ▶ Principal représentant de l'ère Internet des années 1990, **Java** (James Gosling, 1995) a été conçu pour que le code écrit dans ce langage puisse s'exécuter sur n'importe quelle plateforme informatique sans nécessiter une nouvelle compilation. Il est donc très populaire dans les applications web ou embarquées. Sa syntaxe est fortement inspirée du C++.
- ▶ **Visual Basic** (Microsoft, 1991) permet de développer des applications de manière interactive en disposant des composantes sur un canevas. Le langage est aujourd'hui discontinué, mais son dérivé **Visual Basic for Applications** (VBA) demeure beaucoup utilisé dans les applications de la suite bureautique Office.
- ▶ **Python** (Guido van Rossum, 1991) est un langage de haut niveau, orienté objet, multiplateforme et sous licence libre. C'est un des langages les plus utilisés aujourd'hui pour le calcul scientifique et l'analyse de données massives.

Certains langages de programmation ont une vocation généraliste, certains visent des niches particulières, alors que d'autres cherchent surtout à faire progresser l'état des connaissances dans la théorie des langages. Quoi qu'il en soit, un langage de programmation demeure un outil et, en informatique comme dans d'autres domaines, il convient de choisir le meilleur outil pour accomplir une tâche donnée.

J'invite les lecteurs intéressés à en savoir davantage sur l'histoire des langages de programmation en général, ou sur l'un ou l'autre des langages mentionnés ci-dessus en particulier, à débiter par les très complètes entrées de Wikipedia en [français](#) et en [anglais](#). [Oualline](#) (1997, chapitre 2) propose également une excellente introduction à la notion de programmation ainsi qu'une brève, mais instructive, histoire du langage C.

1.4 Systèmes d'exploitation

À titre de (futurs) programmeurs, vous devrez tirer profit au maximum des ressources de votre ordinateur. Ceci exige d'en connaître le fonctionnement mieux que l'utilisateur lambda. Or, vos principaux outils numériques sont aujourd'hui les téléphones et tablettes aux interfaces hautement simplifiées. Connaissez-vous le rôle du système d'exploitation d'un ordinateur ? En connaissez-vous les diverses variantes et leurs caractéristiques principales ? Comprenez-vous bien le système



Tiré de [XKCD.com](https://xkcd.com)

de classification des fichiers de votre ordinateur ? Vous devrez pouvoir répondre « oui » à ces questions. Cette section et la suivante sont là pour vous y aider.

Le système d'exploitation (*operating system*, souvent abrégé « OS ») est un ensemble de programmes qui gère les ressources matérielles et logicielles d'un ordinateur. Premier programme exécuté par celui-ci lors de sa mise en marche, le système d'exploitation reçoit les demandes de ressources des applications — espace mémoire, unité de calcul, disque dur, communication avec les périphériques, utilisation du réseau — et les alloue en fonction de l'état du système et des demandes concurrentes des autres applications. Tous les logiciels applicatifs requièrent un système d'exploitation pour fonctionner.

Il existe une grande variété de systèmes d'exploitation. Les plus connus aujourd'hui sont, pour les ordinateurs personnels et les serveurs, Windows de Microsoft, macOS de Apple et Linux, un logiciel libre créé et toujours administré par Linus Torvalds. Pour les appareils mobiles, deux systèmes d'exploitation se partagent l'essentiel du marché : iOS et Android. Le premier est dérivé de macOS et le second, de Linux.

Les systèmes macOS et Linux appartiennent à la famille des systèmes Unix. Cela ne laisse donc en fait que deux grandes classes de systèmes d'exploitation couramment utilisés : Windows et Unix.

1.4.1 Windows

Le système d'exploitation Windows de Microsoft équipe près de 90 % des ordinateurs personnels dans le monde. Dans les circonstances, rares sont les personnes qui n'ont jamais utilisé le système, ne serait-ce qu'à petites doses.

À l'origine, en 1985, Windows n'était qu'une interface graphique superposée au véritable système d'exploitation des ordinateurs de type « compatibles IBM PC », le DOS. Les versions grand public depuis Windows 2000 sont plutôt basées sur le noyau de Windows NT, un système conçu à partir de zéro et lancé au milieu des années 1990 en tant que système pour les entreprises.

Le plus grand avantage de Windows est son ubiquité. À peu près toutes les applications, sauf peut-être celles s'adressant à un segment de marché très précis, sont disponibles sous Windows, que ce soit en code natif, via un port depuis Unix,


ou encore par l'intermédiaire d'une couche logicielle d'émulation comme Cygwin ou MinGW.

Un système d'exploitation multiutilisateur prend pour acquis que plusieurs utilisateurs se partageront les ressources de l'ordinateur, simultanément lorsqu'il s'agit d'un serveur, ou les uns après les autres dans le cas d'un ordinateur personnel. Dans un tel système, on établit des règles strictes d'accès aux ressources du système (seul l'administrateur peut installer ou supprimer des applications) et d'accès aux ressources des autres utilisateurs (Marianne n'a pas accès aux fichiers d'Alexandre et vice-versa).


Les versions de Windows multiutilisateurs sont apparues relativement tard sur le marché grand public, et ce, sans que le paradigme ne soit imposé aux utilisateurs. Résultat : la plupart des utilisateurs de Windows s'octroient à la ronde les droits d'administrateur et, pire, plusieurs applications requièrent toujours un accès total au système. C'est par ces failles que se propagent virus et autres logiciels malveillants sur la plateforme.

Les systèmes Windows sont livrés avec une trousse de développement très restreinte. Les programmeurs doivent donc installer les outils nécessaires, souvent sous forme de suites logicielles monolithiques, ce qui entraîne dédoublements et, parfois, conflits.

1.4.2 Unix

Le terme *Unix* — ou UNIX qui, en majuscules, est une marque déposée de **Open Group**  — recoupe une famille de systèmes d'exploitation dérivant tous du premier système Unix développé chez Bell Labs dans les années 1970 par Kenneth Thompson, Dennis Ritchie et plusieurs autres.

Les systèmes Unix partagent un certain nombre de caractéristiques communes. D'abord, ils sont intrinsèquement multitâches et multiutilisateurs. Il existe donc une séparation très nette entre les droits accordés à l'administrateur système (communément appelé *superuser* ou *root*) et ceux dévolus aux utilisateurs normaux. Ceux-ci ne peuvent écrire que dans l'espace disque qui leur est réservé, identifié comme le *répertoire personnel* ou le *dossier de départ* (*home directory*).

Ensuite, la **philosophie Unix**  commande la modularité : le système d'exploitation fournit aux utilisateurs et aux applications une multitude de petits outils très spécialisés qui ne réalisent à peu près qu'une seule tâche. Le système rend ensuite simple, via un mécanisme dit de transfert de données (« tuyau », *pipe*), de les combiner pour effectuer des opérations plus complexes.

Par exemple, pour extraire un mot précis d'une phrase, un utilisateur pourrait avoir recours à l'outil `grep` pour extraire la ligne entière où se trouve le mot, puis l'outil `cut` pour isoler le mot visé puis, enfin, l'outil `tr` pour supprimer des symboles de ponctuation qui seraient accolés au mot.

À l'exception notable des Mac, Unix demeure assez peu répandu sur les ordinateurs personnels. En revanche le système d'exploitation, en particulier sa va-

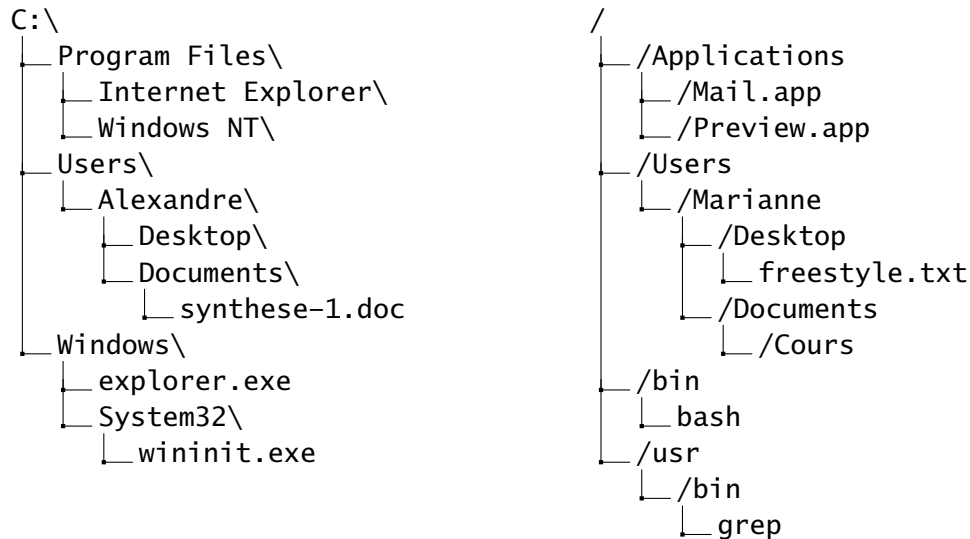


FIG. 1.3 – Extraits des arbres de fichiers de Windows (à gauche) et de macOS (à droite)

riante libre Linux, équipe l'immense majorité des stations de travail, des serveurs et des supercalculateurs. De plus, bon nombre d'applications scientifiques et de langages de programmation sont d'abord développés sur des systèmes Unix. Pour ces raisons, j'estime que tout programmeur devrait connaître quelques-uns des principaux « Unix-ismes ». Nous les passerons en revue à la section suivante avec les détails additionnels sur le système de fichiers de Unix.



Vous savez désormais pourquoi le répertoire personnel, ou *home directory*, est représenté par une maison dans le Finder et que le raccourci-clavier pour y accéder est $\text{⌘} + \text{H}$.

1.5 Systèmes de fichiers

Le système de fichiers d'un ordinateur contrôle l'inscription, l'organisation et la récupération des données sur une unité de stockage, souvent un disque dur.

Nous n'entrerons pas dans les détails techniques des systèmes de fichiers. Seulement, les programmeurs doivent être familiers avec quelques grands principes de l'organisation des données dans un ordinateur.

Les fichiers sont regroupés dans des *répertoires* (ou dossiers). Ceux-ci contiennent eux-mêmes des fichiers ou des sous-répertoires. Ce schéma se répète sans limite pratique. Il en résulte une classification sous forme d'arbre dont la *racine* est le répertoire contenant l'intégralité des fichiers. La [figure 1.3](#) présente des extraits des arbres de fichiers usuels de Windows et de macOS.

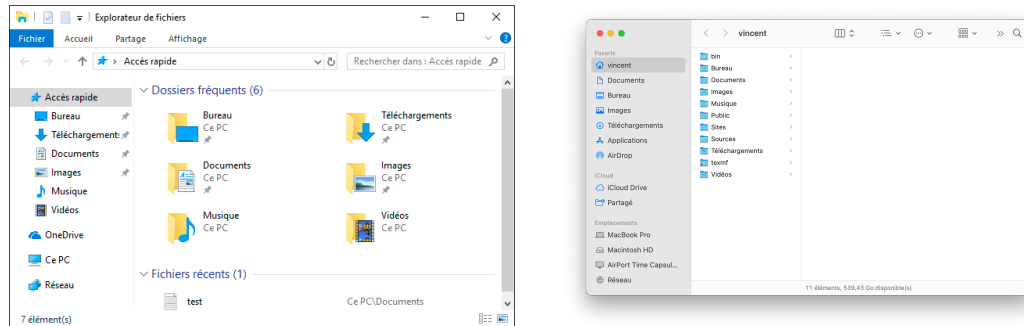


FIG. 1.4 – Interfaces graphiques de navigation dans le système de fichiers : Explorateur Windows (à gauche) et Finder de macOS (à droite)

Les interfaces graphiques des systèmes d'exploitation contiennent toutes une application pour naviguer à l'intérieur du système de fichiers. Sous Windows, il s'agit de l'Explorateur Windows. Sous macOS, l'outil de navigation est le Finder. La [figure 1.4](#) présente des fenêtres types de ces deux applications.



Pour accéder à la racine du système de fichiers dans l'Explorateur Windows, il faut ouvrir l'arborescence sous « Ce PC » ou « Ordinateur ». Dans le Finder de macOS, on atteint la racine en sélectionnant « Macintosh HD » dans la barre latérale.

1.5.1 Particularités du système de fichiers de Windows

Dans Windows, chaque lecteur physique ou lecteur réseau dispose de son propre arbre de fichiers. La racine est identifiée par une lettre. Le premier disque dur est C⁸. Si le système comporte plus d'un disque, l'utilisateur doit savoir sur quel disque retrouver ou enregistrer ses données.

Depuis au moins la version 7 de Windows, le répertoire (ou dossier) personnel des utilisateurs se retrouve sous C:\Users⁹, tel qu'illustré à la [figure 1.3](#). Ainsi, la véritable position du dossier Documents d'Alexandre dans le système de fichiers est C:\Users\Alexandre\Documents.

Le système de fichiers de Windows est insensible à la casse, c'est-à-dire qu'il ne fait aucune distinction entre Program Files, program files et PROGRAM FILES. Dans les chemins d'accès (voir ci-dessous), la barre oblique inversée « \ » sépare les noms de répertoires.

8. La nomenclature des lecteurs date d'une époque où les ordinateurs personnels n'étaient pas tous équipés d'un disque dur, mais plutôt d'un ou deux lecteurs de disquettes. Ceux-ci étaient identifiés par les lettres A et B.

9. Ce répertoire est affiché sous le nom Utilisateurs dans la version française du système d'exploitation.



Le répertoire personnel n'est pas affiché par défaut dans l'Explorateur Windows. Pour y accéder, vous devez naviguer à partir de la racine du système de fichiers dans « Ce PC » ou « Ordinateur ».

1.5.2 Particularités du système de fichiers de Unix

Sous Unix, la racine du système de fichiers est toujours identifiée par le symbole « / » et les lecteurs, physiques ou réseau, sont accessibles par divers *points de montage* dans le système.

Pour illustrer, supposons que le système d'exploitation d'une station de travail réside sur le disque dur de l'ordinateur, mais que les fichiers personnels des utilisateurs se trouvent sur un disque réseau. Dans un tel scénario, le lecteur de disque dur de l'ordinateur sera monté sur le répertoire / et le lecteur réseau, sur le répertoire /home. Le système de fichier dirigera automatiquement les requêtes vers le lecteur approprié. L'utilisateur n'a donc jamais à se préoccuper de l'organisation physique des disques.

Tel que mentionné à la section précédente, il existe dans les systèmes Unix une stricte séparation entre les fichiers du système (modifiables uniquement par l'administrateur système) et ceux des utilisateurs. Ceux-ci ne peuvent créer et modifier des fichiers que dans leur répertoire personnel. Ce répertoire est situé dans /Users sous macOS et habituellement dans /home sous Linux.

À la ligne de commande et dans la configuration de plusieurs applications, le caractère « ~ » représente le répertoire personnel.

Le système de fichiers sous Unix est généralement sensible à la casse. La barre oblique « / » sépare les noms de répertoires dans les chemins d'accès aux fichiers.

Les fichiers dont le nom débute par un point « . » sont des fichiers cachés. À moins de demander explicitement de les afficher, ces fichiers n'apparaissent donc pas dans la liste des fichiers à la ligne de commande ou dans les interfaces graphiques. Les fichiers cachés servent généralement à stocker les options de configuration des applications en format texte brut.



Sous macOS, les clés USB, les disques durs externes et les disques réseau sont montés dans le répertoire /Volumes. Ce répertoire n'existe pas lorsqu'aucun disque externe n'est connecté.

1.5.3 Chemin d'accès

Le chemin d'accès (*path*) d'un fichier ou d'un répertoire décrit la position de la ressource dans le système de fichiers. Un chemin d'accès peut être absolu ou relatif.

Chemin absolu La position d'un fichier est décrite à partir de la racine, de telle sorte que le chemin d'accès demeure valide depuis n'importe quel point dans le système de fichiers.

Dans l'extrait de système de fichier Windows de la [figure 1.3](#), le chemin d'accès absolu vers le fichier `synthese-1.doc` est :

```
C:\Users\Alexandre\Documents\synthese-1.doc
```

Chemin relatif La position d'un fichier est donnée à partir d'un endroit précis dans le système de fichier autre que la racine. Le chemin dépend donc du répertoire courant. Le nom fictif « `..` » identifie le répertoire parent (un niveau supérieur dans l'arbre des fichiers).

Par exemple, à partir du répertoire `Documents/Cours` de Marianne dans le système de fichier Unix de la [figure 1.3](#), le chemin d'accès relatif vers le fichier `freestyle.txt` est :

```
../../Desktop/freestyle.txt
```

La notion de chemin d'accès fait également référence à la liste des répertoires dans lesquels le système d'exploitation recherche une application lorsqu'elle est appelée. Cette liste est conservée dans une variable d'environnement nommée `%PATH%` sous Windows et `$PATH` sous Unix. Il est hors de la portée de cet ouvrage d'expliquer comment modifier la variable d'environnement. La réponse se trouve à une requête près dans un moteur de recherche.

1.6 Exercices

1.1 Des nouveaux langages de programmation apparaissent régulièrement. Deux exemples récents provenant de géants de l'industrie sont [Swift](#), de Apple, et [Go](#), de Google. À partir des informations glanées sur les sites Internet des langages et dans les pages Wikipedia qui leurs sont consacrées, retracer les grands langages ayant servi comme sources d'inspiration à Swift et à Go.

1.2 Les notations infixée (*infix*), préfixée (*prefix*) et suffixée (*postfix*) sont trois manières différentes, mais équivalentes, d'écrire des expressions en mathématiques ou en programmation.

La notation infixée est celle qui nous est la plus familière. L'opérateur y est placé entre les opérandes : $x + y$. En notation préfixée, aussi appelée notation polonaise, l'opérateur est plutôt placé *avant* les opérandes : $+ x y$. On l'aura compris, en notation suffixée, ou notation polonaise inversée, l'opérateur apparaît *après* les opérandes : $x y +$. La notation suffixée n'a jamais recours aux parenthèses.¹⁰

a) Soit A , B , C et D quatre nombres. Observer que l'opération en notation infixée

$$A \times B + C \div D$$

10. La compagnie HP commercialise depuis 1972 de très prisées calculatrices scientifiques utilisant la notation suffixée (libellées RPN pour *Reverse Polish Notation*).

s'écrit en notation préfixée

$$+ \times A B \div C D$$

et en notation suffixée

$$A B \times C D \div +.$$

- b) Exprimer en notations préfixée et suffixée l'opération en notation infixée suivante :

$$A \times (B + C) \div D.$$

- c) Les opérations suivantes exprimées en notation préfixée et suffixée, dans l'ordre, effectuent la même opération arithmétique.

$$\times A + B \div C D$$

$$A B C D \div + \times$$

Exprimer cette opération en notation infixée.

- 1.3** Les répertoires Documents et Desktop (ou Bureau dans la version française) existent par défaut dans le répertoire personnel des utilisateurs tant dans Windows que dans macOS. Localiser ces deux répertoires dans le système de fichier à partir de la racine avec l'Explorateur Windows ou avec le Finder.

- 1.4** (macOS seulement) Dans les versions modernes de macOS, le Finder n'affiche plus par défaut le dossier personnel dans la barre latérale. C'est pourtant pratique de pouvoir y accéder directement. Effectuer les opérations suivantes pour afficher le raccourci vers le répertoire personnel.

1. Ouvrir une fenêtre du Finder et accéder aux options de configuration par le menu standard `Finder | Préférences (⌘ ,)`.
2. Dans l'entête de la fenêtre de configuration, sélectionner la catégorie Barre latérale.
3. Dans la liste des favoris, cocher le répertoire personnel identifié par une maison et votre nom d'utilisateur.

La [figure 1.5](#) illustre cette configuration.

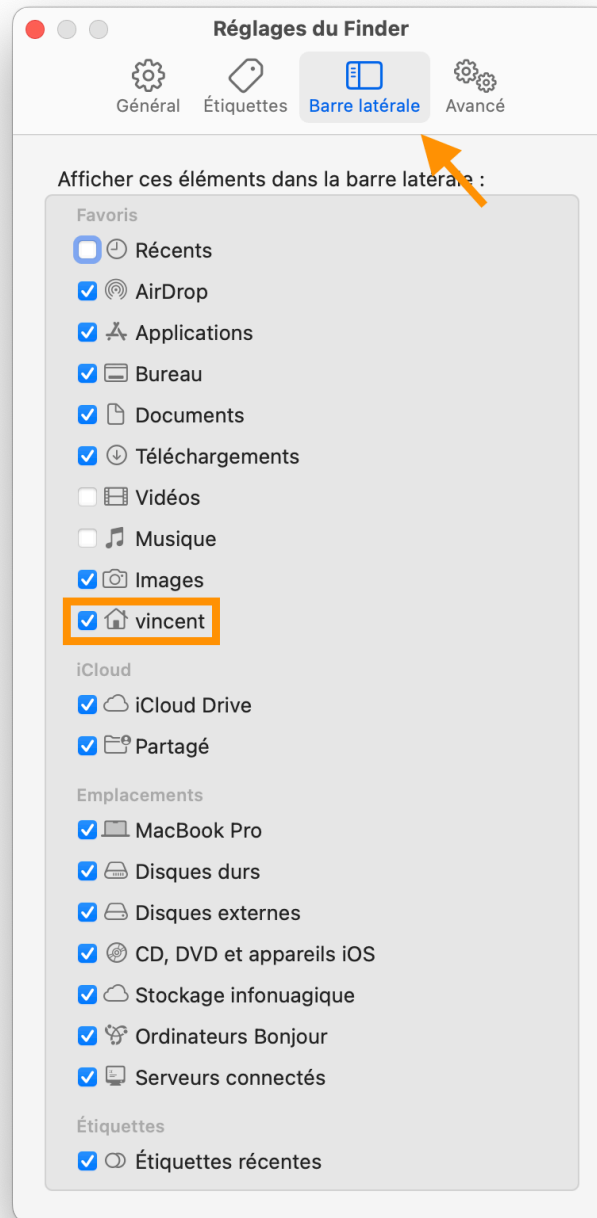


FIG. 1.5 – Réglage du Finder de macOS pour afficher le raccourci vers le répertoire personnel

2 Algorithmes et algorithmique

Objectifs du chapitre

- ▶ Résoudre un problème sous la forme d'un algorithme en langage naturel ou en pseudocode.
- ▶ Utiliser des instructions conditionnelles dans un algorithme.
- ▶ Utiliser la récursion et l'itération pour effectuer des tâches répétitives dans un algorithme.
- ▶ Établir la performance d'un algorithme à l'aide de la notation $O()$.

Marianne et Alexandre doivent fournir une solution logicielle à un problème donné. Sitôt la lecture de l'énoncé du problème complétée, ils se précipitent sur le clavier pour coder leur solution. Rapidement bloqués, pensant qu'ils butent sur un problème de syntaxe ou d'utilisation d'une fonction, ils demandent de l'aide. Lorsqu'on leur demande qu'est-ce qu'ils essaient de faire, quelle est la nature de leur solution, ils ne peuvent répondre que par un long silence.

Dans mon rôle de professeur, j'ai été témoin de la scène ci-dessus à de nombreuses reprises. Marianne et Alexandre font une erreur commune chez les programmeurs — et pas que chez les débutants : commencer à coder sans avoir au préalable suffisamment réfléchi à la solution. Il leur manque un plan clair des étapes à suivre pour résoudre le problème. Or, ce plan, ou la recette à suivre pour transformer les entrants d'un problème en une solution, c'est l'algorithme. Travailler sans algorithme, c'est comme naviguer sans boussole.

Puisque programmer requiert des algorithmes, l'étude de la programmation doit par conséquent s'accompagner de notions d'algorithmique, la science qui étudie les algorithmes et les structures de données. L'algorithmique est une discipline riche en techniques ingénieuses et en analyses mathématiques poussées. Connaître ses principes de base et étudier certains algorithmes classiques vous permettra de mieux planifier vos méthodes de résolution de problème. En effet, un bon algorithme permet de résoudre en quelques secondes un problème qui pourrait autrement prendre des années.

L'étude des algorithmes s'accompagne habituellement de celle des structures de données ou, en d'autres termes, de la manière d'organiser les données dans

un ordinateur. Comme nous le verrons au [chapitre 5](#), il existe bien différentes structures de données en R, mais leur réelle mise en œuvre demeure tout à fait transparente pour les programmeuses et les programmeurs. C'est pourquoi nous ferons l'impasse sur des notions que l'on retrouve dans tous les ouvrages classiques d'algorithmique, comme le tableau (*array*), la liste chaînée (*list*), l'arbre (*tree*) ou la table de hachage (*hashtable*).

Le présent chapitre est inspiré de [Abelson et collab. \(1996\)](#), [Knuth \(1997a\)](#) et [Stephens \(2013\)](#).

2.1 Définition et analogie

Un algorithme est une procédure de calcul permettant de résoudre un problème bien spécifié. L'algorithme explique, de manière non ambiguë et dans un nombre d'opérations fini, comment, à partir d'entrants, obtenir l'extrant solution du problème.

Pour se mériter un algorithme, un problème doit renfermer une dose minimale de complexité : personne n'écrit un algorithme pour extraire le quatrième élément d'un vecteur de données. On suppose que cela fait partie de la définition de vecteur et que vous savez comment effectuer l'opération dès lors que vous connaissez le langage de programmation sous-jacent.

Dans la même veine, un algorithme ne dépend pas du langage de programmation employé, car il fournit les étapes pour résoudre un problème, non pas leur mise en œuvre. En d'autres termes, un algorithme explique *ce qu'il faut faire* pour résoudre un problème et non *comment le faire*.

Illustrons le concept d'algorithme par l'un de ses plus illustres représentants : l'algorithme d'Euclide (c. 300 av. J.-C.), une procédure de calcul du plus grand commun diviseur (PGCD) de deux nombres entiers m et n . Selon [Knuth \(1997b\)](#), il s'agirait de l'un des plus anciens algorithmes. Il repose sur l'idée que le PGCD de m et n est aussi le PGCD de n et du reste de la division de m par n .

Algorithme 2.1 (Algorithme d'Euclide). Calculer le plus grand commun diviseur de deux entiers positifs m et n avec, sans perte de généralité, $n < m$.

1. Diviser m par n et poser r égal au reste de la division. (Nous avons alors $0 \leq r < n$.)
2. Si $r = 0$, retourner la valeur n .
3. Poser $m \leftarrow n$, $n \leftarrow r$, puis retourner à l'étape 1. □

Suivons les étapes de l'algorithme d'Euclide avec $m = 544$ et $n = 119$. À l'étape 1, nous déterminons que $544/119 = 4 + 68/119$, donc nous posons $r \leftarrow 68$. Puisque $r \neq 0$, l'étape 2 de l'algorithme ne s'applique pas. Nous passons donc à l'étape 3 et nous posons $m \leftarrow 119$ et $n \leftarrow 68$. De retour à l'étape 1, nous déterminons cette fois que $r \leftarrow 51$, ce qui mène encore à l'étape 3 et $m \leftarrow 68$ et

TAB. 2.1 – Calcul du plus grand commun diviseur de 544 et 119 avec l'algorithme d'Euclide

Étape	m	n	r
1	544	119	68
3	119	68	68
1	119	68	51
3	68	51	51
1	51	17	0
2	—	17	—

$n \leftarrow 51$. Après un autre cycle, nous en sommes à $m \leftarrow 51$ et $n \leftarrow 17$. Cette fois, nous posons $r \leftarrow 0$ à l'étape 1 et la procédure s'arrête ensuite à l'étape 2. Le plus grand commun diviseur de 544 et 119 est 17. Le [tableau 2.1](#) dresse le sommaire des opérations ci-dessus.



L'ordre des opérations à l'étape 3 de l'[algorithme 2.1](#) est important. En effet, l'action « Poser $m \leftarrow n, n \leftarrow r$ » est très différente de « Poser $n \leftarrow r, m \leftarrow n$ » puisque dans cette dernière, la valeur de n est perdue après la première réaffectation. La seconde action est équivalente à « Poser $n \leftarrow r, m \leftarrow r$ ».

Selon [Knuth \(1997a, section 1.1\)](#), un algorithme devrait réunir cinq caractéristiques.¹

1. *Finitude*. Un algorithme doit toujours se terminer après un nombre fini d'étapes. Ce nombre peut toutefois devenir très grand.
2. *Définition précise*. Chaque étape d'un algorithme doit être définie précisément ; les actions à réaliser doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.
3. *Entrées*. Un algorithme comporte aucune, une ou plusieurs entrées, des quantités fournies à l'algorithme avant qu'il ne commence ou qui sont allouées dynamiquement durant son exécution. Ces entrées proviennent d'un ensemble d'objets bien spécifié. Par exemple, l'[algorithme 2.1](#) comporte deux entrées, les nombres m et n , provenant de l'ensemble des entiers positifs.
4. *Sorties*. Un algorithme comporte une ou plusieurs sorties : des quantités ayant une relation spécifiée avec les entrées. L'[algorithme 2.1](#) a une seule sortie, soit la valeur n de l'étape 2, le plus grand commun diviseur des entrées.
5. *Efficacité*. On s'attend généralement d'un algorithme qu'il soit efficace dans le sens où toutes les opérations qu'il doit accomplir sont suffisamment élémen-

1. Liste adaptée de la version française fournie par [Wikipédia \(2024a\)](#).

taires pour pouvoir être en principe réalisées dans une durée finie par une personne munie de papier et d'un crayon.

Pour bien comprendre la nature et la portée d'un algorithme, il est assez utile de le comparer à une recette de cuisine. Dans une recette, les entrées sont les ingrédients, la procédure les diverses étapes et la sortie, le plat préparé.

Comme un algorithme, une recette repose sur un certain nombre d'instructions élémentaires qui se passent d'explication, comme saisir, mijoter, mélanger, etc. La recette est indépendante des ustensiles utilisés comme l'algorithme l'est du langage de programmation.

Surtout, surtout, une recette explique les étapes à suivre pour obtenir le plat voulu, mais, sauf peut-être pour certaines opérations plus délicates, elle n'explique pas *comment* réaliser chaque étape. S'il faut ajouter du lait dans un mélange à gâteau, la recette n'indique pas de sortir le lait du réfrigérateur, de vérifier la date de péremption et de verser le lait dans la tasse à mesurer jusqu'à la ligne correspondant à la quantité voulue ! Un algorithme ne contient pas plus d'instructions pour vérifier la validité des entrées ou pour effectuer des opérations de base comme les opérations arithmétiques, la lecture de données, l'indilage d'un vecteur, etc.

En revanche, on s'attend généralement d'un algorithme, qui est destiné à un ordinateur, qu'il soit plus précis qu'une recette. En effet, un ordinateur éprouverait beaucoup de difficulté avec une instruction comme « saler et poivrer au goût » !



C'est une erreur commune de fournir des détails de mise en œuvre superflus dans un algorithme. Évitez d'expliquer comment mesurer une quantité de lait avec une tasse à mesurer de marque ACME.

2.2 Pseudocode

Puisqu'il s'agissait du premier exemple d'algorithme complet, j'ai choisi de présenter l'[algorithme 2.1](#) dans un format très près du langage naturel.

Une autre manière très populaire d'exprimer les algorithmes consiste à utiliser du pseudocode, un « langage » similaire à un langage de programmation, mais toujours sans référence à un langage en particulier. L'écriture en pseudocode permet parfois de mieux prendre la mesure de la structure et des détails d'un algorithme avant d'en attaquer la mise en œuvre.

Voici de nouveau l'algorithme d'Euclide, présenté cette fois en pseudocode. Portez-y une attention toute particulière puisqu'il ne s'agit pas d'une transcription directe de l'[algorithme 2.1](#). Vérifiez que les deux versions de l'algorithme retournent bien le même résultat.

Algorithme 2.2 (Algorithme d'Euclide, version en pseudocode). Calculer le plus grand commun diviseur de deux entiers positifs m et n avec, sans perte de généralité, $n < m$.

```

PGCD(entier m, entier n)
  Tant que (n ≠ 0)
    r ← m mod n
    m ← n
    n ← r
  Fin Tant que
  Retourner m
Fin PGCD

```

□



L'opération modulo, notée en mathématiques $m \bmod n$, représente le reste de la division de m par n . Par exemple, tel que calculé précédemment, $544 \bmod 119 = 68$.

La composition de pseudocode n'obéit pas à des normes universelles. L'[algorithme 2.2](#) reprend toutefois quelques-unes des conventions les plus usuelles.

La procédure de calcul faisant l'objet de l'algorithme est rédigée sous forme d'une fonction². L'algorithme débute par la signature de la fonction (son nom avec le nom et le type de tous les arguments). Le code est indenté (décalé vers la droite) après la signature de la fonction pour montrer qu'il fait partie du corps de la fonction.

Le bloc délimité par les instructions **Tant que** (*condition*) et **Fin Tant que** forme une boucle dont le contenu — lui aussi indenté — est exécuté tant et aussi longtemps que la *condition* est vraie.

L'instruction **Retourner** met immédiatement fin à la fonction, ici en retournant la valeur m .

Enfin, on présente le pseudocode comme du code informatique dans une police non proportionnelle qui préserve l'alignement vertical des caractères.

2.3 Évaluation conditionnelle

Plusieurs algorithmes, même les plus simples, nécessitent de pouvoir effectuer des opérations différentes selon le résultat d'un test. Par exemple, pour effectuer le banal calcul de la valeur absolue d'un nombre, nous devons vérifier si le nombre est positif, négatif ou nul pour retourner un résultat en fonction de la règle suivante :

$$|x| = \begin{cases} x, & \text{si } x > 0 \\ 0, & \text{si } x = 0 \\ -x, & \text{si } x < 0. \end{cases}$$

2. C'est la terminologie de R. Une unité de code qui effectue un traitement donné porte divers noms selon le langage de programmation : fonction, routine, sous-routine, méthode, procédure, sous-procédure...

De telles constructions requièrent des instructions conditionnelles qui permettent d'effectuer différents calculs selon le résultat d'une condition booléenne (vraie ou fausse). Il en existe quatre grands types.

- Les instructions conditionnelles à un volet de la forme

Si $\langle condition \rangle$, alors $\langle conséquence \rangle$

Dans cette construction, la $\langle conséquence \rangle$ est évaluée lorsque la $\langle condition \rangle$ est vraie. Dans le cas contraire, le déroulement de l'algorithme se poursuit normalement. Nous avons utilisé une telle instruction conditionnelle à l'étape 2 de l'[algorithme 2.1](#). La locution « alors » est souvent omise dans les algorithmes comme, d'ailleurs, dans la syntaxe de plusieurs langages de programmation.

- Les instructions conditionnelles à deux volets de la forme

Si $\langle condition \rangle$, alors $\langle conséquence \rangle$,
sinon $\langle alternative \rangle$

Cette construction permet d'effectuer un choix entre deux actions : ou bien la $\langle condition \rangle$ est vraie et la $\langle conséquence \rangle$ est évaluée, ou bien la $\langle condition \rangle$ est fausse et c'est alors l' $\langle alternative \rangle$ qui est évaluée. Le déroulement de l'algorithme se poursuit après l'exécution de l'une ou l'autre des deux actions.

- Les instructions conditionnelles imbriquées de la forme

Si $\langle condition_1 \rangle$, alors $\langle conséquence_1 \rangle$,
sinon si $\langle condition_2 \rangle$, alors $\langle conséquence_2 \rangle$,
sinon $\langle alternative \rangle$

Une instruction conditionnelle imbriquée permet de choisir entre trois actions : la $\langle conséquence_1 \rangle$ est évaluée lorsque la $\langle condition_1 \rangle$ est vraie ; la $\langle conséquence_2 \rangle$ est évaluée lorsque la $\langle condition_1 \rangle$ est fausse et que la $\langle condition_2 \rangle$ est vraie ; l' $\langle alternative \rangle$ est évaluée lorsque la $\langle condition_1 \rangle$ et la $\langle condition_2 \rangle$ sont toutes les deux fausses. Pour choisir entre plus de trois actions possibles, il suffit d'ajouter des clauses « sinon si..., alors... ».

- Une variante de l'instruction précédente qui facilite la lecture lorsqu'il y a un grand nombre d'actions possibles :

Selon que
 $\langle condition_1 \rangle$, alors $\langle action_1 \rangle$;
 $\langle condition_2 \rangle$, alors $\langle action_2 \rangle$;
 $\langle condition_3 \rangle$, alors $\langle action_3 \rangle$;
...
autrement, $\langle action \text{ par défaut} \rangle$

Seule l' $\langle action \rangle$ correspondant à la $\langle condition \rangle$ vraie est exécutée. Si aucune $\langle condition \rangle$ n'est vraie, c'est l' $\langle action \text{ par défaut} \rangle$ qui est exécutée. La terminologie française que j'utilise ci-dessus est plutôt rare dans la littérature. Dans les langages de programmation, le mot-clé utilisé pour ce type de construction est généralement l'un ou l'autre de `switch`, `case` ou `cond`.

Afin d'illustrer ce qui précède, exprimons le calcul de la valeur absolue sous forme d'algorithme. Je fournis d'un coup les versions en langage naturel et en pseudocode.

Algorithme 2.3 (Lang. nat.). Calculer la valeur absolue d'un nombre réel x .

1. Si $x > 0$, retourner x ; sinon si $x = 0$, retourner 0 ; sinon retourner $-x$. \square

Algorithme 2.3 (Pseudocode). Calculer la valeur absolue d'un nombre réel x .

```
abs(réel x)
  Si ( $x > 0$ )
    Retourner  $x$ 
  Sinon si ( $x = 0$ )
    Retourner 0
  Sinon
    Retourner  $-x$ 
Fin abs  $\square$ 
```

Pouvons-nous simplifier un peu tout cela ? Vous aurez sans doute remarqué que le cas $x = 0$ ne nécessite pas vraiment de traitement particulier. Il peut être combiné avec le cas $x > 0$ ou avec le cas $x < 0$ sans changer le résultat. C'est une leçon que vous devez retenir : un algorithme simple et efficace ne correspond pas toujours directement à une formule mathématique.

Voici une version de l'algorithme du calcul de la valeur absolue qui n'a recours qu'à une expression conditionnelle à un volet. Vous remarquerez que j'ai aussi inversé la logique. Présenté ainsi, l'algorithme met l'accent sur le fait qu'un traitement ne doit être effectué que pour les nombres négatifs.

Algorithme 2.3a (Lang. nat.). Calculer la valeur absolue d'un nombre réel x .

1. Si $x < 0$, retourner $-x$; sinon retourner x . \square

Algorithme 2.3a (Pseudocode). Calculer la valeur absolue d'un nombre réel x .

```
abs(réel x)
  Si ( $x < 0$ )
    Retourner  $-x$ 
  Sinon
    Retourner  $x$ 
Fin abs  $\square$ 
```

Nous pouvons aussi voir la fonction valeur absolue comme une fonction identité (qui retourne son argument inchangé), sauf lorsque l'argument est négatif. Ceci résulte en une autre version de l'algorithme qui se passe d'une clause « sinon ».³

3. Pensez-y : la clause « sinon » n'est jamais nécessaire lorsque la *(conséquence)* d'une instruction conditionnelle à deux volets contient une instruction « Retourner ».

Algorithme 2.3b (Lang. nat.). Calculer la valeur absolue d'un nombre réel x .

1. Si $x < 0$, retourner $-x$.
2. Retourner x .

Algorithme 2.3b (Pseudocode). Calculer la valeur absolue d'un nombre réel x .

```

□ abs(réel x)
  Si (x < 0)
    Retourner -x
  Retourner x
Fin abs

```

□

Jusqu'à maintenant, nous avons réussi à énoncer la *condition* des instructions conditionnelles uniquement à partir d'opérateurs mathématiques simples comme $<$, $=$ et $>$. Pour construire des conditions composées, il faut avoir recours aux opérateurs logiques ET, OU et NON.

- ▶ $\langle e_1 \rangle$ ET $\langle e_2 \rangle$ est vraie lorsque les expressions $\langle e_1 \rangle$ et $\langle e_2 \rangle$ sont toutes deux vraies ; sinon la condition est fausse.
- ▶ $\langle e_1 \rangle$ OU $\langle e_2 \rangle$ est vraie lorsque l'une ou l'autre des expressions $\langle e_1 \rangle$ et $\langle e_2 \rangle$ est vraie ; la condition est fausse uniquement lorsque $\langle e_1 \rangle$ et $\langle e_2 \rangle$ sont toutes deux fausses.
- ▶ NON $\langle e \rangle$ est vraie lorsque l'expression $\langle e \rangle$ est fausse, et fausse autrement.

Par exemple, la condition que l'on écrit mathématiquement $5 < x < 10$ s'exprime ainsi en algorithmique : $x > 5$ ET $x < 10$.

2.4 Itération et récursion

La plupart des tâches que nous confions à des ordinateurs sont répétitives. Après tout, c'est ce dans quoi ils excellent puisque, contrairement aux humains, ils ne se lassent jamais de répéter toujours la même opération.

Il existe deux grands types de processus répétitifs : les processus itératifs (ou qui utilisent l'itération) et les processus récursifs (ou qui utilisent la récursion).

Sortons un instant du contexte de la programmation informatique pour illustrer chaque type de processus. Nous voulons indiquer à une personne la procédure à suivre pour peindre une clôture longue de n planches. (Pour simplifier, nous allons omettre de peindre les supports horizontaux.)

Dans l'approche itérative, la plus naturelle, nous indiquons à la personne d'appliquer de la peinture sur les planches de la première à la dernière. En supposant que les planches sont numérotées de 1 à n , cela revient à indiquer d'appliquer de la peinture sur la planche i , pour i allant de 1 à n . Transcrites en pseudocode, les instructions vont comme suit.

```

Peinturer(entier n)
  Pour i de 1 à n
    AppliquerPeintureSurPlanche(i)

```

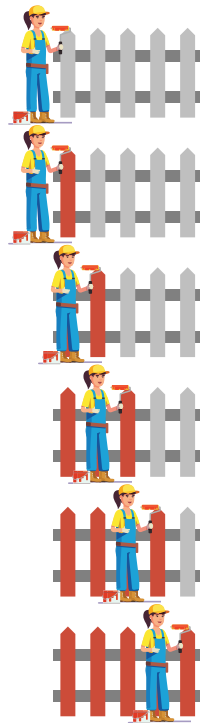


FIG. 2.1 – Processus itératif de peinture d'une clôture

Fin Pour
Fin Peinturer

La [figure 2.1](#) illustre le processus itératif.

L'approche récursive est plus difficile à bien saisir de prime abord. Une vieille blague dit d'ailleurs que pour comprendre la récursion, il faut d'abord comprendre la récursion! Dans cette approche, le processus de peindre une clôture de n planches est découpé en deux étapes : appliquer de la peinture sur une planche ; peindre une clôture de $n - 1$ planches tant qu'il y a des planches à peindre. Remarquez comment la procédure pour peindre une clôture est définie en fonction d'elle-même. Examinez bien le pseudocode correspondant ci-dessous.

```
Peinturer(entier n)
  Si (n = 0)
    Travail terminé
  AppliquerPeintureSurPlanche(n) ET Peinturer(n - 1)
Fin Peinturer
```

La personne qui suit les instructions récursives ne peut commencer à appliquer de la peinture tant qu'elle n'a pas rencontré la condition d'arrêt (l'extrémité de la clôture) puisque la description du travail à effectuer dépend d'elle-même. Une fois arrivée au critère d'arrêt, la personne dispose finalement de toutes les

informations pour faire son travail, qu'elle effectue finalement à rebours. Les processus récursifs se caractérisent par ces phases d'expansion (ou d'accumulation d'information) et de contraction (ou d'exécution des instructions). La [figure 2.2](#) illustre le processus récursif.

Nous pouvons aussi comparer les deux types de processus en observant ce qui se passerait si la personne devait interrompre son travail pour le confier à quelqu'un d'autre. Dans l'approche itérative, elle pourrait simplement indiquer qu'elle en était à appliquer de la peinture sur la planche m dans le processus de peindre une clôture de n planches. Le travail pourrait ensuite se poursuivre à partir de la planche $m + 1$. Dans l'approche récursive, cependant, la personne ne pourrait uniquement dire qu'elle en était à peindre une clôture de m planches, car elle aurait aussi bien pu se trouver dans la phase d'expansion que dans celle de contraction. Elle devrait donc aussi fournir des informations — qu'elle a mémorisées — sur l'« état » dans lequel elle se trouvait au moment d'interrompre son travail.

À ce stade, vous pourriez vous demander pourquoi donc parler de récursion puisque ça semble si compliqué, voire inefficace. Le fait est que la solution de plusieurs problèmes s'exprime beaucoup plus simplement de manière récursive que de manière itérative. Pensez au tri d'une main de cartes : il s'agit de placer la plus petite carte au début de la main, puis de trier le reste de la main, et ainsi de suite jusqu'à ce que la main soit triée au complet. L'algorithme d'Euclide est également récursif. Examinez de nouveau l'[algorithme 2.1](#) : l'étape 2 n'est rien d'autre que le critère d'arrêt et l'étape 3, le rappel de la procédure avec des arguments différents.

Dans l'absolu, les procédures récursives sont moins efficaces en termes de temps de calcul et de consommation de ressources, notamment parce qu'elles nécessitent une pile (*stack*) pour accumuler les informations durant la phase d'expansion. En pratique, toutefois, ces inconvénients tendent à diminuer ou à disparaître avec l'optimisation des compilateurs de plusieurs langages de programmation. De plus, la propriété de récursion terminale (*tail-recursion*) de certains langages, dont Scheme, fait en sorte que certaines procédures récursives sont en tous points équivalentes à des procédures itératives. Nous n'irons pas plus loin, mais si le sujet vous intéresse, je vous recommande de lire la section 1.2.1 de [Abelson et collab. \(1996\)](#). L'article de [Wikipédia \(2024c\)](#) constitue également un bon point de départ pour explorer le sujet plus en profondeur.

Je ne pourrais prétendre avoir écrit un ouvrage d'introduction à la programmation sans traiter de l'exemple classique quand il s'agit de comparer itération et récursion : le calcul de la factorielle. La fonction factorielle est définie par

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1.$$

Or, puisque

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!,$$

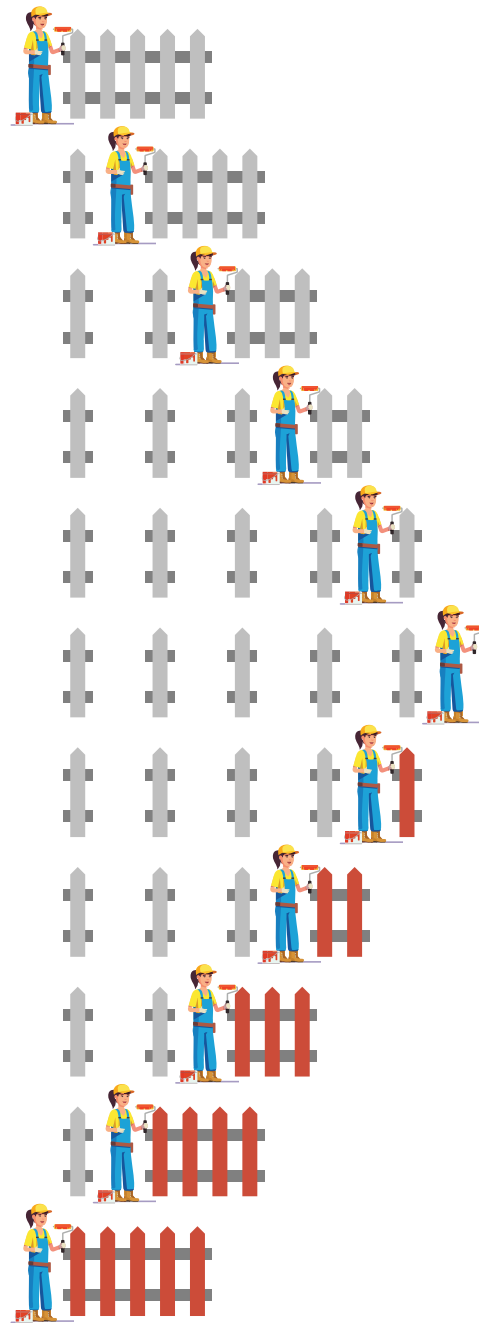


FIG. 2.2 – Processus récursif de peinture d'une clôture. La séparation d'une planche symbolise la mise en attente de cette étape.

```

factorial(6)
6 × factorial(5)
6 × (5 × factorial(4))
6 × (5 × (4 × factorial(3)))
6 × (5 × (4 × (3 × factorial(2))))
6 × (5 × (4 × (3 × (2 × factorial(1)))))
6 × (5 × (4 × (3 × (2 × 1))))
6 × (5 × (4 × (3 × 2)))
6 × (5 × (4 × 6))
6 × (5 × 24)
6 × 120
720

```

FIG. 2.3 – Processus récursif du calcul de 6!

la fonction est récursive et une manière toute naturelle de calculer la factorielle d'un nombre utilise la récursion. Exprimons cela sous forme d'algorithme. Le pseudocode sied un peu mieux aux algorithmes récursifs.

Algorithme 2.4 (Factorielle, approche récursive). Calculer la factorielle d'un entier positif n .

```

factorial(entier positif n)
  Si (n = 0)
    Retourner 1
  Retourner n × factorial(n - 1)
Fin factorial

```

□

La [figure 2.3](#) illustre le processus de calcul récursif de la valeur de 6!. Remarquez comme la structure est similaire à celle du processus de peinture d'une clôture de la [figure 2.2](#).

Nous pouvons également voir $n!$ comme le résultat du produit de 1 par 2, puis le produit du résultat par 3, puis par 4, et ainsi de suite jusqu'à n . Avec cette approche, nous devons enregistrer le résultat courant du produit ainsi qu'un compteur qui nous permet de savoir où nous en sommes dans le calcul. Cela devrait vous rappeler l'approche itérative de la peinture d'une clôture, car c'est bien ce dont il s'agit : le calcul de la factorielle de manière itérative. Exprimons cette fois l'algorithme en langage naturel.

Algorithme 2.5 (Factorielle, approche itérative). Calculer la factorielle d'un entier positif n .

1. Poser $p \leftarrow i \leftarrow 1$.
2. Si $i = n$, retourner la valeur p .
3. Incrémenter le compteur i : $i \leftarrow i + 1$.

```

factorial(6)
i ← 1; p ← 1
i ← 2; p ← 2
i ← 3; p ← 6
i ← 4; p ← 24
i ← 5; p ← 120
i ← 6; p ← 720
720

```

FIG. 2.4 – Processus itératif du calcul de 6!

4. Poser $p \leftarrow p \times i$, puis retourner à l'étape 2. □

La [figure 2.4](#) illustre le processus de calcul itératif de la valeur de 6!. Sa structure est cette fois similaire à celle de [figure 2.1](#). Remarquez comment les valeurs de p et de i enregistrent l'état du processus, ce qui nous permettrait de l'interrompre et de le relancer à tout moment.

2.5 Nombre d'opérations

Imaginons la situation suivantes : vous devez calculer, pour un jeu de données quelconque, l'écart moyen des données supérieures à d par rapport à cette valeur. Le résultat est l'espérance résiduelle empirique des données évaluée à d . L'algorithme de haut niveau suivant permet d'effectuer ce calcul.

Algorithme 2.6. Calculer l'espérance résiduelle empirique à d d'un jeu de données.

1. Sélectionner les données supérieures à d .
2. Soustraire d de chaque valeur retenue à l'étape 1.
3. Retourner la moyenne des valeurs obtenues à l'étape 2. □

Nous pouvons aussi concevoir une variante de l'algorithme ci-dessus en y apportant une toute petite modification.

Algorithme 2.6a. Calculer l'espérance résiduelle empirique à d d'un jeu de données.

1. Sélectionner les données supérieures à d .
2. Effectuer la moyenne des valeurs obtenues à l'étape 1.
3. Retourner la différence entre la moyenne obtenue à l'étape 2 et d . □

Mathématiquement, les deux approches sont tout à fait équivalentes. Cependant, avec l'[algorithme 2.6](#), nous effectuons une soustraction pour chaque valeur

supérieure à d que compte le jeu de données. Avec l'[algorithme 2.6a](#), nous n'effectuons qu'une seule soustraction. Si le jeu de données compte un million d'entrées supérieures à d , c'est un million de soustractions contre une seule. En temps de calcul, cela ne représente qu'un écart de quelques centièmes de secondes sur un ordinateur moderne, mais ces fractions de secondes peuvent finir par faire une différence importante lorsque la taille des jeux de données augmente ou lorsque l'opération se répète un très grand nombre de fois.

Le dénombrement du nombre d'opérations requis par un algorithme est un aspect important de leur analyse. Il se fait généralement en notation $O(f(n))$ qui signifie « de l'ordre de $f(n)$ », où $f(n)$ est une fonction d'un paramètre n qui mesure la taille du problème. Dans l'exemple ci-dessus, n est le nombre d'entrées dans le jeu de données. Pour un algorithme de calcul du produit de deux matrices, n pourrait être le nombre de lignes des matrices.

Attardons-nous à un second exemple intéressant : l'élévation d'une valeur b à la puissance n . L'approche la plus intuitive consisterait à utiliser la relation

$$b^n = b(b(b \cdots (b))) = b \cdot b^{n-1},$$

ce qui mène à un premier algorithme.

Algorithme 2.7. Élever un nombre b à la puissance entière positive n .

Puissance(nombre réel b , entier positif n)

Si ($n = 0$)

Retourner 1

Retourner $b \times$ Puissance(b , $n - 1$)

Fin Puissance

□

Vous aurez tout de suite identifié que l'[algorithme 2.7](#) est récursif. Le nombre d'opérations effectué avec cet algorithme est directement proportionnel à n . Par exemple, il requiert 5 opérations pour élever un nombre à la puissance 6. On dit donc que l'ordre de grandeur de l'algorithme est $O(n)$.

L'[algorithme 2.7](#) peut convenir lorsque n est petit, mais peut-on faire mieux pour une « grande » valeur de n ? Imaginez que vous ne disposez que d'une calculatrice munie des opérations arithmétiques de base et du carré, et que vous devez élever un nombre à la puissance, disons, 21. Comment feriez-vous pour réduire le nombre d'opérations à entrer au clavier ?



Cet exemple est beaucoup moins fantaisiste qu'il n'y paraît ! Jusqu'au milieu des années 1990, les étudiants en actuariat ne disposaient que de ce type de calculatrice pour les examens professionnels. Il s'agissait d'un modèle de Texas Instruments affectueusement surnommé « TI-zéro ».

Vous avez pensé à un algorithme ? Votre idée consiste fort probablement à diviser la puissance par deux autant de fois que possible et à élever au carré par

la suite. Pour $n = 21$, cela donne

$$\begin{aligned} b^{21} &= b(b^{20}) \\ &= b(b^{10})^2 \\ &= b((b^5)^2)^2 \\ &= b((b(b^4))^2)^2 \\ &= b((b(b^2)^2)^2)^2. \end{aligned}$$

Ce calcul se traduit en un nouvel algorithme récursif.

Algorithme 2.7a. Élever un nombre b à la puissance entière positive n .

Puissance(nombre réel b , entier positif n)

Si ($n = 0$)

Retourner 1

Si (n est pair)

Retourner (Puissance(b , $n/2$))²

Si (n est impair)

Retourner $b \times$ Puissance(b , $n - 1$)

Fin Puissance

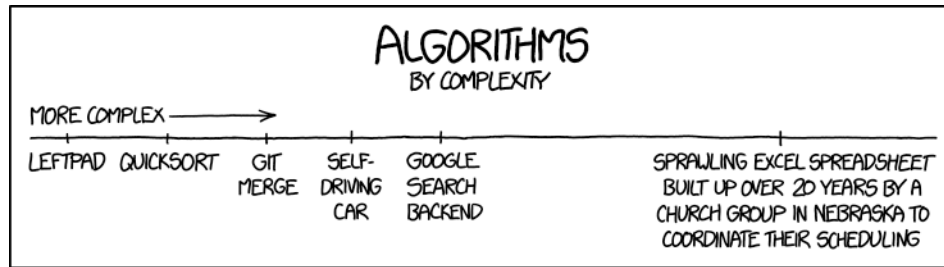
□

Là où l'[algorithme 2.7](#) requiert 20 opérations pour élever un nombre à la puissance 21, l'[algorithme 2.7a](#) n'en demande que 6. Comme la puissance est divisée par deux à répétition, le nombre d'opérations est proportionnel au logarithme (en base deux) de n . Pour vous en convaincre, observez que le calcul de b^{2^n} effectue une seule opération de plus que le calcul de b^n . L'ordre de grandeur de l'algorithme est donc $O(\log n)$.

La mesure d'ordre de grandeur est très approximative. Par exemple, un processus nécessitant n^2 étapes, un processus en nécessitant $1000n^2$ et un autre en nécessitant $3n^2 + 5n + 17$ ont tous un ordre de grandeur $O(n^2)$ puisque les constantes ne sont pas prises en compte.

Quelques ordres de grandeur surviennent plus souvent dans l'étude des algorithmes. En voici une liste partielle, question de vous offrir un peu de perspective si vous lisez que la performance d'un algorithme d'ordre $O(n^3)$ est tout à fait acceptable.

- $O(1)$. Un algorithme d'ordre $O(1)$ prend toujours le même temps pour effectuer ses calculs, peu importe la taille du problème. Il va sans dire que de tels algorithmes sont excessivement rares. Ils effectuent en général des tâches très simples.
- $O(\log n)$. Tel que mentionné précédemment, l'ordre de grandeur $O(\log n)$ survient dès lors que la taille du problème est divisée par deux à chaque étape.
- $O(\sqrt{n})$. Les algorithmes d'ordre $O(\sqrt{n})$ sont rares. La fonction racine carrée croît lentement, mais néanmoins plus rapidement que la fonction logarithme.



Tiré de XKCD.com

- ▶ $O(n)$. Nous avons rencontré un exemple d'algorithme $O(n)$ ci-dessus. Les algorithmes avec cette performance demeurent acceptables étant donné le rythme de croissance somme toute raisonnable de la fonction identité.
- ▶ $O(n \log n)$. Les algorithmes qui répètent n fois un calcul d'ordre $O(\log n)$ ont une performance d'ordre $O(n \log n)$. C'est le cas de plusieurs algorithmes de tri.
- ▶ $O(n^2)$. Les algorithmes qui effectuent un calcul pour toutes les paires d'entrées ont une performance $O(n^2)$. Ici, ça commence à devenir lent.
- ▶ $O(n!)$. Attention, danger. La fonction factorielle croît si rapidement qu'un algorithme ayant ce type de performance ne peut résoudre que de petits problèmes.

Pour en savoir plus sur les principes de base du dénombrement des opérations d'un algorithme, consultez [Stephens \(2013, chapitre 1\)](#).

2.6 Exercices

- 2.1** Écrire l'algorithme d'une fonction `quadroots` qui prend en arguments les coefficients (réels) a , b et c d'un polynôme du second degré $ax^2 + bx + c$ et qui calcule les racines du polynôme avec la formule bien connue

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

La fonction doit retourner deux, une ou aucune racine selon que le discriminant $b^2 - 4ac$ est positif, nul ou négatif, dans l'ordre.

- 2.2** Expliquer dans vos propres mots pourquoi la clause « sinon » n'est pas nécessaire lorsque la *conséquence* d'une instruction conditionnelle à deux volets contient une instruction « Retourner ».
- 2.3** Plusieurs algorithmes comportent une étape d'échange de deux valeurs, que l'on pourrait symboliser par $m \leftrightarrow n$.
- a) Expliquer comment effectuer l'opération $m \leftrightarrow n$ sans perdre l'une ou l'autre des valeurs m et n . *Indice* : il faut avoir recours à une troisième variable temporaire, disons t .

- b) Expliquer comment réordonner les valeurs (a, b, c, d) en (b, c, d, a) en effectuant un minimum de remplacements.

2.4 Soit p et q des expressions logiques. Compléter les tables de vérité ci-dessous.

	p	q	p ET q
	V	V	
a)	V	F	
	F	V	
	F	F	

	p	q	p OU q
	V	V	
b)	V	F	
	F	V	
	F	F	

2.5 La notation en algèbre booléenne est plus compacte et lisible que celle utilisée à la [section 2.3](#). Si p et q sont des expressions logiques, on représente généralement p ET q par $p \wedge q$, p OU q par $p \vee q$, et NON p par \bar{p} .

Soit p, q et r des expressions logiques. Démontrer à l'aide de tables de vérité les identités logiques suivantes.

- $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
- $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
- $\overline{p \wedge q} = \bar{p} \vee \bar{q}$
- $\overline{p \vee q} = \bar{p} \wedge \bar{q}$
- $(p \wedge q) \vee (\bar{p} \wedge r) = (p \wedge q) \vee (\bar{p} \wedge r) \vee (q \wedge r)$

2.6 Écrire une version récursive en pseudocode de l'algorithme d'Euclide.

2.7 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 $\phi = (1 + \sqrt{5})/2$, le nombre d'or⁴. Le calcul de $f(n)$ se prête tout naturellement à une mise en œuvre récursive.

- Proposer un algorithme récursif de calcul de la valeur $f(n)$ de la suite de Fibonacci.
- Calculer les 10 premiers nombres de Fibonacci.

4. On prête au nombre d'or toutes sortes de propriétés, certaines même d'ordre mystique. Il est en tout cas au cœur d'un des romans les plus vendus de tous les temps, *Da Vinci Code* (Dan Brown, 2003).

- c) L'algorithme récursif direct de la suite de Fibonacci est terriblement inefficace puisqu'il répète plusieurs calculs un grand nombre de fois. Pour vous en convaincre, écrivez au long tous les calculs requis par l'algorithme pour calculer $f(5)$.
- d) Proposer un algorithme itératif en langage naturel de calcul des nombres de Fibonacci.

2.8 L'algorithme suivant permet de calculer la valeur de la fonction de Ackermann $A(x, y)$ pour des entiers non négatifs x et y telle que définie dans [Abelson et collab. \(1996\)](#).

```

A(entier x, entier y)
  Si (y = 0)
    Retourner 0
  Sinon Si (x = 0)
    Retourner 2 * y
  Sinon Si (y = 1)
    Retourner 2
  Sinon
    Retourner A(x - 1, A(x, y - 1))
Fin A

```

Calculer la valeur de $A(1, 10)$.

2.9 On vous propose ci-dessous trois algorithmes différents pour calculer et afficher à l'écran les diviseurs d'un nombre naturel n .

Algorithme A. Afficher les diviseurs d'un nombre naturel n .

1. Poser $i \leftarrow 1$.
2. Si $n \bmod i = 0$, afficher la valeur i .
3. Incrémenter $i : i \leftarrow i + 1$.
4. Si $i \leq n$, retourner à l'étape 2. □

Algorithme B. Afficher les diviseurs d'un nombre naturel n .

1. Poser $i \leftarrow 1$.
2. Si $n \bmod i = 0$, afficher les valeurs de i et n/i .
3. Incrémenter $i : i \leftarrow i + 1$.
4. Si $i < n/2$, retourner à l'étape 2. □

Algorithme C. Afficher les diviseurs d'un nombre naturel n .


1. Poser $i \leftarrow 1$.
2. Si $n \bmod i = 0$, afficher les valeurs de i et n/i .
3. Incrémenter $i : i \leftarrow i + 1$.

4. Si $i \leq \lfloor \sqrt{n} \rfloor$, où $\lfloor x \rfloor$ représente le plus grand entier inférieur ou égal à x , retourner à l'étape 2. \square

- a) Calculer la performance de chacun des algorithmes.
- b) Évaluer chacun des algorithmes (à la lettre!) pour $n = 12$ et pour $n = 16$.
- c) Proposer une version de l'algorithme C qui corrige les deux erreurs d'affichage identifiées en b). *Indice* : entreposer les diviseurs dans un « vecteur » (un contenant de valeurs).

2.10 Calculer la performance des algorithmes ci-dessous⁵.

- a) Un algorithme avec n entrées qui génère des valeurs pour chacun des carrés unitaires à la surface d'un cube $n \times n \times n$, tel qu'illustré à la gauche de la [figure 2.5](#).
- b) Un algorithme avec n entrées qui génère des valeurs pour chacun des cubes unitaires sur le pourtour d'un cube $n \times n \times n$, tel qu'illustré à la droite de la [figure 2.5](#).

2.11 Soit un algorithme avec n entrées qui génère des valeurs pour chacun des carrés unitaires d'une pyramide comme dans le jeu classique **Q*bert** , tel qu'illustré à la [figure 2.6](#). Supposer que les cubes invisibles sont présents de telle sorte que les structures sont pleines. Calculer la performance de l'algorithme⁶.

2.12 Comparer à l'aide d'un graphique pour $n \in (0, 20)$ le rythme de croissance des fonctions $\log n$, \sqrt{n} , n , $n \log n$, n^2 et $n!$.

5. Exercice tiré de [Stephens \(2013\)](#).

6. *Idem*.

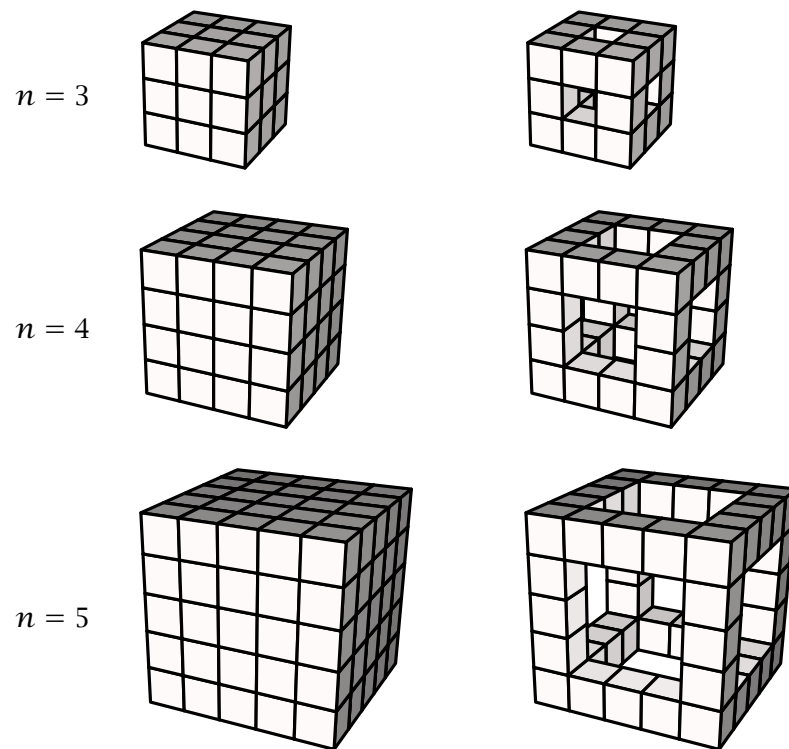


FIG. 2.5 - Illustrations pour des algorithmes qui génèrent des valeurs sur la surface d'un cube (gauche) et sur le pourtour d'un cube (droite)

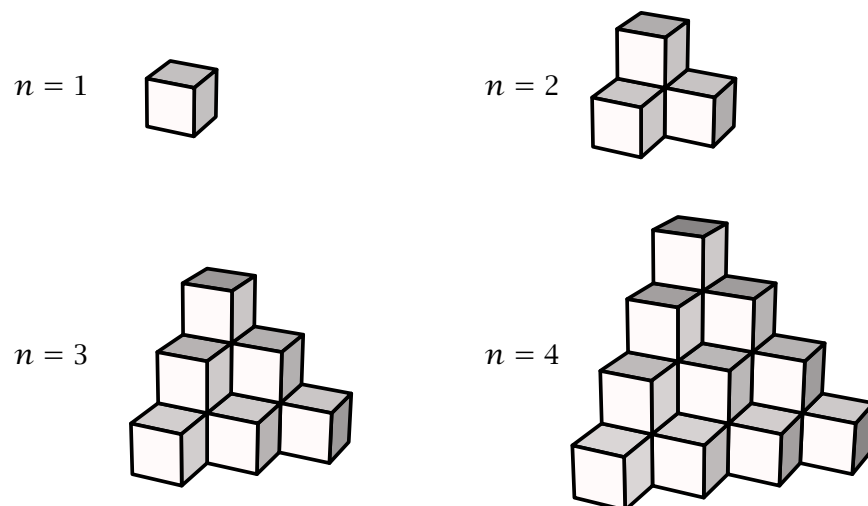


FIG. 2.6 - Illustrations pour un algorithme qui génère des valeurs pour une pyramide comme dans le jeu Q*bert

3 Présentation de R

Objectifs du chapitre

- ▶ Connaître la provenance du langage R et les principes ayant guidé son développement.
- ▶ Utiliser R de manière interactive.
- ▶ Créer, modifier et utiliser des fichiers de script 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. Le présent ouvrage met l'accent sur l'apprentissage du langage de programmation. Néanmoins, vous devrez l'utiliser à l'intérieur de l'interface de l'environnement statistique. Ce chapitre vise à vous familiariser avec celle-ci, avec les fichiers de script et avec l'utilisation d'un éditeur de texte ou d'un environnement de développement intégré.

3.1 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 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. Le système S a ensuite été


```
(define factorial (lambda (n)
  (if (= n 1)
      1
      (* n (factorial (- n 1))))))
```

```
factorial <- function(n)
  if (n == 1)
    1
  else
    n * factorial(n - 1)
```

FIG. 3.1 – Mises en œuvre en Scheme (à gauche) et en S (à droite) de la fonction factorielle

commercialisé de manière relativement confidentielle sous le nom Spotfire S+ par TIBCO Software. Le terme S+ est aujourd’hui disparu du nom du produit.

Ce qui a fortement contribué à la perte d’influence de S-PLUS, c’est une nouvelle mise en œuvre du langage développée à partir du milieu des années 1990 par Ross Ihaka et Robert Gentleman. Ceux-ci basent leur nouveau langage sur Scheme (un dialecte du Lisp), sans doute inspirés en cela par les travaux de Luke Tierney sur le système statistique LISP-STAT (Tierney, 1990). S’ils conservent en grande partie la sémantique de Scheme, Ihaka et Gentleman ont toutefois la bonne idée d’adopter la syntaxe du S, qu’ils jugent plus intuitive pour les statisticiens. La comparaison des mises en œuvre en Scheme et en S de la toute simple fonction factorielle, à la figure 3.1, tend à leur donner raison.

Afin de souligner à la fois l’influence de S et leurs propres efforts, Ross Ihaka et Robert Gentleman choisissent de nommer leur langage pour l’analyse de données et les graphiques avec la première lettre de leur prénom respectif : R (Ihaka et Gentleman, 1996). À la suggestion de Martin Mächler de l’ETH Zurich, ils décident d’intégrer leur nouveau langage au projet GNU , 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).

À partir du début de la décennie 2010, l’analyse de données, l’analyse prédictive et l’intelligence artificielle font une entrée fracassante dans la liste des outils d’affaires incontournables. Déjà bien établi dans ces domaines, R devient, avec Python, la *lingua franca* de la science des données.

3.2 Description sommaire de R

Tel que mentionné précédemment, le R est un langage principalement inspiré de Scheme (Abelson et collab., 1996) et du S. Ce dernier était à son tour inspiré de plusieurs langages, dont l’APL et le Lisp (encore lui). Comme tous ses prédéces-

seurs, R est un langage interprété. Par conséquent, le programme que l'on lance lorsque l'on exécute R est en fait l'interpréteur. Celui-ci attend que l'on lui soumette des commandes dans le langage R — dans la suite nous dirons que ces commandes forment une *expression*. L'interpréteur évalue immédiatement l'expression pour ensuite afficher le résultat.

Par exemple, si l'on entre simplement un nombre à la ligne de commande (identifiée ci-dessous par le symbole >), l'interpréteur retourne la valeur de ce nombre.

```
> 42  
[1] 42
```

Nous pouvons aussi composer des expressions plus élaborées qui combinent des nombres et des opérateurs mathématiques standards.

```
> 2 + 3  
[1] 5  
> 1000 - 958  
[1] 42  
> 5 * 7  
[1] 35  
> 10/5  
[1] 2  
> 2.7 + 12.8  
[1] 15.5
```

L'un des buts de cet ouvrage consiste justement à apprendre à construire des suites d'expressions qui forment des programmes et qui permettent de réaliser une tâche donnée.

Risquons une analogie avec le tableur Excel, qui est aussi un logiciel de manipulation de données, de mise en forme et de préparation de graphiques. Or, au sens large, Excel est également un langage de programmation interprété ! Pensez-y : vous utilisez le langage de programmation lorsque vous entrez des commandes dans une cellule d'une feuille de calcul, puis l'interpréteur évalue les commandes et affiche les résultats dans la cellule.

R est un langage particulièrement puissant pour les applications mathématiques et statistiques, chose peu surprenante puisqu'il a précisément été développé dans ce but. Parmi ses caractéristiques 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;

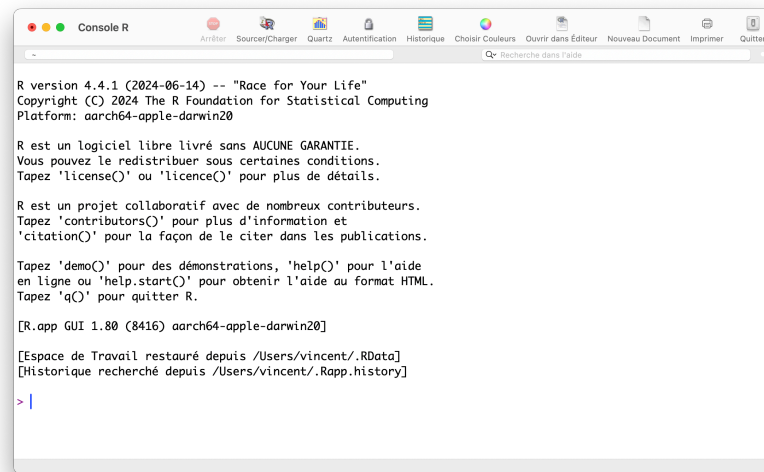


FIG. 3.2 – Fenêtre de la console sous macOS au démarrage de R

- temps de développement très court.



Visionnez la vidéo de [présentation de R](#).

3.3 Interfaces

R est d'abord et avant tout une application en ligne de commande, comme le montre la [figure 3.2](#). 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 commande¹, par ailleurs aux capacités d'édition plutôt réduites.

Sous Windows, l'interface graphique fournie avec R est rudimentaire. Elle facilite certaines opérations tel que l'installation de paquetages externes, mais elle n'offre autrement que peu de fonctionnalités pour l'édition de code R.

L'interface graphique de R sous macOS est la plus élaborée. Outre la console présentée à la [figure 3.2](#), 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. 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 — je recommande d'interagir avec R par le biais d'un éditeur de texte pour

1. Merci à Markus Gesmann pour cette observation.

programmeur ou d'un environnement de développement intégré. Nous y reviendrons à la [section 3.5](#); le choix de cette interface coulera de source une fois que vous connaîtrez la stratégie de travail que nous adopterons avec R.

3.4 Stratégies de travail

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

1. **Code virtuel et objets réels** L'approche dite de « code virtuel et objets réels » semble la plus naturelle avec les interfaces standards de R, mais elle souffre de sévères limites. Selon cette approche, les expressions R sont entrées à la ligne de commande et évaluées immédiatement.

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

Il est aussi possible d'enregistrer le résultat d'un calcul dans un objet. Cet objet est alors stocké en mémoire dans l'espace de travail (*workspace*) de R.

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

À la fin de la session de la session de travail, l'image de l'espace de travail est sauvegardée sur le poste de travail afin de pouvoir conserver les objets pour une future séance de travail. Par défaut, l'image est sauvegardée dans un fichier nommé `.RData` dans le répertoire de travail (plus de détails à la [section 3.6](#)) 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 a toutefois un énorme inconvénient : le code utilisé pour créer les objets n'est pas sauvegardé entre les sessions de travail. Sans accès au code qui a servi à créer l'objet `x`, comment savoir ce que la valeur `7.389056` représente au juste ? Si l'on poursuit l'analogie avec un tableur, cela correspondrait à ne sauvegarder que les valeurs calculées d'une feuille de calcul, sans les formules.

2. **Code réel et objets virtuels** L'approche dite de « code réel et objets virtuels » considère que ce qu'il importe de conserver au terme d'une session de travail, ce ne sont pas tant les objets que le code qui a servi à les créer. Ainsi, nous aurons recours à des fichiers de script pour sauvegarder nos expressions R et

le code de nos fonctions. Par convention, on donne aux fichiers de script un nom se terminant par l'extension `.R`.

Les objets — fonctions et « variables » contenant des résultats de calculs — sont créés au besoin en évaluant à la volée le code des fichiers de script. Comment ? Simplement en copiant le code du fichier de script et en le collant dans la ligne de commande de R. La [figure 3.3](#) illustre schématiquement ce que le programmeur R a constamment sous les yeux : son fichier de script et la ligne de commande R dans laquelle son code a été évalué. Rassurez-vous : les éditeurs de texte et les environnements de développement intégrés dont traite la prochaine section rendent l'opération de copier-coller simple et rapide.

Les tableurs adoptent en fait cette approche « code réel et objets virtuels » : lorsque vous sauvegardez une feuille de calcul, le tableur enregistre les formules qui se trouvent dans les cellules (et non les valeurs) et il réévalue celles-ci à l'ouverture de la feuille, puis constamment et en temps réel durant la session de travail pour afficher les résultats dans les cellules.

La méthode d'apprentissage préconisée dans cet ouvrage suppose que vous utiliserez cette seconde approche d'interaction avec R.



Dans les fichiers de script comme à la ligne de commande, le caractère « # » marque le début d'un commentaire dans R. Ainsi, tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par l'interpréteur.

3.5 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 en format texte brut, 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.

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. Enfin, un éditeur compatible avec R réduira l'opération de copier-coller du fichier de script vers la ligne de commande R à un simple raccourci-clavier.

Vous pouvez utiliser le logiciel de votre choix pour l'édition de code R. Certains éditeurs offrent simplement plus de fonctionnalités que d'autres. Voici quelques bons choix.


```
## Fichier de script simple contenant des expressions R pour
## faire des calculs et créer des objets. Tout le texte qui
## suit le caractère # est ignoré.
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 4.3.2 (2023-10-31) -- "Eye Holes"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

[...]

```
> ## Fichier de script simple contenant des expressions R pour
> ## faire des calculs et créer des objets. Tout le texte qui
> ## suit le caractère # est ignoré.
> 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. 3.3 – Fichier de script (en haut) et ligne de commande R dans laquelle les expressions R ont été évaluées (en bas).

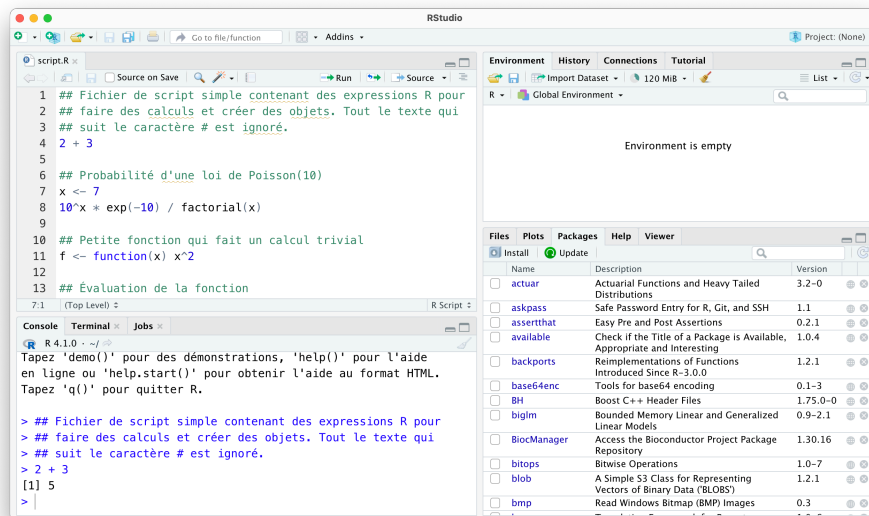


FIG. 3.4 – RStudio sous macOS dans sa configuration par défaut. On retrouve dans la fenêtre, dans le sens des aiguilles d’une montre : le fichier de script de la [figure 3.3](#); la liste des objets de l’environnement (vide, ici); une interface pour charger des paquetages; la ligne de commande R.

- 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 3.4](#). C’est probablement le meilleur choix d’éditeur pour la plupart des personnes qui débutent avec R.

Logiciel libre dans sa version pour ordinateur personnel, RStudio est disponible pour les plateformes Windows, macOS et Linux. L’[annexe A](#) propose une introduction à RStudio.

- Dans la catégorie des éditeurs de texte, je recommande le vénérable et très puissant éditeur pour programmeur GNU Emacs. À la question 6.2 de la foire aux questions de R ([Hornik et R Core Team, 2024](#)), « Devrais-je utiliser R à l’intérieur de Emacs ? », la réponse est : « Oui, assurément. » En effet, combiné avec le mode ESS (*Emacs Speaks Statistics*), Emacs offre un environnement de développement particulièrement riche et efficace. Il a d’ailleurs servi d’inspiration à RStudio à maints égards, notamment pour la disposition du fichier de script et de la ligne de commande R dans une seule fenêtre. La [figure 3.5](#) présente une fenêtre de GNU Emacs en mode d’édition de code R.

Emblème du logiciel libre, Emacs est disponible gratuitement et est identique

sur toutes les plateformes supportées par R, dont Windows, macOS et Linux. Consulter l'[annexe B](#) pour en savoir plus sur GNU Emacs et apprendre les commandes essentielles pour y faire ses premiers pas.

Le principal avantage de Emacs par rapport à RStudio réside dans le fait qu'il s'agit d'un éditeur pour programmeur généraliste qui sait s'adapter à n'importe quel langage — y compris R. Cela permet d'utiliser un seul et même outil pour toutes ses tâches de programmation, tel que recommandé par [Hunt et Thomas \(1999, chapitre 3\)](#) :

Utilisez un seul éditeur et utilisez-le bien. Votre éditeur devrait devenir une extension de votre main; assurez-vous qu'il est configurable, extensible et programmable.

Emacs satisfait les trois conditions ci-dessus. Cependant, la puissance de l'éditeur a un coût : Emacs est un logiciel difficile à apprivoiser, surtout pour les personnes moins à l'aise avec l'informatique. Si vous en êtes à votre première expérience avec un éditeur de texte, optez pour RStudio.

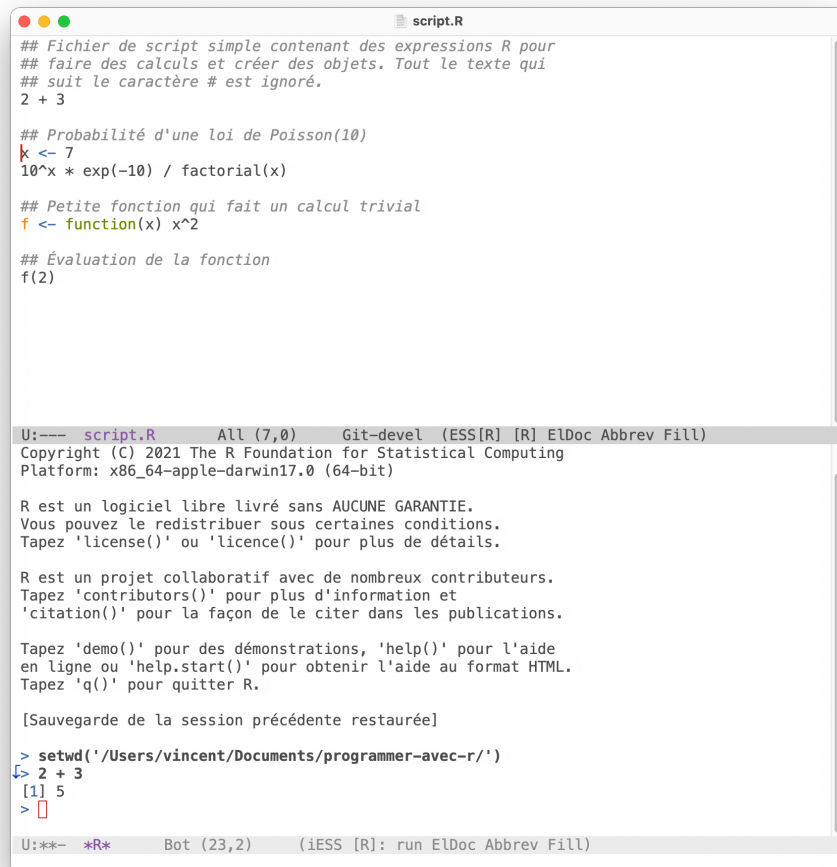
- Il existe plusieurs autres options pour éditer efficacement du code R — et le Bloc-notes de Windows n'en fait *pas* partie! Je recommande plutôt :
 - sous Windows, l'éditeur Notepad++ additionné de l'extension NppToR ([Redd, 2010](#)), tous deux des logiciels libres, ou le partagiciel WinEdt muni de 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.

3.6 Répertoire de travail

Une difficulté à laquelle se butent presque invariablement les personnes qui débutent avec R peut se résumer par ce cri de désespoir : « R ne trouve pas mes fichiers! » Pour éviter de vous retrouver dans cette situation, vous devez dès maintenant comprendre le concept de répertoire de travail (*working directory*) de R.

Durant son exécution, l'interpréteur R garde les yeux rivés sur un seul répertoire de votre poste de travail : c'est son répertoire de travail. C'est dans ce répertoire que R va, d'une part, rechercher par défaut des fichiers de script ou de données, ou, d'autre part, sauvegarder l'espace de travail dans le fichier `.RData`. À chaque session R son propre répertoire de travail : ce n'est pas plus compliqué que ça.

La fonction `getwd` permet d'afficher le répertoire de travail de R.



```

script.R
## Fichier de script simple contenant des expressions R pour
## faire des calculs et créer des objets. Tout le texte qui
## suit le caractère # est ignoré.
2 + 3

## Probabilité d'une loi de Poisson(10)
k <- 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)

U:--- script.R All (7,0) Git-devel (ESS[R] [R] ElDoc Abbrev Fill)
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

[Sauvegarde de la session précédente restaurée]

> setwd('/Users/vincent/Documents/programmer-avec-r/')
[1] 5
>
U:*** *R* Bot (23,2) (iESS [R]: run ElDoc Abbrev Fill)

```

FIG. 3.5 – GNU Emacs sous macOS en mode d'édition de code R. Dans la partie du haut de la fenêtre, on retrouve le fichier de script de la [figure 3.3](#) et dans la partie du bas, la ligne de commande R.

```

> getwd()
[1] "/Users/vincent"

```

Comment le répertoire de travail est-il déterminé ? Cela varie selon l'interface utilisée. Avec les interfaces graphiques de R pour Windows et macOS, ainsi qu'avec RStudio, c'est généralement votre répertoire personnel, comme dans l'exemple ci-dessus². Avec Emacs, c'est plus simple : vous déterminez le répertoire de travail au lancement d'un processus R ; voir l'[annexe B](#).

2. Les foires aux questions spécifiques aux interfaces graphiques de R (Ripley et Murdoch, 2024 ; Iacus et collab., 2023) contiennent des détails additionnels sur la gestion du répertoire de travail.

La fonction `setwd` permet, quant à elle, de changer le répertoire de travail.


```
> setwd("~/Documents/programmer-avec-r")
> getwd()
[1] "/Users/vincent/Documents/programmer-avec-r"
```

Dans les interfaces graphiques de R, vous pouvez aussi changer le répertoire de travail à partir du menu **Fichier** (Windows) ou **Divers** (macOS). Dans RStudio, c'est via le menu **Session**|**Set Working Directory**.

3.7 Organisation des projets et des fichiers

Durant vos premiers pas avec R, vous aurez sans doute tendance à regrouper tous vos fichiers de script dans un même répertoire sur votre poste de travail. Cela risque toutefois de devenir problématique lorsque les projets et les fichiers se multiplieront. Or, votre ordinateur est justement doté d'un puissant système de classement : son système de fichiers ([section 1.5](#)).

Avec sa logique du répertoire de travail, R impose en quelque sorte une organisation des projets et des fichiers qui s'arrime tout naturellement au système de fichiers de l'ordinateur. Veillez donc à adopter dès le départ les bonnes pratiques suivantes en matière d'organisation des projets et des fichiers.

1. Utiliser un répertoire par projet et y regrouper tous les fichiers de ce projet.
2. Faire de ce répertoire le répertoire de travail de R dès le lancement de la session, soit via la commande `setwd`, soit via un menu d'une interface graphique.
3. Placer le code sous contrôle de versions avec un système tel que [Git](#) .

3.8 Programmer pour collaborer

Je l'ai déjà mentionné dans l'introduction de cet ouvrage : la programmation se pratique beaucoup plus souvent en groupe que seul. Autrement dit, vous serez fort probablement appelé à un jour partager vos fichiers de script avec des collaborateurs — ou l'univers entier, via des plateformes de partage de code informatique. Dans de telles circonstances, vous devrez fournir des fichiers de script *portables*, c'est-à-dire susceptibles d'être utilisés sur d'autres systèmes sans modification. Suivez dès maintenant les préceptes suivants dans la rédaction de vos fichiers de script et vous serez sur la bonne voie.

1. Ne jamais utiliser la commande `setwd` dans un fichier de script. Changez plutôt le répertoire de travail au lancement de la session de travail, tel qu'expliqué à la section précédente.
2. Ne jamais utiliser de chemins d'accès absolus ([section 1.5.3](#)). Vous ne pouvez compter sur une organisation des fichiers identique à la vôtre sur les postes de travail des autres. Utilisez exclusivement des chemins d'accès relatifs.

3. Ne jamais utiliser un chemin d'accès vers l'extérieur du projet, donc du répertoire. Les seuls chemins d'accès (relatifs) permis doivent mener vers des sous-répertoires du répertoire courant.
4. Ne jamais dépendre d'objets qui ne sont pas créés dans un script du projet. Vous ne pouvez pas supposer qu'un objet `x` contenant la valeur 5 existe dans la session R d'une collaboratrice.
5. Toujours utiliser un codage de caractères universel tel que **ASCII** ou **UTF-8**, surtout si vos fichiers de script comportent des signes diacritiques (accents, cédille, tréma) ou des symboles spéciaux.



L'UTF-8 n'étant toujours pas le codage par défaut sous Windows, vous devez porter une attention particulière à ce point si vous utilisez ce système d'exploitation.

3.9 Anatomie d'une session de travail

Dans ses grandes lignes, une session de travail avec R comporte 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. Ouvrir un fichier de script existant ou en créer un nouveau.
2. Changer le répertoire de travail de R pour le répertoire contenant le fichier de script.
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. Cette phase de développement comporte généralement de nombreux allers-retours entre la ligne de commande et le fichier de script.
4. Sauvegarder le fichier de script.
5. Si nécessaire — et c'est rarement le cas — sauvegarder l'espace de travail de la session avec la commande `save.image()`.

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

En fait, on ne sauvegarde l'espace de travail que lorsque les objets sont très longs à créer, comme les résultats d'une simulation.

6. Quitter R sans sauvegarder l'espace de travail en fermant l'interface graphique par la procédure usuelle, ou encore avec la commande `q`.

```
| > q("no")
```

L'ordre exact des étapes peut varier. Consultez l'[annexe A](#) pour la procédure détaillée avec l'environnement intégré RStudio, ou l'[annexe B](#) pour celle avec Emacs.

3.10 Obtenir de l'aide

Il est normal, durant l'apprentissage d'un nouveau langage de programmation, d'éprouver par moments un certain vertige devant la masse d'informations à intégrer. Lorsque cela survient, il faut savoir vers où se tourner pour obtenir de l'aide ou de la documentation.

Votre première ressource devrait être non pas Internet, mais bien l'aide en ligne de R. Les rubriques d'aide des fonctions contiennent une foule d'informations ainsi que des exemples d'utilisation. Leur consultation est tout à fait essentielle.

Les fonctions `?` et `help` permettent de consulter la rubrique d'aide d'une fonction ou d'un autre type d'objet.

```
| > ?<fonction>
| > help(<fonction>)
```

La fonction `help.search` permet quant à elle de rechercher de l'aide sur un sujet.

```
| > help.search("<sujet>")
```

Le système d'aide de R contient également bon nombre de documents plus longs en format PDF. Les « vignettes » abordent habituellement une thématique dans son ensemble plutôt que simplement par le biais d'une suite de fonctions.

```
| > vignette("<sujet>")
```

Les livres sur R emplissent aujourd'hui de pleins rayons de bibliothèques. Les ouvrages de [Venables et Ripley \(2000, 2002\)](#) demeurent des références standards *de facto* sur les langages S et R. Plus récents, [Braun et Murdoch \(2007, 2016\)](#) participent du même effort que le présent ouvrage en se concentrant sur la programmation en R plutôt que sur ses applications statistiques.



C'est le temps de vous lancer dans une première session de travail avec R. Le fichier de script `presentation.R` livré avec le présent document et reproduit à la [section 3.11](#) contient du code qui passe en revue quelques-unes des fonctionnalités de R. Ouvrez ce fichier dans votre éditeur de texte, puis évaluez les lignes une par une en prenant bien soin de lire les commentaires et d'examiner les résultats.

3.11 Exemples

📍 Fichier d'accompagnement presentation.R

```

11 ###
12 ### COMMANDES R
13 ###
14
15 ## Les expressions entrées à la ligne de commande sont immédiatement
16 ## évaluées et le résultat est affiché à l'écran, comme avec une
17 ## grosse calculatrice.
18 1 # une constante
19 (2 + 3 * 5)^2/7 # expression mathématique
20 exp(3) # fonction exponentielle
21 sin(pi/2) + cos(pi/2) # fonctions trigonométriques
22 gamma(5) # fonction gamma
23
24 ## Lorsqu'une expression est syntaxiquement incomplète, l'invite de
25 ## commande change de '>' à '+ '.
26 2 - # expression incomplète
27 5 * # toujours incomplète
28 3 # complétée
29
30 ## Taper le nom d'un objet affiche son contenu. Pour une fonction,
31 ## c'est son code source qui est affiché.
32 pi # constante numérique intégrée
33 letters # chaîne de caractères intégrée
34 LETTERS # version en majuscules
35 rnorm # fonction
36
37 ###
38 ### EXEMPLE DE SESSION DE TRAVAIL
39 ###
40 ### (Inspiré de l'annexe A du manuel «An Introduction to R» livré avec
41 ### R.)
42 ###
43
44 ## Afficher le répertoire de travail de R.
45 getwd()
46
47 ## Générer deux vecteurs de nombres pseudo-aléatoires issus d'une loi
48 ## normale centrée réduite.
49 x <- rnorm(50)
50 y <- rnorm(x)
51
52 ## Graphique des couples (x, y).
53 plot(x, y)
54
55 ## Graphique d'une approximation de la densité du vecteur x.
```



```
56 plot(density(x))
57
58 ## Générer la suite 1, 2, ..., 10.
59 1:10
60
61 ## La fonction 'seq' peut générer des suites plus générales.
62 seq(from = -5, to = 10, by = 3)
63 seq(from = -5, length = 10)
64
65 ## La fonction 'rep' répète des valeurs.
66 rep(1, 5)           # répéter 1 cinq fois
67 rep(1:5, 5)         # répéter le vecteur 1,...,5 cinq fois
68 rep(1:5, each = 5) # répéter chaque élément du vecteur cinq fois
69
70 ## Arithmétique vectorielle.
71 v <- 1:12           # initialisation d'un vecteur
72 v + 2               # additionner 2 à chaque élément de v
73 v * -12:-1          # produit élément par élément
74 v + 1:3             # le vecteur le plus court est recyclé
75
76 ## Vecteur de nombres uniformes sur l'intervalle [1, 10].
77 v <- runif(12, min = 1, max = 10)
78 v
79
80 ## Pour afficher le résultat d'une affectation, placer la commande
81 ## entre parenthèses.
82 ( v <- runif(12, min = 1, max = 10) )
83
84 ## Arrondi des valeurs de v à l'entier près.
85 round(v)
86
87 ## Créer une matrice 3 x 4 à partir d'un vecteur de 12 éléments.
88 ## Remarquer que la matrice est remplie par colonne.
89 m <- matrix(c(1, -1, 3, 1, -2, -7, 2, 3, 4, 8, 1, 10),
90             nrow = 3, ncol = 4)
91 m
92
93 ## Les opérateurs arithmétiques de base s'appliquent aux matrices
94 ## comme aux vecteurs.
95 m + 2
96 m * 3
97 m^2
98
99 ## Éliminer la quatrième colonne afin d'obtenir une matrice carrée.
100 ( m <- m[, -4] )
101
102 ## Transposée et inverse de la matrice m.
103 t(m)
```

```
104 solve(m)
105
106 ## Produit matriciel.
107 m %% m           # produit de m avec elle-même
108 m %% solve(m)    # produit de m avec son inverse
109 round(m %% solve(m)) # l'arrondi donne la matrice identité
110
111 ## Consulter la rubrique d'aide de la fonction 'solve'.
112 ?solve
113
114 ## Liste des objets dans l'espace de travail.
115 ls()
116
117 ## Nettoyage.
118 rm(x, y, v, m)
```

3.12 Exercices

3.1 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()
```

3.2 Consulter les rubriques d'aide d'une ou plusieurs des fonctions rencontrées dans le code informatique de la [section 3.11](#). Observer d'abord comment les rubriques d'aide sont toutes structurées à l'identique, puis évaluez quelques expressions tirées des sections d'exemples.

3.3 Identifier le répertoire de travail de votre session de travail R, puis localiser ce répertoire dans le système de fichier de votre poste de travail. Changer ensuite le répertoire de travail de R pour celui contenant le fichier `presentation.R` reproduit à la [section 3.11](#).

3.4 Évaluer le code de l'exemple de session de travail R que l'on trouve à l'annexe A de [Venables et collab. \(2024\)](#). En plus d'aider à se familiariser avec R, cet exercice permet de découvrir les fonctionnalités du logiciel en tant qu'outil statistique.

-
- 3.5** Répéter l'exercice d'évaluation du code du fichier `presentation.R` avec un ou deux autres éditeurs de texte afin de les comparer et de vous permettre d'en choisir un pour la suite.
- 3.6** Si vous utilisez l'environnement de développement intégré RStudio, configurer l'éditeur tel que recommandé à la [section A.7](#).

4 Bases de la programmation

Objectifs du chapitre

- ▶ Écrire et interpréter des expressions R.
- ▶ Utiliser les vecteurs et l'arithmétique vectorielle du langage R dans les calculs.
- ▶ Extraire des données d'un vecteur ou y affecter de nouvelles valeurs à l'aide des méthodes d'indiciage.
- ▶ Définir et appeler une fonction R.

C'est dans ce chapitre que nous débutons réellement l'apprentissage de la programmation. Je l'avoue d'entrée de jeu, la présentation est fortement influencée par l'ouvrage magistral de [Abelson et collab. \(1996\)](#).

Les humains créent des programmes informatiques pour contrôler, à l'aide d'un ensemble de règles, les processus de calcul et de manipulation de données d'un ordinateur. Ces programmes sont rédigés dans un langage de programmation.

Un langage est toutefois plus qu'une simple manière de transmettre des instructions à un ordinateur, c'est aussi une façon de conceptualiser les procédures que l'ordinateur devra effectuer. Autrement dit, le type de langage de programmation que nous utilisons influence directement la solution que nous proposerons à un problème — et vice versa. Comme nous l'avons déjà fait à la [section 1.2.2](#), dressons un parallèle avec les langues parlées et écrites : une langue ne constitue pas seulement un moyen de transmettre une idée, mais bien une façon de concevoir le monde. Ce n'est pas pour rien qu'il est parfois impossible de faire passer une idée d'une langue vers une autre — ce que l'on appelle couramment une « expression intraduisible ».

Le langage de programmation étudié ici est R. Sa syntaxe, celle du langage S, s'apparente au C. En revanche, la sémantique de R s'inspire du paradigme de la programmation fonctionnelle, ce qui lui confère plus d'affinités avec le Lisp et l'APL.

4.1 Données et procédures fondamentales

À sa plus simple expression, la programmation est un exercice de manipulation de données à l'aide de procédures. Les données représentent ce que nous voulons manipuler et les procédures, les descriptions des règles pour manipuler les données.

Tout langage de programmation moindrement puissant fournit au programmeur des expressions primitives ou génériques (les unités de traitement les plus simples du langage), des manières de les combiner pour former des éléments composés, ainsi qu'un mécanisme d'abstraction permettant de nommer et de manipuler ces éléments composés.

Les données fondamentales de R sont les suivantes :

- ▶ nombres réels : 0, 1, 2, 78.42, -1.39, ...;
- ▶ chaînes de caractères : "a", "foo", "Foobar", ...;
- ▶ valeurs booléennes : TRUE, FALSE;
- ▶ donnée manquante : NA;
- ▶ infini positif et négatif : Inf, -Inf;
- ▶ valeur indéterminée : NaN;
- ▶ « néant » : NULL;
- ▶ nombres complexes : $1 + 2i$.

La grande majorité des langages de programmation offrent les deux premiers types de données ci-dessus. Les autres types se révèlent très utiles pour la programmation mathématique et l'analyse de données. Nous reviendrons sur leurs caractéristiques à la [section 5.1](#).

Nous pouvons classer les procédures fondamentales — ou *opérateurs* — en quatre grandes catégories :

- ▶ arithmétique : + - * / ^ < >= ==, etc.;
- ▶ logique : & | !;
- ▶ indilage : [] \$;
- ▶ affectation : <-.

Les opérateurs arithmétiques sont applicables aux nombres réels ou complexes, alors que les opérateurs logiques ne sont applicables qu'aux valeurs booléennes. Nous traitons de l'opérateur d'affectation en détail à la section suivante et de l'indilage à la [section 4.3](#).

Le [tableau 4.1](#) présente les opérateurs les plus fréquemment employés en ordre décroissant de priorité des opérations. Ils sont accompagnés d'une description succincte.

TAB. 4.1 – Principaux opérateurs du langage R, en ordre décroissant de priorité

Opérateur	Fonction
\$	extraction d'une liste
[[]	indixage
^	puissance
-	changement de signe
:	génération de suites
%% %/% %o%	produit matriciel, modulo, division entière, produit extérieur
* /	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
-> ->>	affectation
<- <<-	affectation



Les opérateurs de puissance « ^ » et d'affectation « <- » sont évalués de droite à gauche; tous les autres le sont 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 .



Le fichier de script `bases.R` reproduit à la [section 4.8](#) fournit des détails additionnels sur les données fondamentales de R ainsi que des exemples d'utilisation de la plupart des opérateurs ci-dessus. Étudiez attentivement les lignes [12-139](#).

4.2 Commandes R

Nous l'avons déjà vu au [chapitre 3](#), l'interaction avec l'interpréteur R se fait par l'intermédiaire de commandes entrées à la ligne de commande. Or, toute commande R est soit une *expression*, soit une *affectation*.

4.2.1 Expression

Une expression R est une combinaison de symboles (noms de variables) et de procédures. Toute expression a une valeur. Le symbole d'une donnée fondamen-

talement représente cette donnée, comme on pourrait s'y attendre.

Lorsqu'une expression est entrée à la ligne de commande de l'interpréteur, elle est immédiatement évaluée et le résultat est affiché sous l'invite de commande `>` (le symbole `>` suivi d'une espace).

```
> 42
[1] 42
> 3 + 2i
[1] 3+2i
> 2 + 3
[1] 5
> pi
[1] 3.141593
> cos(pi/4)
[1] 0.7071068
```

Lorsqu'une commande n'est pas syntaxiquement complète, l'invite de commande se change en `+>` pour nous inciter à compléter la commande.

```
> 2 *
+ 3
[1] 6
```

Il est possible de combiner plusieurs expressions ensemble pour en faire une expression composée. Celle-ci est évaluée de gauche à droite, à moins que des parenthèses ne viennent changer l'ordre d'évaluation, comme en mathématiques.

```
> (2 + ((2 + (4 * 6)) * (3 + 5)))/2
[1] 105
```

Examinons un peu plus en détail comment sont évaluées les expressions composées. L'exercice permet de faire appel aux notions d'algorithmique du [chapitre 2](#). Pour évaluer une expression composée, l'interpréteur R doit suivre la procédure suivante :

1. Évaluer les sous-expressions de l'expression composée.
2. Appliquer de gauche à droite sur la sous-expression l'opération qui prend les autres sous-expressions en opérandes.

Vous aurez reconnu ici une procédure récursive puisque la première étape consiste à appliquer la procédure elle-même sur les sous-expressions. Reprenons l'expression composée ci-dessus. Nous pouvons représenter graphiquement la procédure d'évaluation à l'aide d'un arbre, comme à la [figure 4.1](#). Chaque combinaison y est représentée par un nœud, alors que les branches qui partent d'un nœud correspondent à l'opérateur et aux opérandes de la combinaison. Les nœuds ter-

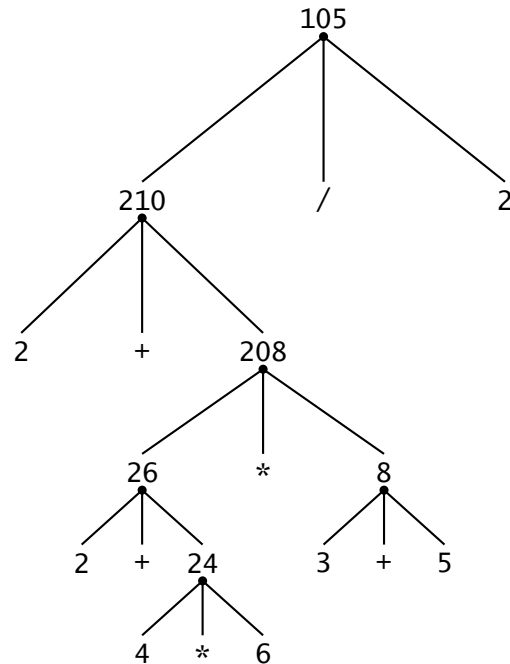


FIG. 4.1 – Représentation en arbre de l'évaluation de l'expression composée $(2 + ((2 + (4 * 6)) * (3 + 5))) / 2$

minaux — desquels ne partent aucune branche — représentent des nombres ou des opérateurs. En examinant l'arbre de la [figure 4.1](#), vous pouvez aisément imaginer que les opérations percolent du bas vers le haut.



Une manière alternative de concevoir l'ordre d'évaluation d'une expression composée est : de l'intérieur vers l'extérieur. En général, les opérations les plus basses dans l'arbre d'évaluation sont également situées vers le centre de l'expression.

4.2.2 Affectation

Dans une affectation, une expression est évaluée, mais le résultat est stocké dans un *objet* (ou *variable*) dans l'espace de travail et rien n'est affiché à l'écran. Tel que mentionné précédemment, l'opérateur d'affectation est « <- », c'est-à-dire les deux caractères « < » et « - » placés obligatoirement l'un à la suite de l'autre, et l'opération d'affectation est évaluée de droite à gauche. Ainsi, toute expression est évaluée avant que son résultat ne soit placé dans l'objet.

```
> a <- 5
> a
[1] 5
> b <- a
> b
[1] 5
> circ <- 2 * pi * 10
> circ
[1] 62.83185
```

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 expression¹.

```
> (circ <- 2 * pi * 10)
[1] 62.83185
```

L'opérateur d'affectation inversé « `->` » existe aussi, mais il est rarement utilisé.



Évitez d'utiliser l'opérateur `=` pour affecter une valeur à une variable. Cette pratique est susceptible d'engendrer de la confusion avec les constructions `symbole = valeur` dans les appels de fonction. Les règles de syntaxe de R commandent d'utiliser l'opérateur `<-` pour l'affectation, point.

J'ai mentionné au début de la [section 4.1](#) qu'un langage de programmation doit fournir un mécanisme d'abstraction pour utiliser les éléments composés. Une première technique d'abstraction de R — la plus simple en fait — est cette possibilité de définir un objet qui contient le résultat d'une expression. Par exemple, une fois que nous avons calculé la circonférence d'un cercle et stocké sa valeur dans un objet `circ`, comme ci-dessus, nous pouvons en quelque sorte « oublier » comment nous avons obtenu cette valeur pour nous concentrer sur l'utilisation de celle-ci à l'intérieur d'une autre procédure, et ainsi de suite jusqu'à obtenir un programme très complexe. Par sa nature interactive, R encourage un tel développement pas à pas des programmes. Cette approche fait aussi en sorte qu'un programme (ou script) R contient souvent plusieurs objets et procédures.

L'interpréteur R doit à tout moment associer correctement les valeurs et les noms d'objets. Il y arrive par le biais d'un dictionnaire et d'un environnement. La mécanique d'appariement demeure généralement transparente, mais au fur et à

1. En fait, l'opérateur d'affectation « `<-` » retourne un résultat — la valeur affectée —, mais de manière invisible. Une affectation placée entre parenthèses devient un appel à l'opérateur « `()` » qui, lui, ne fait que retourner son argument.

mesure que votre apprentissage du langage progressera, le rôle de l'environnement deviendra plus important. Nous étudierons le concept au [chapitre 12](#).

4.2.3 Regroupement de commandes

Dans les fichiers de script ou à la ligne de commande, on sépare généralement les commandes R les unes des autres par un retour à la ligne. Il est également possible de séparer les commandes par un point-virgule. Employer les deux — placer des points-virgules à la fin de chaque ligne de code — est considéré comme du mauvais style, surtout dans les fichiers de script. Le point-virgule peut être utile pour séparer deux courtes expressions ou plus sur une même ligne. C'est le seul emploi que je fais du point-virgule.

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

Pour regrouper plusieurs commandes et en former une seule expression, il faut entourer les commandes d'accolades « { } ». Le résultat du regroupement est la valeur de la *dernière* commande. Par conséquent, si le regroupement se termine par une affectation, aucune valeur n'est retournée ni affichée à l'écran.

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

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

Les accolades joueront un rôle très important dans la construction des fonctions et des structures de contrôle. Je vous renverrai ici en temps et lieu pour vous rappeler les règles de leur utilisation.



Étudiez les lignes [142-192](#) du fichier de script `bases.R` reproduit à la [section 4.8](#).

4.3 Vecteurs et arithmétique vectorielle

Nous interrompons notre programmation régulière pour vous présenter une émission spéciale sur un type de donnée — ou d'objet — qui joue un rôle fondamental dans R : le vecteur. Cet objet constitue la véritable unité de traitement de

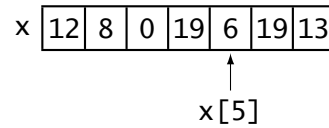


FIG. 4.2 – Représentation d'un vecteur et de l'opération d'indilage

base dans le langage R. De plus, le langage contient des règles d'arithmétique spécifiques pour le vecteur qui permettent de prendre automatiquement en charge un vaste éventail de calculs répétitifs.

Si vous avez étudié l'algèbre linéaire, vous connaissez déjà le concept mathématique de vecteur : un n -uplet de valeurs, souvent des coordonnées d'un plan cartésien. Dans R, la définition du vecteur est encore plus large. En fait, *tout* est un vecteur dans R et il s'agit simplement d'une collection de données contigües (numériques ou non) auxquelles il est possible d'accéder directement par une opération d'indilage. La [figure 4.2](#) représente schématiquement un vecteur simple dans R.

Nous allons nous restreindre pour le moment aux vecteurs simples (atomiques) dans lesquels tous les éléments sont du même type ou, plus précisément, du même *mode*². Les vecteurs plus généraux feront leur entrée au [chapitre 5](#).

4.3.1 Création de vecteurs

La fonction de base pour créer un vecteur est la fonction de concaténation `c`.

```
> (x <- c(2, 5.1, 42))
[1] 2.0 5.1 42.0
```

Les fonctions `numeric`, `logical` et `character` permettent également de créer des vecteurs de données numériques, booléennes et alphanumériques, respectivement. Les fonctions prennent en argument la longueur du vecteur à créer. Celui-ci contiendra des valeurs initiales prédéterminées.

```
> numeric(5)
[1] 0 0 0 0 0
> logical(7)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> character(10)
[1] "" "" "" "" "" "" "" "" "" ""
```

Il est possible — et souvent souhaitable — de nommer les éléments d'un vecteur; on dit qu'on leur attribue une étiquette. Le résultat est un vecteur étiqueté,

2. Nous reviendrons sur le concept de mode à la [section 5.1.2](#).

ou vecteur nommé. Il y a deux grandes façons de procéder : en spécifiant les noms dès la création du vecteur, ou à posteriori en ajoutant l'attribut au vecteur à l'aide de la fonction `names`. Celle-ci permet aussi d'extraire les étiquettes du vecteur.

```
> (x <- c(a = 1, b = 2, c = 5))
a b c
1 2 5
> names(x)
[1] "a" "b" "c"
```

```
> (x <- c(1, 2, 5))
[1] 1 2 5
> names(x) <- c("a", "b", "c")
> x
a b c
1 2 5
```

Si vous tentez de créer un vecteur simple avec des données de modes différents, R effectuera une conversion forcée (*coercion*) vers un mode unique. Celui-ci est choisi pour minimiser la perte d'information sur les objets originaux.

```
> c(2, TRUE, FALSE)
[1] 2 1 0
> c(2, FALSE, "a")
[1] "2"      "FALSE" "a"
```

Vous pouvez aussi convertir un vecteur d'un mode vers un autre avec les fonctions `as.numeric`, `as.logical` et `as.character`.

```
> as.logical(1)
[1] TRUE
> as.numeric(FALSE)
[1] 0
> as.character(42)
[1] "42"
```

De leur côté, les fonctions `is.numeric`, `is.logical` et `is.character` permettent de valider si un vecteur est d'un certain mode.

```
> is.numeric(c(42, NA, 3))
[1] TRUE
> is.logical(c(42, NA, 3))
[1] FALSE
> is.character(c("42", NA, "3"))
```

```
[1] TRUE
```

4.3.2 Arithmétique vectorielle

L'arithmétique vectorielle de R constitue l'une des grandes forces du langage. Elle permet de réaliser une grande variété de calculs sans avoir recours à des procédures itératives ou récursives. En soi, il s'agit d'une couche d'abstraction directement intégrée au langage. Pour prétendre savoir programmer en R, vous devez absolument tirer profit de l'arithmétique vectorielle.

Les procédures fondamentales de la [section 4.1](#) peuvent toutes opérer sur les vecteurs en effectuant les opérations *élément par élément*. C'est la première règle de base de l'arithmétique vectorielle dans R.

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

La seconde règle de base se rapporte aux opérations entre des vecteurs de longueurs différentes. Dans de tels cas, les vecteurs les plus courts sont *recyclés* autant de fois que nécessaire pour 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
```

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)
[1] 4 8 8 12 12 11 11 15 15 19
```

Autrement, le plus court vecteur est recyclé un nombre fractionnaire de fois, mais comme ce résultat est rarement celui souhaité et qu'il 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
```



La règle de recyclage des vecteurs fait en sorte qu'il y a très peu d'erreurs de longueur dans R. Qu'une expression soit valide ne signifie donc pas qu'elle effectue le bon calcul !

Il tombe sous le sens que les opérateurs arithmétiques sont conçus pour des arguments numériques et les opérateurs logiques, pour des arguments booléens. Si un argument n'est pas du bon mode, R effectuera la conversion forcée vers le mode approprié. En particulier, les valeurs booléennes TRUE et FALSE se verront converties en 1 et 0, respectivement, dans les opérations arithmétiques. À l'inverse, dans les opérations logiques, 0 est converti en FALSE et *tout* autre nombre est converti en TRUE.

```
> 2 + c(TRUE, FALSE)
[1] 3 2
> c(0, 5, -1) & TRUE
[1] FALSE TRUE TRUE
```

4.3.3 Indixage

L'indixage des vecteurs est une procédure beaucoup utilisée dans R, aussi est-il important d'en maîtriser toutes les subtilités. L'opération sert principalement à deux choses : extraire des éléments d'un objet avec la construction `x[i]`, ou remplacer des éléments avec la construction `x[i] <- y`.

Dans un cas comme dans l'autre, il faut d'abord indiquer le vecteur. Il existe cinq façons de le faire, toujours à l'intérieur de crochets « `[]` ».

1. **Extraction par position** 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[2]
B
4
> x[c(1, 3)]
A C
2 -1
```

2. **Suppression par position** avec un vecteur d'entiers négatifs. Les éléments se trouvant aux positions correspondant aux entiers négatifs sont *éliminés* du vecteur.

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

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

3. **Extraction par critère** avec un vecteur booléen. Le vecteur d'indilage 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. **Extraction par étiquette** avec un vecteur de chaînes de caractères. Les éléments dont l'étiquette correspond à l'une des chaînes sont extraits du vecteur. Cette méthode d'indilage a comme principal avantage de permettre l'extraction d'éléments d'un vecteur indépendamment de leur position dans celui-ci.

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

5. **Sélection de tous les éléments** en laissant l'indice vide.

```
> 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; nous y reviendrons au [chapitre 5](#). Laisser l'indice vide est différent d'indicer avec un vecteur vide. Cette dernière opération retourne un vecteur vide.



Il n'est pas inutile de savoir que les opérations d'extraction et de remplacement sont en fait traduites par l'interpréteur R en des appels à des fonctions nommées « `[]` » et « `[<-]` », dans l'ordre.



La présente section est très importante pour la suite. Étudiez donc en y portant une attention toute particulière les lignes 195-330 du fichier de script `bases.R` reproduit à la [section 4.8](#).

4.4 Fonctions

À ce stade, nous savons comment utiliser les opérateurs de base du [tableau 4.1](#) ainsi que l'arithmétique vectorielle. Le langage R serait toutefois très restreint s'il ne nous donnait pas aussi accès à des fonctions³ plus élaborées. Or, le langage R compte justement un très grand nombre (des milliers!) de fonctions internes. Vous en avez déjà rencontré quelques-unes dans les exemples, comme `exp` ou `cos`, et vous en découvrirez de nouvelles à la [section 4.7](#). Chose plus importante, R nous permet aussi de définir nos propres fonctions. C'est là l'une des techniques d'abstraction les plus puissantes du langage puisqu'elle permet de faire référence par un seul nom à toute une suite d'opérations.

Cette section explique comment définir vos propres fonctions et retourner des résultats, puis comment faire appel à vos fonctions ou aux fonctions internes.

4.4.1 Programmation fonctionnelle

Nous adoptons en bonne partie avec le langage R le paradigme de la programmation fonctionnelle. Tel qu'expliqué sommairement à la [section 1.2.4](#), dans ce paradigme un programme est constitué d'une suite d'appels de fonctions.

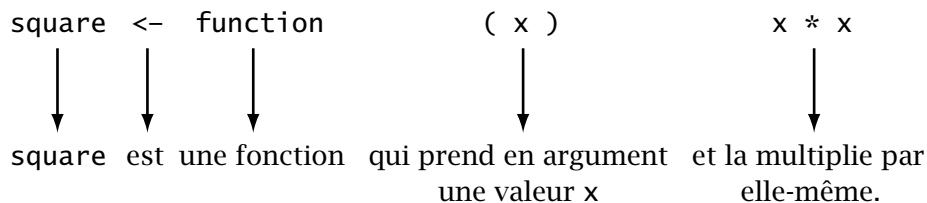
L'approche de programmation fonctionnelle dans R fait aussi en sorte qu'une fonction est traitée comme n'importe quel autre objet. Entre autres choses, cela signifie que :

- ▶ le contenu d'une fonction (son code source) est toujours accessible;
- ▶ une fonction peut accepter en argument une autre fonction;
- ▶ une fonction peut retourner une fonction comme résultat;
- ▶ l'utilisateur peut définir de nouvelles fonctions.

Étudions un exemple très simple, pas véritablement utile en pratique au-delà de ses vertus pédagogiques. Nous voulons élever un nombre au carré. Pour faire ce calcul à répétition et de manière abstraite — c'est-à-dire, sans avoir à se demander à chaque fois comment le calcul est fait —, nous définissons une nouvelle fonction `square`.

```
> square <- function(x) x * x
```

L'expression ci-dessus peut se lire ainsi :



3. Le nom que nous donnons dorénavant au concept de procédure.

Nous pouvons ensuite utiliser la fonction `square` de manière intuitive comme n'importe quelle autre fonction dans R (nous verrons les règles d'appel en détail à la [section 4.4.4](#)).

```
> square(5)
[1] 25
```

L'approche de programmation fonctionnelle fait en sorte que les appels de fonctions peuvent être placés les uns à la suite des autres, comme la composition de fonctions $g \circ f$ en mathématiques. Pour élever un nombre à la puissance 4, nous pouvons appliquer deux fois la fonction `square` à un argument.

```
> square(square(5))
[1] 625
```

4.4.2 Définition d'une fonction

Comme nous l'avons vu dans l'exemple ci-dessus, on définit une nouvelle fonction dans R à l'aide de la fonction `function`. La syntaxe exacte de la définition d'une fonction est la suivante :

```
<nom> <- function(<arguments>) <corps>
```

- *<nom>* est le nom de la fonction (les règles pour les noms de fonctions sont les mêmes que celles présentées à la [section 5.1.1](#) pour tout autre objet).

Nous souhaitons généralement attribuer un nom à une fonction, ne serait-ce que pour pouvoir la réutiliser. Cela dit, c'est tout à fait optionnel. En effet, l'appel à `function` retourne déjà une fonction et si aucun nom n'est attribué à cet objet, on obtient ce qui est appelé une *fonction anonyme*. Les fonctions anonymes sont particulièrement utiles avec les fonctions d'application ([chapitre 5](#)).

- *<arguments>* est la liste des arguments formels de la fonction, séparés par des virgules.

Une fonction peut n'avoir aucun argument formel ou plusieurs. Il n'y a pas de limite pratique au nombre d'arguments formels que peut avoir une fonction. Les arguments peuvent être des symboles, des constructions de la forme `symbole = défaut`, ou encore l'argument formel spécial « `...` ».

Lorsqu'un symbole seul est utilisé comme argument, il indique le nom de celui-ci. L'objet de ce nom sera disponible à l'intérieur de la fonction.

```
> f <- function(x) 2 * x + 1
> f(2)
[1] 5
```

La construction `symbole = défaut` indique la valeur par défaut de l'argument `symbole`, c'est-à-dire la valeur de l'argument si aucune n'est spécifiée dans l'appel de la fonction.

```
> f <- function(x = 0) 2 * x + 1
> f(2)
[1] 5
> f()
[1] 1
```

Enfin, l'argument spécial « ... » (point-point-point) peut contenir un nombre arbitraire d'éléments. La fonction `y` ayant recours se trouve donc à accepter un nombre variable d'arguments. On utilise généralement « ... » pour passer des arguments tels quels à une autre fonction qui, elle, saura quoi faire de ceux-ci. Vous trouverez un meilleur exemple à la [section 4.8](#) que ce que nous pourrions faire ici en quelques lignes.

- `<corps>` est une expression ou un groupe d'expressions réunies par des accolades, tel qu'expliqué à la [section 4.2.3](#).

4.4.3 Retourner des résultats

La plupart des fonctions sont écrites dans le but de retourner un résultat. Or, les règles d'interprétation d'un regroupement d'expressions présentées à la [section 4.2.3](#) s'appliquent ici au corps de la fonction : une fonction retourne tout simplement le résultat de sa *dernière expression*.

Sous peine de causer bien des interrogations, évitez que la dernière expression d'une fonction soit une affectation. En effet, pour les raisons mentionnées à la [section 4.2.2](#), la fonction ne retournerait alors aucun résultat visible susceptible d'être affiché à l'écran.

Lorsque le résultat de l'évaluation d'une fonction n'est pas affecté à une variable, R affiche par défaut le résultat à l'écran.



N'utilisez *jamais* la fonction `print` à l'intérieur d'une fonction pour afficher ses résultats à l'écran. Cette fonction sert principalement pour le débogage ([section 8.1](#)) ou pour des méthodes spécialisées d'affichage des résultats — un sujet qui n'est pas abordé dans le présent ouvrage.

Les fonctions suivantes permettent de retourner un résultat, un avertissement ou un message d'erreur avant même la fin d'une fonction. Vous les utiliserez à peu près exclusivement à l'intérieur d'expressions conditionnelles ([section 4.5](#)).

return Force la sortie immédiate de la fonction et retourne l'objet en argument.

- stop** Force la sortie immédiate de la fonction avec le message d'erreur donné en argument.
- warning** Envoie le message d'avertissement donné en argument à R. L'exécution de la fonction se poursuit normalement. L'effet dépend de la valeur de `options("warn")`. Par défaut, tous les avertissements sont affichés à la console après l'exécution de la fonction.



L'utilisation de `return` à la toute fin d'une fonction est tout à fait inutile et c'est considéré comme du mauvais style en R.

4.4.4 Appel d'une fonction

Je m'en suis remis à votre bonne intuition jusqu'ici lorsqu'il s'agissait d'appeler une fonction dans R. Précisons maintenant les règles d'appel, règles auxquelles obéissent autant les fonctions internes que vos propres fonctions.

Un appel de fonction est constitué du nom de l'objet suivi obligatoirement de parenthèses () et, le cas échéant, d'arguments entre ces parenthèses.

`<nom_fonction>(<arguments>)`

Regardons cela de plus près. À des fins d'illustration, je définis une petite fonction `dist` qui calcule la quantité

$$d = \sum_{i=1}^n |x_i - y_i|^p$$

pour deux vecteurs $\mathbf{x} = (x_1, \dots, x_n)$ et $\mathbf{y} = (y_1, \dots, y_n)$ et une valeur p .

```
> dist <- function(x, y = 0, p = 2) sum(abs(x - y)^p)
```

Vous remarquerez que la fonction fait elle-même appel à deux fonctions internes (`sum` et `abs`) ainsi qu'à deux opérateurs de base (« - » et « ^ »). Elle repose également sur l'arithmétique vectorielle pour calculer toutes les différences $x_i - y_i$.

La fonction compte trois arguments nommés `x`, `y` et `p`. La construction `symbole = défaut` indique la valeur par défaut de l'argument `symbole`, c'est-à-dire la valeur de l'argument si aucune n'est spécifiée dans l'appel de la fonction. Ainsi, seul le premier argument de `dist` n'a pas de valeur par défaut. La valeur par défaut de l'argument `y` est 0 et celle de l'argument `p` est 2.

L'ordre des arguments dans la définition de la fonction est important. Nous avons deux choix au moment d'appeler une fonction :

1. Spécifier les valeurs des arguments dans le bon ordre.
2. Nommer explicitement les arguments avec `symbole = expression`, auquel cas l'ordre n'importe plus.

Si certains arguments sont nommés et d'autre pas, ces derniers reçoivent des valeurs selon leur ordre dans la définition de la fonction, et non selon leur position dans l'appel.

Il est beaucoup plus prudent et *fortement recommandé* de spécifier les arguments par leur nom, surtout après les deux ou trois premiers arguments.

```
> dist(c(3, 2), c(1, 1), 2)
[1] 5
> dist(y = c(1, 1), c(3, 2), 2)
[1] 5
> dist(y = c(1, 1), p = 2, c(3, 2))
[1] 5
```

Il n'est pas nécessaire de spécifier une valeur pour un argument disposant d'une valeur par défaut. Cependant, si l'on omet un tel argument, il faudra nommer les arguments qui suivent, le cas échéant.

```
> dist(c(3, 2))
[1] 13
> dist(c(3, 2), p = 3)
[1] 35
```

Le manuel officiel *R Language Definition* (R Core Team, 2020, section 4.3.2) fournit les détails sur le mécanisme de pairage entre les arguments d'une fonction et les valeurs qui lui sont passées.



Étudiez les lignes 333–576 du fichier de script `bases.R` reproduit à la [section 4.8](#) pour des exemples additionnels de définitions de fonctions et, surtout, d'utilisation des fonctions anonymes et de l'argument « ... ».

4.5 Expressions conditionnelles

Le langage R permet d'écrire des expressions conditionnelles dont la syntaxe se rapproche beaucoup des constructions que nous avons étudiées à la [section 2.3](#). Les structures conditionnelles sont les premières structures de contrôle que nous étudierons. Les boucles itératives feront l'objet de la [section 7.7](#).

La structure `if ... else ...` permet d'exécuter une expression conditionnelle à un ou deux volets, selon que la clause alternative soit présente ou non. Les syntaxes des deux formulations sont les suivantes :

```
if (<condition>)
  <conséquence>
```

```
if (<condition>)
  <conséquence>
else
  <alternative>
```

- ▶ *<condition>* est une expression dont le résultat est une valeur TRUE ou FALSE unique. Une erreur fréquente consiste à construire un test de telle sorte que la *<condition>* est un vecteur, ce qui n'a guère de sens. Les fonctions `any`, `all` (section 4.7.5) se révèlent utiles dans les clauses `if` pour réduire les vecteurs booléens à une valeur unique. Les fonctions `isTRUE` et `isFALSE`, quant à elles, permettent de s'assurer que la condition est un vecteur booléen de longueur 1 différent de NA.
- ▶ *<conséquence>* est une expression, ou un groupe d'expressions regroupées entre accolades « { } », qui sont exécutées lorsque la *<condition>* est TRUE.
- ▶ *<alternative>* est une expression, ou un groupe d'expressions regroupées entre accolades « { } », qui sont exécutées lorsque la *<condition>* est FALSE.



Il tombe sous le sens que « vrai » est vrai, n'est-ce pas ? Prenez donc garde de ne pas écrire des expressions qui reviennent à tester `if (TRUE == TRUE)` ou `if (TRUE != FALSE)`. Oui, j'ai souvent rencontré de telles constructions !

Pour choisir entre plus de deux possibilités, vous pouvez simplement imbriquer les structures `if ... else ...` les unes dans les autres.

```
if (<condition 1>)
  <conséquence 1>
else if (<condition 2>)
  <conséquence 2>
else
  <alternative>
```

Les constructions de type « Selon que » se conçoivent en R avec la fonction `switch`. Sa syntaxe est la suivante :

```
switch(<expression>, <cas 1> = <action 1>, <cas 2> = <action 2>, ...)
```

La fonction évalue d'abord l'*<expression>*. Si le résultat est une valeur numérique *k*, alors l'*<action k>* est exécutée. Si le résultat de l'*<expression>* est l'une des chaînes de caractères *<cas 1>*, *<cas 2>*, ..., alors c'est l'action correspondante qui est exécutée.

```
> switch(2, 2 + 3, mean(1:10), 5:1)
[1] 5.5
```

```
> switch("foo", foo = 2 + 3, bar = mean(1:10))
[1] 5
```



Les fonctions `return`, `stop` et `warning` de la [section 4.4.3](#) se retrouvent habituellement dans la *conséquence* ou dans l'*alternative* d'une expression conditionnelle, en particulier dans des tests de validité des arguments d'une fonction.

Enfin, la fonction `ifelse` — une sorte de version vectorielle de la structure `if ... else ...` — permet de calculer des valeurs selon les résultats d'un test sur un vecteur. Sa syntaxe est la suivante :

```
ifelse(<test>, <oui>, <non>)
```

- ▶ *<test>* est un vecteur booléen.
- ▶ *<oui>* est un vecteur de la même longueur que *<test>* duquel sont extraites les valeurs correspondant à des éléments TRUE dans *<test>*.
- ▶ *<non>* est aussi un vecteur de la même longueur que *<test>*, mais duquel sont extraites les valeurs correspondant à des éléments FALSE dans *<test>*.

```
> x <- c(-2, 3, -1, 0, 5, 1)
> ifelse(x < 0, -x, x)
[1] 2 3 1 0 5 1
```

Une fois que l'on a compris son fonctionnement, la fonction `ifelse` peut apparaître comme un outil de choix dans notre arsenal de fonctions R, en particulier pour la programmation de fonctions mathématiques définies par branches. En réalité, c'est plus ou moins le cas, car `ifelse` est une fonction très lente. Dans la plupart des cas, il vaut mieux lui préférer des constructions moins élégantes, mais bien plus rapides. En particulier, la rubrique d'aide de la fonction souligne que la structure `if ... else ...` est beaucoup plus efficace si *<test>* est un vecteur de longueur 1.



Étudiez les lignes [579-784](#) du fichier de script `bases.R` reproduit à la [section 4.8](#).

4.6 Récursion

En tant que langage de programmation fonctionnel, R admet tout naturellement les procédures récursives. La fonction récursive s'invoque elle-même avec la fonction `Recall`, plutôt que par son nom réel. Cela rend l'appel récursif indépendant du nom de la fonction, ce qui peut s'avérer bien utile en cas de changement du nom de la fonction.

Par exemple, nous pouvons réécrire ainsi la mise en œuvre récursive de la fonction factorielle de la [figure 3.1](#), à la [page 40](#) :

```
factorial <- function(n)
  if (n == 1) 1 else n * Recall(n - 1)
```



Le temps système requis pour la création des environnements lors de chaque appel de fonction ([chapitre 12](#)) rend les fonctions récursives peu efficaces en R.

4.7 Fonctions internes utiles

Cette section présente quelques fonctions internes souvent utilisées pour programmer en R et pour manipuler des données. Elle reviendra à intervalles réguliers dans le document au fur et à mesure que nous découvrirons de nouvelles fonctionnalités du langage.

La liste de fonctions ci-dessous est évidemment loin d'être exhaustive. Un des meilleurs endroits pour découvrir des fonctions demeure la section *See Also* des rubriques d'aide. Celle-ci offre des hyperliens vers des fonctions apparentées au sujet de la rubrique.

Pour chaque fonction présentée dans cette section, je fournis un ou deux exemples d'utilisation. Ces exemples sont loin de couvrir toutes les utilisations possibles d'une fonction. La [section 4.8](#) fournit des exemples additionnels. Consultez les rubriques d'aide des fonctions pour connaître toutes leurs options.



Vraiment, consultez les rubriques d'aide.

4.7.1 Fonctions mathématiques et trigonométriques

exp fonction exponentielle e^x .

```
> exp(1)
[1] 2.718282
```

log logarithme naturel ou dans une base quelconque.

```
> log(exp(1))
[1] 1
> log(9, base = 3)
[1] 2
```

sqrt racine carrée.

	<pre>> sqrt(9) [1] 3</pre>
abs	valeur absolue. <pre>> abs(c(-1, 0, 2)) [1] 1 0 2</pre>
gamma	fonction gamma $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$ (et les fonctions apparentées <code>lgamma</code> , <code>digamma</code> , <code>trigamma</code>). <pre>> gamma(5) [1] 24</pre>
factorial	factorielle $x!$ (et la fonction apparentée <code>lfactorial</code>). <pre>> factorial(4) [1] 24</pre>
cos	cosinus en radians (et les autres fonctions trigonométriques : <code>sin</code> , <code>tan</code> , <code>acos</code> , etc.). <pre>> cos(pi/4) [1] 0.7071068</pre>

4.7.2 Suites et répétition

seq	suite de nombres générale. <pre>> seq(1, 9, by = 2) [1] 1 3 5 7 9</pre>
seq_len	suite de nombres de la longueur donnée en argument à partir de 1 (plus rapide que <code>seq</code>). <pre>> seq_len(10) [1] 1 2 3 4 5 6 7 8 9 10</pre>
seq_along	suite de nombres de la longueur du vecteur en argument à partir de 1 (plus rapide que <code>seq</code>). <pre>> seq_along(c(-1, 0, 2)) [1] 1 2 3</pre>
rep	répétition de vecteurs, de chaque élément d'un vecteur ou d'une combinaison des deux.

```
> rep(2, 10)
[1] 2 2 2 2 2 2 2 2 2 2
> rep(c(1, 3), each = 4)
[1] 1 1 1 1 3 3 3 3
> rep(c(1, 3), times = 2, each = 4)
[1] 1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3
```

`rep.int` répétition de vecteurs complets uniquement (plus rapide que `rep`).

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

`rep_len` répétition de vecteurs jusqu'à une certaine longueur (plus rapide que `rep`).

```
> rep_len(1:3, 10)
[1] 1 2 3 1 2 3 1 2 3 1
```



Les fonctions `seq_len` et `seq_along` sont plus robustes que `seq` ou que l'opérateur « : » pour la programmation. Lorsque leur argument est, dans l'ordre, 0 et un vecteur de longueur nulle, les fonctions retournent un vecteur de longueur nulle. En revanche, le résultat de `seq(0)` ou de `1:0` est le vecteur `c(1, 0)`.

4.7.3 Extraction du début et de la fin d'un objet

`head` avec second argument $n > 0$ ($n = 6$ par défaut) : n premières composantes d'un objet (éléments d'un vecteur, lignes d'une matrice ou d'un *data frame*);

```
> head(1:10, 3)
[1] 1 2 3
```

avec second argument $n < 0$: objet sans les $|n|$ dernières composantes.

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

`tail` avec second argument $n > 0$ ($n = 6$ par défaut) : n dernières composantes d'un objet (éléments d'un vecteur, lignes d'une matrice ou d'un *data frame*);

```
> tail(1:10, 3)
[1] 8 9 10
```

avec second argument $n < 0$: objet sans les $|n|$ premières composantes.

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

4.7.4 Arrondi

Les exemples de cette sous-section utilisent le vecteur suivant.

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

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

```
> 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
```

4.7.5 Tests logiques

Les exemples de cette sous-section utilisent toujours le vecteur suivant.

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

`any` vrai si au moins une valeur du vecteur booléen en argument est vraie.

```
> any(x < 0)
[1] TRUE
> any(x > 10)
[1] FALSE
```

`all` vrai si toutes les valeurs du vecteur booléen en argument sont vraies.

```
> all(x < 0)
[1] FALSE
> all(x > -5)
[1] TRUE
```

4.7.6 Sommaires et statistiques descriptives

Les exemples de cette sous-section utilisent le vecteur suivant.

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

`sum` somme des éléments.

```
> sum(x)
[1] 135
```

`prod` produit des éléments.

```
> prod(x)
[1] 24938020800
```

`diff` différences entre les éléments (opérateur mathématique ∇).

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

`mean` moyenne arithmétique.

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

`var` variance (et écart type avec `sd`).

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

`min` minimum.

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

`max` maximum.

```
> max(x)
[1] 25
```

`range` étendue (minimum et maximum).

```
> 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` principales statistiques descriptives.

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

4.7.7 Sommaires cumulatifs et comparaisons élément par élément

Les exemples de cette sous-section utilisent le vecteur suivant.

```
> x
[1] 14 17 7 9 3
```

`cumsum` somme cumulative.

```
> cumsum(x)
[1] 14 31 38 47 50
```

`cumprod` produit cumulatif.

	<pre>> cumprod(x) [1] 14 238 1666 14994 44982</pre>
<code>cummin</code>	<p>minimum cumulatif.</p> <pre>> cummin(x) [1] 14 14 7 7 3</pre>
<code>cummax</code>	<p>maximum cumulatif.</p> <pre>> cummax(x) [1] 14 17 17 17 17</pre>
<code>pmin</code>	<p>minimum élément par élément (en parallèle).</p> <pre>> pmin(x, 12) [1] 12 12 7 9 3</pre>
<code>pmax</code>	<p>maximum élément par élément (en parallèle).</p> <pre>> pmax(x, c(16, 23, 4, 12, 3)) [1] 16 23 7 12 3</pre>

4.7.8 Manipulation de chaînes de caractères

<code>paste</code>	<p>concaténation de vecteurs après les avoir convertis en chaînes de caractères, séparés par défaut par des espaces</p> <pre>> paste(c("Camille", "Marianne", "Alexandre"), + "porte le numéro", c(8, 7, 2)) [1] "Camille porte le numéro 8" [2] "Marianne porte le numéro 7" [3] "Alexandre porte le numéro 2"</pre>
<code>paste0</code>	<p>comme <code>paste</code>, mais sans séparateur entre les éléments</p> <pre>> paste0(1:5, c("er", rep("e", 4))) [1] "1er" "2e" "3e" "4e" "5e"</pre>
<code>nchar</code>	<p>nombre de caractères dans une chaîne de caractères</p> <pre>> nchar(c("foo", "foobar")) [1] 3 6</pre>
<code>substr</code>	<p>sous-chaîne (partie) d'une chaîne de caractères</p>

```
> substr("abcdef", 2, 4)
[1] "bcd"
```

`strsplit` séparation d'une chaîne de caractères en parties sur un caractère quelconque

```
> strsplit("Luc va à l'école", " ")
[[1]]
[1] "Luc"      "va"      "à"      "l'école"
```



Vous trouverez plusieurs exemples additionnels d'utilisation des fonctions présentées dans cette section dans le code des lignes 787-963 du fichier de script `bases.R`.

4.8 Exemples

📍 Fichier d'accompagnement `bases.R`

```
11 ###
12 ### DONNÉES ET PROCÉDURES FONDAMENTALES
13 ###
14
15 ## NOMBRES ET OPÉRATEURS ARITHMÉTIQUES
16
17 ## Tous les nombres réels sont stockés en double précision dans R,
18 ## entiers comme fractionnaires. R permet aussi de définir des nombres
19 ## en notation scientifique et des nombres complexes.
20 486                # nombre réel entier
21 0.3324             # nombre réel fractionnaire
22 2e-3               # notation scientifique
23 1 + 2i             # nombre complexe
24
25 ## La définition des opérateurs arithmétiques standards coule de
26 ## source.
27 137 + 349           # addition
28 1000 - 334          # soustraction
29 5 * 42              # multiplication
30 10/4                # division
31 2^3                 # puissance
32 -42                 # changement de signe
33
34 ## Les opérateurs de comparaison (égalité et inégalité) retournent une
35 ## valeur booléenne.
36 486 < 521           # plus petit
37 486 >= 521          # plus grand ou égal
38 486 != 521          # différent de
```

```
39
40 ## Attention à cette erreur commune --- et parfois difficile à
41 ## détecter: '=' n'est PAS l'opérateur de comparaison de l'égalité
42 ## entre deux valeurs.
43 5 = 2                                # erreur de syntaxe
44
45 ## L'opérateur de comparaison est plutôt '=='.
46 486 == 486                           # comparaison
47 y = 486                             # pas un test...
48 y                                    # ... plutôt une affectation
49
50 ## Les opérateurs '==' et '!=' vérifient l'égalité (ou non) bit pour
51 ## bit dans la représentation interne des nombres dans l'ordinateur.
52 ## Ça fonctionne bien pour les entiers ou les valeurs booléennes, mais
53 ## pas pour les nombres réels ou, plus insidieux, pour les nombres
54 ## entiers provenant d'un calcul et qui ne sont entiers qu'en
55 ## apparence.
56 ##
57 ## [Pour en savoir (un peu) plus:
58 ## https://floating-point-gui.de/formats/fp/]
59 1.2 + 1.4 + 2.8                       # 5.4 en apparence
60 1.2 + 1.4 + 2.8 == 5.4                # non?!?
61 0.3/0.1 == 3                          # à gauche: faux entier
62
63 ## L'opérateur ':' est très utile pour générer des suites de nombres
64 ## avec un pas fixe de 1 entre chaque nombre.
65 1:10                                  # entiers de 1 à 10
66 5:-2                                  # entiers de 5 à -2
67
68 ## L'opérateur modulo retourne le reste d'une division.
69 544 %% 119                            # 544/119 = 4 reste 68
70 119 %% 544                            # x %% y = x si x < y
71
72 ## La division entière est l'opération duale du modulo: elle retourne
73 ## la partie entière d'une division.
74 544 %/% 119                           # 544/119 = 4 reste 68
75 119 %/% 544                           # x %/% y = 0 si x < y
76
77 ## L'opérateur modulo est souvent utilisé pour déterminer si un nombre
78 ## est pair ou impair. Un nombre 'x' est pair si 'x mod 2 = 0' et il
79 ## est impair si 'x mod 2 = 1'.
80 554 %% 2                               # pair
81 119 %% 2                               # impair
82
83 ## CHAINES DE CARACTÈRES
84
85 ## On crée une chaîne de caractères en l'entourant de guillemets
86 ## doubles " ".
```



```

87 "a"                # chaîne de 1 caractère
88 "foobar"          # chaîne de 6 caractères
89 "486"             # chaîne de 3 caractères
90
91 ## Les opérateurs de comparaison sont également définis pour les
92 ## chaînes de caractères en fonction de l'ordre lexicographique.
93 ## Attention: c'est un terrain miné selon la langue utilisée;
94 ## consultez la rubrique d'aide de 'Comparaison' au besoin (qui vous
95 ## dira de ne pas faire d'hypothèses sur l'ordre lexicographique entre
96 ## deux chaînes de caractères). Quelques exemples simples.
97 "a" < "d"          # ordre alphabétique
98 "a" < "A"          # minuscules avant majuscules
99 "1" < "a"          # chiffres avant lettres
100
101 ## VALEURS BOOLÉENNES
102
103 ## 'TRUE' et 'FALSE' sont des noms réservés pour identifier les
104 ## valeurs booléennes correspondantes.
105 TRUE               # vrai
106 FALSE              # faux
107 !TRUE              # négation logique
108 TRUE & FALSE       # «et» logique
109 TRUE | FALSE       # «ou» logique
110
111 ## [Les expressions suivantes (qui anticipent sur la suite)
112 ## construisent les tables de vérité du «et» logique et du «ou»
113 ## logique.]
114 p <- c(TRUE, TRUE, FALSE, FALSE)
115 q <- c(TRUE, FALSE, TRUE, FALSE)
116 cbind("p" = p, "q" = q, "p ET q" = p & q)
117 cbind("p" = p, "q" = q, "p OU q" = p | q)
118
119 ## AUTRES DONNÉES FONDAMENTALES DE R
120
121 ## Donnée manquante. 'NA' est un nom réservé pour représenter une
122 ## donnée manquante.
123 NA                 # valeur admissible
124 NA + 2             # tout calcul avec 'NA' donne NA
125 NA == 2            # 'NA' n'est égale à rien...
126 NA == NA          # ... pas même elle-même!
127 is.na(NA)          # test si une valeur est 'NA'
128 is.na(2)           # '2' n'est pas manquante
129
130 ## Valeurs infinies et indéterminée. 'Inf', '-Inf' et 'NaN' sont des
131 ## noms réservés.
132 1/0                # +infini
133 -1/0               # -infini
134 0/0                # indétermination

```

```

135
136 ## Valeur "néant". 'NULL' est un nom réservé pour représenter le
137 ## néant, rien.
138 NULL # valeur admissible
139 NULL + 2 # aucun calcul possible avec néant
140
141 ###
142 ### COMMANDES R
143 ###
144
145 ## Les expressions entrées à la ligne de commande sont immédiatement
146 ## évaluées et le résultat est affiché à l'écran, comme avec une
147 ## grosse calculatrice.
148 1 # une constante
149 (2 + 3 * 5)/7 # priorité des opérations
150 3^5 # puissance
151 exp(3) # fonction exponentielle
152 sin(pi/2) + cos(pi/2) # fonctions trigonométriques
153 gamma(5) # fonction gamma
154
155 ## Lorsqu'une expression est syntaxiquement incomplète, l'invite de
156 ## commande change de '>' à '+ '.
157 2 - # expression incomplète
158 5 * # toujours incomplète
159 3 # complétée
160
161 ## Entrer le nom d'un objet affiche son contenu. Pour une fonction,
162 ## c'est son code source qui est affiché.
163 pi # constante numérique intégrée
164 letters # chaîne de caractères intégrée
165 LETTERS # version en majuscules
166 matrix # fonction interne
167
168 ## On crée des nouveaux objets en leur affectant une valeur avec
169 ## l'opérateur '<-'. *Ne pas* utiliser '=' pour l'affectation.
170 x <- 5 # affectation de 5 à l'objet 'x'
171 5 -> x # idem, mais peu utilisé
172 x # voir le contenu
173 (x <- 5) # affectation et affichage
174 y <- x # affecter la valeur de 'x' à 'y'
175 x <- y <- 5 # idem, en une seule expression
176 x <- (y <- 5) # équivalent
177 y # 5
178 x <- 0 # changer la valeur de 'x'...
179 y # ... ne change pas celle de 'y'
180
181 ## Pour regrouper plusieurs expressions en une seule commande, il faut
182 ## soit les séparer par un point-virgule ';', soit les regrouper à

```

```

183 ## l'intérieur d'accolades { } et les séparer par des retours à la
184 ## ligne.
185 x <- 5; y <- 2; x + y      # compact; éviter dans les scripts
186 x <- 5;                    # éviter les ';' superflus
187 {                          # début d'un groupe
188     x <- 5                  # première expression du groupe
189     y <- 2                  # seconde expression du groupe
190     x + y                  # dernière expression du groupe
191 }                          # fin du groupe et résultat
192 {x <- 5; y <- 2; x + y}    # valide, mais redondant
193
194 ###
195 ### VECTEURS
196 ###
197
198 ## CRÉATION DE VECTEURS
199
200 ## La fonction de base pour créer des vecteurs est 'c'. Il peut
201 ## s'avérer utile de nommer les éléments d'un vecteur.
202 x <- c(A = -1, B = 2, C = 8, D = 10) # création d'un vecteur
203 names(x)                            # extraire les noms
204 names(x) <- letters[1:length(x)]    # changer les noms
205 x                                    # nouveau vecteur
206
207 ## Attention aux chaînes de caractères! Dans R, une chaîne de
208 ## caractères n'est pas un vecteur de caractères. Une chaîne est un
209 ## vecteur de longueur 1 comptant plusieurs caractères. Un vecteur
210 ## peut contenir plusieurs chaînes de caractères.
211 length("foobar")                    # *une* chaîne de 6 caractères
212 c("foo", "bar")                     # *deux* chaînes de 3 caractères
213 length(c("foo", "bar"))              # longueur de 2
214
215 ## La fonction 'vector' sert à initialiser des vecteurs avec des
216 ## valeurs prédéterminées. Elle compte deux arguments: le mode du
217 ## vecteur et sa longueur. Les fonctions 'numeric', 'logical',
218 ## 'complex' et 'character' constituent des raccourcis pour des appels
219 ## à 'vector'.
220 vector("numeric", 5)                # vecteur initialisé avec des 0
221 numeric(5)                          # équivalent
222 numeric                             # en effet, voici la fonction
223 logical(5)                          # initialisé avec FALSE
224 complex(5)                          # initialisé avec 0 + 0i
225 character(5)                       # initialisé avec chaînes vides
226
227 ## Si l'on mélange dans un même vecteur des objets de mode différents,
228 ## il y a conversion forcée vers le mode pour lequel il y a le moins
229 ## de perte d'information, c'est-à-dire vers le mode qui permet le
230 ## mieux de retrouver la valeur originale des éléments.

```

```

231 c(5, TRUE, FALSE)      # conversion en mode 'numeric'
232 c(5, "z")              # conversion en mode 'character'
233 c(TRUE, "z")           # conversion en mode 'character'
234 c(5, TRUE, "z")        # conversion en mode 'character'
235
236 ## Les fonctions 'as.numeric', 'as.logical' et 'as.character' sont
237 ## utiles pour forcer la conversion d'un mode vers un autre.
238 as.logical(1)           # conversion en booléen
239 as.numeric(TRUE)        # conversion en numérique
240 as.character(1)         # conversion en chaîne de caractères
241
242 ## ARITHMÉTIQUE VECTORIELLE
243
244 ## L'unité de base de l'arithmétique en R est le vecteur. Cela rend
245 ## très simple et intuitif de faire des opérations mathématiques
246 ## courantes.
247 ##
248 ## Là où plusieurs langages de programmation exigent des boucles, R
249 ## fait le calcul directement.
250 ##
251 ## En effet, les règles de l'arithmétique en R sont globalement les
252 ## mêmes qu'en algèbre vectorielle et matricielle.
253 5 * c(2, 3, 8, 10)      # multiplication par une constante
254 c(2, 6, 8) + c(1, 4, 9) # addition de deux vecteurs
255 c(0, 3, -1, 4)^2        # élévation à une puissance
256 c(0, 3, -1, 4) > 0      # inégalité
257
258 ## Il n'est pas possible d'écrire des inégalités doubles du type  $0 < x$ 
259 ##  $< 10$  en R (comme, par ailleurs, dans la plupart des langages de
260 ## programmation). L'inégalité précédente se lit: « $x > 0$  ET  $x < 10$ ».
261 ## Pour calculer l'inégalité double, il s'agit donc de calculer les
262 ## deux inégalités et de les relier par un «et» logique qui fera le
263 ## calcul élément par élément.
264 x <- c(-2, 3, 0, -1, 12, 8) # vecteur
265 0 < x                       # première inégalité
266 x < 10                      # seconde inégalité
267 0 < x & x < 10              # inégalité double
268 x <= 0 | x >= 10           # négation de l'inégalité
269
270 ## Dans les règles de l'algèbre vectorielle, les longueurs des
271 ## vecteurs doivent toujours concorder.
272 ##
273 ## R permet plus de flexibilité en recyclant les vecteurs les plus
274 ## courts dans une opération.
275 ##
276 ## Il n'y a donc à peu près jamais d'erreurs de longueur en R! C'est
277 ## une arme à deux tranchants: le recyclage des vecteurs facilite le
278 ## codage, mais peut aussi résulter en des réponses complètement

```

```

279 ## erreurs sans que le système ne détecte d'erreur.
280 8 + 1:10                # 8 est recyclé 10 fois
281 c(2, 5) * 1:10          # c(2, 5) est recyclé 5 fois
282 c(-2, 3, -1, 4)^(1:4)   # quatre puissances différentes
283 c(-2, 3, -1, 4) <= 1:4   # quatre inégalités différentes
284 c(TRUE, FALSE) | FALSE  # deux «ou» logiques
285
286 ## Dans les opérations arithmétiques (ou, plus généralement, les
287 ## opérations conçues pour travailler avec des nombres), les valeurs
288 ## booléennes TRUE et FALSE sont automatiquement converties en 1 et 0,
289 ## respectivement. Conséquence: il est possible de faire des calculs
290 ## avec des valeurs booléennes!
291 c(5, 3) + c(TRUE, FALSE) # équivalent à c(5, 3) + c(1, 0)
292 5 + (3 < 4)              # (3 < 4) vaut TRUE
293 5 + 3 < 4                # priorité des opérations!
294
295 ## Dans les opérations logiques, ce sont les nombres qui sont
296 ## convertis en valeurs booléennes. Dans ce cas, zéro est traité comme
297 ## FALSE et tous les autres nombres comme TRUE.
298 0:5 & 5:0
299 0:5 | 5:0
300 !0:5
301
302 ## INDICAGE
303
304 ## L'indication est une opération importante et beaucoup utilisée. Elle
305 ## sert à extraire des éléments d'un vecteur avec la construction
306 ## 'x[i]', ou à les remplacer avec la construction 'x[i] <- y'. Les
307 ## fonctions sous-jacentes sont '[' et '[<-'.
308 ##
309 ## Les expressions suivantes illustrent les cinq méthodes d'indication.
310 x                        # le vecteur
311 x[1]                    # extraction par position
312 "["(x, 1)              # idem avec la fonction '['
313 x[-2]                   # suppression par position
314 x[x > 5]                # extraction par critère
315 x[0 < x & x < 10]       # autre extraction par critère
316 x["c"]                  # extraction par étiquette
317 x[]                     # tous les éléments
318 x[numeric(0)]           # différent d'indice vide
319
320 ## Laissons tomber les noms de l'objet.
321 names(x) <- NULL        # suppression de l'attribut 'names'
322
323 ## Quelques cas spéciaux d'indication.
324 length(x)               # rappel de la longueur
325 x[1:8]                  # vecteur allongé avec des 'NA'
326 x[0]                    # extraction de rien

```

```

327 x[0] <- 1; x           # affectation de rien
328 x[c(0, 1, 2)]         # indice 0 ignoré
329 x[c(1, NA, 5)]        # indice 'NA' retourne NA
330 x[2.6]                # fractions tronquées vers 0
331
332 ###
333 ### FONCTIONS
334 ###
335
336 ## PROGRAMMATION FONCTIONNELLE
337
338 ## Les fonctions sont des objets comme les autres dans R. Cela
339 ## signifie que:
340 ##
341 ## - le contenu d'une fonction (son code source) est toujours
342 ##   accessible;
343 ## - une fonction peut accepter en argument une autre fonction;
344 ## - une fonction peut retourner une fonction comme résultat;
345 ## - l'utilisateur peut définir de nouvelles fonctions.
346 seq                    # contenu est le code source
347 mode(seq)              # mode est "function"
348 rep(seq(5), 3)         # fonction argument d'une fonction
349 lapply(1:5, seq)       # idem
350 mode(ecdf(rpois(100, 1))) # résultat de ecdf est une fonction
351 ecdf(rpois(100, 1))(5)  # évaluation en un point
352 c(seq, rep)            # vecteur de fonctions!
353
354 ## DÉFINITION D'UNE FONCTION
355
356 ## On définit une nouvelle fonction avec la syntaxe suivante:
357 ##
358 ##   <nom> <- function(<arguments>) <corps>
359 ##
360 ## où
361 ##
362 ## - 'nom' est le nom de la fonction;
363 ## - 'arguments' est la liste des arguments, séparés par des virgules;
364 ## - 'corps' est le corps de la fonction, soit une expression ou un
365 ##   groupe d'expressions réunies par des accolades { }.
366 ##
367 ## Une fonction retourne toujours la valeur de la *dernière*
368 ## expression de celle-ci.
369 ##
370 ## Voici un exemple trivial.
371 square <- function(x) x * x
372 square(10)
373
374 ## Suite à l'appel de fonction ci-dessus, R a automatiquement affiché

```

```

375 ## le résultat à l'écran, et ce, parce que le résultat n'est pas
376 ## affecté à une variable. L'expression suivante effectue le même
377 ## calcul, mais en affectant le résultat dans un objet. Comme à
378 ## l'habitude, aucun résultat ne s'affiche alors à l'écran.
379 y <- square(10)
380
381 ## Supposons que l'on veut écrire une fonction pour calculer
382 ##
383 ##  $f(x, y) = x (1 + xy)^2 + y (1 - y) + (1 + xy)(1 - y)$ .
384 ##
385 ## Deux termes sont répétés dans cette expression. On a donc
386 ##
387 ##  $a = 1 + xy$ 
388 ##  $b = 1 - y$ 
389 ##
390 ## et  $f(x, y) = x a^2 + yb + ab$ .
391 ##
392 ## Une manière élégante de procéder au calcul de  $f(x, y)$  qui adopte
393 ## l'approche fonctionnelle fait appel à une fonction intermédiaire à
394 ## l'intérieur de la première fonction. (Il y a ici des enjeux de
395 ## «portée lexicale» sur lesquels nous reviendrons en détail dans un
396 ## chapitre ultérieur.)
397 f <- function(x, y)
398 {
399   g <- function(a, b)
400     x * a^2 + y * b + a * b
401   g(1 + x * y, 1 - y)
402 }
403 f(2, 3)
404
405 ## FONCTION ANONYME
406
407 ## Comme le nom du concept l'indique, une fonction anonyme est une
408 ## fonction qui n'a pas de nom. C'est parfois utile pour des fonctions
409 ## courtes utilisées dans une autre fonction.
410 ##
411 ## Reprenons l'exemple précédent en généralisant les expressions des
412 ## termes 'a' et 'b'. La fonction 'f' pourrait maintenant prendre en
413 ## arguments 'x', 'y' et des fonctions pour calculer 'a' et 'b'.
414 f <- function(x, y, fa, fb)
415 {
416   g <- function(a, b)
417     x * a^2 + y * b + a * b
418   g(fa(x, y), fb(x, y))
419 }
420
421 ## Plutôt que de définir deux fonctions pour les arguments 'fa' et
422 ## 'fb', on passe directement des fonctions anonymes en argument.

```

```

423 f(2, 3,
424     function(x, y) 1 + x * y,
425     function(x, y) 1 - y)
426
427 ## VALEUR PAR DÉFAUT D'UN ARGUMENT
428
429 ## La fonction suivante calcule la distance entre deux points dans
430 ## l'espace euclidien à 'n' dimensions, par défaut par rapport à
431 ## l'origine.
432 ##
433 ## Remarquez comment nous spécifions une valeur par défaut, l'origine,
434 ## pour l'argument 'y'.
435 ##
436 ## (Note: la fonction 'sum' effectue... la somme de tous les éléments
437 ## d'un vecteur.)
438 dist <- function(x, y = 0) sum((x - y)^2)
439
440 ## Quelques calculs de distances.
441 dist(c(1, 1)) # (1, 1) par rapport à l'origine
442 dist(c(1, 1, 1), c(3, 1, 2)) # entre (1, 1, 1) et (3, 1, 2)
443
444 ## Certaines fonctions dans R ont un vecteur de valeurs comme valeur
445 ## par défaut pour un argument. Par convention, cela sert à identifier
446 ## les valeurs admissibles de l'argument.
447 ?order # voir l'argument 'method'
448 ?norm # voir l'argument 'type'
449
450 ## Ajoutons une fonctionnalité à la fonction 'dist' qui permet de
451 ## fournir la position d'un point non plus seulement en coordonnées
452 ## cartésiennes, mais aussi en coordonnées polaires (rayon et angle;
453 ## deux dimensions seulement). Les coordonnées polaire (r, a)
454 ## correspondent aux coordonnées cartésiennes (r cos a, r sin a). Si
455 ## une seule des coordonnées polaires est fournie, nous allons
456 ## supposer qu'il s'agit du rayon et que l'angle vaut 0 (c'est
457 ## discutable comme choix).
458 ##
459 ## Le calcul en coordonnées polaires ne permet malheureusement pas de
460 ## traiter le cas du calcul par rapport à l'origine aussi élégamment
461 ## que ci-dessus. En revanche, l'ajout de la fonctionnalité nous
462 ## permet d'illustrer plusieurs autres concepts:
463 ##
464 ## 1. l'utilisation de la fonction 'match.arg' pour déterminer la
465 ## valeur d'un argument parmi les valeurs possibles;
466 ## 2. les expressions conditionnelles (abordées en détail plus loin);
467 ## 3. l'utilisation d'une fonction auxiliaire ('p2c') à l'intérieur de
468 ## la fonction principale pour la conversion des coordonnées
469 ## polaires en coordonnées cartésiennes;
470 ## 4. la sortie immédiate d'une fonction avec 'return' et avec 'stop'.

```



```

471 ##
472 ## Étudiez donc attentivement le code de la fonction ci-dessous avant
473 ## d'évaluer les exemples d'utilisation.
474 dist <- function(x, y = 0, coord = c("cartesian", "polar"))
475 {
476   coord <- match.arg(coord)
477   if (coord == "polar")
478   {
479     if (max(length(x), length(y)) != 2)
480       stop("coordonnées polaires permises ",
481            "en deux dimensions seulement")
482     p2c <- function(x)
483     {
484       if (length(x) == 1)
485         return(c(x, 0))
486       x[1] * c(cos(x[2]), sin(x[2]))
487     }
488     x <- p2c(x)
489     y <- p2c(y)
490   }
491   sum((x - y)^2)
492 }
493
494 ## Quelques calculs. Le point (1, 1) en coordonnées cartésiennes
495 ## correspond au point (sqrt(2), pi/4) en coordonnées polaires. La
496 ## valeur par défaut de l'argument 'coord' est la première du vecteur,
497 ## soit "cartesian" dans le cas présent.
498 dist(c(1, 1)) # comme précédemment
499 dist(c(1, 1), coord = "cartesian") # idem
500 dist(c(sqrt(2), pi/4), coord = "polar") # même point
501 dist(2) # valide en coord. cart.
502 dist(2, coord = "polar") # erreur en coord. pol.
503
504 ## La fonction 'match.arg' permet d'abréger la valeur d'un argument
505 ## jusqu'à un nom qui permet de l'identifier de manière unique.
506 dist(c(1, 1), coord = "c") # argument abrégé
507 dist(c(sqrt(2), pi/4), coord = "p") # argument abrégé
508
509 ## La fonction 'match.arg' se charge également de vérifier que la
510 ## valeur d'un argument est bien une des valeurs permises.
511 dist(c(1, 1), coord = "polaire") # argument invalide
512
513 ## ARGUMENT '...'
514
515 ## Nous illustrons l'utilisation de l'argument '...' de la manière
516 ## suivante pour le moment. Nous utiliserons davantage cet argument
517 ## avec les fonctions d'application.
518 ##

```

```
519 ## La fonction 'curve' prend en argument une expression mathématique
520 ## et trace la fonction pour un intervalle donné.
521 curve(x^2, from = 0, to = 2)
522
523 ## Nous souhaitons, pour une raison quelconque, que tous nos
524 ## graphiques de ce type (et seulement de ce type) soient tracés en
525 ## orange.
526 curve(x^2, from = 0, to = 2, col = "orange")
527
528 ## Plutôt que de redéfinir entièrement la fonction 'curve' avec tous
529 ## ses arguments (et il y en a plusieurs), nous pouvons écrire une
530 ## petite fonction qui, grâce à l'argument '...', accepte tous les
531 ## arguments de 'curve'.
532 ocurve <- function(...) curve(..., col = "orange")
533 ocurve(x^2, from = 0, to = 2)
534
535 ## APPEL D'UNE FONCTION
536
537 ## L'interpréteur R reconnaît un appel de fonction au fait que le nom
538 ## de l'objet est suivi de parenthèses ( ).
539 ##
540 ## Une fonction peut n'avoir aucun argument ou plusieurs. Il n'y a pas
541 ## de limite pratique au nombre d'arguments.
542 ##
543 ## Les arguments d'une fonction peuvent être spécifiés selon l'ordre
544 ## établi dans la définition de la fonction.
545 ##
546 ## Cependant, il est beaucoup plus prudent et *fortement recommandé*
547 ## de spécifier les arguments par leur nom avec une construction de la
548 ## forme 'nom = valeur', surtout après les deux ou trois premiers
549 ## arguments.
550 ##
551 ## L'ordre des arguments est important; il est donc nécessaire de les
552 ## nommer s'ils ne sont pas appelés dans l'ordre.
553 ##
554 ## Certains arguments ont une valeur par défaut qui sera utilisée si
555 ## l'argument n'est pas spécifié dans l'appel de la fonction.
556 ##
557 ## Examinons la définition de la fonction 'matrix', qui sert à créer
558 ## une matrice à partir d'un vecteur de valeurs.
559 args(matrix)
560
561 ## La fonction compte cinq arguments et chacun a une valeur par défaut
562 ## (ce n'est pas toujours le cas).
563 ##
564 ## Quel sera le résultat de l'appel ci-dessous?
565 matrix()
566
```

```
567 ## Les invocations de la fonction 'matrix' ci-dessous sont toutes
568 ## équivalentes.
569 ##
570 ## Portez attention si les arguments sont spécifiés par nom ou
571 ## par position.
572 matrix(1:12, 3, 4)
573 matrix(1:12, ncol = 4, nrow = 3)
574 matrix(nrow = 3, ncol = 4, data = 1:12)
575 matrix(nrow = 3, ncol = 4, byrow = FALSE, 1:12)
576 matrix(nrow = 3, ncol = 4, 1:12, FALSE)
577
578 ###
579 ### EXPRESSIONS CONDITIONNELLES
580 ###
581
582 ## Débutons par deux petits exemples qui démontrent un usage adéquat
583 ## de 'if'.
584 x <- c(-1, 2, 3)
585 if (any(x < 0)) print("il y a des nombres négatifs")
586 if (all(x > 0)) print("tous les nombres sont positifs")
587
588 ## Première erreur fréquente dans l'utilisation de 'if': la condition
589 ## en argument n'est pas une valeur unique. Il n'y a pas de scénario
590 ## dans lequel ce pourrait être souhaitable de procéder ainsi. R
591 ## signale donc une erreur.
592 if (x < 0) print("il y a des nombres négatifs")
593
594 ## Seconde erreur fréquente: tester que vrai est vrai. (Ce n'est pas
595 ## une «erreur» au sens propre puisque la syntaxe est valide, mais
596 ## c'est un non-sens sémantique, une forme de pléonasme comme «monter
597 ## en haut» ou «deux jumeaux».)
598 ##
599 ## Voici un exemple de construction avec un test inutile. Le résultat
600 ## de 'any' est déjà TRUE ou FALSE, alors pas besoin de vérifier si
601 ## TRUE == TRUE ou si FALSE == TRUE. Comparez avec la version
602 ## sémantiquement correcte, ci-dessus.
603 if (any(x < 0) == TRUE) print("il y a des nombres négatifs")
604
605 ## Voici trois mises en oeuvre de la fonction valeur absolue
606 ## accompagnées de leur algorithme. Elles vont de la plus
607 ## (inutilement) compliquée à la plus simple (sans aller jusqu'à
608 ## utiliser la fonction interne 'abs').
609 ##
610 ## Attention: ces fonctions ne sont pas vectorielles. (Pourquoi?)
611 ##
612 ## Algorithme 1 (trois clauses)
613 ##   abs(réel x)
614 ##   Si (x > 0)
```

```
615  ##      Retourner x
616  ##      Sinon si (x = 0)
617  ##      Retourner 0
618  ##      Sinon
619  ##      Retourner -x
620  ##      Fin abs
621  abs <- function(x)
622  {
623      if (x > 0)
624          x
625      else if (x == 0)
626          0
627      else
628          -x
629  }
630  abs(5)
631  abs(0)
632  abs(-2)
633
634  ## Algorithme 2 (deux clauses)
635  ##      abs(réel x)
636  ##      Si (x < 0)
637  ##      Retourner -x
638  ##      Sinon
639  ##      Retourner x
640  ##      Fin abs
641  abs <- function(x)
642  {
643      if (x < 0)
644          -x
645      else
646          x
647  }
648  abs(5)
649  abs(0)
650  abs(-2)
651
652  ## Algorithme 3 (une seule clause; requiert 'return')
653  ##      abs(réel x)
654  ##      Si (x < 0)
655  ##      Retourner -x
656  ##      Retourner x
657  ##      Fin abs
658  abs <- function(x)
659  {
660      if (x < 0)
661          return(-x)
662      x
```

```
663 }
664 abs(5)
665 abs(0)
666 abs(-2)
667
668 ## Supprimons la fonction 'abs' pour éviter qu'elle entre en conflit
669 ## avec la fonction interne de R du même nom.
670 rm("abs")
671
672 ## L'exemple suivant vise à vous faire réaliser que les valeurs
673 ## booléennes constituent implicitement des résultats de tests
674 ## logiques.
675 ##
676 ## Nous voulons écrire une fonction 'is.odd' qui détermine si les
677 ## éléments d'un vecteur sont impairs ou non. Un nombre est impair si
678 ## le reste de sa division par 2 (le modulo) est égal à 1.
679 ##
680 ## Voici une première mise en oeuvre naïve.
681 is.odd <- function(x)
682 {
683     if (x %% 2 == 1)
684         TRUE
685     else
686         FALSE
687 }
688
689 ## Le résultat de 'x %% 2' est nécessairement 0 ou 1. Ces valeurs
690 ## étant automatiquement converties en FALSE et TRUE dans une
691 ## opération logique, nous pouvons les utiliser directement à
692 ## l'intérieur de la condition d'une clause 'if' sans tester leur
693 ## égalité avec 0 ou 1.
694 ##
695 ## De plus, comme le résultat d'une condition est nécessairement TRUE
696 ## ou FALSE, la construction ci-dessus revient à dire: «si la
697 ## condition est vraie, retourner VRAI, sinon retourner FAUX».
698 ## N'est-ce pas redondant?
699 ##
700 ## Enfin, la mise en oeuvre ci-dessus n'est pas vectorielle puisqu'il
701 ## ne peut y avoir qu'un seul test à l'intérieur de la clause 'if'.
702 is.odd(c(0, 4, 3, 6, 8, 9, 1))
703
704 ## Avant d'étudier une meilleure mise en oeuvre, voyons comment ça
705 ## aurait pu être pire en testant explicitement 'TRUE == TRUE'.
706 is.odd <- function(x)
707 {
708     if ((x %% 2 == 1) == TRUE)
709         TRUE
710     else
```

```
711         FALSE
712     }
713
714     ## Ou pire encore en utilisant une logique tordue où l'on retourne
715     ## FAUX lorsque la condition est vraie et VRAI lorsqu'elle est fausse!
716     ## (Je n'invente rien, c'est un cas rencontré.)
717     is.odd <- function(x)
718     {
719         if ((x %% 2 == 0) == TRUE)
720             FALSE
721         else
722             TRUE
723     }
724
725     ## En fait, la solution simple, élégante et vectorielle retourne
726     ## simplement les résultats de l'opération modulo convertis en valeurs
727     ## booléennes, chacune constituant implicitement le résultat d'un test
728     ## logique.
729     is.odd <- function(x) as.logical(x %% 2)
730     is.odd(c(0, 4, 3, 6, 8, 9, 1))
731
732     ## Détail intéressant sur la structure 'if ... else ...': il est
733     ## possible de l'utiliser comme une fonction normale, c'est-à-dire
734     ## d'affecter le résultat de la structure à une variable.
735     ##
736     ## D'abord, le style de programmation le plus usuel: l'affectation est
737     ## effectuée à l'intérieur des clauses 'if' et 'else'.
738     f <- function(y)
739     {
740         if (y < 0)
741             x <- "rouge"
742         else
743             x <- "jaune"
744         paste("la couleur est:", x)
745     }
746     f(-2)
747     f(3)
748
749     ## Ensuite, la version où le résultat de 'if ... else ...' est
750     ## directement affecté dans la variable. C'est plus compact et très
751     ## lisible si la conséquence et l'alternative sont des expressions
752     ## courtes.
753     f <- function(y)
754     {
755         x <- if (y < 0) "rouge" else "jaune"
756         paste("la couleur est:", x)
757     }
758     f(-2)
```

```

759 f(3)
760
761 ## De l'inefficacité de 'ifelse'.
762 ##
763 ## Supposons que l'on veut une fonction *vectorielle* pour calculer
764 ##
765 ##  $f(x) = x + 2$ , si  $x < 0$ 
766 ##  $= x^2$ , si  $x \geq 0$ .
767 ##
768 ## On se tourne naturellement vers ifelse() pour ce genre de calcul.
769 ## Voyons voir le temps de calcul.
770 x <- sample(-10:10, 1e6, replace = TRUE)
771 system.time(ifelse(x < 0, x + 2, x^2))
772
773 ## Solution alternative n'ayant pas recours à ifelse(). C'est plus
774 ## long à programmer, mais l'exécution est néanmoins plus rapide.
775 f <- function(x)
776 {
777   y <- numeric(length(x)) # contenant
778   w <- x < 0 # x < 0 ou non
779   y[w] <- x[w] + 2 # calcul pour les x < 0
780   w <- !w # x >= 0 ou non
781   y[w] <- x[w]^2 # calcul pour les x >= 0
782   y
783 }
784 system.time(f(x))
785
786 ###
787 ### FONCTIONS INTERNES UTILES
788 ###
789
790 ## On se donne un vecteur pour les exemples qui suivent.
791 x <- c(50, 30, 10, 20, 60, 30, 20, 40)
792
793 ## FONCTIONS MATHÉMATIQUES ET TRIGONOMÉTRIQUES
794
795 ## R contient des fonctions pour calculer la plupart des fonctions
796 ## mathématiques et trigonométriques usuelles.
797 exp(c(1, 2, -1)) # exponentielle
798 log(exp(c(1, 2, -1))) # logarithme naturel
799 log10(c(1, 10, 100)) # logarithme en base 10
800 log(c(1, 5, 25), base = 5) # logarithme en base quelconque
801 sqrt(x) # racine carrée
802 abs(x - mean(x)) # valeur absolue
803 gamma(1:5) # fonction gamma
804 factorial(0:4) # factorielle
805 ?gamma # toutes les fonctions apparentées
806 cos(seq(0, pi, by = pi/4)) # cosinus

```

```

807 sin(seq(0, pi, by = pi/4)) # sinus
808 tan(seq(0, pi, by = pi/4)) # tangente
809 ?Trig                      # toutes les fonctions apparentées
810
811 ## SUITES ET RÉPÉTITION
812
813 ## La fonction 'seq' sert à générer des suites générales. Ses
814 ## principaux arguments sont 'from', 'to' et 'by'.
815 seq(from = 1, to = 10)      # équivalent à 1:10
816 seq(10)                    # idem
817 seq(1, 10, by = 2)          # avec incrément autre que 1
818 seq(-10, 10, length.out = 5) # incrément automatique
819
820 ## La fonction 'seq_len' génère une suite de longueur 'n' à partir de
821 ## 1. C'est une version simplifiée et plus rapide de 'seq(...,
822 ## length.out = n)'. De plus, elle est plus robuste lorsque l'argument
823 ## est 0.
824 seq(10)                     # suite 1, 2, ..., 10
825 seq(1, length.out = 10)     # idem robuste
826 seq_len(10)                 # équivalent et plus rapide
827 seq(0)                      # pas ce que l'on penserait!
828 seq(1, length.out = 0)      # plus prudent
829 seq_len(0)                  # plus simple, plus robuste
830
831 ## La fonction 'seq_along' génère une suite de la longueur du vecteur
832 ## en argument à partir de 1. C'est une version simplifiée et plus
833 ## rapide de 'seq(..., along = x)' et de 'seq_len(length(x))'. De
834 ## plus, elle est plus robuste que 'seq' lorsque la longueur du
835 ## vecteur est 0.
836 seq(1, along = x)            # suite de la longueur de x
837 seq_len(length(x))           # idem, mais deux fonctions
838 seq_along(x)                 # plus rapide, plus simple
839 y <- numeric(0)              # vecteur de longueur nulle
840 seq(length(y))               # pas ce qui souhaité!
841 1:length(y)                  # ça non plus!
842 seq_along(y)                 # plus simple, plus robuste
843
844 ## La fonction 'rep' permet de répéter des vecteurs de plusieurs
845 ## manières différentes.
846 rep(1, 10)                   # utilisation de base
847 rep(x, 2)                    # répéter un vecteur
848 rep(x, each = 4)              # répéter chaque élément
849 rep(x, times = 2, each = 4)   # combinaison des arguments
850 rep(x, length.out = 20)       # résultat de longueur déterminée
851 rep(x, times = 1:8)           # nombre de répétitions différent
852                               # pour chaque élément de 'x'
853
854 ## Pour les deux types de répétitions les plus usuels, il y a les

```



```
855 ## fonctions 'rep.int' et 'rep_len' qui sont plus rapides que 'rep'.
856 rep.int(x, 2)                # seulement répétition 'times'
857 rep_len(x, 10)              # seulement répétition 'length.out'
858
859 ## EXTRACTION DU DÉBUT ET DE LA FIN D'UN OBJET
860
861 ## L'idée des fonctions 'head' et 'tail', c'est que l'on se positionne
862 ## en tête ou en queue d'un objet pour effectuer des extractions ou
863 ## des suppressions de composantes.
864 ##
865 ## Avec un argument positif, les fonctions extraient des composantes
866 ## depuis la tête ou la queue de l'objet. Avec un argument négatif,
867 ## elles suppriment des composantes à l'«autre bout» de l'objet.
868 head(x, 3)                   # trois premiers éléments
869 head(x, -2)                  # tous sauf les deux derniers
870 tail(x, 3)                   # trois derniers éléments
871 tail(x, -2)                  # tous sauf les deux premiers
872
873 ## Les fonctions sont aussi valides sur les matrices et les data
874 ## frames. Elles extraient ou suppriment alors des lignes entières.
875 m <- matrix(1:30, 5, 6)      # matrice 5 x 6
876 head(m, 3)                   # trois premières lignes
877 tail(m, -2)                  # sans les deux premières lignes
878
879 ## ARRONDI
880 (x <- c(-21.2, -pi, -1.5, -0.2, 0, 0.2, 1.7823, 315))
881 round(x)                     # arrondi à l'entier
882 round(x, 2)                  # arrondi à la seconde décimale
883 round(x, -1)                 # arrondi aux dizaines
884 ceiling(x)                   # plus petit entier supérieur
885 floor(x)                     # plus grand entier inférieur
886 trunc(x)                     # troncature des décimales
887
888 ## TESTS LOGIQUES
889
890 ## Les fonctions 'any' et 'all' prennent en argument un vecteur
891 ## booléen et elles indiquent, respectivement, si au moins une ou si
892 ## toutes les valeurs sont TRUE.
893 any(c(TRUE, FALSE, FALSE))   # au moins une valeur TRUE
894 any(c(FALSE, FALSE, FALSE))  # aucune valeur TRUE
895 all(c(TRUE, TRUE, TRUE))     # toutes les valeurs TRUE
896 all(c(TRUE, FALSE, TRUE))    # pas toutes les valeurs TRUE
897
898 ## Les fonctions sont des compléments l'une de l'autre: si 'any(x)'
899 ## est TRUE, alors 'all(!x)' est FALSE, et vice-versa.
900 any(c(TRUE, FALSE, FALSE))    # TRUE
901 all(!c(TRUE, FALSE, FALSE))   # complément: FALSE
902 any(c(FALSE, FALSE, FALSE))   # FALSE
```

```

903 all(!c(FALSE, FALSE, FALSE)) # complément: TRUE
904
905 ## Les fonctions sont habituellement utilisées avec une expression
906 ## logique en argument.
907 x                                # rappel
908 x > 50                            # valeurs > 50?
909 x <= 50                          # valeurs <= 50?
910 any(x > 50)                      # y a-t-il des valeurs > 50?
911 all(x <= 50)                    # complément
912 all(x > 50)                     # toutes les valeurs > 50?
913 any(x <= 50)                   # complément
914
915 ## SOMMAIRES ET STATISTIQUES DESCRIPTIVES
916 sum(x)                          # somme des éléments
917 prod(x)                        # produit des éléments
918 diff(x)                       # x[2] - x[1], x[3] - x[2], etc.
919 mean(x)                       # moyenne des éléments
920 mean(x, trim = 0.125)         # moyenne sans minimum et maximum
921 var(x)                        # variance (sans biais)
922 sd(x)                         # écart type
923 max(x)                        # maximum
924 min(x)                        # minimum
925 range(x)                     # c(min(x), max(x))
926 diff(range(x))               # étendue de 'x'
927 median(x)                    # médiane (50e quantile) empirique
928 quantile(x)                  # quantiles empiriques
929 quantile(x, 1:10/10)         # on peut spécifier les quantiles
930 summary(x)                   # plusieurs des résultats ci-dessus
931
932 ## SOMMAIRES CUMULATIFS ET COMPARAISONS ÉLÉMENT PAR ÉLÉMENT
933 (x <- sample(1:20, 6))
934 (y <- sample(1:20, 6))
935 cumsum(x)                    # somme cumulative de 'x'
936 cumprod(y)                   # produit cumulatif de 'y'
937 rev(cumprod(y))              # produit cumulatif (de g. à d.)
938 rev(cumprod(rev(y)))         # produit cumulatif (de d. à g.)
939 cummin(x)                    # minimum cumulatif
940 cummax(y)                    # maximum cumulatif
941 pmin(x, y)                   # minimum élément par élément
942 pmax(x, y)                   # maximum élément par élément
943
944 ## MANIPULATION DE CHAINES DE CARACTÈRES
945
946 ## Une chaîne de caractères est un regroupement entre guillemets " "
947 ## de 0, 1 ou plusieurs caractères. Puisque tout objet dans R est un
948 ## vecteur, une chaîne de caractères seule est un vecteur contenant
949 ## une seule chaîne de caractères. Un vecteur peut également contenir
950 ## plusieurs chaînes de caractères de longueurs différentes.

```

```

951 "foo"                                # vecteur d'une seule chaîne
952 c("", "foo", "foobar")              # vecteur de 3 chaînes
953
954 ## Les fonctions ci-dessous permettent de manipuler une chaîne de
955 ## caractères. Elles sont vectorielles, c'est-à-dire qu'elles peuvent
956 ## effectuer leur traitement sur chaque chaîne d'un vecteur.
957 s <- c("Programmation", "analyse", "données")
958 paste(1:3, "-", s)                  # création de 3 chaînes
959 paste0(1:3, "-", s)                 # idem, sans espace
960 paste0(s, collapse = " ")          # concaténation des 3 chaînes
961 nchar(s)                            # nombres de caractères
962 substr(s, 2, 5)                     # sous-chaînes
963 strsplit("2001-05", "-")            # séparation en sous-chaînes

```

4.9 Exercices

4.1 Évaluer « à la main » les expressions suivantes comme le ferait l'interpréteur R.

- a) $1:5 * c(0, 1, 0, 1, 0)$
- b) $c(2, 7, 1, 4) > 3$
- c) $c(2, 7, 1, 4) \leq c(10, 4)$
- d) $c(-1, 2, 4)^{(3:-2)}$
- e) $c(TRUE, FALSE, FALSE) \& c(TRUE, TRUE, FALSE)$
- f) $c(TRUE, FALSE, FALSE) \mid !2 < 3$
- g) $-1:1/0$

4.2 Soit x un vecteur contenant un jeu de données :

```

> x
[1] 18 11 10  2 19  9 12 15 13 12  1  6

```

Écrire des expressions R pour créer ce vecteur dans l'espace de travail, puis pour extraire les données suivantes.

- a) La deuxième donnée du vecteur.
- b) Les cinq premières données du vecteur.
- c) Les données strictement supérieures à 14.
- d) Toutes les données sauf la sixième, la septième et la douzième.

4.3 Soit $\mathbf{x} = (x_1, \dots, x_n)$ et $\mathbf{y} = (y_1, \dots, y_n)$ deux vecteurs de nombres réels. Composer des expressions R pour effectuer les calculs mathématiques ci-dessous. Vous pouvez utiliser les fonctions R suivantes : `sum`, `prod`, `max`, `abs`.

- a) $3(\mathbf{x} + \mathbf{y})$

- b) $n^{-1} \sum_{i=1}^n x_i$ (moyenne arithmétique de \mathbf{x})
- c) $(\prod_{i=1}^n x_i)^{1/n}$ (moyenne géométrique de \mathbf{x})
- d) $n / (\sum_{i=1}^n x_i^{-1})$ (moyenne harmonique de \mathbf{x})
- e) $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^n x_i y_i$ (produit scalaire entre \mathbf{x} et \mathbf{y})
- f) $\|\mathbf{x} - \mathbf{y}\|_1 = \sum_{i=1}^n |x_i - y_i|$ (norme 1 entre \mathbf{x} et \mathbf{y})
- g) $\|\mathbf{x} - \mathbf{y}\|_\infty = \max_{i=1, \dots, n} (|x_i - y_i|)$ (norme « infini » entre \mathbf{x} et \mathbf{y})

- 4.4** Pour chacune des parties b)-g) de l'exercice 4.3, composer une fonction R pour effectuer le calcul demandé. Nommer les fonctions `amean`, `gmean`, `hmean`, `pscal`, `norm1` et `normINF`, dans l'ordre.
- 4.5** En vous basant sur l'algorithme 2.6a, composer une fonction `emr1` pour calculer l'espérance résiduelle empirique d'un vecteur \mathbf{x} à un seuil d .
- 4.6** Le fichier de script `bases.R` propose des exemples de mises en œuvre des algorithmes 2.3, 2.3a et 2.3b pour le calcul de la valeur absolue.
- a) Pourquoi les fonctions `abs` de `bases.R` ne sont-elles pas vectorielles, alors que la fonction `abs` de R, elle, l'est ?
 - b) Pourquoi la mise en œuvre de l'algorithme 2.3b doit-elle avoir recours à la fonction `return`, et pas celles des algorithmes 2.3 et 2.3a ?
- 4.7** Soit b un nombre réel et n , un entier positif. Composer deux versions d'une fonction `pow` pour calculer b^n , l'une utilisant l'algorithme 2.7 et l'autre, l'algorithme 2.7a et la fonction `is.odd` du code informatique de la section 4.8.
- 4.8** Le plus petit commun multiple (PPCM) de deux entiers positifs m et n est égal à mn/d , où d est le plus grand commun diviseur (PGCD) de m et n .
- a) Composer un algorithme pour effectuer le calcul du plus petit commun multiple de deux entiers.
 - b) Composer une fonction PPCM qui permet d'effectuer ce calcul. La fonction devrait retourner la réponse sans effectuer aucun calcul pour les cas triviaux suivants :
 1. si $m = 0$ ou $n = 0$, le PPCM est 0 ;
 2. si $n = 1$, le PPCM est m (et vice-versa) ;
 3. si $m = n$, le PPCM est m .

Composer toutes les fonctions intermédiaires, le cas échéant.

- 4.9** On vous donne la définition d'une fonction R :

```
> f
function(x = 0, y = NULL, z) <corps>
```

Déterminer la valeur des arguments \mathbf{x} , \mathbf{y} et \mathbf{z} dans les appels de fonction ci-dessous.

- a) $f(2, 3, 4)$
- b) $f(2, z = 3, y = 4)$
- c) $f(x = 2, z = 3)$
- d) $f(z = 2, 3, x = 4)$
- e) $f(z = 2)$
- f) $f(z = 2, 3, 4)$
- g) $f(2, 3)$

4.10 Soit la fonction f suivante :

```
> f
function(x, y)
{
  g <- function(a)
    (x - y) * a + a * x
  g(1 + x * y)
}
```

Évaluer les résultats des appels ci-dessous.

- a) $f(1, 1)$
- b) $f(f(1, 1), 1)$
- c) $f(1, f(f(1, 1), 1))$

4.11 Générer les suites suivantes à l'aide des fonctions `rep`, `seq` et `c` seulement.

- 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

4.12 Générer les suites de nombres suivantes à l'aide de l'opérateur `:` et de la fonction `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

4.13 Sans utiliser les fonctions `factorial`, `lfactorial`, `gamma` ou `lgamma`, générer la suite $1!, 2!, \dots, 10!$, où $n! = n(n-1)(n-2) \cdots 2 \cdot 1$ est la fonction factorielle.

- 4.14** Trouver une relation entre x , y ($y \neq 0$), $x \% y$ (modulo) et $x \% \% y$ (division entière).
- 4.15** Soit x un vecteur de longueur 20. Écrire des expressions R permettant d'obtenir ou de calculer chacun des résultats demandés ci-dessous.
- a) Les cinq premiers éléments de x .
 - b) La valeur maximale de x .
 - c) La moyenne des cinq premiers éléments de x .
 - d) La moyenne des cinq derniers éléments de x .

5 Structures de données de R et fonctions d'application

Objectifs du chapitre

- ▶ Créer et manipuler les structures de données suivantes de R : vecteur, matrice, tableau, liste, tableau de données, facteur et date.
- ▶ Extraire des données des divers types d'objet ou y affecter de nouvelles valeurs à l'aide des méthodes d'indiciage.
- ▶ Utiliser les fonctions d'application pour effectuer des sommaires ou réduire des structures de données de R.

Ce chapitre regroupe deux sujets bien distincts d'un point de vue conceptuel, mais intimement reliés d'un point de vue fonctionnel : les structures de données de R et les fonctions d'application correspondantes.

Une structure de données est une manière d'organiser les données dans un ordinateur afin de pouvoir les traiter efficacement. L'étude des structures de données va généralement de pair avec celle des algorithmes puisqu'elles influencent directement leur performance. Le bon choix de structure de données peut faire la différence entre un programme performant et un autre trop lent ou inapte à traiter des grandes quantités de données.

Le modèle de données de R repose sur des structures abstraites et spécialisées appelées *objets*. Jusqu'ici, nous n'avons étudié que le vecteur simple (*atomic*). Il existe d'autres structures de données en R, mais leur mise en œuvre demeure tout à fait transparente pour les programmeuses et les programmeurs. Nous ferons donc l'impasse sur les notions de tableau (*array*), de liste chaînée (*linked list*), d'arbre (*tree*) ou de table de hachage (*hashtable*) dont traitent habituellement les ouvrages d'algorithmique et de programmation.

Le vecteur, la matrice, le tableau et la liste sont les types d'objets les plus fréquemment utilisés en programmation en R. Le facteur, le tableau de données (*data frame*) et la date sont davantage des structures de données spécialisées pour l'analyse de données.

Vous savez déjà que certains types de calculs répétitifs sont automatiquement pris en charge par R via la couche d'abstraction de l'arithmétique vectorielle (section 4.3.2). Pour les cas où l'arithmétique vectorielle ne s'applique pas, un grand nombre de calculs répétitifs peut être réduit à quelques fonctions abstraites dites d'*application* (*mapping*, un autre concept hérité du Lisp). Les fonctions d'application permettent de masquer, par une nouvelle couche d'abstraction, les calculs répétitifs sur les dimensions des matrices et des tableaux, ainsi que ceux sur les éléments d'un vecteur ou d'une liste. De plus, elles se prêtent tout naturellement à la programmation vectorielle.

5.1 Objets R

Cette section passe en revue certains détails sur les objets R que nous avons jusqu'à maintenant laissé dans l'ombre par souci de simplicité.

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 de plusieurs *attributs*.

5.1.1 Règles 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 dans les noms d'objet, mais je recommande fortement d'éviter cette pratique qui nuit à la portabilité du code. Le nom d'un objet ne peut débuter par un chiffre. Si le nom débute par un point, alors le second caractère ne peut être un chiffre.



R est sensible à la casse, ce qui signifie que `foo`, `Foo` et `FOO` sont trois objets distincts.

Certains noms sont utilisés par le système R, aussi vaut-il mieux éviter de les utiliser comme nom de variable ou de fonction. En particulier, évitez :

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

De plus, 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`, `while`,
`TRUE`, `FALSE`,
`Inf`, `NaN`, `NULL`,
`NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`,
`...`, `..1`, `..2`, etc.

Oui, l'argument spécial « ... » rencontré à la [section 4.4.2](#) est un objet dans R !

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
```



Écrivez toujours les valeurs booléennes TRUE et FALSE au long pour éviter des bogues difficiles à détecter.



Que signifient ces noms *foo*, *bar*, *foobar* et autres variantes que l'on rencontre souvent dans la documentation informatique ? Rien de particulier, justement ! Ce sont des **variables métasyntaxiques** [↗](#) sans signification propre. Chez les francophones, l'équivalent le plus répandu est sans doute *toto*.

5.1.2 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 5.1](#) 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. Le mode d'un objet est obtenu avec la fonction `mode`.

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

Les objets de mode `numeric`, `complex`, `logical` et `character` sont des objets *simples* (*atomic*) qui contiennent des données d'un seul type. En revanche, les objets de mode `list` ou `expression` sont des objets *récurifs* qui peuvent contenir d'autres objets. Par exemple, une liste peut contenir une ou plusieurs autres listes ; nous y reviendrons plus loin.

TAB. 5.1 – Modes disponibles et contenus correspondants

Mode	Contenu de l'objet
<code>numeric</code>	nombres réels
<code>complex</code>	nombres complexes
<code>logical</code>	valeurs booléennes
<code>character</code>	chaines de caractères
<code>function</code>	fonction
<code>list</code>	liste
<code>expression</code>	expressions non évaluées



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.

5.1.3 Longueur

La longueur d'un objet est égale au nombre d'éléments qu'il contient. La longueur d'un objet est obtenue avec la fonction `length`.

```
> v <- c(1, 2, 5, 9)
> length(v)
[1] 4
```

Au sens R du terme, la longueur d'une chaîne de caractères est toujours 1. Un objet de mode `character` doit contenir plusieurs chaînes de caractères pour que sa longueur soit supérieure à 1. Il faut utiliser la fonction `nchar` pour obtenir le nombre de caractères dans une chaîne.

```
> v1 <- "foobar"
> length(v1)
[1] 1
> nchar(v1)
[1] 6
```

```
> v2 <- c("f", "o", "o", "b", "a", "r")
> length(v2)
[1] 6
> nchar(v2)
[1] 1 1 1 1 1 1
```



Il est permis — et parfois utile — de créer un objet de longueur nulle, c'est-à-dire un objet qui existe, mais qui est vide.

5.1.4 Valeurs spéciales

Les objets valeur manquante (NA), infini (Inf, -Inf), valeur indéterminée (NaN) et néant (NULL) permettent de représenter de manière intuitive des quantités souvent utilisées dans les applications statistiques et en analyse de données. Vous devez connaître les caractéristiques spéciales de ces objets afin d'en tirer pleinement profit.

L'objet NA sert à représenter une donnée manquante. Chose quelque peu surprenante, c'est un objet de mode `logical`. Chose encore plus surprenante, NA n'est cependant ni TRUE, ni FALSE.

```
> mode(NA)
[1] "logical"
> length(NA)
[1] 1
> isTRUE(NA)
[1] FALSE
> isFALSE(NA)
[1] FALSE
```

Toute opération, y compris la comparaison, impliquant la valeur NA a comme résultat NA. Par conséquent, la valeur NA n'est égale à aucune autre, *pas même à elle-même*! Pour tester si une valeur est manquante, vous devez nécessairement avoir recours à la fonction `is.na`.

```
> x <- NA
> x == NA
[1] NA
> is.na(NA)
[1] TRUE
```

Les objets Inf, -Inf et NaN permettent de représenter les valeurs mathématiques spéciales prévues dans la norme IEEE 754 régissant la représentation interne des nombres dans un ordinateur (IEEE, 2003). De manière intuitive, l'objet Inf représente $+\infty$, -Inf représente $-\infty$ et NaN (*Not a Number*) représente une forme indéterminée du type $\frac{0}{0}$ ou $\infty - \infty$. Utilisez les fonctions `is.infinite`, `is.finite` et `is.nan` pour tester de manière robuste ces valeurs.

```
> is.infinite(1/0)
[1] TRUE
> is.finite(-1/0)
[1] FALSE
> is.nan(0/0)
```

TAB. 5.2 - Attributs les plus usuels d'un objet

Attribut	Utilisation
<code>class</code>	affecte le comportement d'un objet
<code>dim</code>	dimensions des matrices et tableaux
<code>dimnames</code>	étiquettes des dimensions des matrices et tableaux
<code>names</code>	étiquettes des éléments d'un objet

```
[1] TRUE
> is.nan(Inf - Inf)
[1] TRUE
```

Enfin, l'objet spécial `NULL` représente « rien », ou le vide. Son mode est `NULL` et sa longueur est 0. Il est toutefois différent d'un objet vide : un objet de longueur 0 est un contenant vide, alors que `NULL` est « pas de contenant ».

```
> mode(NULL)
[1] "NULL"
> length(NULL)
[1] 0
> 1 + NULL
numeric(0)
```

Ajouter `NULL` à un objet n'ajoute rien. Par ailleurs, la seule façon de tester si un objet est `NULL` est avec la fonction `is.null`.

```
> c(3, NULL)
[1] 3
> x <- NULL
> x == NULL
logical(0)
> is.null(NULL)
[1] TRUE
```

5.1.5 Attributs

Les attributs d'un objet sont des éléments d'information additionnels attachés à cet objet. Le [tableau 5.2](#) fournit la liste des attributs les plus fréquemment utilisés. À chacun des attributs du tableau correspond une fonction du même nom servant à extraire l'attribut d'un objet.

La fonction `attributes` permet d'extraire ou de modifier la liste des attributs

d'un objet, alors que la fonction `attr` permet de travailler sur un seul attribut à la fois. Les programmeurs peuvent ajouter à peu près n'importe quoi à la liste des attributs d'un objet. Par exemple, nous pourrions 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"
```

L'extraction d'un attribut qui n'existe pas retourne `NULL`, alors qu'à l'inverse, affecter à un attribut la valeur `NULL` efface cet attribut.

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



Étudiez les lignes 12–151 du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.2 Matrice et tableau

R étant un langage spécialisé dans les calculs mathématiques, il supporte tout naturellement et de manière intuitive — à une exception près, comme nous le verrons — les matrices et les tableaux. Dans la terminologie de R, une matrice est un tableau rectangulaire de nombres à deux dimensions, alors qu'un tableau (*array*) est la généralisation à plus de deux dimensions¹,

Nous avons établi d'entrée de jeu à la [section 4.3](#) qu'en R, tout est un vecteur. C'est aussi vrai pour les matrices et tableaux, qui 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. La présence de cet attribut change implicitement la classe de l'objet et, de ce fait, son mode d'affichage (sous forme de tableau rectangulaire) et son interaction avec plusieurs

1. Il est donc correct de dire qu'une matrice est un tableau à deux dimensions.

opérateurs et fonctions, notamment les opérateurs d'indexage. La fonction de base pour créer une matrice 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
```

Tel que mentionné précédemment, la généralisation d'une matrice à plus de deux dimensions est un tableau. Le nombre de dimensions du tableau est toujours égal à la longueur de l'attribut `dim`. 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
```

Formellement, un tableau peut aussi compter une seule dimension. Un tel tableau comporte un attribut `dim` de longueur 1, alors qu'un vecteur simple n'a pas d'attribut `dim`.



Vous aurez remarqué que R remplit les matrices et tableaux en faisant d'abord varier la première dimension, puis la seconde, etc. Pour les matrices, cela revient à les remplir par colonne. Cette convention, héritée du Fortran, n'est pas des plus intuitives, mais elle a ses avantages dans plusieurs applications. La fonction `matrix` possède un argument `byrow` qui permet d'inverser l'ordre de remplissage, mais il vaut mieux vous habituer à la convention de R que d'essayer constamment de la contourner.

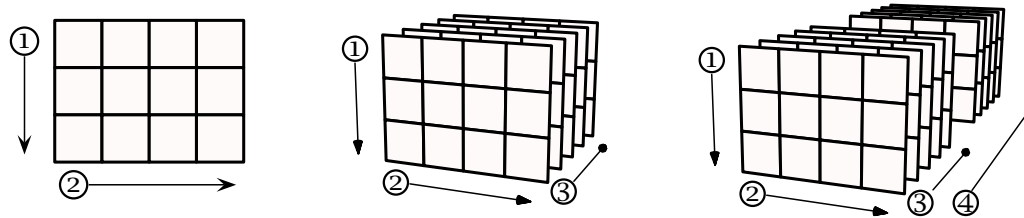


FIG. 5.1 – Représentations d'une matrice, d'un tableau à trois dimensions et d'un tableau à quatre dimensions. Les chiffres encadrés identifient l'ordre de remplissage des dimensions.

L'ordre inhabituel de remplissage des matrices et des tableaux vous causera des maux de tête si vous ne les visualisez pas correctement. Les tableaux à trois dimensions sont des prismes rectangulaires formés de matrices (remplies par colonne!) placées les unes *derrière* les autres. L'ajout d'une quatrième dimension revient à aligner des prismes les uns derrière les autres, et ainsi de suite. La [figure 5.1](#) fournit une représentation schématique des matrices et des tableaux à trois et à quatre dimensions.

L'indéçage dans les matrices et dans les tableaux se fait avec les crochets `[]`, comme pour les vecteurs simples. D'ailleurs, l'ensemble des règles d'indéçage de la [section 4.3.3](#) s'appliquent à l'identique aux matrices et tableaux, sinon qu'il faut maintenant préciser des indices pour chaque dimension, séparés par des virgules. Par exemple, on extrait un élément d'une matrice en précisant sa position dans chaque dimension de celle-ci.

```
> (m <- matrix(c(85, 37, 89, 37, 34, 89, 44, 79,
+               33, 84, 35, 70, 74, 42, 38),
+               nrow = 3))
      [,1] [,2] [,3] [,4] [,5]
[1,]  85   37   44   84   74
[2,]  37   34   79   35   42
[3,]  89   89   33   70   38
> m[1, 3]
[1] 44
```

Il est équivalent — quoique moins usité — d'utiliser la position de l'élément dans le vecteur sous-jacent à la matrice.

```
> m[7]
[1] 44
```

Les indices négatifs suppriment des dimensions et les vecteurs booléens sélectionnent des dimensions, comme d'habitude.

```
> m[-1, -3]
      [,1] [,2] [,3] [,4]
[1,]   37   34   35   42
[2,]   89   89   70   38
> m[c(FALSE, TRUE, TRUE),
+   c(FALSE, TRUE, TRUE, FALSE, TRUE)]
      [,1] [,2] [,3]
[1,]   34   79   42
[2,]   89   33   38
```

En vertu de la cinquième règle d'indilage de la [section 4.3.3](#), lorsqu'une dimension est laissée vide dans les crochets, tous les éléments de cette dimension sont extraits. C'est ainsi que nous pouvons extraire une ligne ou une colonne entière, par exemple.

```
> m[1, ]
[1] 85 37 44 84 74
> m[, 3]
[1] 44 79 33
> m[-1, ]
      [,1] [,2] [,3] [,4] [,5]
[1,]   37   34   79   35   42
[2,]   89   89   33   70   38
> m[, -3]
      [,1] [,2] [,3] [,4]
[1,]   85   37   84   74
[2,]   37   34   35   42
[3,]   89   89   70   38
```

Dans un tableau à trois dimensions, l'indilage sur une seule dimension sélectionne des « tranches » du tableau, alors que l'indilage sur deux dimensions simultanément sélectionne des « carottes »² ou des tubes. Ces idées se généralisent aisément à des tableaux à plus de trois dimensions. Les figures [5.2](#) et [5.3](#) proposent respectivement des représentations schématiques des opérations d'indilage d'une matrice et d'un tableau à trois dimensions.

Les fonctions `rbind` et `cbind` permettent de fusionner des matrices et des tableaux ayant au moins une dimension en commun. La fonction `rbind` (*row bind*) fusionne verticalement (empile) deux matrices (ou plus) ayant le même nombre de colonnes.

2. Un terme que j'emprunte à la géologie, où une carotte est un échantillon de roche cylindrique extrait d'un sol par forage.

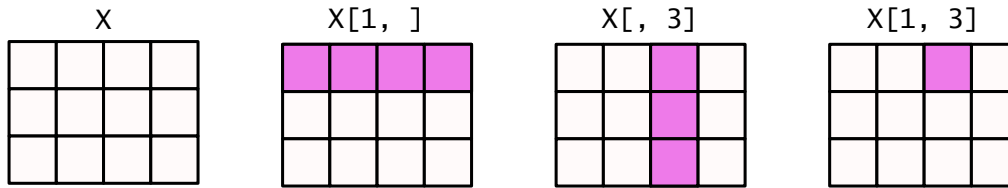


FIG. 5.2 - Représentations d'opérations d'indexage sur une matrice. De gauche à droite : matrice complète ; sélection d'une ligne ; sélection d'une colonne ; sélection d'un élément unique.

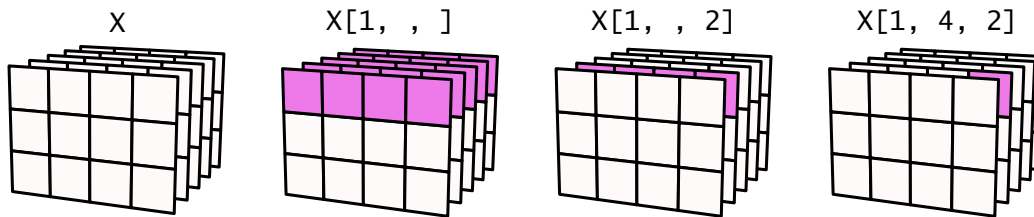


FIG. 5.3 - Représentations d'opérations d'indexage sur un tableau à trois dimensions. De gauche à droite : tableau complet ; sélection d'une tranche horizontale ; sélection d'une carotène transversale ; sélection d'un élément unique.

```
> n <- matrix(1:5, nrow = 1)
> rbind(m, n)
      [,1] [,2] [,3] [,4] [,5]
[1,]   85   37   44   84   74
[2,]   37   34   79   35   42
[3,]   89   89   33   70   38
[4,]    1    2    3    4    5
```

La fonction `cbind` (*column bind*), quant à elle, fusionne horizontalement (place côte à côte) deux matrices (ou plus) ayant le même nombre de lignes.

```
> n <- matrix(1:3, ncol = 1)
> cbind(m, n)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   85   37   44   84   74    1
[2,]   37   34   79   35   42    2
[3,]   89   89   33   70   38    3
```



Quand il s'agit de bien comprendre les opérations de création et d'indexage des matrices et des tableaux, une image vaut mille mots. Visionnez donc la vidéo sur les **matrices et tableaux** [🔗](#) en complément de la présentation ci-dessus.



Étudiez les lignes 154-273 du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.3 Application pour les matrices et les tableaux

Une matrice `X` contient le nombre de réclamations en assurance dommages de trois entreprises pour les cinq dernières années.

```
> X
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    1    1
[2,]    3    1    4    7    5
[3,]    0    3    2    2    1
```

Une analyste souhaite connaître le nombre total de réclamations par entreprise, d'une part, et par année, d'autre part. Cela consiste à calculer les sommes par ligne et par colonne, dans l'ordre. Or, aucun opérateur usuel de R ne permet d'effectuer directement de tels calculs. Notre analyste prend donc le temps de concevoir les algorithmes des procédures qu'elle devrait suivre pour effectuer ses calculs. (Par fantaisie, elle en écrit un en langage naturel et l'autre en pseudocode.)

Algorithme 5.1. Calculer les sommes par ligne d'une matrice `A`.

1. Poser n égal au nombre de lignes de `A`.
2. Définir un vecteur `s` de longueur n .
3. Poser l'élément i de `s` égal à la somme des éléments de la ligne i de `A`, $i = 1, 2, \dots, n$.
4. Retourner `s`.

Algorithme 5.2. Calculer les sommes par colonne d'une matrice `A`.

```
colsum(matrice A)
  n ← nombre de colonnes de A
  Sommes ← vecteur de longueur n
  Pour i = 1, 2, ..., n
    y ← colonne i de la matrice A
    Sommes[i] ← somme des éléments y
  Fin Pour
  Retourner Sommes
Fin colsum
```

Vous pouvez vérifier que les résultats de ces algorithmes sont (3,20,8) et (4,4,6,10,7), dans l'ordre.

L'analyste réalise ensuite rapidement qu'elle devrait utiliser ce type de procédure pour tout sommaire par ligne ou par colonne d'une matrice. Par exemple,

pour calculer le nombre moyen de réclamations par entreprise et par année, le seul changement qu'elle devrait apporter aux algorithmes ci-dessus consisterait à remplacer la somme des éléments (d'une ligne ou d'une colonne) par la moyenne. Elle se dit alors que tout ce qu'elle souhaite, c'est pouvoir facilement *appliquer* une fonction quelconque sur l'une ou l'autre des deux dimensions d'une matrice. Si vous avez bien suivi l'analyste dans son raisonnement, vous avez compris le concept d'application, soit : utiliser à répétition une même fonction sur les sous-ensembles d'une structure de données.

La fonction d'application pour les matrices et les tableaux est `apply`. Utilisée sur une matrice, elle permet d'effectuer des calculs sur les lignes ou les colonnes de celle-ci.

```
> apply(X, 1, sum)
[1] 3 20 8
> apply(X, 2, sum)
[1] 4 4 6 10 7
```

La syntaxe complète de `apply` est la suivante :

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

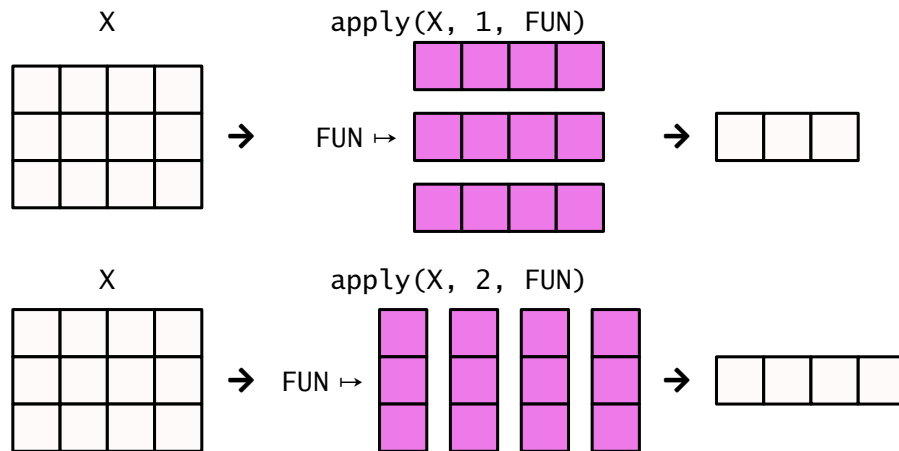
- ▶ X est une matrice ou un tableau.
- ▶ MARGIN est un vecteur d'entiers contenant la ou les dimensions de la matrice ou du tableau sur lesquelles la fonction doit s'appliquer.
- ▶ FUN est la fonction à appliquer sur la ou les dimensions. La fonction reçoit les sous-ensembles de X sans que ceux-ci ne soient nommés. Les règles habituelles d'évaluation d'un appel de fonction s'appliquent : les données constituent le premier argument de FUN, à moins que des arguments nommés dans '...' n'aient préséance.
- ▶ '...' contient des arguments supplémentaires à passer à FUN, séparés par des virgules.

La [figure 5.4](#) représente schématiquement des opérations d'applications sur les lignes et les colonnes d'une matrice.

Le quatrième argument de `apply` est l'argument formel spécial '...' dont nous avons traité à la [section 4.4.2](#). Il devient très utile, ici, lorsque la fonction FUN prend plus d'un argument.

Pour illustrer, reprenons l'exemple précédent en supposant maintenant que les entreprises n'ont pas toutes cinq années de données.

```
> X
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0   NA   NA
```

FIG. 5.4 – Représentations de la fonction d'application `apply` avec une matrice

[2,]	NA	NA	4	7	5
[3,]	0	3	2	2	NA

Si l'analyste a recours aux mêmes expressions que ci-dessus pour calculer les sommaires par ligne et par colonne, elle obtiendra des résultats sans grand intérêt puisque, comme on le sait, tout calcul avec des données manquantes retourne la valeur NA.

```
> apply(X, 1, sum)
[1] NA NA NA
> apply(X, 2, sum)
[1] NA NA 6 NA NA
```

Fort heureusement, la fonction `sum` possède un argument `na.rm` qui indique de supprimer les données manquantes avant d'effectuer la somme. L'analyste spécifie donc cet argument de la fonction `sum` dans l'argument `'...'` de `apply`.

```
> apply(X, 1, sum, na.rm = TRUE)
[1] 1 16 7
> apply(X, 2, sum, na.rm = TRUE)
[1] 1 3 6 9 5
```



Les sommaires les plus usuels pour les matrices sont la somme et la moyenne, par ligne et par colonne. Les fonctions `rowSums`, `colSums`, `rowMeans` et `colMeans` permettent de calculer plus directement ces sommaires.

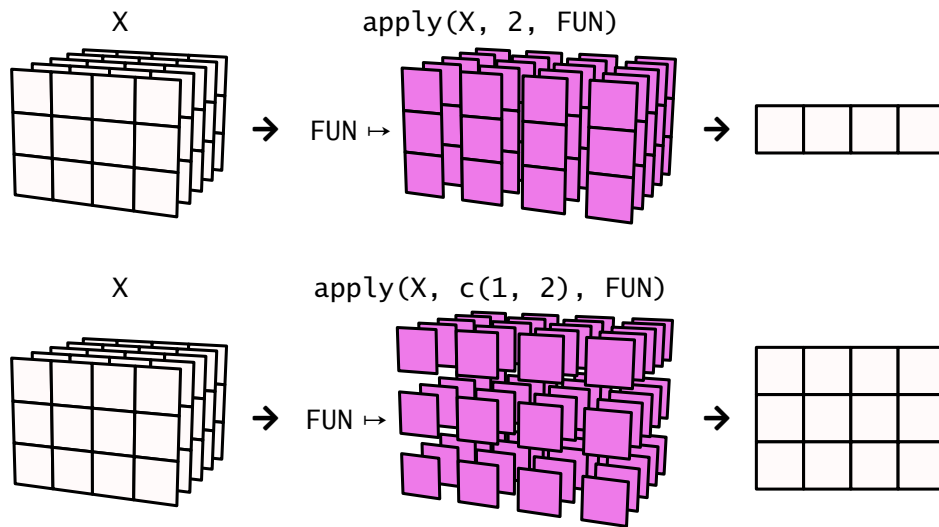


FIG. 5.5 - Représentations de la fonction d'application `apply` avec un tableau à trois dimensions

Lorsque l'on applique une fonction à un tableau à plus de deux dimensions, l'objet passé à la fonction peut s'avérer être une matrice ou un tableau, selon le contenu de l'argument `MARGIN`. Tel que mentionné à la section précédente lorsque nous discutons d'indexage des tableaux, si `X` est un tableau à trois dimensions et que la longueur de `MARGIN` est 1, on applique `FUN` sur les tranches de `X` (des matrices). Si la longueur de `MARGIN` est 2, on applique plutôt `FUN` aux carottes tirées de `X` (des vecteurs). La [figure 5.5](#) illustre ces principes schématiquement.

Illustrons ces propos à partir du tableau $3 \times 4 \times 2$ suivant.

```
> X
, , 1
      [,1] [,2] [,3] [,4]
[1,]    4    8    1    0
[2,]    4    4    9    3
[3,]    8    4    8    2

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

Les expressions ci-dessous calculent les quantités suivantes :

1. sommes des trois tranches horizontales (de l'avant vers l'arrière) dans le tableau;
2. sommes des quatre tranches verticales (du haut vers le bas);
3. sommes des deux tranches transversales (de la gauche vers la droite);
4. sommes des douze carottes horizontales;
5. sommes des huit carottes verticales;
6. sommes des six carottes transversales.

```
> apply(X, 1, sum)
[1] 37 48 47
> apply(X, 2, sum)
[1] 39 27 41 25
> apply(X, 3, sum)
[1] 55 77
> apply(X, c(1, 2), sum)
      [,1] [,2] [,3] [,4]
[1,]    9   13   10    5
[2,]   13    7   17   11
[3,]   17    7   14    9
> apply(X, c(2, 3), sum)
      [,1] [,2]
[1,]   16   23
[2,]   16   11
[3,]   18   23
[4,]    5   20
> apply(X, c(1, 3), sum)
      [,1] [,2]
[1,]   13   24
[2,]   20   28
[3,]   22   25
```



La vidéo sur la **fonction apply** [🔗](#) fournit une représentation animée des opérations d'application sur les matrices et les tableaux.



Truc mnémotechnique : la ou les dimensions figurant dans l'argument MARGIN sont celles retenues par le passage de `apply`. En d'autres termes, les dimensions du résultat de `apply` sont `dim(X) [MARGIN]`.

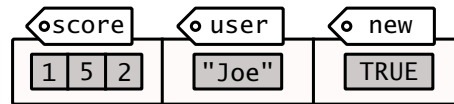


FIG. 5.6 – Représentation d'une liste composée d'un vecteur numérique, d'un vecteur caractère et d'un vecteur booléen, chaque élément de la liste étant étiqueté



Étudiez les lignes 276-334 du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.4 Liste

La liste est la structure de données la plus générale et polyvalente du langage R. Si les éléments d'un vecteur simple doivent tous être du même mode, ceux de la liste peuvent être de n'importe quel mode, y compris le mode `list`. Par conséquent, il est possible d'emboîter des listes, d'où le qualificatif de récursif pour ce type d'objet.

La liste est la structure de données par excellence pour réunir dans un objet plusieurs types d'objets différents. La fonction de base pour créer des listes est `list`.

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

$user
[1] "Joe"

$new
[1] TRUE
```

Dans l'exemple ci-dessus, la liste compte trois éléments. Le premier, étiqueté `score`, est un vecteur de mode `numeric`; le second, étiqueté `user`, est un vecteur de mode `character`; le troisième, étiqueté `new` est un vecteur de mode `logical`. La [figure 5.6](#) propose une représentation schématique de cette liste.



Étiquetez les éléments des listes puisqu'il peut s'avérer difficile autrement de les identifier. De plus, comme nous le verrons ci-dessous, il est très simple d'extraire les éléments d'une liste par leur étiquette.

L'indilage des listes recèle quelques subtilités auxquelles il faut porter une attention particulière. Tout d'abord, il existe trois opérateurs d'indilage : les crochets simples usuels « `[]` », les crochets doubles « `[[]]` » et le signe de dollar

« \$ » (ces deux derniers opérateurs figurent au [tableau 4.1](#), mais nous n'en avons pas parlé jusqu'ici).

- Les crochets simples fonctionnent avec la liste comme avec tout vecteur. Le résultat de l'indilage avec « [] » est une liste. Quand on y pense, c'est le seul résultat logique lorsque l'on indice une liste avec un vecteur. Toutefois, cette règle fait en sorte que le résultat de l'extraction d'un seul élément d'une liste avec les crochets simples est non pas l'objet en question, mais bien une liste d'un élément contenant l'objet. C'est rarement ce que l'on souhaite obtenir.

```
> x[1]
$score
[1] 1 5 2
> x[-2]
$score
[1] 1 5 2

$new
[1] TRUE
> x[c(1, 3)]
$score
[1] 1 5 2

$new
[1] TRUE
```

- Les crochets doubles permettent d'extraire un, et un seul, élément d'une liste à l'aide d'un entier positif ou d'une chaîne de caractères. Le résultat est l'objet « sorti » de la liste.

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

Petite fonctionnalité peu employée, mais élégante des crochets doubles : si l'indice est un vecteur d'entiers positifs, R utilisera les valeurs de celui-ci pour indiquer récursivement la liste. Autrement dit, R 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.

```
> x[[c(1, 2)]]
[1] 5
```

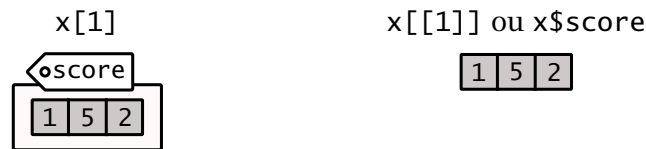



FIG. 5.7 – Représentations d'opérations d'indilage sur une liste

- Finalement, l'autre façon — la meilleure, en fait — de sélectionner un seul élément d'une liste est par son étiquette avec l'opérateur « \$ ».

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

Consultez la [figure 5.7](#) pour une comparaison illustrée des méthodes d'indilage d'une liste.

Mentionnons en terminant la fonction `unlist` qui convertit une liste en un vecteur simple. Elle est surtout utile pour concaténer les éléments d'une liste lorsque ceux-ci sont des vecteurs simples, idéalement de même mode, car il y aura autrement conversion forcée. Attention : cette fonction est destructrice si la structure interne de la liste est importante.

```
> unlist(x)
score1 score2 score3 user new
"1"    "5"    "2"   "Joe" "TRUE"
```



La liste est une structure de données très importante en R. Prenez le temps de bien étudier les lignes [337-404](#) du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.5 Application pour les listes et les vecteurs

Les idées de la [section 5.3](#) se transposent sans mal aux listes et aux vecteurs. En effet, il n'y a pas d'arithmétique vectorielle qui tienne pour les listes. Pour effectuer une opération sur chaque élément d'une liste, il faut utiliser une procédure itérative, procédure que R masque avec des fonctions d'application.

La fonction d'application de base pour les listes et les vecteurs est `lapply`. Elle applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous forme de liste. La syntaxe de `lapply` est similaire à celle de `apply`, l'argument `MARGIN` en moins :

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

- `X` est un vecteur ou une liste.

- FUN est la fonction à appliquer à chacun des éléments de X. Ceux-ci ne sont pas nommés lorsqu'ils sont passés à la fonction. Les règles habituelles d'évaluation d'un appel de fonction s'appliquent : les données constituent le premier argument de FUN, à moins que des arguments nommés dans « ... » n'aient préséance.

Vous avez déjà rencontré dans les exemples la fonction `sample` qui permet de tirer un échantillon aléatoire parmi un ensemble de valeurs. Le premier argument de `sample` est `x`, l'ensemble de valeurs. Le second argument est `size`, la taille de l'échantillon. Les expressions ci-dessous permettent, dans l'ordre : de créer une liste formée de quatre vecteurs aléatoires de taille 5, 6, 7 et 8 ; de calculer la somme de chacun de ces vecteurs ; de trier chacun des vecteurs.

```
> (x <- lapply(5:8, sample, x = 1:10))
[[1]]
[1] 2 10 3 1 9

[[2]]
[1] 3 10 6 9 5 7

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

[[4]]
[1] 4 5 1 9 7 6 10 2
> lapply(x, sum)
[[1]]
[1] 25

[[2]]
[1] 40

[[3]]
[1] 42

[[4]]
[1] 44
> lapply(x, sort)
[[1]]
[1] 1 2 3 9 10

[[2]]
[1] 3 5 6 7 9 10
```

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

[[4]]
[1] 1 2 4 5 6 7 9 10
```

Sœur siamoise de `lapply`, la fonction `sapply` retourne, lorsque c'est possible, son résultat sous forme de vecteur ou de matrice. Le résultat est donc *simplifié* par rapport à celui de `lapply`, d'où le nom de la fonction. La syntaxe ne change pas :

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

- Si les résultats de l'application sont tous des vecteurs de longueur 1, alors `sapply` retourne un vecteur.
- Si les résultats de l'application sont des vecteurs de la même longueur supérieure à 1, alors `sapply` retourne une matrice, remplie comme toujours par colonne.
- Si les résultats de l'application sont des vecteurs de longueurs différentes, alors `sapply` est identique à `lapply`.

```
> (x <- lapply(rep(5, 3), sample, x = 1:10))
[[1]]
[1] 6 4 8 7 2

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

[[3]]
[1] 1 4 7 5 10
> sapply(x, sum)
[1] 27 28 27
> sapply(x, sort)
      [,1] [,2] [,3]
[1,]    2    1    1
[2,]    4    3    4
[3,]    6    6    5
[4,]    7    8    7
[5,]    8   10   10
```

La [figure 5.8](#) illustre l'effet respectif des fonctions d'application `lapply` et `sapply` sur une liste.

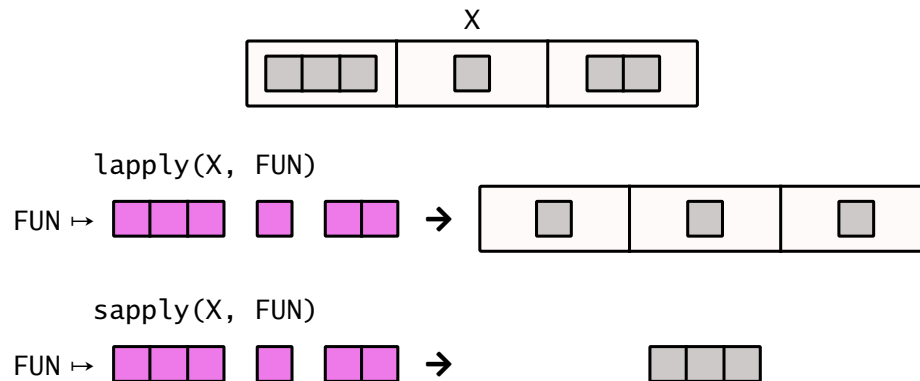


FIG. 5.8 – Représentations des fonctions d'application `lapply` et `sapply` avec une liste



La fonction `sapply` est souvent utile pour rendre vectorielle une fonction `f` qui ne le serait pas autrement. La stratégie consiste généralement à vectoriser les calculs en intégrant à la fonction `f` un appel à `sapply`.



Outre l'arithmétique vectorielle, l'application avec `lapply` et `sapply` constitue le principal moyen d'effectuer des calculs répétitifs dans R. On ne saurait donc trop insister sur l'importance de ces deux fonctions.

Autre membre de la famille des fonctions d'application sur les listes et les vecteurs, `mapply` est une version multidimensionnelle de `sapply`. Sa syntaxe est, pour l'essentiel :

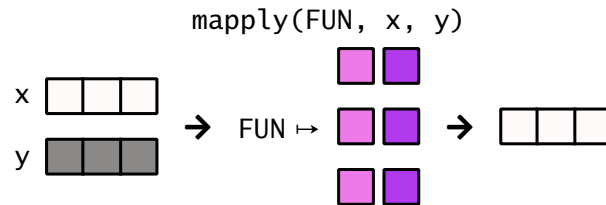
```
mapply(FUN, ...)
```

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.

Par exemple, si `v` et `w` sont des vecteurs, `mapply(FUN, v, w)` retourne sous forme de liste, de vecteur ou de matrice `FUN(v[1], w[1])`, `FUN(v[2], w[2])`, `FUN(v[3], w[3])`, etc. La [figure 5.9](#) offre une représentation de cette fonction d'application.



Étudiez les lignes [407-550](#) du fichier de script `donnees.R` reproduit à la [section 5.12](#).

FIG. 5.9 – Représentation de la fonction d'application `mapply`

5.6 Tableau de données

Le tableau de données (*data frame*³) est une structure de données de R sur laquelle reposent plusieurs procédures statistiques comme la régression linéaire. Un tableau de données est un hybride entre la matrice et la liste : il s'agit d'un tableau rectangulaire de données, mais dont chaque colonne peut être d'un mode différent. Pensons, par exemple, à un tableau avec des noms (mode `character`) dans une colonne et des notes (mode `numeric`) dans une autre.

Formellement, un tableau de données 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). Chaque élément de la liste correspond à une colonne du tableau. La fonction `data.frame` crée un tableau de données, alors que la fonction `as.data.frame` convertit un autre type d'objet (notamment les matrices) en tableau de données.

```
> (x <- data.frame(Nom = c("Pierre", "Jean", "Jacques"),
+                 Age = c(42, 34, 19),
+                 Fumeur = c(TRUE, TRUE, FALSE)))
```

	Nom	Age	Fumeur
1	Pierre	42	TRUE
2	Jean	34	TRUE
3	Jacques	19	FALSE

Le tableau de données peut être indicé à la fois comme une liste (pour l'extraction des colonnes) et comme une matrice (pour l'extraction des lignes ou des colonnes).

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

	Nom	Age	Fumeur
1	Pierre	42	TRUE

3. Il n'y a pas de traduction française universelle pour *data frame*, mais « tableau de données » semble la plus répandue. Ne vous surprenez pas de voir tout bonnement « data frame » dans les messages français de R. Comme c'est souvent le cas, des nuances se perdent dans la traduction puisque ce n'est pas pour rien si la structure de données s'appelle *data frame* et non *data table*. Discuter de la raison est toutefois hors de la portée de cet ouvrage.

```
> x[, 1]
[1] "Pierre" "Jean"   "Jacques"
> x$Age
[1] 42 34 19
```

Il est possible d'ajouter des lignes ou des colonnes à un tableau de données avec les fonctions `rbind` et `cbind`.



Étudiez les quelques exemples de tableaux de données présentés aux lignes 553-588 du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.7 Facteur

Il est très courant en analyse de données de travailler avec des catégories comme la couleur (« rouge », « vert », « bleu »), la taille (« petit », « moyen », « grand ») ou la position géographique (pays, province, état). Bien sûr, un vecteur de mode `character` suffit pour stocker ce genre de données. Cependant, R propose une structure de données dédiée pour les données catégorielles qui permet d'automatiser plusieurs traitements : le facteur.

Un facteur est un vecteur de données correspondant à des catégories. À ce titre, il devrait normalement afficher un fort taux de redondance : inutile de définir des catégories s'il y en a une différente pour chaque donnée, n'est-ce pas ?

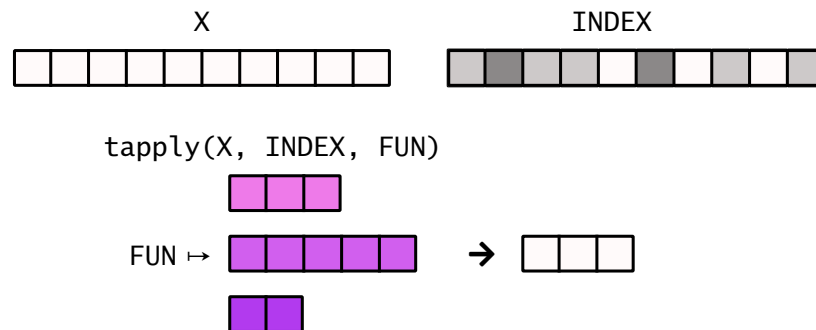
Les fonctions de base pour créer et manipuler des facteurs sont `factor` et `levels`. R représente les données sous forme de chaîne de caractères, mais il les stocke et les traite à l'interne comme un vecteur d'entiers.

```
> (grandeurs <- factor(c("S", "S", "L", "XL",
+                        "M", "M", "L", "L")))
[1] S  S  L  XL M  M  L  L
Levels: L M S XL
> levels(grandeurs)
[1] "L" "M" "S" "XL"
```

Les facteurs sont peu utilisés en tant que structure de données en programmation. En revanche, ils s'avèrent essentiels pour les calculs par groupe de données à l'aide d'une fonction d'application.



Évitez d'effectuer des calculs avec les codes des catégories d'un facteur. Ça peut être très tentant lorsque les codes sont des valeurs numériques ("0" et "1", par exemple). Vous devez d'abord récupérer les véritables valeurs numériques correspondant aux codes à l'aide de la recette recommandée dans la rubrique d'aide de `factor`.

FIG. 5.10 – Représentation de la fonction d'application `tapply`

5.8 Application pour les groupes de données

La fonction `tapply` applique une fonction à chacun des groupes de données définis par les catégories d'un facteur ou d'une combinaison de facteurs. Pour mieux comprendre, étudions la syntaxe de la fonction :

```
tapply(X, INDEX, FUN, ...)
```

- ▶ `X` est un vecteur.
- ▶ `INDEX` est un facteur ou une liste de facteurs, chacun de la même longueur que `X`.
- ▶ `FUN` est la fonction à appliquer à chacun des groupes de données définis par les catégories des facteurs dans `INDEX`.

```
> x <- data.frame(couleur = c("r", "v", "r", "r", "v"),
+               score = c(2, 0, 4, 3, 10))
> tapply(x$score, x$couleur, mean)
 r v 
3 5
```

La mécanique de `tapply` est illustrée à la [figure 5.10](#).



Étudiez les lignes [591-651](#) du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.9 Date

Plusieurs analyses de données impliquent l'enregistrement et le traitement des dates. Or, les dates figurent parmi les données les plus complexes à traiter, et ce, dans n'importe quel langage de programmation ou outil d'analyse. Pensez seulement : il faut tenir compte du nombre variable de jours par mois, des années bis-

sextiles, des secondes intercalaires, des fuseaux horaires et du type de calendrier. Nous pourrions y consacrer un chapitre entier.

R contient dans le système de base les fonctionnalités essentielles pour manipuler les dates : stockage dans divers formats, tri, comparaison, génération de suites, etc. L'infrastructure repose sur trois classes d'objets principales : `Date`, `POSIXct` et `POSIXlt`⁴. Les fonctions `as.Date`, `as.POSIXct` et `as.POSIXlt` permettent de convertir un objet (notamment une chaîne de caractères) vers la classe correspondante.


- La classe `Date` permet de représenter une date au jour près (sans l'heure, donc). À l'interne, la date est enregistrée comme le nombre de jours depuis l'époque⁵ POSIX, le 1^{er} janvier 1970. Les opérateurs suivants sont définis pour les dates : `+`, `-` (avec un nombre de jours), `==`, `!=`, `<`, `<=`, `>` et `>=` (entre deux dates). La fonction `seq` peut également générer des suites d'une date à une autre pour divers intervalles.

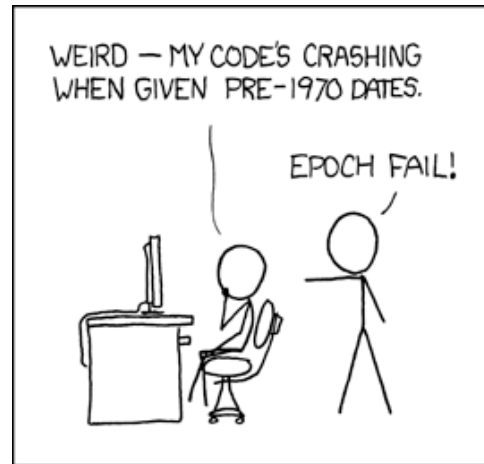
```
> (d <- as.Date("2000-02-29"))
[1] "2000-02-29"
> d + 1
[1] "2000-03-01"
> as.Date("2019-09-01") > d
[1] TRUE
> seq(as.Date("2019-09-01"), as.Date("2020-05-31"),
+     by = "month")
[1] "2019-09-01" "2019-10-01" "2019-11-01" "2019-12-01"
[5] "2020-01-01" "2020-02-01" "2020-03-01" "2020-04-01"
[9] "2020-05-01"
```

- La classe `POSIXct` permet de représenter une date et une heure à la seconde près (incluant le fuseau horaire). C'est habituellement le type de représentation privilégié dans les tableaux de données. À l'interne, la date est enregistrée en nombre de secondes depuis l'époque POSIX. Certains opérateurs sont définis pour les objets `POSIXct`, mais vous devez tenir compte que tous les calculs sont effectués en secondes.

```
> (d <- as.POSIXct("2000-02-29 10:51:27"))
[1] "2000-02-29 10:51:27 EST"
> d + 1
```

4. POSIX est une famille de normes techniques de standardisation des interfaces de programmation des logiciels destinés à fonctionner sur les variantes du système d'exploitation Unix.

5. **époque**, n. f. Moment choisi arbitrairement dans le temps auquel se réfèrent les mesures de position. ([Grand dictionnaire terminologique](#) )



Tiré de [XKCD.com](https://xkcd.com/107/)

```
[1] "2000-02-29 10:51:28 EST"
> d + 3600
[1] "2000-02-29 11:51:27 EST"
> seq(as.POSIXct("2019-09-01 08:30"),
+     as.POSIXct("2019-09-01 13:00"),
+     by = "hour")
[1] "2019-09-01 08:30:00 EDT" "2019-09-01 09:30:00 EDT"
[3] "2019-09-01 10:30:00 EDT" "2019-09-01 11:30:00 EDT"
[5] "2019-09-01 12:30:00 EDT"
```

- Un objet de classe `POSIXlt` contient les mêmes informations qu'un objet de classe `POSIXct`, mais sous forme de liste à raison d'un élément par composante de la date et de l'heure : année (depuis 1900), mois (après le premier, 0-11), jour (1-31), heure (0-23), minute (0-59), seconde (0-61), fuseau horaire, etc. La représentation en format `POSIXlt` est surtout utile pour extraire des informations d'une date. N'hésitez donc pas à convertir un objet `POSIXct` en objet `POSIXlt` pour extraire des composantes d'une date.

```
> (d <- as.POSIXlt("2000-02-29 10:51:27"))
[1] "2000-02-29 10:51:27 EST"
> d$year
[1] 100
> d$mon
[1] 1
> d$hour
```



Les exemples de la [section 5.9](#) utilisent tous la date du 29 février 2000. Que s'est-il passé de spécial ce jour-là? Les années multiples de 100 ne sont pas bissextiles, sauf celles multiples de 400. C'est pourquoi 2000 était bel et bien une année bissextile. La prochaine du type aura donc lieu en... 2400. Certains logiciels de l'époque étaient affligés d'un bogue qui faisait en sorte que le 29 février 2000 n'existait pas dans leur calendrier. C'est beaucoup par espièglerie que le *R Core Team* a justement choisi cette journée pour lancer la **version 1.0.0 de R** [🔗](#).

```
[1] 10
```



Les rubriques d'aide de `Date` et de `DateTimeClasses` contiennent une foule de détails additionnels sur la représentation et le traitement des dates dans R.



Les lignes [654-749](#) du fichier de script `donnees.R` reproduit à la [section 5.12](#) contiennent des exemples additionnels de manipulation des dates.

5.10 Fonctions internes utiles

Cette section présente les fonctions internes couramment utilisées pour les opérations sur les matrices. Les exemples utilisent la matrice suivante.

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

`nrow` nombre de lignes.

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

`ncol` nombre de colonnes.

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

`rowSums` sommes par ligne.

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

colSums sommes par colonne.

```
> colSums(x)
[1] 3 7
```

rowMeans moyennes par ligne.

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

colMeans moyennes par colonne.

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

%*% produit matriciel.

```
> x %*% x
      [,1] [,2]
[1,]    8  20
[2,]    5  13
```

t transposée.

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

det déterminant.

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

solve avec un seul argument (une matrice carrée) : inverse;

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

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

```

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

```

diag avec une matrice en argument : diagonale de la matrice;

```

> diag(x)
[1] 2 3

```

avec un vecteur en argument : matrice diagonale formée avec le vecteur;

```

> diag(c(5, 1))
      [,1] [,2]
[1,]    5    0
[2,]    0    1

```

avec un scalaire p en argument : matrice identité $p \times p$.

```

> diag(2)
      [,1] [,2]
[1,]    1    0
[2,]    0    1

```



Étudiez les lignes 752–773 du fichier de script `donnees.R` reproduit à la [section 5.12](#).

5.11 Produit extérieur

En étirant un peu le concept, la fonction de produit extérieur `outer` peut être considérée comme une fonction d'application. Son utilisation, du moins, est très similaire aux fonctions d'application. Je la présente seulement ici car son résultat est une matrice ou un tableau.

La fonction est très utile pour effectuer plusieurs opérations en une seule expression. Elle calcule le produit extérieur entre deux vecteurs, c'est à dire le résultat de `FUN(X[i], Y[j])` pour toutes les valeurs des indices i et j . La syntaxe de la fonction est la suivante :

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

- ▶ X et Y sont deux objets, normalement des vecteurs ou des matrices.
- ▶ `FUN` est la fonction à appliquer ("*", ou le produit, par défaut) entre chacun des éléments de X et chacun des éléments de Y . Lorsque `FUN` est un opérateur arithmétique du [tableau 4.1](#), il faut placer le symbole entre guillemets : "*", "+", "<=", etc.

La dimension du résultat de `outer` est `c(dim(X), dim(Y))`.

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

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



Visionnez la vidéo sur la **fonction `outer`** [🔗](#) pour mieux comprendre le rôle de cette fonction.



Étudiez les lignes [776-793](#) du fichier de script `donnees.R`.

5.12 Exemples

📍 Fichier d'accompagnement `donnees.R`

```
11 ###
12 ### OBJETS R
13 ###
14
15 ## NOMS D'OBJETS
16
17 ## Quelques exemples de noms valides et invalides.
18 foo <- 5 # valide
19 foo.123 <- 5 # valide
20 foo_123 <- 5 # valide
21 .foo <- 5 # valide
22 123foo <- 5 # invalide; commence par un chiffre
23 .123foo <- 5 # invalide; point suivi d'un chiffre
24
25 ## Liste des objets dans l'espace de travail. Les objets dont le nom
26 ## commence par un point sont cachés, comme à la ligne de commande
27 ## Unix.
28 ls() # l'objet '.foo' n'est pas affiché
29 ls(all.names = TRUE) # objets cachés aussi affichés
30
31 ## R est sensible à la casse
32 foo <- 1
33 Foo
34 F00
35
36 ## MODES ET TYPES DE DONNÉES
```

```
37
38 ## Le mode d'un objet détermine ce qu'il peut contenir. Les vecteurs
39 ## simples ("atomic") contiennent des données d'un seul type.
40 mode(c(1, 4.1, pi))      # nombres réels
41 mode(c(2, 1 + 5i))      # nombres complexes
42 mode(c(TRUE, FALSE, TRUE)) # valeurs booléennes
43 mode("foobar")          # chaînes de caractères
44
45 ## Par défaut, tous les nombres sont représentés en double précision
46 ## dans R. Il n'y a donc pas de différence entre un nombre entier et
47 ## un nombre réel.
48 typeof(486)              # nombre réel en double précision
49 typeof(0.3324)           # idem
50
51 ## Il est possible de définir des «vrais» entiers dans R en faisant
52 ## suivre la valeur immédiatement de la lettre «L», sans espace. Le
53 ## mode de ces valeurs est toujours "numeric", mais leur type (ou
54 ## représentation interne) est "integer".
55 mode(486L)               # nombre...
56 typeof(486L)             # ... entier
57
58 ## La plupart des autres types d'objets sont récursifs. Voici quelques
59 ## autres modes.
60 mode(seq)                # une fonction
61 mode(list(5, "foo", TRUE)) # une liste
62 mode(expression(x <- 5)) # une expression non évaluée
63
64 ## LONGUEUR
65
66 ## La longueur d'un vecteur est égale au nombre d'éléments dans le
67 ## vecteur.
68 (x <- 1:4)
69 length(x)
70
71 ## Une chaîne de caractères ne compte que pour un seul élément.
72 (x <- "foobar")
73 length(x)
74
75 ## Pour obtenir la longueur de la chaîne, il faut utiliser nchar().
76 nchar(x)
77
78 ## Un objet peut néanmoins contenir plusieurs chaînes de caractères.
79 (x <- c("f", "o", "o", "b", "a", "r"))
80 length(x)
81
82 ## La longueur peut être 0, auquel cas on a un objet vide, mais qui
83 ## existe.
84 (x <- numeric(0))        # création du contenant
```

```

85 length(x)           # l'objet 'x' existe...
86 x[1] <- 1           # définition du premier élément
87
88 ## Si un objet n'existe pas au préalable, il est impossible d'affecter
89 ## directement la valeur d'un élément.
90 X[1] <- 1           # impossible, 'X' n'existe pas
91
92 ## VALEURS SPÉCIALES
93
94 ## Donnée manquante. 'NA' est un nom réservé pour représenter une
95 ## donnée manquante.
96 c(65, NA, 72, 88)   # traité comme une valeur
97 NA + 2              # tout calcul avec 'NA' donne NA
98 is.na(c(65, NA))    # test si les données sont NA
99
100 ## Il arrive souvent de vouloir indiquer spécifiquement les données
101 ## manquantes d'un vecteur (pour les éliminer ou pour les remplacer
102 ## par une autre valeur, par exemple).
103 ##
104 ## Pour ce faire, on utilise la fonction 'is.na' et l'indilage par un
105 ## vecteur booléen.
106 x <- c(NA, 12, 55, NA, 4) # vecteur contenant des NA
107 is.na(x)              # positions des données manquantes
108 x[!is.na(x)]          # suppression des données manquantes
109 x[is.na(x)] <- 0; x    # remplacement des NA par des 0
110
111 ## Valeurs infinies et indéterminée. 'Inf', '-Inf' et 'NaN' sont des
112 ## noms réservés.
113 1/0                  # +infini
114 -1/0                 # -infini
115 0/0                  # indétermination
116 x <- c(65, Inf, NaN, 88) # s'utilisent comme des valeurs
117 is.finite(x)          # quels sont les nombres réels?
118 is.nan(x)            # lesquels sont indéterminés?
119
120 ## Valeur "néant". 'NULL' est un nom réservé pour représenter le
121 ## néant, rien.
122 mode(NULL)           # le mode de 'NULL' est NULL
123 length(NULL)          # longueur nulle
124 c(NULL, NULL)         # du néant ne résulte que le néant
125
126 ## ATTRIBUTS
127
128 ## Les objets peuvent être dotés d'un ou plusieurs attributs.
129 data(cars)            # jeu de données intégré
130 attributes(cars)      # liste de tous les attributs
131 attr(cars, "class")   # extraction d'un seul attribut
132

```

```

133 ## L'attribut 'names' conserve les étiquettes des éléments d'un
134 ## vecteur.
135 x <- 1:24 # un vecteur
136 names(x) <- letters[1:24] # attribution d'étiquettes
137 names(x) # extraction des étiquettes
138 x # identification facilitée
139
140 ## À l'instar de quelques autres attributs, l'attribut 'names' est
141 ## spécial en ce qu'une fonction dédiée permet de le définir et d'en
142 ## extraire la valeur. De manière générale, la fonction 'attr' permet
143 ## de définir un attribut.
144 x <- 1:24 # un vecteur
145 attr(x, "names") <- letters[1:24] # définition de l'attribut 'names'
146 x # comme ci-dessus
147
148 ## La fonction 'structure' permet quant à elle de définir la valeur
149 ## d'un objet ainsi que son ou ses attributs en une seule étape.
150 x <- structure(1:24, names = letters[1:24])
151 x
152
153 ###
154 ### MATRICE ET TABLEAU
155 ###
156
157 ## Une matrice est un vecteur avec un attribut 'dim' de longueur 2 et
158 ## une classe implicite "matrix". La manière naturelle de créer une
159 ## matrice est avec la fonction 'matrix'.
160 (x <- matrix(1:12, nrow = 3, ncol = 4))
161 length(x) # longueur du vecteur sous-jacent
162 attributes(x) # objet muni d'un attribut 'dim'
163 dim(x) # deux dimensions
164
165 ## Les matrices sont remplies par colonne par défaut. L'option 'byrow'
166 ## permet de les remplir par ligne, si nécessaire.
167 (x <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE))
168
169 ## Il n'est pas nécessaire de préciser les deux dimensions de la
170 ## matrice s'il est possible d'en déduire une à partir de l'autre et
171 ## de la longueur du vecteur de données. Les expressions ci-dessous
172 ## sont toutes équivalentes.
173 matrix(1:12, nrow = 3, ncol = 4)
174 matrix(1:12, nrow = 3)
175 matrix(1:12, ncol = 4)
176
177 ## À l'inverse, s'il n'y a pas assez de données pour remplir les
178 ## dimensions précisées, les données seront recyclées, comme
179 ## d'habitude.
180 matrix(1, nrow = 3, ncol = 4)

```



```

181 matrix(1:3, nrow = 3, ncol = 4)
182 matrix(1:4, nrow = 3, ncol = 4, byrow = TRUE)
183
184 ## Dans l'indication des matrices et tableaux, l'indice de chaque
185 ## dimension obéit aux règles usuelles d'indication des vecteurs.
186 x[1, 2]                # élément en position (1, 2)
187 x[1, -2]               # 1ère rangée sans 2e colonne
188 x[c(1, 3), ]           # 1ère et 3e rangées
189 x[-1, ]                # supprimer 1ère rangée
190 x[, -2]                # supprimer 2e colonne
191 x[x[, 1] > 2, ]        # lignes avec 1er élément > 2
192
193 ## Indicer la matrice ou le vecteur sous-jacent est équivalent.
194 ## Utiliser l'approche la plus simple selon le contexte.
195 x[1, 3]                # l'élément en position (1, 3)...
196 x[7]                   # ... est le 7e élément du vecteur
197
198 ## Détail additionnel sur l'indication des matrices et tableaux: il est
199 ## aussi possible de les indiquer avec une matrice. Chaque ligne de la
200 ## matrice d'indication fournit alors la position d'un élément à
201 ## sélectionner.
202 ##
203 ## Consulter au besoin la rubrique d'aide de la fonction '[' (ou de
204 ## 'Extract').
205 x[rbind(c(1, 1), c(2, 2))] # éléments x[1, 1] et x[2, 2]
206 x[cbind(1:3, 1:3)]        # éléments x[i, i] («diagonale»)
207 diag(x)                   # idem et plus explicite
208
209 ## Quelques fonctions pour travailler avec les dimensions des
210 ## matrices.
211 nrow(x)                   # nombre de lignes
212 dim(x)[1]                # idem
213 ncol(x)                   # nombre de colonnes
214 dim(x)[2]                # idem
215
216 ## Les matrices et les tableaux étant des vecteurs, ils sont soumis
217 ## aux règles usuelles de l'arithmétique vectorielle. Certaines des
218 ## opérations qui en résultent ne sont pas définies en algèbre
219 ## linéaire usuelle.
220 (x <- matrix(1:4, 2))    # matrice 2 x 2
221 (y <- matrix(3:6, 2))    # autre matrice 2 x 2
222 5 * x                    # multiplication par une constante
223 x + y                    # addition matricielle
224 x * y                    # produit *élément par élément*
225 x %*% y                  # produit matriciel
226 x / y                    # division *élément par élément*
227 x * c(2, 3)              # produit par colonne
228

```

```

229 ## La fonction 'rbind' ("row bind") permet d'«empiler» des matrices
230 ## comptant le même nombre de colonnes.
231 ##
232 ## De manière similaire, la fonction 'cbind' ("column bind") permet de
233 ## concaténer des matrices comptant le même nombre de lignes.
234 ##
235 ## Utilisées avec un seul argument, 'rbind' et 'cbind' créent des
236 ## vecteurs ligne et colonne, respectivement. Ceux-ci sont rarement
237 ## nécessaires.
238 x <- matrix(1:12, 3, 4)    # 'x' est une matrice 3 x 4
239 y <- matrix(1:8, 2, 4)    # 'y' est une matrice 2 x 4
240 z <- matrix(1:6, 3, 2)    # 'z' est une matrice 3 x 2
241 rbind(x, 99)              # ajout d'une ligne à 'x'
242 rbind(x, y)               # fusion verticale de 'x' et 'y'
243 cbind(x, 99)              # ajout d'une colonne à 'x'
244 cbind(x, z)               # concaténation de 'x' et 'z'
245 rbind(x, z)               # dimensions incompatibles
246 cbind(x, y)               # dimensions incompatibles
247 rbind(1:3)                # vecteur ligne
248 cbind(1:3)                # vecteur colonne
249
250 ## Un tableau (array) est un vecteur avec plus de deux dimensions.
251 ## Pour le reste, la manipulation des tableaux est en tous points
252 ## identique à celle des matrices. Ne pas oublier: les tableaux sont
253 ## remplis de la première dimension à la dernière!
254 (x <- array(1:60, 3:5))    # tableau 3 x 4 x 5
255 length(x)                 # longueur du vecteur sous-jacent
256 dim(x)                    # trois dimensions
257 x[1, 3, 2]                # l'élément en position (1, 3, 2)...
258 x[19]                     # ... est le 19e élément du vecteur
259
260 ## Le tableau ci-dessus est un prisme rectangulaire 3 unités de haut,
261 ## 4 de large et 5 de profond. Indicer ce prisme avec un seul indice
262 ## équivaut à en extraire des «tranches», alors qu'utiliser deux
263 ## indices équivaut à en tirer des «carottes» (au sens géologique du
264 ## terme) ou des tubes. Il est laissé en exercice de généraliser à
265 ## plus de dimensions...
266 x                          # les cinq matrices
267 x[, , 1]                   # tranche transversale
268 x[, 1, ]                   # tranche verticale
269 x[1, , ]                   # tranche horizontale
270 x[, 1, 1]                  # carotte de haut en bas
271 x[1, 1, ]                  # carotte de devant à derrière
272 x[1, , 1]                  # carotte de gauche à droite
273 x[1, 1, 1]                 # donnée unique
274
275 ###
276 ### APPLICATION POUR LES MATRICES ET LES TABLEAUX

```

```
277 ###
278
279 ## La fonction 'apply' applique une fonction sur une ou plusieurs
280 ## dimensions d'une matrice ou d'un tableau.
281 ##
282 ## Création d'une matrice et d'un tableau à trois dimensions pour les
283 ## exemples.
284 m <- matrix(sample(1:100, 20), nrow = 4, ncol = 5)
285 a <- array(sample(1:100, 60), dim = 3:5)
286
287 ## Les fonctions 'rowSums', 'colSums', 'rowMeans' et 'colMeans' sont
288 ## des raccourcis pour des utilisations fréquentes de 'apply'.
289 apply(m, 1, sum)           # sommes par ligne
290 rowSums(m)                # idem, plus lisible
291 apply(m, 2, mean)         # moyennes par colonne
292 colMeans(m)               # idem, plus lisible
293
294 ## Puisqu'il n'existe pas de fonctions comme 'rowMax' ou 'colProds',
295 ## il faut utiliser 'apply'.
296 apply(m, 1, max)          # maximums par ligne
297 apply(m, 2, prod)         # produits par colonne
298
299 ## L'argument '...' de 'apply' permet de passer des arguments à la
300 ## fonction FUN.
301 f <- function(x, y) x + 2 * y # fonction à deux arguments
302 apply(m, 1, f, y = 2)     # argument 'y' passé dans '...'
303
304 ## Lorsque 'apply' est utilisée sur un tableau, son résultat est de
305 ## dimensions dim(X)[MARGIN], d'où le truc mnémotechnique donné dans
306 ## le texte du chapitre.
307 apply(a, c(2, 3), sum)    # le résultat est une matrice
308 apply(a, 1, prod)        # le résultat est un vecteur
309
310 ## L'utilisation de 'apply' avec les tableaux peut rapidement devenir
311 ## confondante si l'on ne visualise pas les calculs qui sont réalisés.
312 ##
313 ## Reprenons ici les exemples du chapitre en montrant comment calculer
314 ## le premier élément de chaque utilisation de 'apply'.
315 ##
316 ## Au besoin, réviser l'indichage des tableaux au chapitre 3.
317 (x <- array(sample(0:10, 24, rep = TRUE), c(3, 4, 2)))
318 apply(x, 1, sum)          # sommes des 3 tranches horizontales
319 sum(x[1, , ])            # équivalent pour la première somme
320
321 apply(x, 2, sum)          # sommes des 4 tranches verticales
322 sum(x[, 1, ])            # équivalent pour la première somme
323
324 apply(x, 3, sum)          # sommes des 2 tranches transversales
```

```

325 sum(x[, , 1])          # équivalent pour la première somme
326
327 apply(x, c(1, 2), sum) # sommes des 12 carottes horizontales
328 sum(x[1, 1, ])         # équivalent pour la première somme
329
330 apply(x, c(2, 3), sum) # sommes des 8 carottes verticales
331 sum(x[, 1, 1])         # équivalent pour la première somme
332
333 apply(x, c(1, 3), sum) # sommes des 6 carottes transversales
334 sum(x[1, , 1])         # équivalent pour la première somme
335
336 ###
337 ### LISTE
338 ###
339
340 ## La liste est l'objet le plus général en R. C'est un objet récursif
341 ## qui peut contenir des objets de n'importe quel mode (y compris la
342 ## liste) et de n'importe quelle longueur.
343 (x <- list(joueur = c("V", "C", "C", "M", "A"),
344           score = c(10, 12, 11, 8, 15),
345           expert = c(FALSE, TRUE, FALSE, TRUE, TRUE),
346           niveau = 2))
347 is.vector(x)           # liste est un vecteur...
348 is.recursive(x)        # ... récursif...
349 length(x)              # ... de quatre éléments...
350 mode(x)                # ... de mode "list"
351
352 ## Comme tout autre vecteur, une liste peut être concaténée avec un
353 ## autre vecteur avec la fonction 'c'.
354 y <- list(TRUE, 1:5)    # liste de deux éléments
355 c(x, y)                 # liste de six éléments
356
357 ## Pour initialiser une liste d'une longueur donnée, on utilise la
358 ## fonction 'vector'.
359 vector("list", 5)
360
361 ## Les crochets simples [ ] permettent d'extraire un ou plusieurs
362 ## éléments d'une liste. Le résultat est toujours une liste, même si
363 ## l'on extrait un seul élément.
364 x[c(1, 2)]             # deux premiers éléments
365 x[1]                   # premier élément: une liste
366
367 ## Lorsque l'on veut extraire un, et un seul, élément d'une liste et
368 ## obtenir l'objet lui-même (et non une liste contenant l'objet), il
369 ## faut utiliser les crochets doubles [[ ]].
370 x[[1]]                 # comparer avec ci-dessus
371
372 ## Jolie fonctionnalité: les crochets doubles permettent d'indicer

```

```

373 ## récursivement la liste, c'est-à-dire d'extraire un objet de la
374 ## liste, puis un élément de l'objet, et ainsi de suite.
375 x[[1]][2]          # 2e élément du 1er élément
376 x[[c(1, 2)]]      # idem, par indexage récursif
377
378 ## Les éléments d'une liste étant généralement nommés (c'est une bonne
379 ## habitude à prendre!), il est souvent plus simple et, surtout, plus
380 ## sûr d'extraire les éléments d'une liste par leur étiquette avec
381 ## l'opérateur $.
382 x$joueur           # équivalent à x[[1]]
383 x$joueur[2]        # équivalent à x[[c(1, 2)]]
384 x[["expert"]]      # aussi valide, mais peu usité
385 x$level <- 1       # aussi pour l'affectation
386
387 ## Une liste peut contenir n'importe quoi...
388 x[[5]] <- matrix(1, 2, 2) # ... une matrice...
389 x[[6]] <- list(0:5, TRUE) # ... une autre liste...
390 x[[7]] <- seq           # ... même le code d'une fonction!
391 x                      # eh ben!
392 x[[c(6, 1, 3)]]       # de quel élément s'agit-il?
393
394 ## Il est possible de supprimer un élément d'une liste en lui
395 ## affectant la valeur 'NULL'.
396 x[[7]] <- NULL; length(x) # suppression du 7e élément
397
398 ## Il est parfois utile de convertir une liste en un simple vecteur.
399 ## Les éléments de la liste sont alors «déroulés», y compris la
400 ## matrice en position 5 dans notre exemple (qui n'est rien d'autre
401 ## qu'un vecteur, on s'en souviendra).
402 unlist(x)           # remarquer la conversion
403 unlist(x, recursive = FALSE) # ne pas appliquer aux sous-listes
404 unlist(x, use.names = FALSE) # éliminer les étiquettes
405
406 ###
407 ### APPLICATION POUR LES LISTES ET LES VECTEURS
408 ###
409
410 ## FONCTIONS 'lapply' ET 'sapply'
411
412 ## La fonction 'lapply' applique une fonction à tous les éléments d'un
413 ## vecteur ou d'une liste et retourne une liste, peu importe les
414 ## dimensions des résultats.
415 ##
416 ## La fonction 'sapply' retourne un vecteur ou une matrice, si
417 ## possible.
418 ##
419 ## Somme «interne» des éléments d'une liste.
420 (x <- list(1:10, c(-2, 5, 6), matrix(3, 4, 5)))

```

```

421 sum(x)                                # erreur
422 lapply(x, sum)                        # sommes internes (liste)
423 sapply(x, sum)                        # sommes internes (vecteur)
424
425 ## Création de la suite 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, ..., 1, 2, ...,
426 ## 9, 10.
427 lapply(1:10, seq)                    # résultat sous forme de liste
428 unlist(lapply(1:10, seq))             # résultat sous forme de vecteur
429
430 ## Soit une fonction calculant la moyenne pondérée d'un vecteur. Cette
431 ## fonction prend en argument une liste de deux éléments: 'donnees' et
432 ## 'poids'.
433 fun <- function(x)
434   sum(x$donnees * x$poids)/sum(x$poids)
435
436 ## Nous pouvons maintenant calculer la moyenne pondérée de plusieurs
437 ## ensembles de données réunis dans une liste itérée.
438 (x <- list(list(donnees = 1:7,
439               poids = (5:11)/56),
440           list(donnees = sample(1:100, 12),
441               poids = 1:12),
442           list(donnees = c(1, 4, 0, 2, 2),
443               poids = c(12, 3, 17, 6, 2))))
444 sapply(x, fun)                        # aucune boucle explicite!
445
446 ## EXEMPLES ADDITIONNELS SUR L'UTILISATION DE L'ARGUMENT '...' AVEC
447 ## 'lapply' ET 'sapply'
448
449 ## Aux fins des exemples ci-dessous, créons d'abord une liste formée
450 ## de nombres aléatoires.
451 ##
452 ## L'expression ci-dessous fait usage de l'argument '...' de 'lapply'.
453 ## Pouvez-vous la décoder? Nous y reviendrons plus loin, ce qui compte
454 ## pour le moment c'est simplement de l'exécuter.
455 x <- lapply(c(8, 12, 10, 9), sample, x = 1:10, replace = TRUE)
456
457 ## Soit maintenant une fonction qui calcule la moyenne arithmétique
458 ## des données d'un vecteur 'x' supérieures à une valeur 'y'.
459 ##
460 ## Vous remarquerez que cette fonction n'est pas vectorielle pour 'y',
461 ## c'est-à-dire qu'elle n'est valide que lorsque 'y' est un vecteur de
462 ## longueur 1.
463 fun <- function(x, y) mean(x[x > y])
464
465 ## Pour effectuer ce calcul sur chaque élément de la liste 'x', nous
466 ## pouvons utiliser 'sapply' plutôt que 'lapply', car chaque résultat
467 ## est de longueur 1.
468 ##

```

```
469 ## Cependant, il faut passer la valeur de 'y' à la fonction 'fun'.
470 ## C'est là qu'entre en jeu l'argument '...' de 'sapply'.
471 sapply(x, fun, 7)           # moyennes des données > 7
472
473 ## Les fonctions 'lapply' et 'sapply' passent tour à tour les éléments
474 ## de leur premier argument comme *premier* argument à la fonction,
475 ## sans le nommer explicitement. L'expression ci-dessus est donc
476 ## équivalente à
477 ##
478 ##   c(fun(x[[1]], 7), ..., fun(x[[4]], 7)
479 ##
480 ## Que se passe-t-il si l'on souhaite passer les valeurs à un argument
481 ## de la fonction autre que le premier? Par exemple, supposons que
482 ## l'ordre des arguments de la fonction 'fun' ci-dessus est inversé.
483 fun <- function(y, x) mean(x[x > y])
484
485 ## Les règles de pairage des arguments des fonctions en R font en
486 ## sorte que lorsque les arguments sont nommés dans l'appel de
487 ## fonction, leur ordre n'a pas d'importance. Par conséquent, un appel
488 ## de la forme
489 ##
490 ##   fun(x, y = 7)
491 ##
492 ## est tout à fait équivalent à fun(7, x). Pour effectuer les calculs
493 ##
494 ##   c(fun(x[[1]], y = 7), ..., fun(x[[4]], y = 7)
495 ##
496 ## avec la liste définie plus haut, il s'agit de nommer l'argument 'y'
497 ## dans '...' de 'sapply'.
498 sapply(x, fun, y = 7)
499
500 ## Décodons maintenant l'expression
501 ##
502 ##   lapply(c(8, 12, 10, 9), sample, x = 1:10, replace = TRUE)
503 ##
504 ## qui a servi à créer la liste. La définition de la fonction 'sample'
505 ## est la suivante:
506 ##
507 ##   sample(x, size, replace = FALSE, prob = NULL)
508 ##
509 ## L'appel à 'lapply' est équivalent à
510 ##
511 ##   list(sample(8, x = 1:10, replace = TRUE),
512 ##       ...,
513 ##       sample(9, x = 1:10, replace = TRUE))
514 ##
515 ## Toujours selon les règles d'appariement des arguments, vous
516 ## constaterez que les valeurs 8, 12, 10, 9 seront attribuées à
```

```
517 ## l'argument 'size', soit la taille de l'échantillon.
518 ##
519 ## L'expression crée donc une liste comprenant quatre échantillons
520 ## aléatoires de tailles différentes des nombres de 1 à 10 pigés avec
521 ## remise.
522 ##
523 ## Une expression équivalente, quoique moins élégante, aurait recours
524 ## à une fonction anonyme pour replacer les arguments de 'sample' dans
525 ## l'ordre voulu.
526 lapply(c(8, 12, 10, 9),
527        function(x) sample(1:10, x, replace = TRUE))
528
529 ## La fonction 'sapply' est aussi très utile pour vectoriser une
530 ## fonction qui n'est pas vectorielle. Supposons que l'on veut
531 ## généraliser la fonction 'fun' pour qu'elle accepte un vecteur de
532 ## seuils 'y'.
533 fun <- function(x, y)
534     sapply(y, function(y) mean(x[x > y]))
535
536 ## Utilisation sur la liste 'x' avec trois seuils.
537 sapply(x, fun, y = c(3, 5, 7))
538
539 ## FONCTION 'mapply'
540
541 ## Application de la fonction 'fun' sur les échantillons de la liste
542 ## 'x' avec un seuil différent pour chacun.
543 mapply(fun, x, c(3, 5, 7, 7))
544
545 ## Création de quatre échantillons aléatoires de taille 12.
546 x <- lapply(rep(12, 4), sample, x = 1:100)
547
548 ## Moyennes tronquées à 0, 10, 20 et 30 %, respectivement, de ces
549 ## quatre échantillons aléatoires.
550 mapply(mean, x, 0:3/10)
551
552 ###
553 ### TABLEAU DE DONNÉES
554 ###
555
556 ## Un tableau de données (data frame) est une liste dont les éléments
557 ## sont tous de la même longueur. Il comporte un attribut 'dim', ce
558 ## qui fait qu'il est représenté comme une matrice. Cependant, les
559 ## colonnes peuvent être de modes différents.
560 data.frame(Nom = c("Pierre", "Jean", "Jacques"),
561            Age = c(42, 34, 19),
562            Fumeur = c(TRUE, TRUE, FALSE))
563
564 ## R est livré avec plusieurs jeux de données, la plupart sous forme
```



```

565 ## de tableaux de données.
566 data()                # liste complète
567
568 ## Nous allons illustrer certaines manipulations des tableaux de
569 ## données avec le jeu de données 'USArrests'.
570 USArrests              # jeu de données
571
572 ## Analyse succincte de l'objet.
573 mode(USArrests)        # une liste...
574 length(USArrests)      # ... de quatre éléments...
575 class(USArrests)       # ... de classe 'data.frame'
576 dim(USArrests)         # dimensions implicites
577 names(USArrests)       # titres des colonnes
578 row.names(USArrests)   # titres des lignes
579 USArrests[, 1]         # première colonne
580 USArrests$Murder       # idem, plus simple
581 USArrests[1, ]        # première ligne
582
583 ## La fonction 'subset' permet d'extraire des lignes et des colonnes
584 ## d'un tableau de données de manière très intuitive.
585 ##
586 ## Par exemple, nous pouvons extraire ainsi le nombre d'assauts dans
587 ## les états comptant un taux de meurtre supérieur à 10.
588 subset(USArrests, Murder > 10, select = Assault)
589
590 ###
591 ### FACTEUR
592 ###
593
594 ## Les facteurs jouent un rôle important en analyse de données,
595 ## surtout pour classer des données en diverses catégories. Les
596 ## données d'un facteur devraient normalement afficher un fort taux de
597 ## redondance.
598 ##
599 ## Reprenons l'exemple du chapitre.
600 (grandeurs <-
601   factor(c("S", "S", "L", "XL", "M", "M", "L", "L")))
602 levels(grandeurs)      # catégories
603 as.integer(grandeurs)  # représentation interne
604
605 ## Dans le présent exemple, nous pourrions souhaiter que R reconnaisse
606 ## le fait que S < M < L < XL. C'est possible avec les facteurs
607 ## *ordonnés*.
608 factor(c("S", "S", "L", "XL", "M", "M", "L", "L"),
609       levels = c("S", "M", "L", "XL"),
610       ordered = TRUE)
611
612 ## Il arrive que les codes des catégories soient des valeurs

```

```

613 ## numériques, surtout lorsque les données proviennent de systèmes
614 ## externes (autres que R) qui ne reconnaissent pas les facteurs ou
615 ## les données booléennes.
616 ##
617 ## Par exemple, les catégories «non membre» et «membre» représentées
618 ## dans un système par des 0 et des 1 deviennent, dans R, un facteur
619 ## avec les catégories "0" et "1".
620 (is_member <- factor(c(1, 0, 1, 1, 0, 0, 1, 0, 1, 1)))
621 levels(is_member)
622
623 ## Les valeurs numériques sous-jacentes n'ont toutefois aucun lien
624 ## avec les codes des catégories.
625 as.integer(is_member)
626
627 ## La manière sécuritaire de récupérer les valeurs numériques des
628 ## codes est fournie dans ?factor. Il est ensuite possible d'effectuer
629 ## des calculs avec les valeurs numériques comme à l'accoutumée.
630 (x <- as.numeric(levels(is_member))[is_member])
631 sum(x) # nombre de membres
632 mean(x) # proportion de membres
633
634 ###
635 ### APPLICATION POUR LES GROUPES DE DONNÉES
636 ###
637
638 ## Le jeu de données 'airquality' livré avec R contient les mesures
639 ## quotidiennes de la qualité de l'air à New York entre mai et
640 ## septembre 1973.
641 ?airquality # rubrique d'aide du jeu de données
642
643 ## La colonne 'Temp' contient la température du jour et la colonne
644 ## 'Month', le mois (sous forme d'entier de 5 à 9).
645 ##
646 ## La fonction 'tapply' permet de calculer facilement la température
647 ## moyenne par mois.
648 tapply(airquality$Temp, airquality$Month, mean)
649
650 ## Équivalent (sauf pour la présentation des résultats).
651 by(airquality$Temp, airquality$Month, mean)
652
653 ###
654 ### DATE
655 ###
656
657 ## Votre premier réflexe pour représenter une date pourrait être
658 ## d'utiliser simplement une chaîne de caractères. Ça suffit si les
659 ## dates sont à intervalles égaux et pour identifier les points sur un
660 ## graphique. Dès lors que vous devez tenir compte de l'écart entre

```

```
661 ## les dates ou effectuer des calculs avec les dates, il vous faut une
662 ## solution plus robuste.
663 ##
664 ## R utilise la représentation standard des systèmes Unix consignée
665 ## dans la norme POSIX. En simplifiant, le temps y est compté (en
666 ## jours pour une date seule, en secondes pour une date et une heure)
667 ## à partir du 1er janvier 1970.
668 ##
669 ## Une date n'est donc rien d'autre qu'un entier à l'interne pour R.
670 ## C'est ce qui permet d'effectuer facilement des calculs et des
671 ## comparaisons. Cependant, l'objet est muni d'un attribut "class" qui
672 ## fait en sorte de modifier l'affichage et l'interaction avec
673 ## certaines fonctions.
674 ##
675 ## Pour les dates seules, la classe de base est "Date".
676 d <- "2000-02-29"          # chaîne de caractères
677 d + 1                     # opération invalide
678 d <- as.Date(d)           # conversion en date
679 d + 1                     # opération valide; jour suivant
680 d - 1                     # jour précédent
681 as.numeric(d)             # nombre de jours depuis 1970-01-01
682 d - as.Date("1970-01-01") # vérification
683 (auj <- Sys.Date())       # date du jour
684 auj >= d                  # après le 2000-02-29?
685
686 ## Chose particulièrement utile, la fonction 'seq' est munie d'une
687 ## méthode pour la classe "Date", ce qui permet de générer des suites
688 ## de dates.
689 ##
690 ## Exemple: générer les dates des 10 prochaines semaines à partir
691 ## d'aujourd'hui.
692 (dixsem <- seq(Sys.Date(), length.out = 10, by = "1 week"))
693 ?seq.Date                 # voir les autres possibilités
694
695 ## Attention: dans la génération de suites par mois, le jour du mois
696 ## ne change pas sauf si cela résulte en un jour invalide. Cela arrive
697 ## seulement si la date de départ est le 31 du mois. Dans ce cas,
698 ## c'est le jour du mois suivant qui est utilisé.
699 seq(as.Date("2042-05-31"), length.out = 4, by = "+1 month")
700
701 ## Un truc pour se prémunir contre cet éventuel problème consiste à
702 ## ajouter un jour à la date de départ avant de générer la suite, puis
703 ## à soustraire un jour ensuite.
704 ##
705 ## Comme ça, si la date de départ est un 31, on génère la suite à
706 ## partir du 1er du mois suivant pour revenir ensuite en arrière; si
707 ## la date de départ est n'importe quel autre jour dans le mois, ça ne
708 ## change rien.
```

```

709 seq(as.Date("2042-05-31") + 1, length.out = 4, by = "+1 month") - 1
710
711 ## Quelques fonctions d'extraction utiles.
712 weekdays(dixsem)           # jours de la semaine
713 months(dixsem)             # mois
714 quarters(dixsem)           # trimestres
715
716 ## Pour enregistrer non seulement une date, mais aussi une heure, vous
717 ## devez utiliser les représentations POSIXct et POSIXlt. Il s'agit de
718 ## classes d'objets très puissantes qui permettent d'enregistrer une
719 ## heure jusqu'à la fraction de seconde, le fuseau horaire, s'il
720 ## s'agit de l'heure d'été ou non, etc.
721 ?DateTimeClasses           # *tous* les détails
722
723 ## Les objets POSIXct et POSIXlt représentent un nombre de secondes
724 ## depuis le 1er janvier 1970.
725 (auj <- Sys.time())         # date et heure courante
726 auj + 3600                  # une heure plus tard
727 auj - 24 * 3600             # hier, même heure
728 auj - as.POSIXct("2000-02-29") # écart entre deux dates
729 difftime(auj, as.POSIXct("2000-02-29")) # idem
730
731 ## La classe POSIXlt est une représentation sous forme de liste des
732 ## informations contenues dans un objet POSIXct. Elle est surtout
733 ## utile pour extraire facilement des informations d'une date sous
734 ## forme numérique, ce qui permet ensuite d'effectuer des calculs.
735 ##
736 ## N'hésitez pas à convertir d'un type vers un autre.
737 class(auj)                  # objet POSIXct
738 auj <- as.POSIXlt(auj)      # conversion en POSIXlt
739 unclass(auj)                # c'est une liste
740 auj$hour                    # extraction de l'heure
741 (m <- auj$mon)              # nombre de mois après janvier
742 11 - m                      # nombre de mois fin d'année
743 (y <- auj$year)             # nombre d'année depuis 1900 (!)
744 y - (2000 - 1900)           # nombre d'années depuis 2000
745
746 ## La conversion de la classe POSIXct ou POSIXlt vers la classe Date
747 ## laisse tomber l'heure.
748 d <- as.POSIXct("2000-02-29 10:51:27") # objet POSIXct
749 as.Date(d)                  # conversion en date seule
750
751 ###
752 #### FONCTIONS INTERNES UTILES
753 ###
754
755 ## On se donne un vecteur de 16 éléments.
756 (A <- sample(1:10, 16, replace = TRUE))

```

```

757
758 ## Opérations sur les matrices.
759 dim(A) <- c(4, 4)      # conversion en une matrice 4 x 4
760 b <- c(10, 5, 3, 1)   # vecteur quelconque
761 A                     # matrice 'A'
762 t(A)                  # transposée
763 solve(A)              # inverse
764 solve(A, b)           # solution de Ax = b
765 A %% solve(A, b)      # vérification de la réponse
766 diag(A)               # extraction de la diagonale de 'A'
767 diag(b)               # matrice diagonale formée avec 'b'
768 diag(4)               # matrice identité 4 x 4
769 (A <- cbind(A, b))     # matrice 4 x 5
770 nrow(A)               # nombre de lignes de 'A'
771 ncol(A)               # nombre de colonnes de 'A'
772 rowSums(A)            # sommes par ligne
773 colSums(A)            # sommes par colonne
774
775 ###
776 ### FONCTION 'outer'
777 ###
778
779 ## La fonction 'outer' applique une fonction (le produit par défaut,
780 ## d'où le nom de la fonction, dérivé de «produit extérieur») à toutes
781 ## les combinaisons des éléments de ses deux premiers arguments.
782 x <- c(1, 2, 4, 7, 10, 12)
783 y <- c(2, 3, 6, 7, 9, 11)
784 outer(x, y)           # produit extérieur
785 x %o% y               # équivalent plus court
786
787 ## Pour effectuer un calcul autre que le produit, on spécifie la
788 ## fonction à appliquer en troisième argument. Si la fonction est un
789 ## des opérateurs arithmétiques de base, il faut placer le symbole
790 ## entre guillemets " ".
791 outer(x, y, "+")      # «somme extérieure»
792 outer(x, y, "<=")      # toutes les comparaisons possibles
793 outer(x, y, function(x, y) x + 2 * y) # fonction quelconque

```

5.13 Exercices

5.1 Évaluer les expressions suivantes comme le ferait l'interpréteur R.

- a) `length(c("a", "abc", "ab"))`
- b) `x <- c(-1, -2, -3); length(x)`
- c) `length(c(c(1, 3, 6, 7), NULL))`
- d) `mode(c(45.44, pi, TRUE))`

- e) `mode(c(5, "5", "cinq"))`
- f) `sum(is.na(c(4, NA, 3, 8, NA)))`
- g) `matrix(1:6, 2)[, 2]`
- h) `as.POSIXlt(as.Date("2001-05-09"))$mday`
- i) `list(NA, 1:10, list(NA, 1:10))`
- j) `x <- data.frame(A = 1:3, B = 5:7); x$A`

5.2 Soit `x` une matrice 7×10 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.
- 5.3** 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 $I \times J \times K$.
- 5.4** Soit `m` une matrice 10×7 quelconque. Écrire des expressions R permettant d'effectuer les tâches demandées ci-dessous.
- a) Calculer la somme des éléments de chacune des lignes de la matrice.
 - b) Calculer la moyenne des éléments de chacune 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 de la matrice.
 - d) Extraire toutes les lignes de la matrice dont la moyenne des éléments est supérieure à 7.
- 5.5** Écrire une fonction `sort.matrix` servant à ordonner les lignes d'une matrice selon les valeurs des éléments de la première colonne.

Par défaut, la fonction trie en ordre croissant. Elle peut également trier en ordre décroissant si son argument `decreasing` est `TRUE`.

```
> m
      [,1] [,2] [,3] [,4]
[1,]    8    7    5    8
[2,]    3    3   10    6
[3,]    3    6    3    2
> sort.matrix(m)
```

```

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

> sort.matrix(m, decreasing = TRUE)

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

```

5.6 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.

5.7 La valeur actuelle d'une série de n paiements de 1 versés à la fin des années $1, 2, \dots, n$ à un taux d'intérêt i est

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

où $v = (1 + i)^{-1}$. À partir de vecteurs n et i contenant les durées et les taux d'intérêt, calculer en une seule expression —et sans boucle! — un tableau des valeurs actuelles de séries de $n = 1, 2, \dots, 10$ paiements à chacun des taux d'intérêt $i = 0,05, 0,06, \dots, 0,10$.

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

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

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

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

- 5.9** Nous pouvons généraliser aux matrices et aux tableaux de données la définition de moyenne pondérée de l'exercice 5.8.

Dans le cas de matrices $n \times p$ d'observations X_{ij} et de poids w_{ij} correspondants, on définit les moyennes pondérées suivantes :

$$X_{iw} = \sum_{j=1}^p \frac{w_{ij}}{w_{i\Sigma}} X_{ij}, \quad w_{i\Sigma} = \sum_{j=1}^p w_{ij}$$

$$X_{wj} = \sum_{i=1}^n \frac{w_{ij}}{w_{\Sigma j}} X_{ij}, \quad w_{\Sigma j} = \sum_{i=1}^n w_{ij}$$

et

$$X_{ww} = \sum_{i=1}^n \sum_{j=1}^p \frac{w_{ij}}{w_{\Sigma\Sigma}} X_{ij}, \quad w_{\Sigma\Sigma} = \sum_{i=1}^n \sum_{j=1}^p w_{ij}.$$

En suivant la même logique, il est possible de définir des moyennes pondérées pour des tableaux de données X_{ijk} de dimensions $n \times p \times r$ et de poids w_{ijk} correspondants.

Écrire des expressions R pour calculer, sans boucle, les moyennes pondérées suivantes.

- X_{iw} en supposant une matrice de données $n \times p$.
- X_{wj} en supposant une matrice de données $n \times p$.
- X_{ww} en supposant une matrice de données $n \times p$.
- X_{ijw} en supposant un tableau de données $n \times p \times r$.
- X_{iww} en supposant un tableau de données $n \times p \times r$.
- X_{wjwt} en supposant un tableau de données $n \times p \times r$.
- X_{www} en supposant un tableau de données $n \times p \times r$.

5.10 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, ..., 0, 1, 2, 3, ..., 10.
- b) 10, 9, 8, ..., 2, 1, 10, 9, 8, ..., 3, 2, ..., 10, 9, 10.
- c) 10, 9, 8, ..., 2, 1, 9, 8, ..., 2, 1, ..., 2, 1, 1.

5.11 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 $\alpha = \text{shape}$ et $\lambda = \text{scale}$:

```
> 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) Nommer les éléments de la liste créée en a) `sample1`, ..., `sample5` à l'aide de la fonction `paste` présentée à la [section 4.7.8](#).
- 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) 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.

5.12 La fonction `emr1` de l'[exercice 4.5](#) ne permettait de calculer l'espérance résiduelle empirique d'un vecteur x que pour un seul seuil d . Composer une version qui permet d'effectuer le calcul pour un vecteur de seuils d .

6 Bonnes pratiques de la programmation

Objectifs du chapitre

- ▶ Adopter des bonnes pratiques de programmation en contexte de travail collaboratif.
- ▶ Respecter les normes de programmation reconnues en matière de style, de présentation du code et de documentation.
- ▶ Interpréter les résultats de tests unitaires.

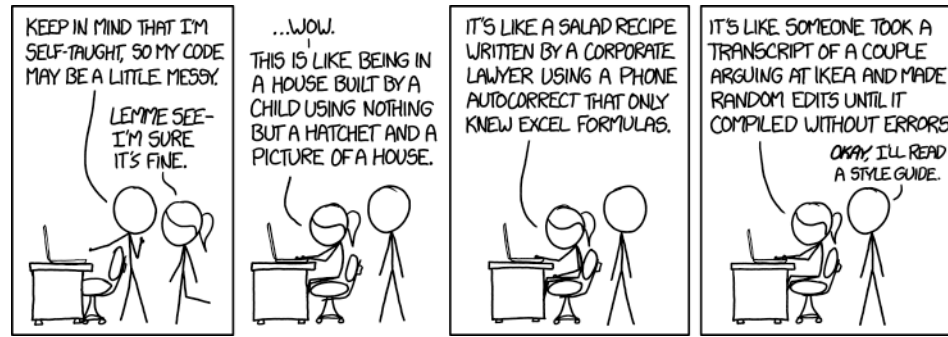
De manière générale, le développement et la maintenance de code informatique reposent sur la contribution de plusieurs personnes. En effet, il est plutôt rare, dans le milieu professionnel, d'être appelé à concevoir un programme informatique à partir d'une page blanche et en complète autarcie, c'est-à-dire sans que quiconque n'ait à interagir avec le code à un stade ou à un autre. Une grande part du travail de programmation consiste à corriger, à mettre à jour ou à améliorer du code existant. Dans ce contexte, l'adhésion à un certain nombre de normes et de bonnes pratiques permet de faciliter le travail de tous les intervenants et de réduire les risques d'erreurs. Ce chapitre présente quelques-unes de ces bonnes pratiques à adopter en matière de style de programmation, de présentation du code et de documentation.



Je me suis efforcé de respecter les bonnes pratiques présentées dans ce chapitre dans les fichiers de script fournis avec l'ouvrage. Prêtez donc attention à la manière dont le code qui vous est fourni est construit et présenté.

6.1 Style

Il en va du code informatique comme de la prose : si le style peut varier d'un auteur à l'autre, l'œuvre doit toujours être à la fois agréable à lire et facile à comprendre. Bref, les meilleurs programmeurs préfèrent la lisibilité de leur code aux



Tiré de XKCD.com

effets de style qui auraient pour seul mérite de faire étalage de leur maîtrise du langage.

Tout programmeur devrait constamment garder en tête les trois objectifs suivants en effectuant son travail : simplicité, clarté, concision. Ces objectifs entrent souvent en conflit les uns avec les autres ! Tout l'art de la bonne programmation consiste donc à trouver un juste équilibre entre les trois pôles.

Kernighan et Pike (1999), Oualline (1997, 2003), Kernighan et Plauger (1978) proposent d'excellents chapitres sur le style en programmation ; Hunt et Thomas (1999) offrent quasiment un traité sur le sujet ; Martin (2009) est la bible des ingénieurs logiciels¹. Je ne saurais être aussi exhaustif que ces auteurs établis. Néanmoins, je vous incite à porter une attention particulière aux quelques points de style livrés en vrac ci-dessous.

- Utilisez des noms de variables significatifs. Ne soyez pas ce collègue qui nomme les variables d'un programme `x`, `xx` et `xxx` (cas vécu). Attention, toutefois, de ne pas pousser le concept trop loin. Ici comme ailleurs, la clarté peut provenir de la concision ; la terminologie²

```
| xlen <- length(x)
```

est aussi claire que

```
| length_of_x <- length(x)
```

et bien plus simple à utiliser au fil d'un programme.

Certains noms d'objets sans réelle signification sont tellement usuels qu'il devient contreproductif de leur préférer des versions plus explicites. Pensons, ici, à `x` comme premier argument d'une fonction R ou à `i`, `j` et `k` comme compteurs dans les boucles itératives.

1. Les disciples appellent d'ailleurs affectueusement l'auteur *Uncle Bob*.

2. Vous remarquerez que je préfère utiliser l'anglais pour les noms d'objets, question d'uniformité avec les identificateurs du langage. Chose certaine, évitez à tout prix les accents dans les noms d'objets.

Quant à la composition des noms d'objets formés de plusieurs mots, divers styles s'affrontent, notamment :

```
variable.name
variable_name
variableName
VariableName
```

Assurez-vous simplement de suivre le standard en vigueur dans votre équipe de travail, et, par-dessus tout, soyez constant. Ma préférence, qui concorde avec une grande partie du code source de R, va aux noms d'objets courts et entièrement en minuscules.

- Dès qu'elles sont disponibles, utilisez les fonctions internes de R au lieu de reprogrammer certaines procédures. Non seulement bénéficierez-vous de l'optimisation des fonctions internes, mais votre code gagnera également en lisibilité. Comparez

```
| sum(x)/length(x)
```

à

```
| mean(x)
```

- Évitez de vous répéter. Évitez de vous répéter. Si vous utilisez une expression ou un groupe d'expressions plus d'une fois dans un programme et que vous devez un jour y changer quelque chose, il vous faudra penser à effectuer le changement partout. Tôt ou tard, vous allez oublier un cas et introduire un bogue ; c'est une règle de l'univers, ne pensez pas y échapper. Comment éviter les périls de la répétition ?
 1. Révisez votre algorithme pour éliminer la répétition. Voici un exemple que je rencontre parfois dans des travaux d'étudiants : une expression présente à la fois dans la conséquence (`if`) et dans l'alternative (`else`) d'une structure conditionnelle. Il suffit de sortir l'expression en question de la structure pour éviter la répétition.
 2. Placez plutôt le code dans une fonction auxiliaire. Vous pourrez ensuite utiliser celle-ci autant de fois que vous voudrez en sachant que toute modification que vous y apporterez se répercutera automatiquement partout.
 3. Ne prenez pas de raccourcis — ils sont connus pour faire de longs délais³. Une répétition insérée volontairement pour espérer sauver du temps risque fort de vous causer des soucis plus tard. Prenez le temps de l'éviter ou de l'éliminer.
- Connaitre sur le bout des doigts la priorité des opérateurs du [tableau 4.1](#), c'est bien ; rendre explicite l'ordre des opérations dans une expression à l'aide de parenthèses, c'est mieux. N'hésitez pas à utiliser des parenthèses dès que l'ombre

3. « *Les raccourcis font de longs délais*, argumenta Pippin. » — *Le Seigneur des anneaux*, *La Communauté de l'anneau*, chapitre 4.

d'un doute pourrait planer sur l'ordre des opérations. D'ailleurs, à ce propos, [Oualline \(1997\)](#) ramène la quinzaine de règles de priorité des opérations (du langage C) à seulement deux :

1. La multiplication et la division précèdent l'addition et la soustraction.
 2. Placer tout le reste entre parenthèses.
- Évitez les *nombres magiques*. Un nombre magique est une constante numérique non nommée ou mal documentée dans le code source d'un programme ([Wikipédia, 2023](#)). Les nombres magiques rendent les programmes difficiles à lire, à interpréter et à déboguer. Voici une expression qui en est truffée :

```
cost <- 4.79 + 3 * floor(x/900)
```

Que représentent au juste les trois constantes ? Difficile de savoir sans la documentation appropriée. Dès qu'une constante dans une ligne de code nécessite de documenter sa signification, vous avez affaire à un nombre magique. Cependant, la documentation n'est pas une panacée : elle pourrait ne plus être à jour ou se trouver loin de l'endroit où les constantes apparaissent dans le code.

Afin de déjouer à la fois les périls des nombres magiques et de la répétition, affectez les constantes à des variables, puis employez ces variables dans les opérations arithmétiques. Utilisez des noms qui expliquent le rôle — ou la sémantique — des nombres magiques, pas leur valeur numérique (il ne sert à rien d'affecter le nombre magique 42 à une variable nommée QUARANTEDEUX). Par convention, vous devriez écrire les noms des variables contenant des nombres magiques en majuscules.

Le code ci-dessous est fonctionnellement équivalent à l'expression précédente, mais vous pouvez sans doute maintenant décoder sa signification même sans documentation :

```
BASE <- 4.79
EXTRA <- 3
TIMESLICE <- 30 * 60
cost <- BASE + EXTRA * floor(x/TIMESLICE)
```

Règle générale, les nombres -1 , 0 et 1 ne sont pas considérés comme des nombres magiques, tout comme 2 (souvent utilisé pour déterminer si un nombre est pair) et 100 (beaucoup utilisé dans la conversion des pourcentages). Selon le contexte, il peut convenir d'omettre d'autres valeurs.

- Évitez les expressions logiques complexes, surtout celles reposant sur la double négation. Par exemple, pour exécuter une expression si un vecteur contient des données manquantes, la condition

```
if (any(is.na(x)))
```

est beaucoup plus facile à déchiffrer que la version équivalente d'un point de vue logique

```
| if (!all(!is.na(x)))
```

En revanche, s'il s'agit plutôt d'exécuter une expression quand un vecteur ne contient aucune donnée manquante, alors

```
| if (all(!is.na(x)))
```

est plus simple que

```
| if (!any(is.na(x)))
```

De plus, le conseil précédent sur la priorité des opérations est particulièrement indiqué avec les opérations logiques. Sauriez-vous confirmer, sans consulter le [tableau 4.1](#), l'ordre des opérations dans l'expression logique suivante ? ⁴

```
| ! p | q & r
```


- Utilisez des fonctions d'application plutôt que des boucles explicites (nous étudierons les boucles au [chapitre 7](#)). Une expression ayant recours à une fonction d'application est plus concise et plus simple à décoder. Même si vous ne connaissez pas encore la syntaxe exacte des boucles, vous pouvez d'ores et déjà apprécier la différence de lisibilité entre

```
| z <- numeric(n)
| for (i in seq_len(n))
|   z[i] <- mean(x[[i]])
```

et

```
| z <- sapply(x, mean)
```

Ici encore, évitez de pousser la logique trop loin. Si une boucle est plus naturelle et plus simple à comprendre qu'une fonction d'application, optez pour la boucle. En particulier, une fonction d'application `sapply` à l'intérieur d'une autre fonction `sapply`, ce n'est généralement ni plus efficace, ni plus simple à déchiffrer qu'une double boucle `for`.

- Adoptez la [philosophie Unix](#) , notamment le précepte qui appelle à créer des programmes qui effectuent une seule chose et qui le font bien. Lorsqu'une fonction devient « longue » — cela dépend du contexte, mais généralement dès une vingtaine de lignes en R — il convient de la scinder en plusieurs blocs logiques.
- Enfin, utilisez `return` uniquement pour provoquer la sortie anticipée d'une fonction, habituellement à l'intérieur d'une clause `if`. En d'autres termes, en R, `return` n'a pas sa place à la toute fin d'une fonction.
- Évitez de vous répéter.

4. C'est `(!p) | (q & r)`.



La vidéo « [Le guide de style R du *geek* de placard](#) » propose, sur un ton humoristique, trois autres éléments de style à surveiller en programmation R.

6.2 Présentation du code

Le code bien mis en forme est plus facile et agréable à consulter. Il existe plusieurs chapelles dans le monde des programmeurs quant à la « bonne façon » de présenter et, surtout, d'indenter le code informatique.

Voyons d'abord ce qui rallie tout le monde.

En premier lieu, veillez à limiter la longueur des lignes de code à environ 80 caractères. Ce standard remonte à l'époque des terminaux en format texte qui ne pouvaient afficher de l'information que sur 80 colonnes. Pourquoi s'y tenir encore aujourd'hui, alors que nos écrans d'ordinateur sont très larges ? Parce que les longues lignes de texte sont difficiles à suivre, notre œil ayant tendance à sauter à la ligne inférieure en se déplaçant de la gauche vers la droite⁵. Profitez donc plutôt de l'espace horizontal à l'écran pour afficher des fenêtres côte à côte.

Si de multiples niveaux d'indentation (voir plus bas) font en sorte qu'il manque de place à droite pour écrire du code, le problème n'est peut-être pas tant la limite sur la longueur des lignes que la conception même du programme. Simplifiez l'algorithme ou scindez le programme en plusieurs fonctions.

Ensuite, aérez le code avec des lignes blanches entre les blocs logiques et, surtout, avec des espaces. Les espaces en programmation jouent le même rôle que dans du texte normal : elles facilitent la lecture. En particulier, utilisez des espaces dans les circonstances suivantes :

- ▶ de part et d'autre du symbole d'affectation « `<-` » ; ces espaces sont ajoutées automatiquement avec les raccourcis-clavier des éditeurs spécialisés GNU Emacs ([section B.5.7](#)) et RStudio ([section A.4](#)) ;
- ▶ de part et d'autre de tous les opérateurs⁶ ;
- ▶ après les virgules (mais jamais avant) ;
- ▶ avant la parenthèse ouvrante « `(` », sauf dans les appels de fonction.

Comparez les deux blocs de code de la [figure 6.1](#). Vous serez sans doute d'accord que celui qui respecte les indications ci-dessus s'avère bien plus lisible.

Passons maintenant au dossier chaud parmi les programmeurs : l'indentation du code et la position des accolades. Tous s'entendent au moins sur un point : il est absolument essentiel d'indenter les blocs de code pour mettre la structure d'un programme en évidence. En clair, cela signifie que toute expression — ou groupe d'expressions entre accolades — doit être placée en retrait de la marge de

5. C'est pourquoi les journaux et les magazines sont composés en colonnes de texte étroites.

6. Sauf peut-être la division : je préfère $(x + y)/z$ à $(x + y) / z$.

<pre>f<-function(x,y) { if(y<0) y<--y x*(1+x*y)^2 }</pre>	<pre>f <- function(x, y) { if (y < 0) y <- -y x * (1 + x * y)^2 }</pre>
--	--

FIG. 6.1 – Blocs de code sans (à gauche) et avec (à droite) les espaces appropriées. Le code de droite est plus lisible.

gauche dès lors qu'elle fait partie d'une structure de contrôle ou de la définition d'une fonction. Le code de la [figure 6.1](#) est correctement indenté.



Ne pas du tout indenter son code est passible de la peine capitale, d'excommunication, de bannissement de la Terre du Milieu... choisissez votre châtiment.

La source des insolubles débats se situe, comme souvent, dans les détails : le nombre d'espaces dont il convient d'indenter le code et la position des accolades, surtout l'accolade ouvrante. À titre d'exemples, l'éditeur GNU Emacs reconnaît et supporte au moins les styles d'indentation suivants :

C++

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

K&R (1TBS⁷)

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

Whitesmith

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

GNU

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

Le code source de R est entièrement composé dans un style analogue aux style C++, ci-dessus, ou RRR du mode ESS de Emacs :

- ▶ le code est indenté de quatre (4) espaces ;
- ▶ les accolades ouvrante et fermante sont placées sur leurs propres lignes.

Ce style peut être considéré comme standard pour la programmation en R.

7. *One True Bracing Style*. C'est dire combien les amateurs de ce style le tiennent en haute estime.

En définitive, le style d'indentation utilisé n'a pas tellement d'importance. Ce qui compte, c'est de se conformer au style en vigueur dans son domaine et de demeurer constant au fil de son code.



Les bons éditeurs pour programmeurs permettent de configurer le niveau d'indentation. Consultez la documentation de votre éditeur.

6.3 Commentaires et documentation

Les commentaires dans le code servent à guider le lecteur — peut-être vous-même, quelque temps après la rédaction — dans la lecture d'un programme. Le niveau de détails que devraient comporter les commentaires fait, comme le style d'indentation, l'objet de vifs débats.

Certains affirment qu'un bon programme se passe d'explications et que, par conséquent, les commentaires sont en grande partie inutiles. Or, comme le mentionne [Oualline \(1997\)](#), un programme sans commentaires constitue une bombe en attente d'exploser. Un jour ou l'autre, quelqu'un devra modifier ledit programme et l'absence de commentaires rendra la tâche beaucoup plus ardue que nécessaire.

À l'autre bout du spectre, on trouve les tenants du tout, tout commenter, jusqu'à l'évidence.

```
## calculer la somme de x  
z <- sum(x)
```

Cette pratique s'avère plus souvent qu'autrement contreproductive : non seulement vous force-t-elle à passer du temps à rédiger des commentaires sans véritable utilité, mais elle surcharge également le code, le rendant de ce fait plus difficile à lire.

Comme bien des choses en ce monde, la meilleure solution se trouve dans le juste milieu : commentez ni trop, ni trop peu. Les quelques préceptes suivants, dont certains sont tirés de [Kernighan et Pike \(1999\)](#), devraient vous aider à trouver un juste équilibre.

- Documentez non pas ce que *fait* le programme, mais *pourquoi* il le fait. Lire qu'un bloc de code effectue tel calcul s'avère de peu de secours si l'on ne sait pas dans quel but le calcul est effectué.
- N'enfoncez pas de portes ouvertes. Indiquer que l'expression `i <- i + 1` incrémente le compteur `i` n'est pas utile. Les commentaires doivent fournir de l'information qui ne saute pas aux yeux ou qui se trouve éparpillée dans le code.
- Définissez ce que fait chaque fonction, la nature de ses arguments et la valeur retournée. Fournissez aussi des exemples d'utilisation utiles et pertinents. Si une fonction R fait partie d'un paquetage, vous devrez nécessairement placer ces informations dans l'obligatoire rubrique d'aide de la fonction. Autrement, pla-

cez ces informations en commentaires avant la définition de la fonction. Vous devriez pouvoir expliquer ce que fait une fonction en une phrase.

- Éclairez les zones d'ombre, ne les rendez pas plus opaques. Des commentaires confus, imprécis ou qui entrent carrément en contradiction avec le code nuisent davantage qu'ils n'aident. Soyez concis et gardez toujours à l'esprit de fournir au lecteur des informations justes et pertinentes.
- Ne documentez pas du mauvais code, réécrivez-le. Si les commentaires sont beaucoup plus longs que le code auquel ils se rapportent, c'est probablement qu'il est temps de réviser le code.

À moins qu'une rubrique d'aide en bonne et due forme n'accompagne une fonction, sa définition devrait toujours être précédée d'un bloc de documentation contenant au minimum :

1. La *signature* de la fonction (son nom suivi, entre parenthèses, de tous les arguments avec leur valeur par défaut, le cas échéant);
2. Une courte description de ce que fait la fonction (en mode infinitif, de manière à compléter la phrase « Cette fonction permet de... »);
3. La liste des arguments, de leur signification et des valeurs admissibles;
4. La valeur retournée par la fonction;
5. Un ou plusieurs exemples d'utilisation de la fonction, selon le niveau de complexité de celle-ci.

Il s'agit de la structure générale des rubriques d'aide de R. La [figure 6.2](#) fournit un exemple pour une fonction simple.



Utilisez le fichier `gabarit-documentation-fonction.R` livré avec cet ouvrage pour insérer la structure de base de la documentation d'une fonction dans vos fichiers de script.

Dans R, le symbole carré « # » — ou numéro — marque le début d'un commentaire, et ce, peu importe où le symbole se trouve sur la ligne. Il est possible de combiner les symboles « # » pour former une hiérarchie dans les commentaires ou pour délimiter différentes sections d'un fichier de script. J'ai utilisé dans cet ouvrage la [convention de l'éditeur GNU Emacs](#).

- Les commentaires qui débutent par trois carrés, « ### », sont toujours alignés sur la marge de gauche. Ils ne sont utilisés qu'à l'extérieur des fonctions. Ils marquent soit des sections, soit des entêtes de fonctions.
- Les commentaires qui débutent par deux carrés, « ## », sont alignés sur le niveau d'indentation courant. Lorsqu'ils apparaissent à l'intérieur d'une fonction, ils décrivent le rôle du bloc de code qui suit ou l'état de la fonction à ce stade. À l'extérieur des fonctions, ils marquent des sous-sections du code source.

```
###
### square(x)
###
## Éléver au carré.
##
## Arguments
##
## x: vecteur de nombres réels.
##
## Valeur
##
## Vecteur de nombres réels contenant les carrés
## des arguments.
##
## Exemples
##
## square(2)
##
square <- function(x) x * x
```

FIG. 6.2 – Exemple de documentation pour une fonction simple

- Les commentaires qui débutent par un seul carré, « # », sont alignés sur une colonne à droite du code source. Ils servent à clarifier, très succinctement, ce qu'effectue une ligne de code.

L'éditeur RStudio, pour sa part, utilise par défaut les niveaux de titres du langage de balisage [Markdown](#), dont la hiérarchie est exactement l'inverse de celle de Emacs. Ainsi, les commentaires de premier niveau débutent par un seul carré, « # » ; les commentaires de deuxième niveau débutent par deux carrés, « ## », etc.



Le fichier de script pratiques.R reproduit à la [section 6.5](#) contient, aux lignes [12-240](#), un long exemple d'amélioration du style, de la présentation et de la documentation d'une fonction.

6.4 Tests unitaires

Lorsque vous rédigez un programme, vous savez ce que celui-ci doit accomplir⁸. Il tombe sous le sens qu'une fois la programmation complétée, vous devriez

8. Si ce n'est pas le cas, retournez soit à la table à dessin, soit consulter la personne qui vous a confié le mandat de programmation !

tester que le programme effectue le traitement adéquat, non ? Et pourtant... Tout bogue dans un logiciel signale un test non ou mal exécuté.

Cette section explique brièvement comment mettre en place une infrastructure de tests pour du code R. Comme [Hunt et Thomas \(1999\)](#) le soulignent, effectuer des tests est davantage une question culturelle que technique, en ce sens que le plus difficile n'est pas tant d'apprendre à tester le code dans un langage donné que de prendre l'habitude de tester son code. Efforcez-vous donc d'intégrer les tests dès maintenant dans votre routine de programmation.

Compte tenu de la portée de cet ouvrage, nous allons limiter la discussion aux *tests unitaires* (*unit tests*). Ceux-ci permettent de vérifier le bon fonctionnement d'une partie d'un grand programme. Dans notre contexte, un test unitaire porte sur une fonction R.

Au moment d'écrire une fonction, vous devriez avoir en main — ou au moins en tête — sa spécification. Celle-ci contient normalement :

1. La signature de la fonction ;
2. La liste des arguments de la fonction ;
3. La valeur retournée par la fonction ;
4. Une description détaillée de ce que doit réaliser la fonction, y compris les cas limites et la gestion des erreurs, le cas échéant.

Vous aurez reconnu plusieurs des éléments de la documentation d'une fonction mentionnés à la [section 6.3](#). En effet, une large part de la documentation s'écrit toute seule à partir de la spécification.

Les tests unitaires confrontent la réalisation d'une fonction à sa spécification. Les bons tests ne valident pas uniquement les cas « normaux », mais également les cas limites et les erreurs, lorsque la fonction se doit de les traiter. Si la fonction est vectorielle, assurez-vous aussi de la tester tant avec un vecteur de longueur 1 qu'avec un vecteur de longueur supérieure à 1.



Durant le codage d'une fonction, vous effectuez sans doute des petits tests pour valider votre fonction. Conservez ces bouts de code ! Ils formeront la base de vos tests unitaires.

Prenons l'exemple d'une fonction simple qui calcule la racine carrée d'un seul nombre. Vous devriez alors écrire des tests unitaires pour les cas suivants :

1. La fonction retourne une erreur (ou NaN) si l'argument est strictement négatif ;
2. La fonction retourne un véritable 0 (et non une approximation) pour le cas limite où l'argument est nul ;
3. le carré de la valeur retournée par la fonction est approximativement égal à l'argument lorsque celui-ci est strictement supérieur à 0.

Concrètement, les tests consistent en des expressions qui retournent une erreur uniquement lorsque la réalisation d'une fonction n'est pas conforme à la spécification. Si l'évaluation de l'ensemble des expressions se déroule sans erreur, vous savez que tous les tests sont réussis.

Reste à savoir quelles fonctions utiliser pour rédiger les tests unitaires. Plusieurs paquetages externes⁹ proposent des outils pour enjoliver leur écriture ou leurs résultats, par exemple **tinytest** (Loo, 2021), **testthat** (Wickham, 2011) ou **RUnit** (Burger et collab., 2024), pour n'en nommer que trois. Pour ma part, j'estime que les fonctions ci-dessous de la distribution R de base s'avèrent généralement suffisantes pour la mise en place d'une infrastructure de tests.

source évaluation séquentielle d'un fichier de script.

```
> source("tests.R")
```

stopifnot erreur dès qu'une expression n'est pas vraie.

```
> stopifnot(1 == 1, 1 < 2)
> stopifnot(3 < 2)
Erreur : 3 < 2 n'est pas TRUE
```

assertError validation qu'une expression retourne une erreur.

```
> assertError(sqrt("abc"))
> assertError(sqrt(4))
Erreur : Aucune erreur n'a été renvoyée lors
de l'évaluation de sqrt(4)
```

assertWarning validation qu'une expression retourne un avertissement.

```
> assertWarning(matrix(1:8, 4, 3))
> assertWarning(matrix(1:12, 4, 3))
Error in [...] : warning n'a pas été obtenu
lors de l'évaluation de matrix(1:12, 4, 3)
```

all.equal égalité entre deux nombres à une (petite) différence près.

```
> tan(pi/4) == 1
[1] FALSE
> 1 - tan(pi/4)
[1] 1.110223e-16
> all.equal(tan(pi/4), 1)
[1] TRUE
```

9. L'utilisation de paquetages fait l'objet du [chapitre 10](#).

```
> all.equal(pi, 355/113)
[1] "Mean relative difference: 8.491368e-08"
```

`identical` égalité exacte et complète entre deux objets.

```
> identical("abc", "abc")
[1] TRUE
> identical(1, as.integer(1))
[1] FALSE
```



Lorsqu'il s'agit de comparer des valeurs numériques résultant de calculs, vous devez absolument éviter d'utiliser l'opérateur « `==` ». Il est hors de la portée de cet ouvrage de discuter d'arithmétique en virgule flottante, mais consultez déjà l'[entrée 7.31](#) (la plus célèbre de toutes !) de la foire aux questions de R ([Hornik et R Core Team, 2024](#)).



Deux objets vous apparaissent identiques et `identical` persiste à dire que ce n'est pas le cas ? La fonction `str`, qui révèle la structure interne des objets, pourrait vous aider à identifier les différences que R y voit.

Terminons avec une question toute pratique : où placer les tests unitaires ? Pour les projets simples, vous pouvez les insérer dans le fichier de script contenant la définition de votre fonction. Cette stratégie convient toutefois mal aux grands projets ou aux tests longs à exécuter. Une meilleure pratique consiste à placer les tests dans des fichiers de script dédiés. Si un jour vous développez votre propre paquetage R, vous pourrez déposer les fichiers dans un répertoire spécial justement nommé `tests`.

Un fichier de tests dédié doit pouvoir être évalué intégralement dans une session R vierge. Par conséquent, assurez-vous d'inclure des appels à `source` (pour du code libre) ou à `library` (pour du code dans un paquetage) pour faire en sorte que les fonctions testées existent dans l'espace de travail de la session R.



Difficile de bien saisir comment concevoir un fichier de tests sans un exemple. Il s'en trouve un complet pour une fonction simple aux lignes [243-403](#) du fichier de script `pratiques.R` reproduit à la section suivante.

6.5 Exemples

📍 Fichier d'accompagnement `pratiques.R`

```

11  ###
12  ### STYLE, PRÉSENTATION, COMMENTAIRES ET DOCUMENTATION
13  ###
14
15  ## Afin d'illustrer l'utilité de bien présenter et de commenter son
16  ## code, nous allons prendre une fonction dans un état assez pitoyable
17  ## et l'améliorer graduellement.
18  ##
19  ## Bien qu'il existe une fonction 'sqrt' dans R pour calculer la
20  ## racine carrée d'un nombre, nous allons programmer notre propre
21  ## version. Elle est basée sur la méthode des approximations
22  ## successives de Newton dont l'algorithme est le suivant:
23  ##
24  ## ALGORITHME Calculer la racine carrée d'un nombre x, c'est-à-dire la
25  ## valeur y tel que  $y \geq 0$  et  $y^2 = x$ .
26  ##
27  ## 1. Poser y égal à une valeur de départ quelconque.
28  ## 2. Si  $|y^2 - x| < e$ , retourner la valeur y.
29  ## 3. Poser  $y \leftarrow (y + x/y)/2$  et retourner à l'étape 2.
30  ##
31  ## La valeur 'e' dans l'algorithme représente la précision de
32  ## l'approximation. Dans notre mise en oeuvre ci-dessous, nous
33  ## utiliserons  $e = 0.001$ . Quant à la valeur de départ de l'étape 1,
34  ## nous utiliserons  $y = 1$ .
35  ##
36  ## Notre mise en oeuvre de l'algorithme est itérative. Elle repose sur
37  ## l'idée de répéter l'étape 3 «tant que» la condition de l'étape 2
38  ## n'est pas remplie. Ceci se traduit en langage informatique en
39  ## boucle 'while'. Nous n'avons pas encore étudié cette structure de
40  ## contrôle, mais vous devriez néanmoins pouvoir suivre l'exemple
41  ## aisément.
42  ##
43  ## Dernière chose: notre fonction 'sqrt' fait appel à une seconde
44  ## fonction interne pour effectuer le calcul de l'étape 3 de
45  ## l'algorithme.
46
47  ### PRÉSENTATION DU CODE
48
49  ## La première version du code ne respecte pas les règles de base
50  ## d'indentation et d'«aération» du code. Résultat: un fouillis
51  ## difficile à consulter.
52  sqrt<-function(x)
53  {if(x<0) stop("x doit être un nombre positif")
54    improve<-function(guess,x)
55      mean(c(guess,x/guess))
56  y<-1
57  while(!abs(y^2-x)>=0.001)y<-improve(y, x)
58      return(y)

```



```
59
60 ## Réviser ne serait-ce que l'indentation permet déjà d'y voir plus
61 ## clair. Tous les bons éditeurs de texte pour programmeurs sont en
62 ## mesure d'indenter le code pour vous, à la volée ou de manière
63 ## asynchrone.
64 ##
65 ## Vous pouvez arriver au résultat ci-dessous avec RStudio en
66 ## sélectionnant le code ci-dessus et en exécutant l'option du menu
67 ## Code|Reindent Lines.
68 ##
69 ## Dans Emacs, l'indentation se fait automatiquement au fur et à
70 ## mesure que l'on entre du code ou, autrement, en appuyant sur la
71 ## touche de tabulation.
72 sqrt<-function(x)
73 {if(x<0) stop("x doit être un nombre positif")
74   improve<-function(guess,x)
75     mean(c(guess,x/guess))
76   y<-1
77   while(!abs(y^2-x)>=0.001)y<-improve(y, x)
78   return(y)
79
80 ## La simple indentation du code nous permet déjà de découvrir un
81 ## bogue dans le code: il manque une accolade fermante } à la fin de
82 ## la fonction.
83 ##
84 ## Corrigeons déjà le code.
85 sqrt<-function(x)
86 {if(x<0) stop("x doit être un nombre positif")
87   improve<-function(guess,x)
88     mean(c(guess,x/guess))
89   y<-1
90   while(!abs(y^2-x)>=0.001)y<-improve(y, x)
91   return(y)
92 }
93
94 ## Les normes usuelles de présentation du code informatique exigent
95 ## également de placer les accolades ouvrantes et fermantes seules sur
96 ## leur ligne et d'aérer le code avec des espaces autour des
97 ## opérateurs et des structures de contrôle, après les virgules, etc.
98 ## Comme pour du texte normal, les espaces rendent le code plus facile
99 ## à lire.
100 ##
101 ## Dans RStudio, vous pouvez parvenir à la présentation ci-dessous
102 ## avec la commande du menu Code|Reformat Code.
103 sqrt <- function(x)
104 {
105   if (x < 0)
106     stop("x doit être un nombre positif")
```

```
107     improve <- function(guess, x)
108         mean(c(guess, x/guess))
109     y <- 1
110     while (!abs(y^2 - x) >= 0.001)
111         y <- improve(y, x)
112     return(y)
113 }
114
115 ## Je recommande d'ajouter des lignes blanches additionnelles dans le
116 ## code afin de bien séparer les blocs logiques. Dans le cas présent,
117 ## ces blocs sont:
118 ##
119 ## 1. le test de validité de l'argument;
120 ## 2. la définition de la fonction interne 'improve';
121 ## 3. l'affectation de la valeur de départ;
122 ## 4. les approximations successives;
123 ## 5. la valeur finale.
124 sqrt <- function(x)
125 {
126     if (x < 0)
127         stop("x doit être un nombre positif")
128
129     improve <- function(guess, x)
130         mean(c(guess, x/guess))
131
132     y <- 1
133
134     while (!abs(y^2 - x) >= 0.001)
135         y <- improve(y, x)
136
137     return(y)
138 }
139
140 ### STYLE
141
142 ## Il y a quelque chose à redire sur le style de cette fonction?
143 ## Pourtant, les noms d'objets sont raisonnables, le coeur de la
144 ## fonction n'est pas inutilement placé dans une clause 'else' après
145 ## le test de validité de l'argument 'x', le calcul de la nouvelle
146 ## approximation est placé sous une couche d'abstraction dans une
147 ## fonction interne...
148 ##
149 ## Il reste tout de même quatre choses à améliorer.
150 ##
151 ## 1. Nous pouvons traiter séparément le cas trivial où l'argument est
152 ## égal à 0. Il n'y a aucun calcul à faire dans ce cas et la
153 ## fonction peut directement retourner la réponse 0.
154 ## 2. L'expression logique dans la clause 'while' qui utilise une
```

```
155 ##      formulation «n'est pas plus petite que 0.001» est inutilement
156 ##      compliquée. De plus, elle repose sur la priorité des opérations:
157 ##      la négation logique a-t-elle bel et bien une priorité plus
158 ##      faible que l'inégalité? Il suffit de réécrire l'expression sous
159 ##      la forme «est plus grand ou égal à 0.001» et tout sera plus
160 ##      clair.
161 ## 3. Ce 0.001, justement, est un nombre magique dans notre fonction.
162 ##      Affectons plutôt la valeur à une variable qui clarifiera la
163 ##      sémantique du nombre.
164 ## 4. Il y a ce 'return' à la dernière ligne de la fonction. Ça, c'est
165 ##      un gros non en R.
166 ##
167 ## Améliorons le style.
168 sqrt <- function(x)
169 {
170     ERROR_MAX <- 0.001
171
172     if (x < 0)
173         stop("x doit être un nombre positif")
174
175     if (x == 0)
176         return(0)
177
178     improve <- function(guess, x)
179         mean(c(guess, x/guess))
180
181     y <- 1
182
183     while (abs(y^2 - x) >= ERROR_MAX)
184         y <- improve(y, x)
185
186     y
187 }
188
189 ### COMMENTAIRES ET DOCUMENTATION
190
191 ## Dernier élément manquant dans notre code: les commentaires et la
192 ## documentation.
193
194 ###
195 ### sqrt(x)
196 ###
197 ## Calculer la racine carrée d'un nombre.
198 ##
199 ## Argument
200 ##
201 ## x: nombre réel positif.
202 ##
```

```

203 ## Valeur
204 ##
205 ## Nombre y tel que  $y^2 = x$ .
206 ##
207 ## Exemples
208 ##
209 ## sqrt(9)
210 ##
211 sqrt <- function(x)
212 {
213     ## Précision de l'approximation.
214     ERROR_MAX <- 0.001
215
216     ## Vérification de la validité de 'x'.
217     if (x < 0)
218         stop("'x' doit être un nombre positif")
219
220     ## Cas trivial.
221     if (x == 0)
222         return(0)
223
224     ## Fonction pour calculer la prochaine approximation selon la
225     ## méthode de Newton. Si 'y' est l'approximation actuelle de la
226     ## racine carrée de 'x', alors la nouvelle approximation est
227     ##  $(y + x/y)/2$ .
228     improve <- function(guess, x)
229         mean(c(guess, x/guess))
230
231     ## Valeur de départ de la procédure itérative.
232     y <- 1
233
234     ## Approximations successives.
235     while (abs(y^2 - x) >= ERROR_MAX)
236         y <- improve(y, x)
237
238     ## Retourner la valeur tel que  $y^2 - x < 0.001$ .
239     y
240 }
241
242 ###
243 ### TESTS UNITAIRES
244 ###
245
246 ## FONCTIONS UTILES
247
248 ## La fonction 'source' permet d'évaluer intégralement un fichier de
249 ## script. C'est l'équivalent, en une commande, d'évaluer un fichier
250 ## ligne par ligne de manière interactive. La fonction est

```

```
251 ## particulièrement utile pour définir rapidement des fonctions dans
252 ## un espace de travail ou pour exécuter des tests unitaires.
253 ##
254 ## Le fichier 'sqrt.R' contient le code et la documentation de la
255 ## fonction 'sqrt' ci-dessus. Nous pouvons définir la fonction dans
256 ## l'espace de travail rapidement avec l'expression suivante.
257 source("sqrt.R")
258
259 ## La fonction 'stopifnot' permet de générer une erreur dès qu'une des
260 ## expressions en argument n'est pas vraie.
261 ##
262 ## Vous pouvez l'utiliser dans vos fonctions lorsqu'il y a plusieurs
263 ## tests de validité des arguments, mais elle sert surtout pour les
264 ## tests unitaires.
265 stopifnot(1 == 2) # erreur
266 stopifnot(1 < 2) # pas d'erreur
267
268 ## Pour tester plusieurs expressions à la fois dans 'stopifnot', deux
269 ## options. La première, les séparer par des virgules, possiblement
270 ## sur plusieurs lignes.
271 stopifnot(1 < 2, 2 == 2, "a" < "b")
272 stopifnot(1 < 2,
273           2 == 2,
274           "a" < "b")
275
276 ## L'autre option permet d'éviter les virgules, ce qui est plus
277 ## pratique pour l'évaluation interactive ligne par ligne. Il s'agit
278 ## de fournir à l'argument 'exprs' (qu'il faut nommer explicitement)
279 ## un groupe d'expressions entre accolades «{ }» et séparées les unes
280 ## des autres par un retour à la ligne.
281 stopifnot(exprs = {
282     1 < 2
283     2 == 2
284     "a" < "b"
285 })
286
287 ## Les fonctions 'assertError' et 'assertWarning' font partie de la
288 ## distribution R de base, mais elles se trouvent dans le paquetage
289 ## 'tools' qui n'est pas chargé par défaut. Les appeler sous la forme
290 ## 'tools::assertError' et 'tools::asserWarning' permet d'éviter de
291 ## devoir charger le paquetage en mémoire.
292 ##
293 ## Attention avec 'assertError': pour cette fonction le comportement
294 ## correct de l'expression en argument consiste à... retourner une
295 ## erreur! (C'est ce que l'on veut tester.)
296 TRUE <- 3 # erreur
297 tools::assertError(TRUE <- 3) # ok!
298 T <- 3 # valide
```

```

299 tools::assertError(T <- 3)      # erreur: pas d'erreur!
300
301 ## Similaire pour 'assertWarning'.
302 1:4 + 1:3                        # avertissement
303 tools::assertWarning(1:4 + 1:3) # ok!
304 1:4 + 1:2                        # valide
305 tools::assertWarning(1:4 + 1:2) # erreur: pas d'avertissement!
306
307 ## La fonction 'all.equal' est extrêmement importante pour vérifier
308 ## l'égalité ou, plutôt, la presque égalité de deux nombres en virgule
309 ## flottante dans R.
310 ##
311 ## Ne vérifiez JAMAIS l'égalité de deux nombres obtenus à partir de
312 ## calculs avec '==' (à moins d'avoir la certitude que les calculs
313 ## n'impliquent que des entiers).
314 1.2 + 1.4 + 2.8                  # 5.4
315 1.2 + 1.4 + 2.8 == 5.4           # non?!?
316 5.4 - (1.2 + 1.4 + 2.8)         # petit écart
317 all.equal(1.2 + 1.4 + 2.8, 5.4) # ok
318
319 ## Vous pouvez utiliser '==' pour les chaînes de caractères, mais
320 ## n'oubliez pas que la comparaison se fait élément par élément.
321 letters[1:3] == c("a", "b", "c")
322
323 ## La fonction 'identical', comme son nom l'indique, vérifie que deux
324 ## objets sont rigoureusement identiques.
325 identical(letters[1:3], c("a", "b", "c"))
326 identical(42, 40 + 2)
327
328 ## Une différence dans le type d'objet, les attributs ou la classe et
329 ## le résultat est FALSE.
330 (x <- 42)                        # nombre réel en double précision
331 str(x)                          # confirmation
332 (y <- as.integer(42))           # véritable entier
333 str(y)                          # confirmation
334 x == y                          # visuellement pareils et égaux...
335 identical(x, y)                # ... mais pas identiques
336
337 (x <- c(a = 2))                 # nombre étiqueté
338 (y <- c(b = 2))                 # idem
339 x == y                          # nombres égaux...
340 identical(x, y)                # ... mais objets non identiques
341
342 ## TESTS
343
344 ## Nous avons maintenant tout en main pour rédiger des tests
345 ## unitaires.
346 ##

```

```
347 ## À titre d'exemple, nous allons composer les tests pour la fonction
348 ## 'sqrt' ci-dessus (ou qui se trouve dans le fichier d'accompagnement
349 ## 'sqrt.R').
350 ##
351 ## La fonction n'est PAS vectorielle. Nous n'avons donc pas à tester
352 ## sa validité avec un vecteur de nombres de longueur supérieure à 1.
353 ##
354 ## Premier test: la fonction retourne une erreur pour un argument
355 ## strictement négatif.
356 tools::assertError(sqrt(-1))
357
358 ## Deuxième test: la fonction retourne un véritable 0 pour un argument
359 ## nul.
360 stopifnot(identical(0, sqrt(0)))
361
362 ## Troisième test: le carré de la réponse est (approximativement) égal
363 ## à l'argument.
364 ##
365 ## Dans cet exemple précis, la précision de la fonction 'sqrt' est
366 ## limitée à 0,001. Nous devons donc fournir une tolérance à
367 ## 'all.equal'. Prenez note que ce n'est généralement pas nécessaire.
368 stopifnot(all.equal(3, sqrt(3)^2,
369                   tolerance = 0.001))
370
371 ## Nous pouvons aussi regrouper les deux derniers tests en un seul
372 ## appel à 'stopifnot'.
373 stopifnot(exprs = {
374     identical(0, sqrt(0))
375     all.equal(3, sqrt(3)^2,
376               tolerance = 0.001)
377 })
378
379 ## Comme les tests ci-dessus sont rapides à évaluer, nous aurions pu
380 ## les placer directement dans le fichier 'sqrt.R'. Ainsi,
381 ## 'source("sqrt.R")' permet à la fois de définir la fonction et de
382 ## vérifier sa validité. C'est suffisant pour les petits projets ou
383 ## les fonctions seules.
384 ##
385 ## De manière plus générale, je vous recommande de placer les tests
386 ## dans des fichiers de script dédiés et intégralement évaluables dans
387 ## une session R vierge.
388 ##
389 ## Le fichier de script 'tests-sqrt.R' distribué avec l'ouvrage
390 ## reprend les tests ci-dessus. Le fichier a aussi recours à 'source'
391 ## pour faire en sorte que la fonction 'sqrt' existe dans l'espace de
392 ## travail de la session R.
393 ##
394 ## Vous pouvez ainsi ensuite évaluer les tests simplement en évaluant
```

```

395 ## séquentiellement le fichier de tests.
396 source("tests-sqrt.R")      # exécution des tests
397
398 ### NETTOYAGE
399
400 ## La fonction 'sqrt' ci-dessus masque la fonction interne de R.
401 ## Question d'éviter les éventuels problèmes, effaçons notre fonction
402 ## moins performante.
403 rm("sqrt")

```

6.6 Exercices

6.1 Présenter le code ci-dessous selon les normes d'espacement, d'indentation et de positionnement des accolades mentionnées dans ce chapitre. Pour ajuster automatiquement l'indentation avec RStudio, sélectionner le bloc de code et choisir dans les menus Code|Reformat Code.

```

f <- function(x){
  if(all(x>=0)|| all(x<=0))
  { stop("all x are the same sign")
  }
  if (sum(diff(sign(x[x!=0]))!=0)>1)
  warning("more than one sign change")
  r<-polyroot(x)
  i<-1/Re(r)[abs(Im(r))< 1.5e-8]-1
  i[i > -1]
}

```

6.2 Rédiger des tests unitaires pour la fonction de l'exercice 6.1 à partir de la lecture du code de la fonction et des informations additionnelles suivantes :

1. La fonction prend en argument un vecteur de nombres réels positifs, négatifs ou nuls ;
2. Elle vérifie le nombre de changements de signe dans le vecteur après avoir éliminé les valeurs nulles ;
3. La réponse correcte correspondant au vecteur $(-10, -7, 12, 0, 11)$ est :

0,128 564 783 114 ;

4. La réponse correcte correspondant au vecteur $(-10, 12, 7, 0, 11)$ est :

0,785 366 742 540.

7 Tri et recherche

Objectifs du chapitre

- ▶ Expliquer les algorithmes classiques de tri et de recherche ; comparer leurs avantages et inconvénients.
- ▶ Concevoir une boucle itérative dans R.
- ▶ Effectuer la mise en oeuvre d'un algorithme itératif dans le langage R.

Nous nous sommes penchés sur le concept d'algorithme et sur les principes de base de l'algorithmique au [chapitre 2](#). Or, pour citer [Kernighan et Pike \(1999\)](#) :

Tous les programmes reposent sur des algorithmes et des structures de données, mais très peu de programmes exigent d'en concevoir de nouveaux.

Autrement dit, aussi complexe soit-il, un programme repose souvent sur quelques algorithmes fondamentaux bien connus et bien établis. À ce titre, les algorithmes de tri et de recherche jouent un rôle particulièrement important. Donald Knuth consacre d'ailleurs un volume entier de son œuvre monumentale *The Art of Computer Programming* à ce seul sujet ([Knuth, 1997c](#)).

Dans l'introduction de son chapitre 5, Knuth rapporte que les grands fabricants d'ordinateurs estimaient dès les années 1960 que 25 % du temps de calcul de leurs ordinateurs était consacré au tri. Dans certaines applications, cela dépassait même la part de 50 % du temps de calcul. Selon Knuth, de telles statistiques signifient ou bien qu'il existe plusieurs applications importantes du tri ; ou bien que plusieurs personnes trient sans que ce ne soit nécessaire ; ou bien que les algorithmes employés sont inefficaces. Il y a sans doute une part de vérité dans chacune de ces hypothèses, mais le fait est que le tri s'avère, d'un point de vue pratique, un sujet d'étude important.

Pour vous convaincre de la variété d'algorithmes de tri, prenez le temps de réaliser la petite expérience suivante :

Choisir au hasard cinq cartes à jouer d'une même enseigne^{1, 2}. Déterminer un algorithme ou une procédure pas à pas pour trier les cartes en ordre croissant. Comparer votre méthode de tri avec celles d'autres personnes de votre entourage³.

Si vous lisez cette phrase sans avoir complété l'expérience, je vous encourage à rebrousser chemin. Elle en vaut le coup pour la suite.

La procédure de tri que vous avez utilisée est fort probablement de l'un ou l'autre des types suivants (Knuth, 1997c, section 5.2) :

- i) tri par insertion, où chaque carte de la main est évaluée tour à tour et placée au bon endroit dans la liste des cartes déjà triées ;
- ii) tri par sélection, où la plus petite carte est identifiée et placée au début de la main, puis la seconde plus petite, et ainsi de suite ;
- iii) tri par échange, où deux cartes qui ne sont pas dans le bon ordre sont échangées, le procédé étant répété tant et aussi longtemps que les cartes ne sont pas entièrement triées ;
- iv) tri par dénombrement, où l'on compte le nombre de cartes plus petites qu'une carte donnée pour déterminer son rang dans la main triée ;
- v) tri par une méthode *ad hoc* qui fonctionne pour une main de cinq cartes, mais qui ne saurait se généraliser à un problème de plus grande taille ;
- vi) tri paresseux, qui consiste à ne pas faire l'expérience recommandée.

La figure 7.1 illustre les quatre premières méthodes de tri ci-dessus.

Ce chapitre est donc consacré à l'étude de quelques-uns des algorithmes classiques de tri et de recherche. Ces opérations étant par nature répétitives, le sujet se prête tout naturellement à aussi aborder, en seconde partie, la conception de boucles itératives dans R.



En complément de la figure 7.1, mais aussi pour préparer la lecture des prochaines sections, je vous recommande de visionner une courte vidéo sur d'illustration de quelques algorithmes de tri. [🔗](#)

7.1 Définition du problème et notation

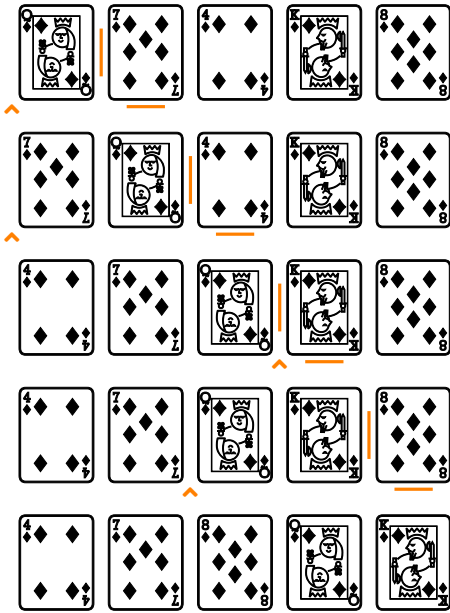
Avant d'aller plus loin, prenons le temps de définir formellement le problème de tri et de nous munir de la notation qui sera utilisée tout au long du chapitre. Nous allons tâcher de trier n enregistrements

$$E_1, E_2, \dots, E_n.$$

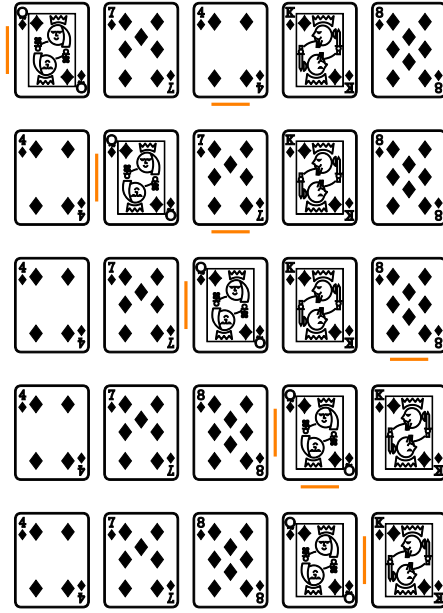
1. Les quatre enseignes d'un jeu de carte de standard sont piques, cœurs, carreaux et trèfles.

2. Si vous n'avez pas de jeu de carte à portée de la main, une simple liste de nombres tirés aléatoirement de votre cerveau fertile et inscrits sur une feuille de papier fera l'affaire.

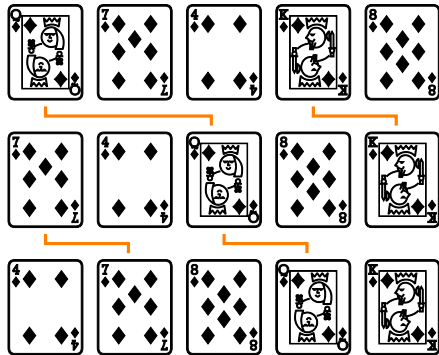
3. À défaut d'entourage apte à trier des cartes, déterminer une ou deux autres procédures de tri de votre cru.



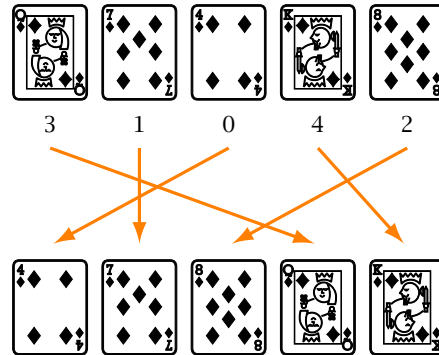
(a) Tri par insertion. Le trait vertical délimite les cartes triées de celles qui ne le sont pas. Le trait horizontal identifie la carte à trier. Le symbole « \wedge » indique l'endroit où insérer la carte non triée.



(b) Tri par sélection. Le trait vertical délimite les cartes triées de celles qui ne le sont pas. Le trait horizontal identifie la plus petite carte.



(c) Tri par échange



(d) Tri par dénombrement

FIG. 7.1 – Illustrations de quatre méthodes de tri pour une main de cinq cartes à jouer

Un enregistrement est un ensemble d'informations formant un tout logique et qui sont manipulées en bloc. Lorsque ce sera nécessaire, l'ensemble des enregistrements sera nommé *fichier*. À chaque enregistrement E_j correspond une *clé* de tri C_j .

Par exemple, dans le fichier des résultats d'un cours universitaire, un enregistrement correspond aux résultats d'un étudiant ou d'une étudiante. Il est généralement formé d'un numéro d'identification unique, du nom, du prénom et des résultats de toutes les évaluations. Chacun de ces éléments peut constituer une clé de tri, afin de classer la liste par numéro d'identification, par nom, par résultat à une évaluation donnée, etc.



Vous avez déjà rencontré ce genre d'interface dans une page web : des données présentées sous forme de tableau avec la possibilité de cliquer sur un titre de colonne pour trier les données selon celle-ci. Chaque ligne du tableau constitue un enregistrement et chaque colonne constitue une éventuelle clé de tri.

Sans perte de généralité, nous allons étudier le tri en ordre croissant. Le problème de tri consiste donc à trouver une permutation $\{(1), (2), \dots, (n)\}$ des indices $\{1, 2, \dots, n\}$ qui ordonne les clés de telle sorte que

$$C_{(1)} \leq C_{(2)} \leq \dots \leq C_{(n)}.$$

Dans la suite, nous allons illustrer les algorithmes en triant le vecteur de seize nombres entiers suivant :

513	780	321	828	623	080	596	423
380	707	693	766	139	161	316	071

7.2 Tri par insertion

Le tri par insertion est l'une des techniques de tri les plus simples et intuitives. C'est probablement celle que vous avez utilisée lors de la petite expérience du début du chapitre. Elle consiste à examiner les enregistrements tour à tour à partir du deuxième et, pour chacun, à trouver sa position dans les enregistrements déjà triés. En d'autres termes, lors de l'examen de l'enregistrement E_j , $j \geq 2$, les enregistrements précédents E_1, \dots, E_{j-1} sont déjà triés et il s'agit de trouver la position de E_j parmi ceux-ci. Dans l'algorithme qui suit, cette position est identifiée en comparant la clé de l'enregistrement E_j aux clés des enregistrements E_{j-1}, \dots, E_1 , dans cet ordre.

Algorithme 7.1 (Tri par insertion). Réorganiser les enregistrements E_1, \dots, E_n de telle sorte qu'en sortie leurs clés soient en ordre croissant, $C_1 \leq \dots \leq C_n$.

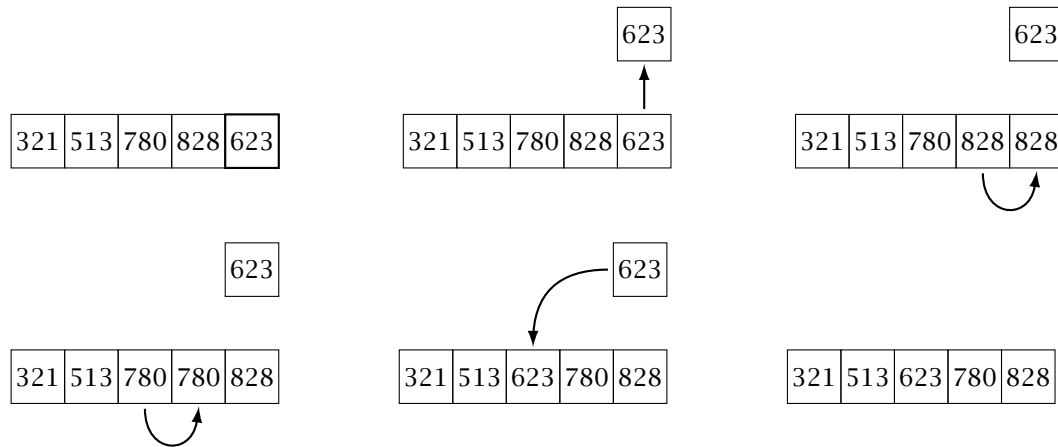


FIG. 7.2 - Illustration du positionnement d'un enregistrement non trié parmi les enregistrements triés à l'aide d'une variable tampon. L'enregistrement non trié est encadré en gras dans la première illustration.

En fait, les enregistrements ne sont pas « déplacés » dans le fichier et aucun espace n'est véritablement « libéré ». Les enregistrements sont plutôt dupliqués, puis leur ancienne valeur est rendue disponible pour être écrasée par un autre enregistrement. La [figure 7.2](#) illustre cette procédure pour la quatrième itération de notre exemple.

L'algorithme doit traiter $n - 1$ enregistrements. Lors du traitement de l'enregistrement j , $j \in \{2, \dots, n\}$, sa clé sera comparée, en moyenne, à celle de $(j - 1)/2$ enregistrements. L'ordre de grandeur du tri par insertion est donc $O(n^2)$. C'est tout à fait raisonnable pour une main de cartes, mais le coût devient rapidement prohibitif pour le tri de grands fichiers.

Le code informatique de la [section 7.9](#) contient, aux lignes 186-227, une mise en œuvre en R du tri par insertion. Vous pouvez déjà y jeter un coup d'œil, mais son examen attentif devrait attendre d'avoir étudié les boucles itératives à la [section 7.7](#).

7.3 Tri par sélection

Le tri par sélection est une autre technique simple et répandue. Contrairement au tri par insertion, elle nécessite que tous les enregistrements soient connus. Elle consiste à retourner l'enregistrement avec la plus petite clé, puis celui avec la deuxième plus petite clé, et ainsi de suite. Vous utilisez sans doute cette technique pour trier des cartes lorsque vous tenez toutes vos cartes en main, sauf que vous remplacez les cartes triées directement dans votre main.

L'algorithme ci-dessous formalise la méthode de tri par sélection, mais avec deux raffinements qui permettent de simplifier la procédure et d'éviter d'avoir à

TAB. 7.2 – Exemple de tri par sélection. Le symbole «|» délimite les enregistrements triés et non triés. Les éléments en gras sont les candidats pour le maximum lors de la recherche de droite à gauche.

513	780	321	828	623	080	596	423	380	707	693	766	139	161	316	071
513	780	321	071	623	080	596	423	380	707	693	766	139	161	316	828
513	316	321	071	623	080	596	423	380	707	693	766	139	161	780	828
513	316	321	071	623	080	596	423	380	707	693	161	139	766	780	828
513	316	321	071	623	080	596	423	380	139	693	161	707	766	780	828
513	316	321	071	623	080	596	423	380	139	161	693	707	766	780	828
.....															
071	080	139	161	316	321	380	423	513	596	623	693	707	766	780	828

prévoir une zone tampon pour recevoir les enregistrements triés : les enregistrements sont triés du plus grand au plus petit ; lorsqu'un enregistrement est sélectionné pour occuper sa position finale, il est échangé avec l'enregistrement qui occupe cette position.

Algorithme 7.2 (Tri par sélection). Réorganiser les enregistrements E_1, \dots, E_n de telle sorte qu'en sortie leurs clés soient en ordre croissant, $C_1 \leq \dots \leq C_n$.

1. Répéter les étapes 2 et 3 pour $j = n, n-1, \dots, 2$. [Trier les enregistrements un à un à partir du plus grand.]
2. Rechercher parmi les clés C_j, C_{j-1}, \dots, C_1 pour identifier la plus grande ; posons qu'il s'agit de la clé C_i , où i est la plus grande valeur possible. [Trouver $\max(C_1, \dots, C_j)$.]
3. Échanger $E_i \leftrightarrow E_j$. [Les enregistrements E_j, \dots, E_n occupent maintenant leur position finale.] \square

Le [tableau 7.2](#) illustre la procédure de tri par sélection pour notre exemple de seize nombres, ainsi que la procédure de recherche du maximum à chaque itération.

Le tri par sélection nécessite de répéter à $n-1$ reprises la recherche d'un maximum parmi $n-j$ clés, $j \in \{1, \dots, n-1\}$. L'ordre de grandeur de l'algorithme est donc de nouveau $O(n^2)$.

7.4 Tri par échange

La famille du tri par échange regroupe plusieurs algorithmes qui reposent sur la permutation de paires d'éléments mal ordonnés jusqu'à ce qu'il n'en subsiste aucune. Nous n'étudierons que deux de ces techniques et une seule en détail. Vous trouverez un traitement exhaustif du sujet dans [Knuth \(1997c, section 5.2.2\)](#).

TAB. 7.3 – Exemple de tri à bulles

071	828	828	828	828	828	828	828	828	828	828	828	828	828	828
316	071	780	780	780	780	780	780	780	780	780	780	780	780	780
161	316	071	766	766	766	766	766	766	766	766	766	766	766	766
139	161	316	071	707	707	707	707	707	707	707	707	707	707	707
766	139	161	316	071	693	693	693	693	693	693	693	693	693	693
693	766	139	161	316	071	623	623	623	623	623	623	623	623	623
707	693	766	139	161	316	071	596	596	596	596	596	596	596	596
380	707	693	707	139	161	316	071	513	513	513	513	513	513	513
423	380	707	693	693	139	161	316	071	423	423	423	423	423	423
596	423	380	623	623	139	161	316	071	380	380	380	380	380	380
080	596	423	380	596	596	139	161	071	321	321	321	321	321	321
623	080	596	423	380	513	513	139	316	071	316	316	316	316	316
828	623	080	596	423	380	423	423	161	316	071	161	161	161	161
321	780	623	080	513	423	380	380	139	161	161	071	139	139	139
780	321	513	513	080	321	321	321	321	139	139	139	071	080	080
513	513	321	321	321	080	080	080	080	080	080	080	080	080	071

La technique du tri à bulles consiste à comparer les clés C_1 et C_2 et à permuter les enregistrements E_1 et E_2 s'ils ne sont pas ordonnés, puis à recommencer avec les enregistrements E_2 et E_3 , E_3 et E_4 , etc. Les enregistrements avec les plus grandes clés se déplaceront vers la fin du fichier. En fait, l'enregistrement avec la plus grande clé terminera sa course en position E_n après une première itération de la procédure. On répète ensuite celle-ci jusqu'à ce que tous les enregistrements soient ordonnés.

La [tableau 7.3](#) illustre la procédure du tri à bulles. Il est intéressant de disposer les nombres de notre exemple à la verticale (de bas en haut) afin de voir comment les plus grandes valeurs ont tendance à « remonter à la surface », comme les bulles dans un verre de champagne.

Formalisons l'algorithme du tri à bulles. La principale difficulté consiste à déterminer s'il faut répéter la procédure d'échanges ou non. Dans [Stephens \(2013\)](#), l'algorithme suppose à chaque itération que les enregistrements sont triés, pour ensuite modifier la valeur d'une variable d'état dès qu'un échange est effectué (et ainsi indiquer que le fichier est possiblement non ordonné). La version ci-dessous, adaptée de [Knuth \(1997c\)](#), a plutôt recours à deux variables qui suivent l'indice du plus grand enregistrement non trié.

Algorithme 7.3 (Tri à bulles). Réorganiser les enregistrements E_1, \dots, E_n de telle sorte qu'en sortie leurs clés soient en ordre croissant, $C_1 \leq \dots \leq C_n$.

1. Poser $u \leftarrow n$. [La variable u va contenir l'indice du plus grand enregistrement dont on ignore s'il est ordonné. Une valeur de n indique que nous ne savons encore rien de l'ordre des enregistrements.]

2. Poser $t \leftarrow 0$. Répéter l'étape 3 pour $j = 1, 2, \dots, u - 1$, puis passer à l'étape 4. (Si $u = 1$, passer directement à l'étape 4.) [Trier les enregistrements un à un.]
3. Si $C_j > C_{j+1}$, échanger $E_j \leftrightarrow E_{j+1}$ et poser $t \leftarrow j$. [Échanger les enregistrements jusqu'à « faire monter » le plus grand enregistrement.]
4. Si $t = 0$, terminer l'algorithme. Sinon, poser $u \leftarrow t$ et retourner à l'étape 2. [Soit les enregistrements sont triés, soit il faut reprendre la procédure.] \square

L'exemple de la [figure 7.1](#) pouvait donner l'impression que le tri à bulles — puisque c'est bien celui qui est illustré — est très efficace. Or, il n'en est rien. De manière générale, et surtout lorsque la taille du fichier augmente, ce n'est pas une très bonne technique de tri. Même en éliminant certaines opérations inutiles, le tri à bulles requiert près de deux fois plus d'opérations que le tri par insertion. Outre son nom joliment évocateur, la technique n'a donc à mettre de l'avant que son intérêt didactique.

En revanche, le concept de tri par échange nous permet d'aborder brièvement l'une des techniques de tri les plus rapides. Son auteur, C. A. R. Hoare, l'a d'ailleurs baptisée *quicksort* ([Hoare, 1962](#)). L'idée consiste à choisir un enregistrement, disons E_1 , et à le placer à sa position finale s dans le fichier trié. Durant cette manipulation, les enregistrements sont également réarrangés de telle sorte qu'il ne subsiste aucun enregistrement plus grand à gauche de la position s , et aucun enregistrement plus petit à droite. Le fichier forme dès lors deux partitions et le problème de tri en est réduit à deux problèmes plus petits : trier E_1, \dots, E_{s-1} et trier E_{s+1}, \dots, E_n . Il s'agit ensuite de réutiliser récursivement la technique sur chaque sous-problème jusqu'à ce que le fichier soit entièrement trié.

Il est hors de la portée de cet ouvrage d'étudier en détail l'algorithme *quicksort* ou le choix de la valeur de partition (ou pivot). La [figure 7.3](#) illustre tout de même la technique pour notre exemple de seize nombres. En moyenne, parce que le problème est à chaque étape divisé en deux, l'ordre de grandeur de l'algorithme *quicksort* est $O(n \log n)$. On peut démontrer qu'il s'agit de la borne inférieure pour les opérations de tri. Dans le pire des cas, lorsque les enregistrements sont déjà triés, l'ordre de grandeur de *quicksort* est $O(n^2)$.

7.5 Tri par dénombrement

La dernière technique de tri que nous étudierons repose sur cette idée toute simple : en supposant que toutes les clés sont différentes, la valeur de la clé j dans l'ordre croissant est supérieure à celles de $j - 1$ clés. En d'autres termes, si une clé est supérieure à 8 clés, alors la place de l'enregistrement correspondant dans l'ordre croissant est en position 9.

La technique de tri par dénombrement consiste donc à comparer toutes les paires de clés (K_j, K_i) , $1 \leq j \leq n$, $1 \leq i \leq n$. Plus de la moitié de ces opérations sont toutefois redondantes : il est inutile de comparer une clé avec elle-même et

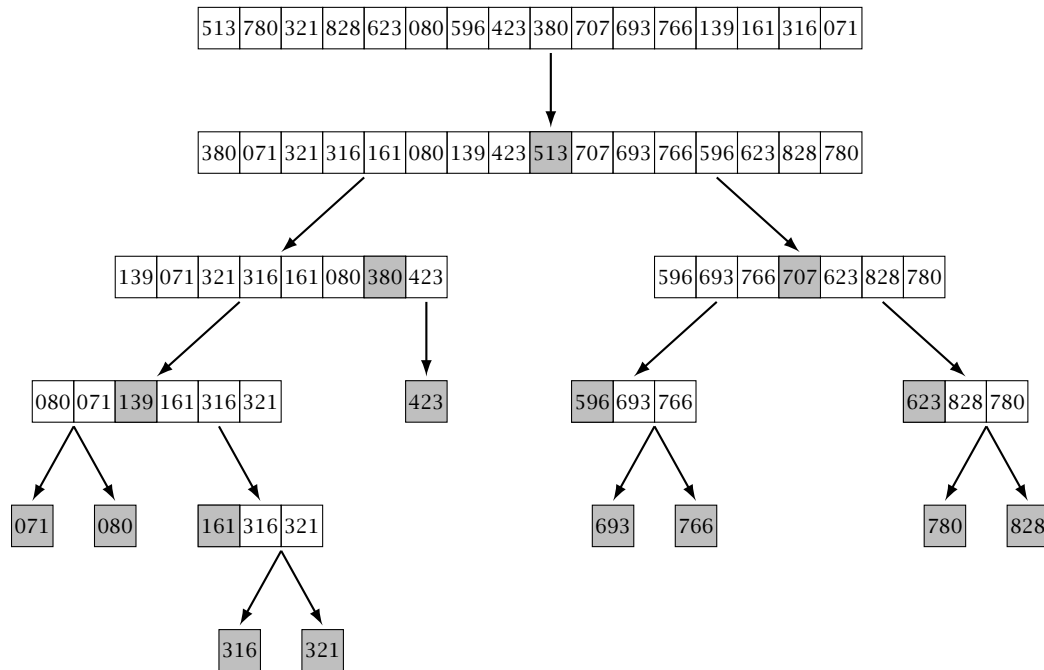


FIG. 7.3 - Illustration de *quicksort*. Les enregistrements dans des cellules grisées sont à leur position dans l'ordre croissant.

il est inutile de comparer K_r à K_s et plus tard K_s à K_r . Il suffit donc de comparer les clés (K_j, K_i) pour $1 \leq j < i$ et $1 < i \leq n$.

Algorithme 7.4 (Tri par dénombrement). Trier les enregistrements E_1, \dots, E_n en fonction de leurs clés C_1, \dots, C_n en créant une liste de compteurs K_1, \dots, K_n du nombre de clés inférieures à une clé donnée. À la fin de l'algorithme, la valeur $K_j + 1$ représente la position de l'enregistrement E_j dans l'ordre croissant.

1. Poser $K_j \leftarrow 0$ pour $j = 1, \dots, n$. [Initialiser les compteurs.]
2. Exécuter l'étape 3 pour $i = n, n-1, \dots, 2$, puis terminer. [Itérer sur tous les enregistrements à partir du dernier.]
3. Exécuter l'étape 4 pour $j = i-1, i-2, \dots, 1$. [Itérer sur tous les enregistrements à la gauche de l'enregistrement i .]
4. Si $C_i < C_j$, poser $K_j \leftarrow K_j + 1$; autrement poser $K_i \leftarrow K_i + 1$. [Incrémenter le compteur K_j ou le compteur K_i .] \square

Remarquez que l'algorithme 7.4 ne trie pas les enregistrements : il produit plutôt une liste qui permet de les trier. Le tableau 7.4 illustre cet algorithme pour notre liste de seize nombres.

Le tri par dénombrement n'est pas une technique particulièrement efficace. Son ordre de grandeur est encore une fois $O(n^2)$. En revanche, une variante s'avère

TAB. 7.4 - Exemple de tri par dénombrement.

	513	780	321	828	623	080	596	423	380	707	693	766	139	161	316	071
K (init.)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K ($i = n$)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
K ($i = n - 1$)	2	2	2	2	2	1	2	2	2	2	2	2	1	1	4	0
K ($i = n - 2$)	3	3	3	3	3	1	3	3	3	3	3	3	1	3	4	0
K ($i = n - 3$)	4	4	4	4	4	1	4	4	4	4	4	4	2	3	4	0
K ($i = n - 4$)	4	5	4	5	4	1	4	4	4	4	4	13	2	3	4	0
K ($i = n - 5$)	4	6	4	6	4	1	4	4	4	5	11	13	2	3	4	0
.....																
K ($i = 2$)	8	14	5	15	10	1	9	7	6	12	11	13	2	3	4	0

très rapide lorsqu'il y a plusieurs enregistrements à trier et que toutes les clés se retrouvent dans un intervalle $u \leq C_j \leq v$, où $v - u$ est petit. Par exemple : trier un million d'entiers entre 1 et 100. Il suffit alors d'un passage à travers le fichier pour compter la fréquence de chaque clé et d'un second passage pour placer les enregistrements en position. L'algorithme suivant décrit en détail cette procédure de tri par dénombrement rapide.

Algorithme 7.5 (Tri par dénombrement rapide). Sous l'hypothèse que toutes les clés sont des entiers et que $u \leq C_j \leq v$, $j \in \{1, \dots, n\}$, trier les enregistrements E_1, \dots, E_n à l'aide de compteurs K_u, \dots, K_v , puis produire une liste S_1, \dots, S_n des enregistrements triés.

1. Poser $K_j \leftarrow 0$ pour $j = u, \dots, v$. [Initialiser les compteurs.]
2. Exécuter l'étape 3 pour $j = 1, \dots, n$, puis passer à l'étape 4. [Itérer sur tous les enregistrements à partir du premier.]
3. Augmenter la valeur de K_{C_j} de 1. [Incrémenter le compteur correspondant à la clé C_j .]
4. (À cette étape, K_i est le nombre de clés égales à i .) Poser $K_i \leftarrow K_i + K_{i-1}$ pour $i = u + 1, \dots, v$. [Effectuer la somme cumulative des compteurs.]
5. (À cette étape, K_i est maintenant le nombre de clés inférieures ou égales à i . En particulier, $K_v = n$.) Exécuter l'étape 6 pour $j = n, n - 1, \dots, 1$, puis terminer l'algorithme. [Itérer sur tous les enregistrements à partir du dernier.]
6. Poser $i \leftarrow K_{C_j}$, $S_i \leftarrow E_j$ et $K_{C_j} \leftarrow i - 1$. [Placer l'enregistrement E_j à sa position dans l'ordre croissant.] \square

L'évaluation de cet algorithme fait l'objet de l'exercice 7.5.

INEFFECTIVE SORTS

```

DEFINE HALFHEARTEDMERGESORT(LIST):
  IF LENGTH(LIST) < 2:
    RETURN LIST
  PIVOT = INT(LENGTH(LIST) / 2)
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
  // UMMMMM
  RETURN [A, B] // HERE. SORRY.

```

```

DEFINE FASTBOGOSORT(LIST):
  // AN OPTIMIZED BOGOSORT
  // RUNS IN O(N LOG N)
  FOR N FROM 1 TO LOG(LENGTH(LIST)):
    SHUFFLE(LIST):
    IF ISSORTED(LIST):
      RETURN LIST
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

```

```

DEFINE JOBINTEVIEWQUICKSORT(LIST):
  OK SO YOU CHOOSE A PIVOT
  THEN DIVIDE THE LIST IN HALF
  FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO, WAIT, IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE EQUAL ONES GO INTO, UH
    THE SECOND LIST FROM BEFORE
    HANG ON, LET ME NAME THE LISTS
    THIS IS LIST A
    THE NEW ONE IS LIST B
    PUT THE BIG ONES INTO LIST B
    NOW TAKE THE SECOND LIST
    CALL IT LIST, UH, A2
    WHICH ONE WAS THE PIVOT IN?
    SCRATCH ALL THAT
    IT JUST RECURSIVELY CALLS ITSELF
    UNTIL BOTH LISTS ARE EMPTY
    RIGHT?
    NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?

```

```

DEFINE PANICSORT(LIST):
  IF ISSORTED(LIST):
    RETURN LIST
  FOR N FROM 1 TO 10000:
    PIVOT = RANDOM(0, LENGTH(LIST))
    LIST = LIST[:PIVOT] + LIST[PIVOT:]
    IF ISSORTED(LIST):
      RETURN LIST
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING
    RETURN LIST
  IF ISSORTED(LIST): // COME ON COME ON
    RETURN LIST
  // OH JEEZ
  // I'M GONNA BE IN SO MUCH TROUBLE
  LIST = [ ]
  SYSTEM("SHUTDOWN -h +5")
  SYSTEM("RM -RF ./")
  SYSTEM("RM -RF ~/*")
  SYSTEM("RM -RF /")
  SYSTEM("RD /S /Q C:\*") // PORTABILITY
  RETURN [1, 2, 3, 4, 5]

```

Tiré de [XKCD.com](http://xkcd.com)

7.6 Recherche

Nous allons consacrer passablement moins de temps aux algorithmes de recherche qu'aux algorithmes de tri. D'une part, c'est un sujet qui a historiquement reçu moins d'attention des chercheurs ; voir l'introduction du chapitre 6 de [Knuth \(1997c\)](#). D'autre part, les algorithmes véritablement intéressants reposent sur la recherche dans des données déjà triées et ils tombent hors de la portée du présent ouvrage, pour plusieurs parce qu'ils reposent sur des structures de données que nous n'avons pas abordées (arbres, piles).

Je vous propose quand même un aperçu de deux méthodes de recherche dans des données triées en faisant uniquement appel à votre expérience et à votre intuition. En premier lieu, imaginez devoir chercher la valeur 1,645 dans une table sans avoir d'idée a priori à quel endroit elle peut s'y trouver. Plutôt que d'examiner les valeurs une à une en partant de la première, vous irez peut-être examiner la valeur qui se trouve au milieu de la table. Selon que celle-ci est plus grande ou plus petite que 1,645, vous poursuivrez votre recherche dans la première ou la se-

conde moitié de la table. Vous pouvez ensuite répéter le processus autant de fois que nécessaire⁴. C'est l'algorithme de recherche dichotomique ou par bisection (*binary search*). L'ordre de grandeur de la recherche dichotomique est $O(\log n)$.

Si vous cherchez plutôt le mot *parapluie* dans un dictionnaire, vous ne débutez sans doute pas votre recherche au milieu du dictionnaire. Sachant que *p* est la seizième lettre de l'alphabet, vous ouvrirez d'abord le dictionnaire un peu après la moitié. Une fois arrivé dans les *p*, vous vous dirigerez immédiatement vers le début puisque la seconde lettre est un *a*, mais vers la fin des *pa* puisque la troisième lettre est un *r*, et ainsi de suite. Cet algorithme qui estime à quel endroit se trouve la cible est la recherche par interpolation. L'ordre de grandeur de la recherche par interpolation est $O(\log(\log n))$, mais le temps de calcul plus grand qu'elle requiert fait en sorte que l'algorithme n'est véritablement plus efficace que la bisection que pour des très grandes bases de données.

Le seul algorithme de recherche que nous étudierons un peu plus en profondeur est à la fois le plus simple et le seul qui convient pour des données non triées. Dans un tel contexte, on ne peut faire beaucoup mieux que débiter la recherche au début du fichier et continuer jusqu'à ce que le bon enregistrement ait été trouvé. C'est la recherche séquentielle ou linéaire. L'algorithme est très simple.

Algorithme 7.6 (Recherche séquentielle). Rechercher parmi les enregistrements E_1, \dots, E_n , dont les clés sont C_1, \dots, C_n , celui pour lequel la clé est C . On suppose que $n \geq 1$.

1. Poser $i \leftarrow 1$. [Initialiser le compteur.]
2. Si $C_i = C$, retourner la valeur i . [Comparer les clés une à une.]
3. Poser $i \leftarrow i + 1$. [Incrémenter le compteur i .]
4. Si $i \leq n$, retourner à l'étape 2. Sinon, l'algorithme est terminé et la recherche a échoué. [Vérifier si la fin du fichier est atteinte.] □

Assez intuitivement, la recherche dans un fichier deux fois plus grand prendra en moyenne deux fois plus de temps. L'ordre de grandeur moyen de l'algorithme de recherche séquentielle est donc $O(n)$.

Comme le rapporte Knuth (1997c, section 6.1), une toute petite modification de l'algorithme 7.6 permet toutefois d'en améliorer la performance. L'idée consiste à réduire les deux tests de la boucle principale (étapes 2 et 4) à un seul. Il s'agit là d'une stratégie d'optimisation applicable dans plusieurs situations.

Algorithme 7.7 (Recherche séquentielle rapide). Rechercher parmi les enregistrements E_1, \dots, E_n , dont les clés sont C_1, \dots, C_n , celui pour lequel la clé est C . On suppose la présence d'un enregistrement fictif E_{n+1} à la fin du fichier.

1. Poser $i \leftarrow 1$ et $C_{n+1} \leftarrow C$. [Initialiser le compteur i et la clé de l'enregistrement fictif.]

4. De ce fait l'algorithme devient récursif!

2. Si $C_i = C$, aller à l'étape 4. [Comparer les clés une à une.]
3. Poser $i \leftarrow i + 1$ et retourner à l'étape 2. [Incrémenter le compteur.]
4. Si $i \leq n$, retourner la valeur i . Sinon, l'algorithme est terminé et la recherche a échoué. [Vérifier si la fin du fichier est atteinte.] \square

Vous doutez du gain d'efficacité? Examinons le nombre d'opérations nécessaires dans chaque algorithme pour trouver la position de l'enregistrement 139 dans la liste de seize nombre que nous utilisons depuis le début du chapitre. En ne tenant pas compte de l'étape 1 d'initialisation, chaque essai infructueux nécessite trois opérations dans l'[algorithme 7.6](#) : la comparaison de l'étape 2 ; l'incréméntation du compteur de l'étape 3 ; la vérification de l'étape 4. La découverte de la valeur recherchée n'exige quant à elle que la comparaison de l'étape 2. La valeur 139 étant la treizième de la liste, la recherche aura nécessité $3 \times 12 + 1 = 37$ opérations.

Dans l'[algorithme 7.7](#), chaque essai infructueux ne requiert que deux opérations : la comparaison de l'étape 2 et l'incréméntation du compteur de l'étape 3. L'identification de la valeur recherchée nécessite toutefois deux opérations : la comparaison de l'étape 2 et celle de l'étape 4. Au final, le nombre d'opérations n'en est pas moins réduit à $2 \times 12 + 2 = 26$, un gain de près de 30 %. Pas mal!

Terminons avec une variante de l'[algorithme 7.7](#) pour des données triées, c'est-à-dire dont les clés se trouvent en ordre croissant.

Algorithme 7.8 (Recherche séquentielle pour données triées). Rechercher parmi les enregistrements E_1, \dots, E_n , dont les clés sont en ordre croissant $C_1 < \dots < C_n$, celui pour lequel la clé est C . On suppose la présence d'un enregistrement fictif E_{n+1} à la fin du fichier dont la clé est $C_{n+1} = \infty > C$.

1. Poser $i \leftarrow 1$. [Initialiser le compteur i .]
2. Si $C_i \geq C$, aller à l'étape 4. [Comparer les clés une à une.]
3. Poser $i \leftarrow i + 1$ et aller à l'étape 2. [Incrémenter le compteur.]
4. Si $C = C_i$, retourner la valeur i . Sinon, l'algorithme est terminé et la recherche a échoué. [Vérifier si la fin du fichier est atteinte.] \square

L'évaluation de cet algorithme fait l'objet de l'[exercice 7.7](#).

Comme le tri par insertion, l'algorithme de recherche séquentielle fait l'objet d'une mise en œuvre en R dans le code informatique de la [section 7.9](#) (lignes 229-257).

7.7 Boucles itératives

Comme nous l'avons déjà vu, il existe deux grandes manières de répéter des calculs en programmation : par itération ou par récursion. La récursion est très simple à mettre en œuvre dans un langage fonctionnel comme R puisqu'il suffit

de faire en sorte qu'une fonction s'invoque elle-même, dans R idéalement avec la fonction `Recall` (section 4.6).

Une procédure itérative répète un bloc d'instructions, habituellement avec un léger changement d'état, jusqu'à ce qu'un nombre de répétitions soit atteint ou qu'une condition soit satisfaite. C'est ce que l'on appelle communément une boucle (*loop*). La présente section étudie les structures de contrôles itératives de R. Elles sont de trois types : la boucle à contrôle par dénombrement « pour » (boucle `for`), la boucle à précondition « tant que » (boucle `while`) et la boucle à condition d'arrêt « jusqu'à » (boucle `repeat`).



Les boucles sont beaucoup moins utilisées dans R que dans la plupart des autres langages de programmation. Avant d'utiliser une boucle, demandez-vous si l'arithmétique vectorielle ou une fonction d'application ne pourrait faire le travail de manière plus claire et plus efficace.

7.7.1 Boucle à dénombrement

La boucle à dénombrement `for` sert à répéter une procédure un nombre prédéterminé de fois. La syntaxe de la boucle `for` est la suivante dans R :

```
for (<variable> in <suite>)  
  <expression>
```

- ▶ `<variable>` est un compteur ou, plus spécifiquement, un itérateur qui prend successivement les valeurs contenues dans `<suite>`. La variable est habituellement — mais pas nécessairement — utilisée dans les calculs à l'intérieur de la boucle.
- ▶ `<suite>` est une expression dont le résultat est un vecteur (y compris une liste). À noter que le résultat n'a pas à être composé de nombres consécutifs, ni même de nombres, en fait.
- ▶ `<expression>` est le contenu de la boucle. L'itérateur `<variable>` apparaît habituellement dans `<expression>`. Comme toujours, si le contenu est constitué de plusieurs expressions, elles doivent être regroupées dans des accolades `{ }`.

La fonction ci-dessous est une mise en œuvre de l'algorithme 2.5 du calcul de la factorielle qui utilise une boucle `for`. Remarquez comment la boucle n'est exécutée que pour $n \geq 2$ afin d'éviter d'effectuer le produit 1×1 dans le calcul de $1!$.

```
factorial <- function(n)  
{  
  p <- 1  
  for (i in seq.int(from = 2, length.out = n - 1))  
    p <- p * i  
}
```

```
p
}
```

7.7.2 Boucle à précondition

La boucle à précondition `while` exécute une procédure tant qu'une condition est satisfaite. La condition étant vérifiée avant d'entrer dans la boucle, celle-ci peut ne jamais s'exécuter. La syntaxe de la boucle `while` est la suivante :

```
while (<condition>)
  <expression>
```

- ▶ *<condition>* est une expression dont le résultat est une valeur TRUE ou FALSE unique.
- ▶ *<expression>* est une expression ou un groupe d'expressions entre accolades { } qui sont exécutées tant que la *<condition>* est TRUE.

Vous avez déjà rencontré la structure « tant que » dans la version en pseudo-code de l'algorithme d'Euclide ([algorithme 2.2](#)). Voici la mise en œuvre correspondante avec une boucle `while`.

```
PGCD <- function(m, n)
{
  while (n != 0)
  {
    r <- m %% n
    m <- n
    n <- r
  }
  m
}
```

7.7.3 Boucle à condition d'arrêt

Plusieurs langages de programmation comportent une structure itérative à postcondition « répéter, tant que » (boucle `do ... while`) ou « répéter, jusqu'à ce que » (boucle `do ... until`), voire les deux. Dans R, la boucle à postcondition prend plutôt la forme d'une boucle à condition d'arrêt `repeat`. La syntaxe de cette boucle est on ne peut plus simple :

```
repeat
  <expression>
```

- ▶ *<expression>* est l'expression ou le groupe d'expressions à répéter. En pratique, pour éviter que la boucle ne se répète indéfiniment, *<expression>* est toujours un groupe d'expressions comportant un test d'arrêt.

Le test d'arrêt d'une boucle `repeat` utilise habituellement la commande `break` ([section 7.7.4](#)) pour forcer la sortie de la boucle. Puisque le test se trouve à l'intérieur de la boucle — généralement à ou vers la fin —, une boucle `repeat` est toujours exécutée au moins une fois. C'est d'ailleurs la grande différence entre une boucle `while` et une boucle `repeat`.

Le fichier de script du [chapitre 6](#) propose une fonction `sqrt` pour calculer la racine carrée d'un nombre par approximations successives. Elle a recours à une boucle à précondition `while`. Voici une variante qui utilise plutôt une boucle à critère d'arrêt `repeat`.

```
sqrt <- function(x)
{
  y <- 1
  repeat
  {
    y <- (y + x/y)/2
    if (abs(y^2 - x) < 0.001)
      break
  }
  y
}
```

7.7.4 Contrôle du flux

Les commandes de contrôle du flux permettent d'infléchir le déroulement normal d'une boucle.

break force la sortie de la boucle courante. La commande `break` peut être utilisée dans les boucles `for` ou `while`, mais elle est presque indissociable de la boucle `repeat`, tel qu'illustré à la [section 7.7.3](#).

next force le passage à la prochaine itération de la boucle `for`, `while` ou `repeat`. Comme `break`, la commande est généralement utilisée à l'intérieur d'une structure conditionnelle.

7.7.5 Syndrome de la plaque à biscuits

Il arrive souvent que les fonctions itératives calculent un à un les éléments d'un vecteur. Une erreur fréquente consiste à utiliser une construction comme celle-ci pour effectuer les calculs.

```
...
x <- numeric(0)
for (i in seq_len(n))
  x <- c(x, calcul)
...
```

Le code ci-dessus est tout à fait valide. Pourquoi parler d'une erreur, alors ? Parce qu'il souffre du gros défaut suivant : la taille du vecteur doit constamment augmenter pour stocker un résultat additionnel.

Tentons une (autre) analogie alimentaire pour cette manière de procéder. Pour ranger dans une boîte des biscuits frais sortis du four, vous prenez un premier biscuit et vous le rangez dans une boîte ne pouvant contenir qu'un seul biscuit. Au second biscuit, constatant que le contenant n'est pas assez grand, vous sortez une boîte pouvant contenir deux biscuits, vous changez le premier biscuit de boîte et vous y rangez aussi le second biscuit. 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 je nomme, non sans un sourire en coin, le Syndrome de la plaque à biscuits™.

Le manège décrit ci-dessus se reproduit à l'identique dans la mémoire de l'ordinateur, l'odeur des biscuits chauds en moins. En effet, pour ajouter des éléments à un vecteur, l'ordinateur doit allouer de la nouvelle mémoire et déplacer les termes déjà sauvegardés avant de pouvoir sauvegarder un terme additionnel. Vous aurez compris que vous devriez absolument éviter de procéder ainsi lorsque c'est possible — et ça l'est souvent.

Quand vous savez quelle sera la longueur finale d'un objet, il vaut mieux créer un contenant vide de la bonne longueur et le remplir par la suite avec une construction comme ci-dessous.

```
...
x <- numeric(n)
for (i in seq_len(n))
  x[i] <- calcul
...
```



Les lignes 12-308 du fichier de script `tri.R` reproduit à la [section 7.9](#) proposent un exemple additionnel de fonction ayant recours à une boucle itérative, ainsi que des mises en œuvre des algorithmes de tri par insertion et de recherche séquentielle.

7.8 Fonctions internes utiles

Le tri et la recherche sont des opérations de base dans l'analyse de données. Il ne faut donc pas se surprendre que R comporte des fonctions internes pour ces opérations.

7.8.1 Tri et opérations apparentées

`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 vecteur renversé.

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

unique éléments uniques d'un vecteur.

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



Visionnez la [vidéo sur la fonction order](#) qui explique plus en détail les différences entre les fonctions `sort`, `order` et `rank`.

7.8.2 Recherche

Les exemples de cette sous-section utilisent le vecteur suivant.

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

match indice de la première occurrence d'un élément.

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

%in% appartenance d'une ou plusieurs valeurs au vecteur.

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

which indices des éléments satisfaisant la condition en argument (ou des valeurs TRUE dans le vecteur).

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

`which.min` indice du minimum.

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

`which.max` indice du maximum.

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



Vous trouverez quelques exemples additionnels d'utilisation des fonctions présentées dans cette section aux lignes 311-366 du fichier de script `tri.R`.

7.9 Exemples

📍 Fichier d'accompagnement `tri.R`

```
11 ###
12 ### BOUCLES ITÉRATIVES
13 ###
14
15 ## BOUCLE À DÉNOMBREMENT
16
17 ## Débutons par illustrer les boucles à dénombrement avec des boucles
18 ## triviales qui ne font qu'afficher des valeurs à l'écran.
19 ##
20 ## La syntaxe de la déclaration d'une boucle 'for' dans R est la
21 ## suivante:
22 ##
23 ##   for(<variable> in <suite>)
24 ##
25 ## La <variable> est un compteur (ou itérateur) et <suite> est une
26 ## expression qui permet de créer un vecteur (ou une liste) des
27 ## valeurs successives du compteur.
28 ##
29 ## Plus souvent qu'autrement, l'expression fait appel à une fonction
30 ## de génération de suites de valeurs.
31 for (n in 1:10)
32   print(n)
33
34 ## Je recommande d'utiliser les fonctions 'seq_len' et 'seq_along', ou
35 ## alors 'seq' avec l'argument 'length.out', afin de vous prémunir
```

```
36 ## contre le risque de générer une suite non vide alors le nombre
37 ## d'itérations de la boucle devrait être zéro.
38 ##
39 ## Voici un exemple où les deux approches sont équivalentes.
40 n <- 5
41 for (i in 1:n) print(i)          # approche avec ':'
42 for (i in seq_len(n)) print(i)  # approche avec 'seq_len'
43
44 ## Si le nombre d'itérations devait toutefois être nul, l'approche
45 ## avec ':' ne donne pas le résultat escompté; celle avec 'seq_len',
46 ## oui.
47 n <- 0
48 for (i in 1:n) print(i)          # deux itérations!
49 for (i in seq_len(n)) print(i)  # aucune itération
50
51 ## De manière équivalente, s'il faut répéter une boucle un nombre de
52 ## fois égal à la longueur d'un vecteur déjà connu, la fonction
53 ## 'seq_along' permet de générer la suite de valeurs de manière
54 ## robuste.
55 x <- c(5, 32, 57, 42, 0)         # vecteur de longueur 5...
56 for (i in seq_along(x)) print(i) # ... 5 itérations
57 x <- numeric(0)                 # vecteur vide...
58 for (i in seq_along(x)) print(i) # ... 0 itération
59
60 ## Il y a une petite subtilité avec les boucles 'for' à laquelle vous
61 ## devez faire bien attention: la classe du vecteur (ou de la liste)
62 ## créé par l'expression n'est PAS prise en compte.
63 ##
64 ## Vous rencontrerez ce cas si, par exemple, vous souhaitez itérer sur
65 ## un vecteur de dates. Rappelez-vous: une date dans R n'est qu'un
66 ## nombre entier «maquillé» pour être plus lisible lorsque présenté à
67 ## l'écran. (Ce «maquillage» est fourni par la classe de l'objet.)
68 dates <- seq(as.Date("2042-05-09"), by = "+1 month",
69             length.out = 6)
70 dates          # objet maquillé
71 unclass(dates) # sans classe, sans maquillage
72
73 ## En itérant sur un vecteur de dates, la classe (le maquillage) ne
74 ## suit pas dans la boucle.
75 for (d in dates)
76   print(d)
77
78 ## Si vous devez utiliser les dates successives sous forme de chaînes
79 ## de caractères à l'intérieur de la boucle, vous pouvez soit indiquer
80 ## le vecteur de dates dans la boucle, soit «cacher» les dates à
81 ## l'intérieur d'une liste.
82 for (i in seq_along(dates))
83   print(dates[i])
```

```

84 for (d in as.list(dates))
85     print(d)
86
87 ## Passons maintenant aux boucles 'while' et 'repeat'.
88 ##
89 ## Nous allons illustrer leur utilisation avec la méthode numérique du
90 ## point fixe. On dit qu'une valeur x est un «point fixe» d'une
91 ## fonction f si cette valeur satisfait l'équation
92 ##
93 ##   x = f(x).
94 ##
95 ## La méthode numérique de recherche du point fixe d'une fonction f
96 ## est simple et puissante: elle consiste à choisir une valeur de
97 ## départ, puis à évaluer successivement f(x), f(f(x)), f(f(f(x))),
98 ## ... jusqu'à ce que la valeur change «peu».
99 ##
100 ## L'algorithme est donc très simple:
101 ##
102 ## 1. Choisir une valeur de départ y.
103 ## 2. Calculer x = f(y)
104 ## 3. Si |x - y|/|x| >= e, poser y <- x et retourner à
105 ##   l'étape 2.
106 ## 4. Retourner x.
107 ##
108 ## Avant de poursuivre votre lecture, tentez d'identifier le meilleur
109 ## type de boucle ('for', 'while' ou 'repeat') à utiliser pour
110 ## programmer cet algorithme.
111
112 ## La méthode de Newton du calcul de la racine carrée par
113 ## approximations successives est un cas spécial de la méthode du
114 ## point fixe. En effet, la racine carrée d'un nombre est la valeur
115 ## positive de y satisfaisant l'équation y^2 = x. Cette équation peut
116 ## se réécrire sous forme de point fixe ainsi:
117 ##
118 ##   y = (y + x/y)/2.
119 ##
120 ## Voici une nouvelle mise en oeuvre de la fonction 'sqrt' qui utilise
121 ## la méthode du point fixe. Le critère d'arrêt y est exprimé non plus
122 ## en fonction de l'écart entre 'y'^2 et 'x', mais plutôt en fonction
123 ## de l'écart entre deux approximations successives. De plus, la
124 ## valeur de départ et l'erreur d'approximation sont passées en
125 ## argument à la fonction.
126 ##
127 ## Puisqu'il faut au minimum vérifier si la valeur initiale est un
128 ## point fixe, nous utilisons une boucle 'repeat'.
129 sqrt <- function(x, start = 1, TOL = 1E-10)
130 {
131     repeat

```

```
132     {
133         y <- (start + x/start)/2
134         if (abs(y - start)/y < TOL)
135             break
136         start <- y
137     }
138     y
139 }
140
141 ## Vérifions la validité de la fonction.
142 sqrt(9, 1)
143 sqrt(225, 1)
144 sqrt(3047, 50)
145
146 ## Formidable. Toutefois, si nous voulions utiliser la méthode du
147 ## point fixe pour résoudre une autre équation, il faudrait écrire une
148 ## nouvelle fonction qui serait pour l'essentiel identique, sinon pour
149 ## le calcul de la fonction (mathématique) f(x) pour laquelle nous
150 ## cherchons le point fixe.
151 ##
152 ## Créons donc une fonction de point fixe générale qui prendra la
153 ## fonction mathématique f(x) en argument.
154 fixed_point <- function(FUN, start, TOL = 1E-10)
155 {
156     repeat
157     {
158         x <- FUN(start)
159         if (abs(x - start)/x < TOL)
160             break
161         start <- x
162     }
163     x
164 }
165
166 ## Nous pouvons ensuite écrire une nouvelle fonction 'sqrt' qui
167 ## utilise 'fixed_point'. Nous y ajoutons un test de validité de
168 ## l'argument, pour faire bonne mesure.
169 sqrt <- function(x)
170 {
171     if (x < 0)
172         stop("cannot compute square root of negative value")
173
174     fixed_point(function(y) (y + x/y)/2, start = 1)
175 }
176
177 ## Validation. Nous obtenons les mêmes résultats que précédemment.
178 sqrt(9)
179 sqrt(25)
```

```

180 sqrt(3047)
181
182 ## Suppression de la fonction pour éviter qu'elle n'entre en conflit
183 ## avec celle de R.
184 rm("sqrt")
185
186 ## TRI PAR INSERTION
187
188 ## Voici une mise en oeuvre de l'algorithme de tri par insertion. Dans
189 ## le contexte de R, les enregistrements et les clés sont identiques:
190 ## il s'agit des éléments du vecteur à trier.
191 ##
192 ## La mise en oeuvre fait appel à une boucle à dénombrement 'for' et à
193 ## une boucle à précondition 'while'. La première sert à itérer sur
194 ## tous les éléments du vecteur à partir du deuxième (nous savons
195 ## combien de fois répéter la boucle). La seconde sert lors de la
196 ## recherche de la position d'un élément dans la liste des éléments
197 ## triés (nous ne savons pas d'avance combien de fois il faudra
198 ## répéter cette étape).
199 ##
200 ## Portez une attention particulière à l'expression qui sert à générer
201 ## la suite dans la boucle 'for'. Pour un vecteur de longueur 1, la
202 ## suite est un vecteur vide et la boucle n'est pas exécutée, comme il
203 ## se doit: un vecteur de longueur 1 est d'office trié. (La fonction
204 ## 'seq.int' est une variante moins générale, mais beaucoup plus
205 ## rapide, de 'seq'.)
206 isort <- function(x)
207 {
208   for (j in seq.int(from = 2, length.out = length(x) - 1))
209   {
210     xj <- x[j]
211     i <- j - 1
212     while (x[i] > xj)
213     {
214       x[i + 1] <- x[i]
215       i <- i - 1
216       if (i == 0)
217         break
218     }
219     x[i + 1] <- xj
220   }
221   x
222 }
223
224 ## Testons notre fonction avec l'exemple du chapitre.
225 x <- c(513, 780, 321, 828, 623, 80, 596, 423,
226       380, 707, 693, 766, 139, 161, 316, 71)
227 isort(x)

```



```
228
229 ## RECHERCHE SÉQUENTIELLE
230
231 ## La fonction 'linsearch' ci-dessous (du nom usuel de l'algorithme en
232 ## anglais, «linear search») est une mise en oeuvre de l'algorithme de
233 ## recherche séquentielle. Elle retourne l'indice de la valeur 'x'
234 ## dans le vecteur 'table' (les noms des arguments sont ceux de la
235 ## fonction analogue 'match') et NA en cas d'échec, comme c'est
236 ## généralement l'usage dans R.
237 ##
238 ## ATTENTION: cette fonction n'est pas vectorielle pour l'argument
239 ## 'x'. (Quels changements faudrait-il apporter à la fonction pour
240 ## qu'elle le soit?)
241 linsearch <- function(x, table)
242 {
243     n <- length(table)
244     i <- 1
245     while (x != table[i])
246     {
247         i <- i + 1
248         if (i == n)
249             return(NA)
250     }
251     i
252 }
253
254 ## Tests avec l'exemple du chapitre.
255 linsearch(139, x)
256 linsearch(513, x)
257 linsearch(42, x)
258
259 ## SYNDROME DE LA PLAQUE À BISCUITS
260
261 ## La fonction ci-dessous calcule les 'nterm' premiers termes de la
262 ## suite de Fibonacci. Elle souffre toutefois du Syndrome de la plaque
263 ## à biscuits. (Identifiez pourquoi.)
264 fibonacci0 <- function(nterm)
265 {
266     if (nterm < 1)
267         stop("'nterm' doit être supérieur ou égal à 1")
268     if (nterm == 1)
269         return(0)
270     x <- c(0, 1)
271     for (i in seq_len(nterm - 2))
272         x[i + 2] <- x[i + 1] + x[i]
273     x
274 }
275
```

```

276 ## Validation de la fonction
277 fibonacci0(1)
278 fibonacci0(2)
279 fibonacci0(5)
280
281 ## Une seconde version de la fonction prend garde de d'abord créer un
282 ## vecteur de la bonne longueur pour stocker tous les résultats, puis
283 ## de le remplir graduellement. (Le premier terme du vecteur est déjà
284 ## 0 suite à l'initialisation avec 'numeric'.)
285 fibonacci <- function(nterm)
286 {
287     if (nterm < 1)
288         stop("'nterm' doit être supérieur ou égal à 1")
289     if (nterm == 1)
290         return(0)
291     x <- numeric(nterm)
292     x[2] <- 1
293     for (i in seq_len(nterm - 2))
294         x[i + 2] <- x[i + 1] + x[i]
295     x
296 }
297
298 ## Validation de la fonction
299 fibonacci(1)
300 fibonacci(2)
301 fibonacci(5)
302
303 ## Avons-nous vraiment gagné en efficacité? En comparant le temps
304 ## requis pour calculer plusieurs valeurs de la suite de Fibonacci
305 ## pour chaque fonction, vous pourrez constater que la seconde version
306 ## est entre trois et quatre fois plus rapide!
307 system.time(fibonacci0(1e6))
308 system.time(fibonacci(1e6))
309
310 ###
311 ### FONCTIONS INTERNES UTILES
312 ###
313
314 ## On se donne un vecteur non ordonné pour les exemples qui suivent.
315 x <- c(50, 30, 10, 20, 60, 30, 20, 40)
316
317 ## TRI ET OPÉRATIONS APPARENTÉES
318
319 ## Classement en ordre croissant ou décroissant.
320 sort(x) # classement en ordre croissant
321 sort(x, decr = TRUE) # classement en ordre décroissant
322 sort(c("abc", "B", "Aunt", "Jemima")) # chaînes de caractères
323 sort(c(TRUE, FALSE)) # FALSE vient avant TRUE

```

```

324
325 ## La fonction 'order' retourne l'indice, dans le vecteur donné donné
326 ## en argument, du premier élément selon l'ordre croissant, puis du
327 ## deuxième, etc. Autrement dit, on obtient l'ordre dans lequel il
328 ## faut extraire les données du vecteur pour les obtenir en ordre
329 ## croissant.
330 order(x)                # regarder dans le blanc des yeux
331 x[order(x)]             # équivalent à 'sort(x)'
332
333 ## Rang des éléments d'un vecteur dans l'ordre croissant.
334 rank(x)                 # rang des élément de 'x'
335
336 ## Renverser l'ordre d'un vecteur.
337 rev(x)
338
339 ## Seulement les éléments différents d'un vecteur.
340 unique(x)
341
342 ## RECHERCHE D'ÉLÉMENTS DANS UN VECTEUR
343
344 ## La fonction de base pour rechercher un ou plusieurs éléments dans
345 ## un vecteur est 'match'.
346 match(20, x)             # indice du premier 20 dans 'x'
347 match(c(20, 30), x)      # aussi pour plusieurs valeurs
348
349 ## Par défaut, la fonction retourne NA lorsque la recherche échoue.
350 ## L'argument 'nomatch' permet de retourner une autre valeur,
351 ## habituellement 0.
352 match(42, x)             # élément absent du vecteur
353 match(42, x, nomatch = 0) # parfois utile, voir ci-dessous
354
355 ## L'opérateur %in%, basé sur 'match', est utile pour déterminer si un
356 ## élément se trouve dans un vecteur.
357 60 %in% x                 # 60 appartient à 'x'
358 70 %in% x                 # 70 n'appartient pas à 'x'
359 match(60, x, nomatch = 0) > 0 # équivalent, mais moins lisible
360 match(70, x, nomatch = 0) > 0 # idem
361
362 ## La fonction 'which' et ses variantes permettent d'identifier les
363 ## indices des éléments qui satisfont une condition.
364 which(x >= 30)             # indices des éléments >= 30
365 which.min(x)              # indice du minimum
366 which.max(x)              # indice du maximum

```

7.10 Exercices

7.1 Effectuer une mise en œuvre dans R de l’[algorithme 7.2](#) de tri par sélection.

- 7.2** Effectuer une mise en œuvre dans R de l'[algorithme 7.3](#) de tri à bulles.
- 7.3** Effectuer une mise en œuvre dans R de l'[algorithme 7.4](#) de tri par dénombrement.
- 7.4** a) Déterminer le lien entre les résultats de l'algorithme de tri par dénombrement et ceux de la fonction `rank` de R, notamment en ce qui a trait à son argument `ties.method`.
b) Déterminer une expression R pour trier un vecteur à partir du résultat de la fonction `rank`.
- 7.5** Évaluer l'[algorithme 7.5](#) de tri par dénombrement rapide étape par étape à la main pour les enregistrements suivants.
- a) 3, 4, 2, 1, 4, 1, 4, 4, 4, 2, 3, 4
b) 5T, 0C, 5U, 0D, 9., 1N, 8S, 2R, 6A, 4A, 1G, 5L, 6T, 6I, 7O, 7N (utiliser la valeur numérique de chaque enregistrement comme clé de tri; adapté de [Knuth \(1997c\)](#))
- 7.6** Effectuer une mise en œuvre dans R de l'[algorithme 7.7](#) de recherche séquentielle rapide. À l'aide de la fonction `system.time`, comparer ensuite sa performance à celle de la fonction `linsearch` du code informatique de la [section 7.9](#), ainsi qu'à celle de la fonction interne `match`.
- 7.7** Évaluer l'[algorithme 7.8](#) de recherche séquentielle pour données triées pour identifier la position de la valeur 316 dans la liste des seize nombres classés en ordre croissant :

071	080	139	161	316	321	380	423
513	596	623	693	707	766	780	828

- 7.8** Le [tableau 7.5](#) contient la date et le temps des 31 meilleurs résultats enregistrés au 100 mètres homme entre 1964 et 2005. Le fichier `100metres.rds` livré avec cet ouvrage contient ces données sous forme de vecteur étiqueté. Les données se trouvent dans un format binaire propre à R. L'expression ci-dessous permet de les importer facilement dans R :

```
> x <- readRDS("100metres.rds")
```

Proposer une expression R pour extraire les dates des records du monde, c'est-à-dire la première fois que chaque temps a été enregistré.

TAB. 7.5 - Meilleurs temps au 100 mètres homme entre 1964 et 2005

Date	Temps (sec.)
1964-10-15	10,06
1968-06-20	10,03
1968-10-13	10,02
1968-10-14	9,95
1968-10-14	10,04
1968-10-14	10,07
1968-10-14	10,08
1975-08-20	10,05
1977-08-11	9,98
1978-07-30	10,09
1979-09-04	10,01
1981-05-16	10,00
1983-05-14	9,97
1983-07-03	9,93
1984-05-05	9,96
1984-05-06	9,99
1988-09-24	9,92
1989-06-16	9,94
1991-06-14	9,90
1991-08-25	9,86
1991-08-25	9,88
1993-08-15	9,87
1994-07-06	9,85
1994-08-23	9,91
1996-07-27	9,84
1996-07-27	9,89
1999-06-16	9,79
1999-08-22	9,80
2001-08-05	9,82
2002-09-14	9,78
2005-06-14	9,77

8 Débogage

Objectifs du chapitre

- ▶ Repérer et corriger des erreurs de programmation, syntaxiques et logiques.
- ▶ Utiliser les outils de débogage de R.

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. En fait, il n'est pas rare que le débogage et les tests de validité accaparent la plus grande part du temps de développement d'un projet.

Les techniques de débogage les plus simples et naïves sont parfois les plus efficaces et certainement les plus faciles à apprendre. Lorsqu'elles ne suffisent pas à la tâche, les outils de débogage de R peuvent accélérer la résolution des problèmes. Je présente dans les sections suivantes trois techniques de débogage que j'utilise régulièrement, de la plus simple à la plus puissante.

Au cours du chapitre, nous allons tâcher de déboguer une fonction de simulation de nombres aléatoires issus de la distribution gamma avec fonction de densité de probabilité

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

basée sur l'algorithme d'acceptation-rejet suivant.

Algorithme 8.1. Simuler un nombre issu d'une distribution gamma de paramètres $0 < \alpha < 1$ et $\lambda > 0$.

1. Simuler u, v d'une distribution uniforme sur $(0, 1)$.
2. Calculer $y = G^{-1}(u)$, où

$$G^{-1}(x) = \begin{cases} \left(\frac{\alpha + e}{e} x \right)^{1/\alpha}, & 0 \leq x \leq e/(\alpha + e) \\ -\ln[((1/\alpha) + (1/e))(1 - x)], & e/(\alpha + e) < x \leq 1. \end{cases}$$

```

1  rgamma <- function(n, shape, rate = 1; scale = 1/rate)
2  {
3      if (shape <= 0 | shape >= 1)
4          stop("valeur de shape inadmissible")
5
6      ratio <- function(x)
7          if (x <= 1) exp(-x) else x^(shape - 1)
8
9      Ginv <- function(x)
10     {
11         k <- 1 + shape * exp(-1)
12         if (x <= 1/k)
13             (k * x)^(1/shape)
14         else
15             -log(((1/shape) + exp(-1)) * (1 - x))
16     }
17
18     x <- numeric(n)
19     i <- 1
20     while (i < n)
21     {
22         y <- Ginv(runif(1))
23         if (runif(1) <= ratio(y))
24             x[i <- i + 1] <- y
25     }
26     x * scale
27 }

```

FIG. 8.1 - Code d'une fonction à déboguer

3. Si

$$v \leq \begin{cases} e^{-y}, & 0 \leq y \leq 1 \\ y^{\alpha-1}, & y > 1, \end{cases}$$

alors poser $x \leftarrow y$. Sinon, retourner à l'étape 1.

4. Retourner λx .

La fonction `rgamma` qui se trouve à la [figure 8.1](#) est une mise en œuvre directe de l'algorithme, à la différence près qu'elle permet de simuler n valeurs plutôt qu'une seule. Surtout, j'y ai également inséré quelques petites anomalies. Voyons voir si nous pouvons les trouver.

8.1 Approche naïve

La première technique de débogage que vous devriez utiliser consiste simplement à vérifier la syntaxe de votre programme et à afficher des résultats intermédiaires pour identifier les sources d'erreurs.

Les erreurs de syntaxe sont de loin les erreurs les plus fréquentes — en particulier l'oubli de parenthèses ou d'accolades. Ici, votre premier outil de débogage demeure votre éditeur de texte pour programmeur — d'où l'importance d'en utiliser un bon. D'ailleurs, je vous recommande de lire l'argumentaire bien plus développé sur le sujet de [Hunt et Thomas \(1999, chapitre 3\)](#).

Un bon éditeur de texte va, entre autres choses, indenter le code automatiquement en fonction de ce qu'il contient, et non en fonction de ce que vous pensez qu'il contient. J'utilise pour ma part l'éditeur GNU Emacs depuis de nombreuses années pour des tâches d'édition de diverses natures et jamais je n'ai pris le moteur d'indentation en défaut. En d'autres termes, quand Emacs ne dispose pas le code comme je pense qu'il devrait l'être, c'est moi qui ai fait une erreur quelque part. Pratique.

Dans RStudio, pour obtenir une fonctionnalité équivalente, il suffit de refaire l'indentation du code avec l'option `Reindent Lines` du menu `Code`.

La simple édition du code de la [figure 8.1](#) avec un bon outil fait immédiatement ressortir un premier problème dans la boucle `while` (lignes 20–25), qui se retrouve indentée comme ci-dessous.

```
while (i < n)
{
  y <- Ginv(runif(1)
            if (runif(1) <= ratio(y))
              x[i <- i + 1] <- y
}
```

C'est ainsi que nous remarquons qu'il manque une parenthèse fermante dans l'expression `y <- Ginv(runif(1)`. Voilà déjà une erreur de corrigée.

Lors de la définition d'une fonction, l'interpréteur R effectue une vérification de la syntaxe. Cela permet de détecter plusieurs types d'erreurs relativement simples à corriger. Prenez simplement garde : une erreur peut prendre sa source plusieurs lignes avant celle que l'interpréteur identifie comme causant problème.

Dans notre exemple, l'interpréteur identifie correctement que nous avons utilisé un point-virgule « ; » en lieu et place d'une virgule « , » pour séparer deux arguments dans la définition de la fonction.

```
Erreur : ';' inattendu(e) in
"rgamma <- fonction(n, shape, rate = 1;"
```

Avec ces deux erreurs maintenant corrigées, nous pouvons définir la fonction dans l'espace de travail et l'utiliser sans erreur d'exécution.

```
> rgamma(5, 0.5, 1)
[1] 0.000000000 0.003292113 0.678709062 0.222413926 0.071747971
```

Les personnes un tant soit peu familières avec la distribution gamma auront tôt fait de remarquer une anomalie dans les résultats ci-dessus : la distribution étant strictement positive, elle ne peut générer la valeur 0 que l'on retrouve en première position du vecteur.

Une des premières choses à examiner lorsqu'un bogue survient, c'est s'il est reproductible, tant avec les mêmes conditions d'entrée (les valeurs des arguments) que sous d'autres conditions. Voyons voir.

```
> rgamma(5, 0.5, 1)
[1] 0.000000000 0.202158699 0.938504690 0.671480802 0.007275548
> rgamma(5, 0.8, 1)
[1] 0.000000000 0.04404955 0.45432025 3.73236138 0.51103648
> rgamma(5, 0.5, 2)
[1] 0.00000000 0.1593141 0.0693786 1.0312206 0.3232272
> rgamma(10, 0.5, 1)
[1] 0.00000000 0.17352536 1.03815695 0.55411331 1.06906215
[6] 0.24818970 0.13203951 0.07188287 0.17966368 0.02836616
> rgamma(1, 0.5, 1)
[1] 0
```

Il semble y avoir une constante, ici : c'est toujours la première valeur du vecteur de résultats qui est nulle, quel que soit le nombre de valeurs aléatoires demandé ou la paramétrisation de la loi gamma. Cela disculpe fort probablement les fonctions internes `ratio` et `Ginv` puisqu'elles semblent faire leur travail correctement pour les cas autres que le tout premier.

Dès qu'une fonction comporte une boucle, vos efforts de débogage devraient se tourner de ce côté. En effet, il est très facile de se tromper dans le critère d'arrêt d'une boucle, de telle sorte qu'il manque une itération ou, à l'inverse, qu'il y en a une de trop. Il faut également examiner l'effet des compteurs de près : seraient-ils incrémentés trop tôt ou trop tard dans la boucle ?

Il est temps de regarder une première technique de débogage, de loin la plus simple. Lorsqu'une fonction ne retourne pas le résultat attendu, placez des commandes utilisant la fonction `print` à des endroits stratégiques dans la fonction de façon à pouvoir suivre les valeurs prises par les différents objets. Dans notre exemple, nous aimerions suivre les valeurs du vecteur `x`, du compteur `i` et de l'objet `y`. Pour ce faire, nous remplaçons les lignes 20 à 25 de la [figure 8.1](#) par l'extrait de code ci-dessous¹, puis nous exécutons de nouveau la fonction.

1. Le code modifié est fourni en exemple dans la [section 8.5](#).

```

while (i < n)
{
  y <- Ginv(runif(1))
  if (runif(1) <= ratio(y))
  {
    print(paste("valeur de y acceptée:", y))
    x[i <- i + 1] <- y
    print(c(i, x))
  }
}

```

```

> rgamma(5, 0.5, 1)
[1] "valeur de y acceptée: 0.0910311655833059"
[1] 2.00000000 0.00000000 0.09103117 0.00000000 0.00000000
[6] 0.00000000
[1] "valeur de y acceptée: 0.137840041022257"
[1] 3.00000000 0.00000000 0.09103117 0.13784004 0.00000000
[6] 0.00000000
[1] "valeur de y acceptée: 0.0715489159587048"
[1] 4.00000000 0.00000000 0.09103117 0.13784004 0.07154892
[6] 0.00000000
[1] "valeur de y acceptée: 0.0664509126403331"
[1] 5.00000000 0.00000000 0.09103117 0.13784004 0.07154892
[6] 0.06645091
[1] 0.00000000 0.09103117 0.13784004 0.07154892 0.06645091

```

Les résultats précédents nous apprennent que lorsque la première valeur y est acceptée, elle est placée en seconde position dans le vecteur x . Le coupable est donc notre compteur i ! En y regardant de plus près, nous réalisons qu'il est initialisé à la valeur 1 à l'extérieur de la boucle `while` (ligne 19 de la [figure 8.1](#)) et qu'il est ensuite incrémenté *avant* que l'affectation dans x ne soit effectuée dans la clause `if` (ligne 24). Le correctif est donc facile à appliquer : initialiser i avec la valeur 0. Ceci fait, la fonction retourne maintenant des valeurs plausibles.

```

> rgamma(5, 0.5, 1)
[1] "valeur de y acceptée: 0.303918278257569"
[1] 1.00000000 0.3039183 0.00000000 0.00000000 0.00000000 0.00000000
[1] "valeur de y acceptée: 0.233544162476529"
[1] 2.00000000 0.3039183 0.2335442 0.00000000 0.00000000 0.00000000
[1] "valeur de y acceptée: 0.125404703312485"
[1] 3.00000000 0.3039183 0.2335442 0.1254047 0.00000000 0.00000000
[1] "valeur de y acceptée: 0.290653330254637"
[1] 4.00000000 0.3039183 0.2335442 0.1254047 0.2906533 0.00000000

```

```
[1] "valeur de y acceptée: 0.363374964992821"  
[1] 5.0000000 0.3039183 0.2335442 0.1254047 0.2906533 0.3633750  
[1] 0.3039183 0.2335442 0.1254047 0.2906533 0.3633750
```

En l'état, la fonction `rgamma` semble valide. Je vous rappelle la mise en garde de la [section 4.3.2](#) : avec le recyclage des vecteurs dans R, ce n'est pas parce qu'une expression — ou une fonction — s'exécute sans erreur apparente que le calcul effectué est bon pour autant. Il faut vérifier l'exactitude des calculs à l'aide de tests unitaires, tel que décrit à la [section 6.4](#).



Lorsqu'une erreur survient, assurez-vous de bien lire les messages d'erreur de R. Il faut parfois apprendre à les reconnaître. Par exemple, le message assez fréquent « `valeur manquante là où TRUE / FALSE est requis` » provient d'une clause `if` dont l'argument vaut `NA` plutôt que `TRUE` ou `FALSE`. En général, c'est parce que des valeurs manquantes se sont faufilées à votre insu dans les calculs jusqu'à l'instruction `if`.

8.2 Évaluation pas à pas

La technique d'affichage de résultats intermédiaires s'est avérée suffisante pour trouver et corriger les bogues de la fonction `rgamma`, à la section précédente. Ce n'est toutefois pas toujours ainsi.

Une autre méthode de débogage simple à mettre en œuvre avec les langages de programmation interprétés comme R consiste à évaluer pas à pas le code d'une fonction. Voici comment procéder.

1. Définir dans l'espace de travail tous les arguments de la fonction en y affectant les valeurs avec lesquelles vous souhaitez tester le code.
2. Exécuter ligne par ligne le corps de la fonction. Après chaque ligne ou groupe de lignes, vérifier les valeurs des différents objets. Cela permet habituellement de détecter les expressions qui causent problème.
3. Lorsque la fonction contient une boucle, sauter par-dessus la ligne contenant l'expression `for`, `while` ou `repeat` (et celle contenant l'éventuelle accolade ouvrante) pour pouvoir exécuter les itérations de la boucle une à une. Porter une attention particulière à la première et à la dernière itération.

Vous trouverez ci-dessous la transcription d'une session de débogage par la méthode pas à pas pour l'exemple de la section précédente. Remarquez comment je n'ai pas évalué les lignes 1–2 et 20–21 de la [figure 8.1](#). J'ai également placé le nombre aléatoire issu de l'expression `runif(1)` dans la clause `if` dans un objet `u` afin de pouvoir valider le test. Au final, la technique permet de détecter le problème de compteur mal initialisé.

```
> n <- 5
> shape <- 0.5
> scale <- 1
>   if (shape <= 0 | shape >= 1)
+     stop("valeur de shape inadmissible")
>   ratio <- function(x)
+     if (x <= 1) exp(-x) else x^(shape - 1)
>   Ginv <- function(x)
+   {
+     k <- 1 + shape * exp(-1)
+     if (x <= 1/k)
+       (k * x)^(1/shape)
+     else
+       -log(((1/shape) + exp(-1)) * (1 - x))
+   }
>   x <- numeric(n)
>   i <- 1
> x
[1] 0 0 0 0 0
> i
[1] 1
>     u <- runif(1)
>     if (u <= ratio(y))
+       x[i <- i + 1] <- y
> u
[1] 0.9470823
> ratio(y)
[1] 0.9426682
> x
[1] 0 0 0 0 0
>     y <- Ginv(runif(1))
>     u <- runif(1)
>     if (u <= ratio(y))
+       x[i <- i + 1] <- y
> u
[1] 0.1297151
> ratio(y)
[1] 0.5810007
> x
[1] 0.0000000 0.5430033 0.0000000 0.0000000 0.0000000
> i
[1] 2
```



Une fois la séance de débogage terminée, supprimez de l'espace de travail les objets que vous avez créés au cours de celle-ci, puis évaluez de nouveau votre fonction. Cela vous prémunira contre une fonction qui, par la magie de la portée lexicale de R ([chapitre 12](#)), serait valide seulement lorsque certains objets existent dans l'espace de travail.

8.3 Navigateur d'environnements

Pour efficace qu'elle soit, la technique d'évaluation pas à pas devient rapidement fastidieuse lorsque la fonction à déboguer est longue ou, surtout, lorsqu'elle contient de nombreux appels à des fonctions dont nous souhaitons également valider le code. Dans de telles situations, les outils de débogage de R peuvent nous simplifier la tâche.

Un premier outil de R très utile dès qu'un appel de fonction retourne un message d'erreur est la fonction `traceback`. Je vous encourage fortement à développer le réflexe de l'utiliser. La fonction affiche la pile des appels de fonction (*call stack*) en ordre chronologique inverse au moment de l'erreur. Cela permet d'identifier avec précision l'appel de fonction qui a causé l'erreur.

```
> foo <- function(x) bar(2 * x)
> bar <- function(x) x + variable.inconnue
> foo(2)
Error in bar(2 * x) (from #1) : objet 'variable.inconnue'
introuvable
> traceback()
2: bar(2 * x) at #1
1: foo(2)
```

Le second — et plus puissant — outil de débogage de R permet en quelque sorte d'automatiser la technique d'évaluation pas à pas. L'idée est la suivante : pour analyser le déroulement d'une fonction `fun`, vous ajoutez un appel à la fonction `browser` en un point quelconque de la fonction et vous définissez la fonction dans l'espace de travail. Vous évaluez ensuite un appel à la fonction. Quand l'interpréteur atteint la commande `browser()`, il interrompt le déroulement de la fonction et il vous remet le contrôle dans un mode d'interaction spécial nommé « navigateur d'environnement ». Celui-ci vous permet d'explorer à loisir l'environnement d'évaluation ([chapitre 12](#)) de la fonction en ce point. Vous pouvez également poursuivre l'exécution de la fonction ligne par ligne comme dans la technique d'évaluation pas à pas. Avantage non négligeable : le navigateur peut entrer récursivement dans l'environnement d'évaluation de chaque appel de fonction.

Une fois entré dans le navigateur d'environnement, diverses commandes spéciales deviennent disponibles. Je ne présente ici que les plus importantes ; consultez la rubrique d'aide de la fonction `browser` pour la liste complète.

- n évaluer la prochaine expression de la fonction (celle qui est affichée à l'invite de commande du navigateur), sans toutefois entrer dans les appels de fonction.
- s évaluer la prochaine expression de la fonction en entrant dans les appels de fonctions.
- c quitter le navigateur et poursuivre l'exécution de la fonction normalement.
- Q quitter le navigateur ainsi que l'évaluation en cours pour retourner à l'invite de commande de R.

Reprenons l'exemple de la section précédente en supposant que nous avons inséré la commande `browser()` à la ligne 17 de la [figure 8.1](#). La transcription d'une séance d'utilisation du navigateur d'environnement suit.

```
> rgamma(5, 0.5, 1)
Called from: rgamma(5, 0.5, 1)
Browse[1]> n
debug à #18 :x <- numeric(n)
Browse[2]> n
debug à #19 :i <- 1
Browse[2]> x
[1] 0 0 0 0 0
Browse[2]> n
debug à #20 :while (i < n) {
  y <- Ginv(runif(1))
  if (runif(1) <= ratio(y))
    x[i <- i + 1] <- y
}
Browse[2]> i
[1] 1
Browse[2]> n
debug à #22 :y <- Ginv(runif(1))
Browse[2]> s
debugging in: Ginv(runif(1))
debug à #9 :{
  k <- 1 + shape * exp(-1)
  if (x <= 1/k)
    (k * x)^(1/shape)
  else -log(((1/shape) + exp(-1)) * (1 - x))
}
Browse[3]> n
debug à #11 :k <- 1 + shape * exp(-1)
Browse[3]> n
debug à #12 :if (x <= 1/k) (k * x)^(1/shape)
  else -log(((1/shape) + exp(-1)) * (1 - x))
```

```


Browse[3]> k
[1] 1.18394
Browse[3]> n
debug à #12 :-log(((1/shape) + exp(-1)) * (1 - x))
Browse[3]> n
exiting from: Ginv(runif(1))
debug à #23 :if (runif(1) <= ratio(y)) x[i <- i + 1] <- y
Browse[2]> n
debug à #23 :x[i <- i + 1] <- y
Browse[2]> n
debug à #20 :(while) i < n
Browse[2]> y
[1] 1.667504
Browse[2]> x
[1] 0.000000 1.667504 0.000000 0.000000 0.000000
Browse[2]> c
debug à #26 :x * scale
Browse[2]> c
[1] 0.000000000 1.667504209 0.164785294 1.176891303
[5] 0.008645285

```



Les environnements de développement intégrés comme Emacs avec le mode ESS ou RStudio facilitent l'utilisation du navigateur d'environnement. Consultez la documentation de votre éditeur sous la rubrique *Debugging* pour en tirer le meilleur parti.

8.4 Méthode du canard en plastique

Ce chapitre ne serait pas complet sans une mention de la méthode de débogage dite du *canard en plastique* . [Hunt et Thomas \(1999, chapitre 3\)](#), encore eux, sont généralement considérés comme les pères de la méthode ou, du moins, de son appellation.

La méthode du canard en plastique consiste simplement à expliquer méticuleusement son code source à une personne qui se contente de vous écouter en hochant la tête ou, de manière à peu près équivalente, à un objet inanimé comme un canard en plastique. L'important, c'est de le faire à voix haute et en explicitant chacune des hypothèses et chacune des étapes du programme. Le regard neuf que vous porterez au code peut vous permettre d'y déceler les problèmes.



Étudiez le fichier de script `debogage.R` reproduit à la [section 8.5](#).

8.5 Exemples

📍 Fichier d'accompagnement debogage.R

```

11 ## Ce fichier fournit le code utile pour refaire les exemples
12 ## du chapitre.
13
14 ###
15 ### VÉRIFICATION DE LA SYNTAXE
16 ###
17
18 ## Code de la fonction 'rgamma' avec des bogues, tel que
19 ## présenté à la figure 10.1.
20 ##
21 ## Refaire l'indentation de la fonction avec votre éditeur
22 ## pour programmeur. Cela devrait immédiatement faire
23 ## ressortir qu'il manque une parenthèse dans l'appel à la
24 ## fonction 'Ginv', à l'intérieur de la boucle 'while'.
25 rgamma <- function(n, shape, rate = 1; scale = 1/rate)
26 {
27     if (shape <= 0 | shape >= 1)
28         stop("valeur de shape inadmissible")
29
30     ratio <- function(x)
31         if (x <= 1) exp(-x) else x^(shape - 1)
32
33     Ginv <- function(x)
34     {
35         k <- 1 + shape * exp(-1)
36         if (x <= 1/k)
37             (k * x)^(1/shape)
38         else
39             -log(((1/shape) + exp(-1)) * (1 - x))
40     }
41
42     x <- numeric(n)
43     i <- 1
44     while (i < n)
45     {
46         y <- Ginv(runif(1))
47         if (runif(1) <= ratio(y))
48             x[i <- i + 1] <- y
49     }
50     x * scale
51 }
52
53 ## La parenthèse manquante a été ajoutée dans la définition
54 ## ci-dessous.
55 ##

```

```

56 ## Essayez de définir la fonction dans l'espace de travail.
57 ## L'interpréteur R devrait attraper l'erreur de syntaxe à la
58 ## toute première ligne, soit le ';' en lieu et place d'une
59 ## ', '.
60 rgamma <- function(n, shape, rate = 1; scale = 1/rate)
61 {
62     if (shape <= 0 | shape >= 1)
63         stop("valeur de shape inadmissible")
64
65     ratio <- function(x)
66         if (x <= 1) exp(-x) else x^(shape - 1)
67
68     Ginv <- function(x)
69     {
70         k <- 1 + shape * exp(-1)
71         if (x <= 1/k)
72             (k * x)^(1/shape)
73         else
74             -log(((1/shape) + exp(-1)) * (1 - x))
75     }
76
77     x <- numeric(n)
78     i <- 1
79     while (i < n)
80     {
81         y <- Ginv(runif(1))
82         if (runif(1) <= ratio(y))
83             x[i <- i + 1] <- y
84     }
85     x * scale
86 }
87
88 ## L'erreur de syntaxe est maintenant corrigée.
89 rgamma <- function(n, shape, rate = 1, scale = 1/rate)
90 {
91     if (shape <= 0 | shape >= 1)
92         stop("valeur de shape inadmissible")
93
94     ratio <- function(x)
95         if (x <= 1) exp(-x) else x^(shape - 1)
96
97     Ginv <- function(x)
98     {
99         k <- 1 + shape * exp(-1)
100         if (x <= 1/k)
101             (k * x)^(1/shape)
102         else
103             -log(((1/shape) + exp(-1)) * (1 - x))

```

```
104     }
105
106     x <- numeric(n)
107     i <- 1
108     while (i < n)
109     {
110         y <- Ginv(runif(1))
111         if (runif(1) <= ratio(y))
112             x[i <- i + 1] <- y
113     }
114     x * scale
115 }
116
117 ## Nous ne sommes pas sortis du bois: l'appel de la fonction
118 ## ci-dessous retourne une valeur de 0, ce qui n'est pas une
119 ## valeur admissible pour une distribution gamma.
120 rgamma(5, 0.5, 1)
121
122 ## Vérification avec d'autres valeurs des arguments. Le bogue
123 ## est aisément reproductible.
124 rgamma(5, 0.5, 1)
125 rgamma(5, 0.8, 1)
126 rgamma(5, 0.5, 2)
127 rgamma(10, 0.5, 1)
128 rgamma(1, 0.5, 1)
129
130 ###
131 ### AFFICHAGE DE RÉSULTATS INTERMÉDIAIRES
132 ###
133
134 ## Afin de découvrir le bogue de la fonction, nous ajoutons
135 ## une commande 'print' à l'intérieur de la boucle 'while'.
136 rgamma <- function(n, shape, rate = 1, scale = 1/rate)
137 {
138     if (shape <= 0 | shape >= 1)
139         stop("valeur de shape inadmissible")
140
141     ratio <- function(x)
142         if (x <= 1) exp(-x) else x^(shape - 1)
143
144     Ginv <- function(x)
145     {
146         k <- 1 + shape * exp(-1)
147         if (x <= 1/k)
148             (k * x)^(1/shape)
149         else
150             -log(((1/shape) + exp(-1)) * (1 - x))
151     }
```

```

152
153     x <- numeric(n)
154     i <- 1
155     while (i < n)
156     {
157         y <- Ginv(runif(1))
158         if (runif(1) <= ratio(y))
159         {
160             print(paste("valeur de y acceptée:", y))
161             x[i <- i + 1] <- y
162             print(c(i, x))
163         }
164     }
165     x * scale
166 }
167
168 ## L'exécution de la commande ci-dessous permet de détecter
169 ## que lorsqu'une première valeur de 'y' est acceptée, elle
170 ## est placée en deuxième position dans le vecteur des
171 ## résultats 'x'. Ah! Ha! Le compteur 'i' est soit incrémenté
172 ## trop tôt, soit initialisé une unité trop haut.
173 rgamma(5, 0.5, 1)
174
175 ## Nous corrigeons la fonction ainsi: le compteur 'i' est
176 ## initialisé à 0 plutôt qu'à 1.
177 rgamma <- function(n, shape, rate = 1, scale = 1/rate)
178 {
179     if (shape <= 0 | shape >= 1)
180         stop("valeur de shape inadmissible")
181
182     ratio <- function(x)
183         if (x <= 1) exp(-x) else x^(shape - 1)
184
185     Ginv <- function(x)
186     {
187         k <- 1 + shape * exp(-1)
188         if (x <= 1/k)
189             (k * x)^(1/shape)
190         else
191             -log(((1/shape) + exp(-1)) * (1 - x))
192     }
193
194     x <- numeric(n)
195     i <- 0
196     while (i < n)
197     {
198         y <- Ginv(runif(1))
199         if (runif(1) <= ratio(y))

```

```

200         x[i <- i + 1] <- y
201     }
202     x * scale
203 }
204
205 ###
206 ### ÉVALUATION PAS À PAS
207 ###
208
209 ## La technique d'évaluation pas à pas consiste à définir tous
210 ## les arguments de la fonction dans l'espace de travail, puis
211 ## à exécuter le corps de la fonction ligne par ligne.
212 ##
213 ## Évaluons l'équivalent de l'appel 'rgamma(5, 0.5, 1)'
214 ## pas à pas avec le bogue 'i <- 1' remis en place.
215 ##
216 ## Explorez les valeurs des objets 'x', 'i' et 'y' après
217 ## l'évaluation de chaque expression, ci-dessous (ou, plus
218 ## précisément, lorsque c'est pertinent).
219 n <- 5
220 shape <- 0.5
221 scale <- 1
222
223 ## rgamma <- function(n, shape, rate = 1, scale = 1/rate)
224 ## {
225     if (shape <= 0 | shape >= 1)
226         stop("valeur de shape inadmissible")
227
228     ratio <- function(x)
229         if (x <= 1) exp(-x) else x^(shape - 1)
230
231     Ginv <- function(x)
232     {
233         k <- 1 + shape * exp(-1)
234         if (x <= 1/k)
235             (k * x)^(1/shape)
236         else
237             -log(((1/shape) + exp(-1)) * (1 - x))
238     }
239
240     x <- numeric(n)
241     i <- 1
242     ## while (i < n)
243     ## {
244         y <- Ginv(runif(1))
245         u <- runif(1)
246         if (u <= ratio(y))
247             x[i <- i + 1] <- y

```

```

248     ## }
249     x * scale
250 ## }
251
252 ###
253 ### PILE DES APPELS DE FONCTIONS
254 ###
255
256 ## J'ai inséré un bogue additionnel dans la fonction interne
257 ## 'Ginv' de 'rgamma', ci-dessous. Définir la fonction.
258 rgamma <- function(n, shape, rate = 1, scale = 1/rate)
259 {
260     if (shape <= 0 | shape >= 1)
261         stop("valeur de shape inadmissible")
262
263     ratio <- function(x)
264         if (x <= 1) exp(-x) else x^(shape - 1)
265
266     Ginv <- function(x)
267     {
268         k <- 1 + shape * exp(-1) * variable.inconnue
269         if (x <= 1/k)
270             (k * x)^(1/shape)
271         else
272             -log(((1/shape) + exp(-1)) * (1 - x))
273     }
274
275     x <- numeric(n)
276     i <- 1
277     while (i < n)
278     {
279         y <- Ginv(runif(1))
280         u <- runif(1)
281         if (u <= ratio(y))
282             x[i <- i + 1] <- y
283     }
284     x * scale
285 }
286
287 ## Évaluer l'appel de fonction ci-dessous. Il y aura une
288 ## erreur.
289 rgamma(5, 0.5, 1)
290
291 ## Afficher la pile des appels de fonctions. On voit que c'est
292 ## l'appel à 'Ginv' qui pose problème.
293 traceback()
294
295 ###

```

```

296 ### NAVIGATEUR D'ENVIRONNEMENT
297 ###
298
299 ## La définition de la fonction 'rgamma' ci-dessous
300 ## contient un appel à la fonction 'browser'.
301 ##
302 ## L'appel à la fonction qui suit provoquera l'entrée dans le
303 ## navigateur d'environnement juste avant l'évaluation de
304 ## l'expression 'x <- numeric(n)'.
305 ##
306 ## L'utilisation du navigateur demande un temps d'adaptation à
307 ## cause de la forme de déphasage que l'on observe entre
308 ## l'affichage de la ligne qui sera évaluée et celle qui vient
309 ## d'être évaluée.
310 ##
311 ## L'interface de RStudio pour utiliser le navigateur est très
312 ## conviviale; consultez la documentation de RStudio pour un
313 ## bon tutoriel.
314 ##
315 ## Également, consultez la rubrique d'aide de 'browser' pour
316 ## connaître les commandes disponibles à l'invite du
317 ## navigateur.
318 rgamma <- function(n, shape, rate = 1, scale = 1/rate)
319 {
320   if (shape <= 0 | shape >= 1)
321     stop("valeur de shape inadmissible")
322
323   ratio <- function(x)
324     if (x <= 1) exp(-x) else x^(shape - 1)
325
326   Ginv <- function(x)
327   {
328     k <- 1 + shape * exp(-1)
329     if (x <= 1/k)
330       (k * x)^(1/shape)
331     else
332       -log(((1/shape) + exp(-1)) * (1 - x))
333   }
334   browser()
335   x <- numeric(n)
336   i <- 1
337   while (i < n)
338   {
339     y <- Ginv(runif(1))
340     if (runif(1) <= ratio(y))
341       x[i <- i + 1] <- y
342   }
343   x * scale

```

```

344 }
345 rgamma(5, 0.5, 1)
346
347 ###
348 ### EXEMPLE ADDITIONNEL
349 ###
350
351 ## Variation sur le thème de la suite de Fibonacci: une
352 ## fonction pour calculer non pas les 'n' premières valeurs
353 ## de la suite de Fibonacci, mais uniquement la 'n'ième
354 ## valeur.
355 ##
356 ## Mais il y a un mais: la fonction 'fibonaci', à l'instar de
357 ## son nom même, est truffée d'erreurs (de syntaxe,
358 ## d'algorithmique, de conception). À vous de trouver les
359 ## bogues. (Afin de préserver cet exemple, copiez le code
360 ## erroné plus bas ou dans un autre fichier avant d'y faire
361 ## les corrections.)
362 fibonaci <- fonction(nb)
363 {
364     x <- 0
365     x1 _ 0
366     x2 <- 1
367     while (n > 0))
368 x <- x1 + x2
369 x2 <- x1
370 x1 <- x
371 n <- n - 1
372 }
373 fibonaci(1)           # devrait donner 0
374 fibonaci(2)           # devrait donner 1
375 fibonaci(5)           # devrait donner 3
376 fibonaci(10)          # devrait donner 34
377 fibonaci(20)          # devrait donner 4181

```

8.6 Exercices

8.1 Trouver les erreurs qui empêchent la définition de la fonction suivante.

```

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

```


- 8.2 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 être 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
```

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

9 Importation et exportation de données

Objectifs du chapitre

- ▶ Importer dans R des données provenant de sources externes.
- ▶ Exporter des données de R vers un format exploitable par d'autres outils.

« Est-ce que je peux transférer mes données de Excel à R ? » Cette question et ses multiples variantes sont assurément parmi les plus fréquemment posées par les utilisateurs de R. La réponse courte : oui, c'est possible d'importer (c'est la terminologie correcte) des données de source externe dans R. La réponse longue : oui, c'est possible, mais c'est souvent compliqué. Il en va de même pour la procédure inverse consistant à exporter des données de R pour usage dans une autre application.

Cette situation n'est pas propre à R. Ce qui rend souvent ardu l'échange de données entre différents systèmes et logiciels, c'est le nombre de facteurs qui influencent l'importation et l'exportation : le type de fichier (en format texte brut ou en format binaire), le codage des caractères (ASCII pur, UTF-8 ou autre), la disposition des données dans le fichier (notamment les matrices et tableaux), le caractère utilisé pour séparer les champs (espace, virgule ou autre), le format du fichier (Excel, SAS ou autre), etc. Difficile, dans les circonstances, de proposer une procédure universelle. C'est d'ailleurs pourquoi la documentation officielle de R consacre le manuel *R Data Import/Export* ([R Core Team, 2024](#)) à ce seul sujet.

À l'exception des fonctions de la [section 9.4](#), je ne traite, ici, que de l'importation et de l'exportation de données sous forme de fichiers en format texte brut¹.



Il existe de multiples autres façons de travailler avec des données externes, notamment l'importation de fichiers de tableurs comme Excel ou, encore mieux, l'interaction directe avec une base de données depuis R. Consultez *R Data Import/Export* pour en savoir plus.

1. C'est-à-dire sans aucune mise en forme, comme les fichiers de script.

vecteur.data	matrice.data
# Vecteur de données	# Données d'une matrice 3 x 5
4267	143 159 121 107 195
4238	114 151 154 173 137
4200	
4233	

FIG. 9.1 – Exemples de fichiers contenant des vecteurs de données en texte brut. Gauche : fichier contenant un vecteur de données à raison d'une donnée par ligne. Droite : fichier contenant les données d'une matrice. Dans les deux cas, le caractère # indique que le reste du texte sur la ligne est un commentaire.

9.1 Importation de vecteurs de données

Le plus simple jeu de données à transférer d'un système à un autre est un vecteur ou une matrice de données toutes du même mode. Pour ce genre d'application, un fichier de données brutes convient parfaitement. Les données y sont disposées les unes à la suite des autres sur une ou plusieurs lignes. Le fichier peut aussi contenir des commentaires, ceux-ci étant normalement délimités par un caractère spécial tel que « # », « % » ou « ; ». La [figure 9.1](#) fournit deux exemples d'un tel fichier de données.

La fonction `scan` permet d'importer des données brutes dans R. Elle lit l'intégralité des données du fichier dont le nom est donné en premier argument, ligne par ligne, puis retourne un vecteur. Seul l'ordre des données dans le fichier est pris en compte, leur disposition n'a aucune importance. La fonction ignore tout texte qui suit le caractère spécifié via l'argument `comment.char`.

```
> scan("vecteur.data", comment.char = "#")
[1] 4267 4238 4200 4233 4286
```

Alternative à `comment.char`, l'argument `skip` permet de sauter un certain nombre de lignes au début du fichier lors de l'importation.

```
> scan("vecteur.data", skip = 1)
[1] 4267 4238 4200 4233 4286
```

Pour recréer à l'identique dans R une matrice importée avec `scan`, il faut s'assurer de la remplir par ligne.

```
> matrix(scan("matrice.data", comment.char = "#"),
+        nrow = 3, byrow = TRUE)
      [,1] [,2] [,3] [,4] [,5]
[1,]  143  159  121  107  195
[2,]  114  151  154  173  137
[3,]  182  185  174  179  187
```

La fonction `scan` compte de nombreux autres arguments; consulter sa ru-

carburant.txt					
"mpg"	"nbcyl"	"cylindree"	"cv"	"poids"	
16.9	8	350	155	4.36	
15.5	8	351	142	4.054	
19.2	8	267	125	3.605	
18.5	8	360	150	3.94	

carburant.csv					
"mpg"	"nbcyl"	"cylindree"	"cv"	"poids"	
16.9,8,350,155,4.36					
15.5,8,351,142,4.054					
19.2,8,267,125,3.605					
18.5,8,360,150,3.94					

FIG. 9.2 – Exemples de tableaux de données en texte brut. Haut : fichier contenant un jeu de données rectangulaire dont les champs sont séparés par des espaces. Bas : fichier contenant le même jeu de données avec une virgule comme séparateur. Dans les deux fichiers, la première ligne contient les titres des colonnes.

brique d'aide si la structure du fichier de données à importer dans R ne correspond pas tout à fait aux cas traités ci-dessus.



Dès qu'il s'agit d'accéder à un fichier, R recherche celui-ci dans son répertoire de travail ([section 3.6](#)). Si le fichier se trouve dans un autre répertoire, vous devez spécifier dans le nom le chemin d'accès ([section 1.5.3](#)) vers le fichier.

9.2 Importation de tableaux de données

Les données des tableurs et des bases de données sont habituellement organisées en tableau rectangulaire où chaque ligne représente un enregistrement et chaque colonne représente un champ (ou un « sujet » et une « caractéristique »). Vous aurez reconnu la structure du tableau de données dans R.

Les tableurs, les bases de données et les logiciels statistiques peuvent exporter leurs tableaux de données sous forme de fichier texte dans lequel les champs sont séparés par un caractère quelconque, souvent une espace. Le format le plus universel utilise toutefois comme séparateur la virgule ou le point-virgule ; c'est le format de fichier CSV (*comma-separated values*). La [figure 9.2](#) présente le même jeu de données² dans deux formats différents.

Les deux principales fonctions d'importation de tableaux de données dans R sont `read.table` et `read.csv`. On choisira l'une ou l'autre selon le format du fichier de données : `read.table` si le séparateur des champs est une espace et

2. Ce jeu de données est une version réduite et simplifiée des données du fichier `carburant.dat` livré avec le présent ouvrage.

TAB. 9.1 – Caractéristiques des fonctions d'importation de données de la famille `read.table`

Fonction d'importation	Séparateur des champs	Séparateur décimal
<code>read.table</code>	espace	.
<code>read.csv</code>	,	.
<code>read.csv2</code>	;	,
<code>read.delim</code>	tabulation	.
<code>read.delim2</code>	tabulation	,

`read.csv` si c'est une virgule. Les nombreux arguments de ces fonctions permettent de spécifier si les titres des lignes ou des colonnes apparaissent ou non dans le fichier, le mode de chaque colonne, le séparateur décimal, le codage, etc. La rubrique d'aide fournit les détails.

```
> read.table("carburant.txt", header = TRUE)
  mpg nbcyl cylindree  cv poids
1 16.9     8      350 155 4.360
2 15.5     8      351 142 4.054
3 19.2     8      267 125 3.605
4 18.5     8      360 150 3.940
5 30.0     4       98  68 2.155

> read.csv("carburant.csv")
  mpg nbcyl cylindree  cv poids
1 16.9     8      350 155 4.360
2 15.5     8      351 142 4.054
3 19.2     8      267 125 3.605
4 18.5     8      360 150 3.940
5 30.0     4       98  68 2.155
```

D'autres variantes de `read.table` existent pour des formats de fichiers courants. Elles diffèrent uniquement par le séparateur des champs et le séparateur décimal; consulter le [tableau 9.1](#).



Si les données à importer comportent des accents et qu'ils n'apparaissent pas correctement dans R, utilisez l'option `encoding` dans les appels des fonctions de la famille `read.table` pour spécifier le type de codage du fichier. Par exemple, pour un fichier de données codé en UTF-8, ajoutez l'option `encoding = "UTF-8"`.

9.3 Exportation de données

R peut évidemment produire les types de fichiers mentionnés aux sections précédentes pour exporter ses données vers d'autres systèmes. Les exemples de la présente section permettent d'ailleurs de recréer les fichiers des figures 9.1 et 9.2 (sans les commentaires).

L'exportation d'un vecteur simple dans un fichier en format texte brut passe par la fonction `cat`. Les principaux arguments de la fonction sont : le ou les objets à exporter; le nom du fichier; le séparateur entre les éléments du vecteur dans le fichier. Le séparateur par défaut est une espace, ce qui fait en sorte que les éléments du vecteur se retrouvent sur une seule ligne dans le fichier d'exportation. Pour placer les éléments sur des lignes différentes, il faut utiliser un retour à la ligne, symbolisé par `"\n"`, comme séparateur.

```
> x
[1] 4267 4238 4200 4233 4286
> cat(x, file = "vecteur.data", sep = "\n")
```

La fonction `write` joue le même rôle que `cat`, sauf que les données sont disposées sous forme de tableau rectangulaire (de cinq colonnes par défaut) dans le fichier d'exportation. Elle est donc mieux appropriée pour exporter les matrices. Étant donné que R remplit les matrices par colonne, il faut prendre soin de les transposer avant de les exporter.

```
> x
      [,1] [,2] [,3] [,4] [,5]
[1,]  143  159  121  107  195
[2,]  114  151  154  173  137
[3,]  182  185  174  179  187
> write(t(x), file = "matrice.data", ncolumns = 5)
```

Enfin, les fonctions analogues à `read.table`, `read.csv` et `read.csv2`, mais dédiées à l'exportation, sont `write.table`, `write.csv` et `write.csv2`, dans l'ordre. Divers arguments permettent de contrôler les informations à transférer du tableau de données au fichier d'exportation, ainsi que le format de ce dernier.

```
> x
      mpg nb cyl cylindree   cv poids
1 16.9      8      350 155 4.360
2 15.5      8      351 142 4.054
3 19.2      8      267 125 3.605
4 18.5      8      360 150 3.940
5 30.0      4       98  68 2.155
> write.table(x, file = "carburant.txt", row.names = FALSE)
> write.csv(x, file = "carburant.csv", row.names = FALSE)
```

9.4 Importation et exportation d'objets R

Les fonctions des sections précédentes permettent l'échange de données entre logiciels. Or, ce que l'on souhaite parfois, c'est simplement transférer des données de R dans un système vers R dans un autre système. Le plus simple devient alors de partager les objets R sans passer par un format intermédiaire. C'est le rôle de la sérialisation, c'est-à-dire le codage d'information pour son transport et son échange.

Dans R, la manière la plus simple d'exporter et d'importer des données sérialisées est via les fonctions `saveRDS` et `readRDS`. La première permet de sauvegarder un objet dans un fichier, par convention muni d'une extension `.rds`.

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

$user
[1] "Joe"

$new
[1] TRUE

> saveRDS(x, "liste.rds")
```

La fonction `readRDS` permet de recréer un objet à l'identique à partir de l'information d'un fichier de données sérialisées.

```
> readRDS("liste.rds")
$score
[1] 1 5 2

$user
[1] "Joe"

$new
[1] TRUE
```

Le format de sérialisation est propre à R : vous ne pouvez donc pas espérer partager un fichier `.rds` avec un autre logiciel. De plus, le format du fichier est binaire : impossible de le lire avec un éditeur de texte.



Les exemples interactifs du fichier de script `import-export.R` reproduit à la section suivante vous permettront de mieux comprendre le fonctionnement des fonctions d'importation et d'exportation.

9.5 Exemples

📍 Fichier d'accompagnement import-export.R

```

11  ## Pour illustrer les procédures d'importation et d'exportation de
12  ## données, nous allons d'abord exporter des données dans des fichiers
13  ## pour ensuite les importer.
14  ##
15  ## Les fichiers seront créés dans le répertoire de travail de R. La
16  ## commande
17  ##
18  ##   getwd()
19  ##
20  ## affiche le nom de ce répertoire.
21  ##
22  ## Après chaque création de fichier d'exportation, ci-dessous, ouvrir
23  ## le fichier correspondant dans votre éditeur pour voir les
24  ## résultats.
25  ##
26  ## Débutons par l'exportation d'un vecteur simple créé à partir d'un
27  ## échantillon aléatoire des nombres de 1 à 100.
28  (x <- sample(1:100, 20))
29
30  ## Exportation avec la fonction 'cat' sans commentaires dans le
31  ## fichier, les données les unes à la suite des autres sur une seule
32  ## ligne.
33  cat(x, file = "vecteur.data")
34
35  ## Pour placer des commentaires au début du fichier, il suffit
36  ## d'exporter deux objets: la chaîne de caractères contenant le
37  ## commentaire et le vecteur. Ici, nous insérons un retour à la ligne
38  ## entre chaque élément.
39  cat("# commentaire", x, file = "vecteur.data", sep = "\n")
40
41  ## La fonction 'write' permet de disposer les données exportées en
42  ## colonnes (cinq par défaut), un peu comme une matrice. Exportons
43  ## exactement le même vecteur de données en lui donnant l'apparence
44  ## d'une matrice 5 x 4 (remplie par ligne).
45  write(x, file = "matrice.data", ncolumns = 4)
46
47  ## Pour insérer des commentaires au début du fichier créé avec
48  ## 'write', le plus simple consiste à procéder en deux étapes: on crée
49  ## d'abord un fichier ne contenant que le commentaire (et le retour à
50  ## la ligne) avec 'cat', puis on y ajoute les données avec 'write' en
51  ## spécifiant 'append = TRUE' pour éviter d'écraser le contenu du
52  ## fichier.
53  cat("# commentaire\n", file = "matrice.data")
54  write(x, file = "matrice.data", ncolumns = 4, append = TRUE)
55

```

```
56 ## Pour illustrer l'exportation avec 'write.table', 'write.csv' et
57 ## 'write.csv2', nous allons exporter le jeu de données 'USArrests'
58 ## livré avec R.
59 ##
60 ## Les titres des lignes sont importants dans ce jeu de données
61 ## puisqu'ils contiennent les noms des États. Par défaut, les
62 ## fonctions exportent tant les titres de lignes que les titres de
63 ## colonnes.
64 ##
65 ## 'write.table' utilise l'espace comme séparateur des champs et le
66 ## point comme séparateur décimal.
67 write.table(USArrests, "USArrests.txt")
68
69 ## 'write.csv' utilise la virgule comme séparateur des champs et le
70 ## point comme séparateur décimal.
71 write.csv(USArrests, "USArrests.csv")
72
73 ## 'write.csv2' utilise le point-virgule comme séparateur des champs
74 ## et la virgule comme séparateur décimal.
75 write.csv2(USArrests, "USArrests.csv2")
76
77 ## Importons maintenant toutes ces données dans notre espace de
78 ## travail.
79 ##
80 ## Les données de 'vecteur.data' (en passant, l'extension dans le nom
81 ## de fichier n'a aucune importance) sont lues et importées avec la
82 ## fonction 'scan'.
83 ##
84 ## Nous devons indiquer à la fonction que la ligne débutant par un #
85 ## est un commentaire.
86 (x <- scan("vecteur.data", comment.char = "#"))
87
88 ## La fonction 'scan' permet aussi de lire les données de
89 ## 'matrice.data'. La disposition des données dans le fichier n'a
90 ## aucune importance pour 'scan'. Il faut donc en recréer la structure
91 ## dans R.
92 ##
93 ## Cette fois, nous sautons simplement la ligne du fichier pour
94 ## omettre le commentaire.
95 (x <- matrix(scan("matrice.data", skip = 1),
96              nrow = 5, ncol = 4, byrow = TRUE))
97
98 ## L'importation des données de 'USArrests.txt', 'USArrests.csv' et
99 ## 'USArrests.csv2' est très simple avec les fonctions 'read.table',
100 ## 'read.csv' et 'read.csv2'.
101 ##
102 ## Prenez toutefois note: l'importation de données n'est pas toujours
103 ## aussi simple. Il faut souvent avoir recours aux multiples autres
```

```

104 ## arguments de 'read.table'
105 read.table("USArrests.txt")
106 read.csv("USArrests.csv")
107 read.csv2("USArrests.csv2")
108
109 ## Pour illustrer l'exportation et l'importation avec 'saveRDS' et
110 ## 'readRDS', nous allons utiliser le jeu de données 'HairEyeColor'
111 ## livré avec R. Il s'agit d'un tableau à trois dimensions 4 x 4 x 2.
112 ## Voilà une structure de données qu'il n'est pas simple d'exporter
113 ## avec les outils précédents: il faudrait d'abord réorganiser les
114 ## données en deux dimensions sous forme de tableau de données ou de
115 ## matrice.
116 HairEyeColor
117 dim(HairEyeColor)
118
119 ## Exportons d'abord la structure de données en format .rds.
120 saveRDS(HairEyeColor, "haireyecolor.rds")
121
122 ## Importons ensuite les données sérialisées.
123 x <- readRDS("haireyecolor.rds")
124
125 ## Les objets 'HairEyeColor' et 'x' sont identiques. La structure de
126 ## données a donc été exportée et recrée à l'identique.
127 identical(HairEyeColor, x)
128
129 ## Nettoyage: la fonction 'unlink' supprime les fichiers spécifiés en
130 ## argument, ici ceux créés précédemment dans le répertoire de
131 ## travail.
132 unlink(c("vecteur.data", "matrice.data",
133          "USArrests.txt", "USArrests.csv", "USArrests.csv2",
134          "haireyecolor.rds"))

```

9.6 Exercices

- 9.1** Importer dans R les données boston du dépôt de jeux de données de [StatLib](#).
Les données débutent à la ligne 23 du fichier. De plus, chaque entrée du fichier compte 14 variables, mais leurs valeurs sont chaque fois réparties sur deux lignes à raison de 11 variables sur une ligne et 3 sur l'autre.
- 9.2** Créer un tableau de données dans R, l'exporter en format CSV, puis ouvrir le fichier dans un tableur. Vérifier que le transfert des données s'est effectué correctement.

10 Bibliothèques et paquetages

Objectifs du chapitre

- ▶ Expliquer les concepts de bibliothèque et de paquetage dans R.
- ▶ Utiliser les fonctionnalités d'un paquetage dans une session de travail R.
- ▶ Installer de nouveaux paquetages R depuis le site *Comprehensive R Archive Network* (CRAN).

L'un des aspects de R ayant sans aucun doute le plus contribué à son succès est la possibilité — et la facilité — d'ajouter des extensions au système de base par le biais de paquetages. Toute personne utilisant R sera un jour appelée à utiliser des paquetages. Ce chapitre explique comment ajouter une bibliothèque personnelle au système et comment la garnir de paquetages téléchargés depuis le site *Comprehensive R Archive Network* (CRAN).

Dans la terminologie de R, un *paquetage* (*package*¹). est un ensemble cohérent de fonctions, de jeux de données et de documentation. Les paquetages sont regroupés dans une *bibliothèque* (*library*).



Insistons sur la terminologie puisque la confusion règne souvent hors de la documentation officielle de R : un paquetage est un ensemble de fonctions, alors qu'une bibliothèque est un ensemble de paquetages.

10.1 Système de base

La bibliothèque de base de R contient une trentaine de paquetages dont certains sont chargés par défaut au démarrage. La fonction `search` fournit la liste des paquetages chargés dans la session de travail, alors que `library` affiche le contenu de la bibliothèque de paquetages.

1. L'équipe de traduction française de R a choisi de conserver le terme anglais. Je ne les suis pas dans cette voie.

```
> search()
[1] ".GlobalEnv"          "package:RweaveExtra"
[3] "package:stats"       "package:graphics"
[5] "package:grDevices"   "package:utils"
[7] "package:datasets"    "package:methods"
[9] "AutoLoads"           "package:base"

> library()
Packages dans la bibliothèque
‘/Library/Frameworks/R.framework/(/...)/library’ :

base          The R Base Package
boot          Bootstrap Functions
              (Originally by Angelo Canty for S)
...
```

10.2 Utilisation d'un paquetage

Pour rendre disponibles les fonctionnalités d'un paquetage, peu importe sa provenance, il faut le charger dans la session de travail. C'est le rôle principal de la fonction `library`.

```
library("<paquetage>")
```

10.3 Création d'une bibliothèque personnelle

Nous verrons à la [section 10.4](#) comment ajouter des paquetages au système de base. Je vous recommande fortement de prévoir une bibliothèque personnelle pour contenir ces nouveaux paquetages. Tout d'abord, cela permet d'éviter les éventuels problèmes d'accès en écriture dans la bibliothèque principale, surtout sur des systèmes partagés. Ensuite, les paquetages placés dans une bibliothèque personnelle résistent mieux au processus de mise à jour de R que ceux ajoutés à la bibliothèque principale. Il s'agit d'un *préréglage absolu*² que vous aurez à effectuer une seule fois. L'investissement en temps — minime — en vaut largement la peine.

La mise sur pied d'une bibliothèque personnelle consiste simplement à créer un répertoire dans son système de fichier pour accueillir la bibliothèque, puis à indiquer à R, via un fichier de configuration, l'existence et l'emplacement de cette bibliothèque.

2. L'expression « *préréglage absolu* » est celle proposée par l'Office québécois de la langue française comme équivalent de l'expression anglaise *set and forget*.

Voici une procédure pas à pas pour compléter les opérations de création d'une bibliothèque personnelle depuis une ligne de commande Unix (Terminal sous macOS; MSYS de [MSYS2](#) ou Git Bash de [Git for Windows](#) sous Windows; terminal RStudio).

1. Créer les répertoires dans votre système de fichiers pour accueillir la bibliothèque personnelle. Sous macOS, je recommande d'utiliser le répertoire `~/Library/R/library`. Pour les autres systèmes d'exploitation, `~/R/library` devrait convenir. (Ici, « `~` » représente le dossier de départ tel qu'expliqué à la [section 1.4.2](#).)

macOS

```
$ mkdir -p ~/Library/R/library
```

autres systèmes d'exploitation

```
$ mkdir -p ~/R/library
```

2. Inscrire l'emplacement de la bibliothèque personnelle dans le fichier de configuration `~/.Renvi` que R lit au démarrage.

macOS

```
$ cat >> ~/.Renvi <<EOF  
R_LIBS_USER="~/Library/R/library"  
EOF
```

autres systèmes d'exploitation

```
$ cat >> ~/.Renvi <<EOF  
R_LIBS_USER="~/R/library"  
EOF
```

3. Démarrer ou redémarrer R et, depuis à la ligne de commande de R, vérifier que le chemin vers la bibliothèque personnelle apparaît dans les résultats de la commande `.libPaths`.

```
> .libPaths()  
[1] "/Users/vincent/Library/R/library"  
[2] "/Library/Frameworks/R.framework/(\...)/library"
```

4. Il n'y a même pas de quatrième étape.

La bibliothèque personnelle a toujours préséance sur la bibliothèque principale.




Consultez la rubrique d'aide de Startup pour les détails sur la syntaxe et l'emplacement des fichiers de configuration et celles de `library` et de `.libPaths` pour la gestion des bibliothèques.



Le fichier de script `paquetages.R` reproduit à la [section 10.6](#) propose, aux lignes 12-52, une procédure alternative — et en tous points équivalente — pour créer une bibliothèque personnelle entièrement depuis la ligne de commande de R.

10.4 Installation de paquets additionnels

Dès les débuts de R, les développeurs et utilisateurs ont mis sur pied le dépôt central de paquets *Comprehensive R Archive Network*  (CRAN). Ce site compte aujourd'hui des milliers de paquets et le nombre ne cesse de croître.

L'installation d'un paquet distribué via CRAN est très simple avec la fonction `install.packages`. Celle-ci télécharge le paquet spécifié en argument depuis CRAN et l'installe, par défaut, dans la première bibliothèque de `.libPaths`.

```
install.packages("<paquetage>")
```

Pour utiliser le paquet, vous devez d'abord le charger en mémoire avec `library`, tel qu'expliqué à la [section 10.2](#). R recherche le paquet spécifié dans les bibliothèques de `.libPaths`.

10.5 Mise à jour des bibliothèques de paquets

La fonction `update.packages` effectue la mise à jour depuis CRAN des bibliothèques fournies en argument ou, si aucun argument n'est fourni, de toutes les bibliothèques de paquets.

```
update.packages()
```



Étudiez l'intégralité du fichier de script `paquetages.R` reproduit à la [section 10.6](#).

10.6 Exemples

📍 Fichier d'accompagnement `paquetages.R`

```
11 ###
12 ### CRÉATION D'UNE BIBLIOTHÈQUE PERSONNELLE
13 ###
14
15 ## Liste des bibliothèques consultées par R. Votre bibliothèque
16 ## personnelle devrait y apparaître si vous avez suivi la procédure
17 ## expliquée dans le chapitre.
18 .libPaths()
19
20 ## Comme alternative à la procédure du chapitre, il est possible de
```



```
21 ## créer le répertoire pour la bibliothèque personnelle et d'inscrire
22 ## son emplacement dans le fichier ~/.Renvirom directement depuis R.
23 ##
24 ## Tout d'abord, la commande 'dir.create' crée un répertoire dans le
25 ## système de fichier. Adaptez les chemins d'accès selon vos besoins.
26 ##
27 ## Si vous êtes sous macOS, utilisez la commande suivante.
28 dir.create("~/Library/R/library", recursive = TRUE) # macOS
29
30 ## Pour tous les autres systèmes d'exploitation, utilisez plutôt la
31 ## commande suivante.
32 dir.create("~/R/library", recursive = TRUE) # autres OS
33
34 ## Ensuite, la commande 'cat' permet d'écrire directement dans un
35 ## fichier. Assurez-vous que le chemin d'accès correspond à celui
36 ## utilisé ci-dessus.
37 ##
38 ## Comme dans les instructions du chapitre, les commandes ci-dessous
39 ## ajoutent du contenu à un fichier si celui-ci existe déjà. Évitez
40 ## donc d'évaluer les expressions plusieurs fois.
41 ##
42 ## Si vous êtes sous macOS, utilisez la commande suivante.
43 cat('R_LIBS_USER=~/.Library/R/library"', file = "~/.Renvirom",
44     append = TRUE)
45
46 ## Pour tous les autres systèmes d'exploitation, utilisez plutôt la
47 ## commande suivante.
48 cat('R_LIBS_USER=~/.R/library"', file = "~/.Renvirom",
49     append = TRUE)
50
51 ## Vérification: afficher le contenu du fichier ~/.Renvirom.
52 file.show("~/Renvirom")
53
54 ###
55 ### SYSTÈME DE BASE
56 ###
57
58 ## La fonction 'search' retourne la liste des environnements dans
59 ## lesquels R va chercher un objet (en particulier une fonction).
60 ## '.GlobalEnv' est l'environnement de travail.
61 search()
62
63 ## Liste de tous les packages installés sur votre système. Noter que
64 ## MASS et tools en font partie. Ce sont des paquets livrés avec R,
65 ## mais qui ne sont pas chargés par défaut.
66 library()
67
68 ###
```

```
69 ### UTILISATION D'UN PAQUETAGE
70 ###
71
72 ## Chargement du package MASS qui contient plusieurs fonctions
73 ## statistiques très utiles.
74 library("MASS")
75
76 ###
77 ### INSTALLATION DE PAQUETAGES ADDITIONNELS
78 ###
79
80 ## Installation du paquetage actuar depuis un site miroir de CRAN
81 ## sélectionné automatiquement.
82 ##
83 ## Si vous avez configuré une bibliothèque personnelle et qu'elle
84 ## apparaît dans le résultat de '.libPaths()', ci-dessus, le paquetage
85 ## sera automatiquement installé dans celle-ci.
86 install.packages("actuar", repos = "https://cloud.r-project.org")
87
88 ## Chargement du paquetage dans la session de travail. R recherche le
89 ## paquetage dans toutes les bibliothèques de '.libPaths()'.
90 library("actuar")
91
92 ###
93 ### MISE À JOUR DES BIBLIOTHÈQUES DE PAQUETAGES
94 ###
95
96 ## Mise à jour de la bibliothèque personnelle.
97 update.packages(.libPaths()[1])
```

10.7 Exercices

- 10.1** Installer un paquetage disponible dans CRAN dans une bibliothèque personnelle sur votre poste de travail.


11 Analyse et contrôle de texte

*And now for something
completely different*

Monty Python

Objectifs du chapitre

- Utiliser les expressions régulières pour décrire ou identifier une chaîne de caractères.
- Effectuer l'analyse et le contrôle de texte à l'aide des outils informatiques standards `grep`, `sed` et `awk`.
- Effectuer des opérations de recherche et de remplacement de texte à l'aide d'expressions régulières dans R.


Vous êtes-vous déjà demandé comment s'effectue la validation de certains champs comme le code postal ou l'adresse de courrier électronique dans les formulaires électroniques ? Surement pas en vérifiant si l'entrée figure dans la liste des quelques 17,5 millions de codes postaux possibles au Canada ou, pire, parmi les milliards d'adresse de courriel que les **règles internationales**  permettent de concevoir ! Non, ce qu'il faut, c'est un « langage » qui permet de décrire *comment* une chaîne de caractères peut être composée, sans toutefois en fixer la composition exacte. Un tel langage existe : ce sont les *expressions régulières* ou expressions rationnelles.

Une expression régulière (*regular expression*; souvent abrégé « `regex` » ou « `rexp` ») est une suite de caractères typographiques qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles ([Wikipédia, 2024b](#)). L'expression elle-même est souvent appelée un « motif » (*pattern*). Les expressions régulières s'avèrent particulièrement utiles dans l'analyse de données textuelles et dans le traitement de la langue naturelle (*natural language processing*, NLP), une branche importante de l'intelligence artificielle.

Voici quelques utilisations possibles des expressions régulières.

- Rechercher du texte pouvant contenir des variations ou des fautes d'orthographe comme « je transfert » plutôt que « je transfère », ou encore les mul-

tiples variations autour du verbe « appeler » : appel, appelle, appellent, appeler, appelez, etc.

- Extraire les coordonnées géographiques d'un lieu (latitude et longitude) de l'URL d'une carte Google Maps. Par exemple, l'URL correspondant à la recherche « Université Laval »  est

```
https://www.google.ca/maps/place/Universite+Laval/
@46.7817463,-71.2769311,17z/data=!3m1!4b1!4m5!3m4!
1s0x4cb896c469ff32f9:0x15feb853bd2f8247!8m2!3d
46.7817463!4d-71.2747424
```

dont on décode que le lieu se trouve à une latitude de 46,781 746 3 et à une longitude de -71,276 931 1 en degrés décimaux.

- Extraire d'une base de données les adresses se trouvant sur une rue ou une avenue numérotée (« 1^{re} Avenue » ou « 4^e Rue »), et ce, peu importe comment est orthographié l'abréviation de l'adjectif numéral ordinal (« re » ou « ère », « e » ou « ième »).
- Sélectionner les entrées d'une base de données qui débutent ou qui se terminent par un caractère spécifique.
- Déterminer par programmation si un dépôt Git contient une branche autre que master en examinant le résultat de la commande `git branch`.
- Mettre en italique tous les mots d'un texte qui se trouvent entre guillemets (mais pas les guillemets eux-mêmes), et ce, quels que soient les mots.

Un jour ou l'autre, vous aurez à traiter des chaînes de caractères en programmation ou en analyse de données. Ce jour-là, connaître les expressions régulières vous sera d'un grand secours.



Dans la littérature informatique, la présentation des expressions régulières s'accompagne généralement de celle du langage de programmation Perl. En effet, Perl a été créé par Larry Wall en 1987 justement pour traiter facilement du texte. Le langage prend en charge les expressions régulières dans sa syntaxe même et il comporte plusieurs extensions aux expressions prises en charge par `grep` et `sed`. Le présent ouvrage ne traite pas de Perl.

11.1 Conventions typographiques et texte des exemples

La présentation des expressions régulières exige souvent de clairement délimiter un motif et d'afficher certains caractères invisibles, comme les espaces ou les retours à la ligne. Ce chapitre utilise les conventions typographiques suivantes.



Tiré de XKCD.com

- Les motifs sont encadrés de « coins » de couleur contrastante, comme « ceci » (une notation que j’emprunte à [Friedl, 2006](#)).
- Les espaces sont représentées par le symbole `␣`, ce qui permet, par exemple, de bien discerner que le motif « #␣␣␣␣A » compte quatre espaces entre # et A.
- La chaîne dans laquelle s’effectue une recherche est représentée en police non proportionnelle et les correspondances d’un motif dans celle-ci sont mis en surbrillance, comme dans l’exemple suivant :

le motif « chat » correspond à Les **chats** sont très mignons.

Dans les exemples, nous utiliserons le texte de la chanson « La journée qui s’en vient est flambant neuve » de l’album *Astronomie* (2012) du groupe québécois Avec pas d’casque. Le texte est fourni avec le présent document dans le fichier `chanson.txt`. Pour référence, il est également reproduit à la [figure 11.1](#).

chanson.txt

Oh comme il est lourd
Le temps qui s'appelle hier
Prends-le ce diamant
Dans ma tête il est pour toi
Je promets, je promets que
la journée qui s'en vient est flambant neuve

Je le connais bien
Le chemin du plus lâche
Mes bras désolés
Rampent comme des chiens
Je promets, je promets que
la journée qui s'en vient est flambant neuve

Même si le mouvement
Meurt mieux qu'il ne se charge
Même si la fatigue
Voudrait que tu deviennes
Je promets, je promets que
la journée qui s'en vient est flambant neuve

Le vent qui est bon
Est le même qui arrache
Nous avons l'outil
Il manque la manière
Je promets, je promets que
la journée qui s'en vient est flambant neuve

Frôle-moi de ton mieux
Frôle-moi davantage
Gave-moi de ton amour
Pour shimmer l'Univers
Je promets, je promets que
la journée qui s'en vient est flambant neuve

Et nous l'habiterons avec
Nos yeux qui s'habituent à la noirceur
Et nous l'abîmerons avec
Nos plus beaux accidents
Je promets, je promets que
la journée qui s'en vient est flambant neuve

FIG. 11.1 – Texte de la chanson *La journée qui s'en vient est flambant neuve* de Avec pas d'casque

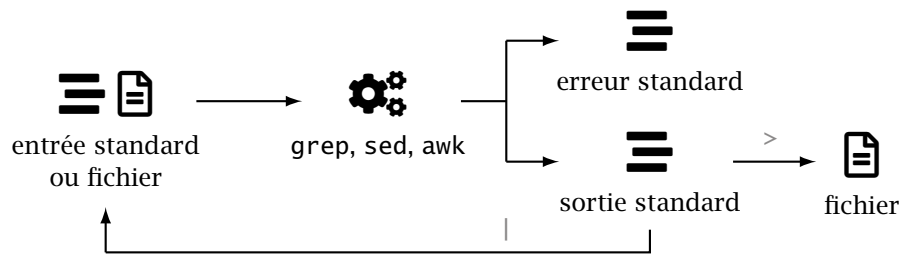


FIG. 11.2 – Flux des données à la ligne de commande Unix

11.2 Outils d'analyse et de contrôle du texte

L'expression régulière n'est qu'un langage de description d'une chaîne de texte. Pour exploiter ce langage, des outils sont nécessaires. Aux fins de cet ouvrage, j'ai choisi de concentrer notre étude sur les utilitaires Unix standards en ligne de commande `grep`, `sed` et `awk`. Ils font partie intégrante des systèmes d'exploitation macOS et Linux et, sous Windows, ils sont livrés avec les interpréteurs de commande Git Bash de [Git for Windows](#) et MSYS de [MSYS2](#).

Il va sans dire que les utilitaires `grep`, `sed` et `awk` s'utilisent depuis une ligne de commande Unix : dans le Terminal sous macOS, ou dans Git Bash ou MSYS sous Windows. Si vous n'êtes pas familier avec la ligne de commande, [Goulet \(2024b\)](#) offre une rapide introduction suffisante pour suivre la matière de ce chapitre. Nous reviendrons à R à la [section 11.5](#) lorsque nous étudierons les fonctions internes de R aux fonctionnalités équivalentes à `grep` et `sed`.

Comme la plupart des utilitaires Unix, `grep`, `sed` et `awk` prennent en entrée un flux de texte ou un ou plusieurs fichiers en format texte brut. Ils opèrent ensuite sur cette entrée *ligne par ligne*. Le résultat du traitement est affiché à la sortie standard (le terminal), transféré à un autre programme avec le tuyau « `|` », ou encore redirigé vers un fichier avec l'opérateur « `>` ». La [figure 11.2](#), reprise de [Goulet \(2024b\)](#) avec quelques simplifications, illustre ce flux des données.

Les outils `grep`, `sed` et `awk` diffèrent quant au type de traitement qu'ils peuvent effectuer.

grep sélectionne les lignes en entrée correspondant à une expression régulière.

sed sert surtout pour rechercher et remplacer du texte.


awk permet de traiter aisément du texte séparé en champs, ainsi que du texte réparti sur plusieurs lignes.

Puisque nous utiliserons les utilitaires `grep` et `sed` à profusion dans la présentation sur les expressions régulières à la [section 11.3](#), prenons un moment pour apprendre comment les utiliser. L'utilitaire `awk`, quant à lui, fera l'objet de la [section 11.4](#).



L'espace horizontal étant compté dans un document en format PDF, il est parfois nécessaire de scinder une commande sur deux lignes ou plus. Vous remarquerez dans ces cas la présence du symbole « \ » en fin de ligne. C'est le symbole de continuation de ligne de Bash qui permet d'écrire une commande sur plus d'une ligne. Il ne fait pas partie de la syntaxe des commandes.

11.2.1 Extraction de lignes avec grep

L'utilitaire `grep`¹ prend en argument une expression régulière (habituellement entre guillemets simples ou doubles), lit l'entrée standard ou une liste de fichiers ligne par ligne, puis retourne sur la sortie standard les lignes qui contiennent des correspondances avec l'expression régulière. C'est très utile pour savoir si un fichier contient — ou quels fichiers contiennent — un mot ou une expression, assez pour que le nom de l'utilitaire soit devenu, en anglais, un verbe dans le **jargon informatique**  (*to grep*).

Pour reprendre plus formellement ce qui précède, la syntaxe d'une commande `grep` est :

```
grep <options> '<motif>' <fichiers>
```


- ▶ *<options>* est une liste facultative d'options (ou drapeaux) qui modifient le comportement de `grep`.
- ▶ *<motif>* est une expression régulière qui définit le texte à rechercher.
- ▶ *<fichiers>* est le ou les fichiers dans lesquels rechercher le texte ; si aucun fichier n'est spécifié, `grep` recherche dans l'entrée standard.

Les trois exemples ci-dessous permettent, dans l'ordre, d'extraire du fichier `chanson.txt` : le vers de la chanson qui contient le mot « temps » ; les vers qui débutent par « M » ; le vers qui contient le mot « chat » (aucun²).

```
$ grep 'temps' chanson.txt
Le temps qui s'appelle hier
```

```
$ grep '^M' chanson.txt
Mes bras désolés
Même si le mouvement
Meurt mieux qu'il ne se charge
Même si la fatigue
```

1. Le nom de l'utilitaire provient de la commande `:g/re/p` de l'antédiluvien éditeur de texte `ed` qui permet de rechercher globalement (`g`) une expression régulière (*regular expression*, `re`) et d'afficher (*print*, `p`) les lignes qui correspondent à l'expression.

2. Sachez que la **vidéo**  de la chanson est, elle, emplie de chatons très mignons.


```
$ grep 'chat' chanson.txt
```

Les *<options>* mentionnées ci-dessus pour modifier le comportement de `grep` se présentent généralement en deux versions : courte sous forme d'une seule lettre précédée d'un tiret, ou longue sous forme d'un mot (ou une expression) précédé de deux tirets. Les options de `grep` sont très nombreuses ; je n'en ai retenu que quelques unes ici.

- E, --extended-regexp interprète le motif comme une expression régulière étendue.
- c, --count retourne seulement le nombre de lignes qui correspondent au motif.
- o, --only-matching retourne uniquement la portion de la ligne qui correspond au motif, plutôt que la ligne au complet.
- v, --invert-match sélectionne les lignes qui ne correspondent *pas* au motif.

Nous aurons l'occasion d'illustrer chacune de ces options dans la suite.

11.2.2 Recherche et remplacement avec `sed`

L'utilitaire `sed`³ est en quelque sorte un éditeur de texte qui opère sur un texte une seule ligne à la fois et de manière non-interactive. (Vous pourriez objecter que cela ne correspond en rien à l'idée que l'on se fait d'un éditeur de texte, mais il n'en reste pas moins que le programme effectue des tâches d'édition sur du texte.) L'éventail des traitements que `sed` peut appliquer à un texte est très vaste, mais la syntaxe des commandes qu'il faut entrer pour y arriver est particulièrement ésothérique⁴. Nous nous attarderons donc à une seule commande de `sed`, celle qui permet de rechercher et de remplacer à l'intérieur d'un texte. La commande est fournie en argument à `sed` entre guillemets (simples ou doubles). L'utilitaire lit ensuite l'entrée standard ou une liste de fichiers ligne par ligne. La syntaxe est donc la suivante :

```
sed <options> 's/<motif>/<remplacement>/<modificateur>'
    <fichiers>
```

- *<options>* est une liste facultative d'options (ou drapeaux) qui modifient le comportement de `sed`.
- `s` est le nom de la commande `sed` qui sert à rechercher et remplacer.
- *<motif>* est une expression régulière qui définit le texte à rechercher.

3. Pour *stream editor*, ou éditeur de flux de texte. Comme `grep`, `sed` tire son origine de l'éditeur `ed`.

4. Jugez-en : la commande pour supprimer l'une de deux lignes blanches consécutives dans un texte est `N;/^\n$/D;P;D;.`

- ▶ *⟨remplacement⟩* est le texte qui sert à remplacer le texte qui correspond à *⟨motif⟩*.
- ▶ *⟨modificateur⟩* est un symbole qui permet de modifier le comportement de la commande; nous utiliserons uniquement le modificateur **g** (pour « global ») qui indique de remplacer toutes les correspondances de *⟨motif⟩* sur une ligne, et non seulement la première.
- ▶ « / » est un symbole qui sert à délimiter les différentes parties de la commande; n'importe quel symbole (qui ne se trouve pas autrement dans la commande) fait l'affaire, mais « / » est le symbole le plus couramment utilisé.
- ▶ *⟨fichiers⟩* est le ou les fichiers dans lesquels rechercher le texte; si aucun fichier n'est spécifié, **sed** recherche dans l'entrée standard.

Contrairement à ce que l'on pourrait penser, **sed** ne modifie pas le ou les fichiers d'origine. Le texte modifié est simplement affiché à la sortie standard. Pour sauvegarder ce texte modifié, vous devez rediriger la sortie standard vers un fichier avec l'opérateur de redirection « > » comme expliqué à la [section 11.2](#). Le nom de ce fichier doit être différent de celui que **sed** est occupé à traiter⁵.

Les trois exemples ci-dessous permettent de faire les changements suivants dans le texte du fichier `chanson.txt`, dans l'ordre : remplacer « Oh » par « Ah »; remplacer « mm » par « MM »; remplacer toutes les occurrences de « promets » sur une ligne par « jure ». (J'ai abrégé les sorties des commandes pour économiser de l'espace.)

```
$ sed 's/Oh/Ah/' chanson.txt
Ah comme il est lourd
Le temps qui s'appelle hier
...
```

```
$ sed 's/mm/MM/' chanson.txt
Oh coMMe il est lourd
Le temps qui s'appelle hier
...
Mes bras désolés
Rampent coMMe des chiens
...
Gave-moi de ton amour
Pour shiMMer l'Univers
...
```

```
$ sed 's/promets/jure/g' chanson.txt
...
```

5. Il existe une option pour remplacer directement dans le fichier d'origine, mais elle n'est pas standard dans toutes les versions de **sed**. Je vous recommande de ne pas vous y fier.

```
Dans ma tête il est pour toi
Je jure, je jure que
la journée qui s'en vient est flambant neuve
...
Rampent comme des chiens
Je jure, je jure que
la journée qui s'en vient est flambant neuve
...
Voudrait que tu deviennes
Je jure, je jure que
la journée qui s'en vient est flambant neuve
...
Il manque la manière
Je jure, je jure que
la journée qui s'en vient est flambant neuve
...
Pour shimmer l'Univers
Je jure, je jure que
la journée qui s'en vient est flambant neuve
...
Nos plus beaux accidents
Je jure, je jure que
la journée qui s'en vient est flambant neuve
```

Il n'est pas tout à fait exact de dire que `sed` traite une seule ligne à la fois. En effet, il est possible d'appliquer des commandes sur un intervalle de lignes par le biais d'une mémoire tampon (*hold space*). Nous n'aurons pas recours à la mémoire tampon de `sed` dans le présent ouvrage.




Il existe plusieurs versions différentes de `grep`, `sed` et `awk`, parfois avec des différences importantes, surtout dans le cas de `sed`. Les versions sous Linux, MSYS2 et Git Bash sont généralement celles du **projet GNU** [↗](#). Dans macOS, elles proviennent de **BSD** [↗](#). J'ai tâché de fournir ici des expressions universelles qui fonctionnent dans ces deux principales variantes.



Le fichier de script `texte.sh` reproduit à la [section 11.6](#) contient des exemples d'utilisation de `grep`, `sed` et `awk`. Vous pouvez ouvrir le fichier de script dans votre éditeur de texte pour facilement copier les commandes à la ligne de commande, ou encore évaluer celle-ci directement dans le terminal RStudio tel qu'expliqué à la [section A.6](#). Étudiez pour l'instant les lignes [28–128](#).

11.3 Expressions régulières

Tel que mentionné dans l'introduction de ce chapitre, une expression régulière, ou expression rationnelle, est un langage, muni d'une syntaxe précise, qui permet de décrire un ensemble de chaînes de caractères possibles. Vous avez rencontré quelques expressions régulières très simples et intuitives à la section précédente. Ainsi, l'expression régulière «`chat`» correspond à `chat`, mais aussi à `chats`, `chaton` ou `achat`. Nous verrons dans la suite comment restreindre l'expression au seul mot « chat ».

Cette section propose une introduction au langage des expressions régulières. Il existe une multitude de tutoriels et de documents de référence sur le sujet dans Internet et sous forme de livre. Pour vous accompagner dans l'atteinte des objectifs d'apprentissage mentionnés au début du chapitre, je me suis inspiré de l'excellent *Shell Scripting Primer*  de la bibliothèque pour développeurs de Apple (Apple). Si vous souhaitez étudier les expressions régulières vraiment en profondeur, je vous recommande l'ouvrage classique de Friedl (2006).

11.3.1 Types d'expressions régulières

Avant d'aller plus loin, vous devez savoir qu'il existe trois grands types d'expressions régulières : les expressions régulières basiques ou classiques (*Basic Regular Expressions*, BRE), les expressions régulières étendues (*Extended Regular Expressions*, ERE) et les expressions régulières de Perl (*Perl Compatible Regular Expressions*, PCRE). Les utilitaires `grep` et `sed` utilisent les expressions régulières basiques par défaut.

Voici les principales différences entre les expressions régulières basiques et étendues. Si ce qui suit n'est pas tout à fait clair à la première lecture, ça le deviendra dès la prochaine section lorsque nous entamerons l'étude des opérateurs des expressions régulières.

- Dans les expressions régulières étendues, les symboles «`+`» et «`?`» sont des opérateurs de répétition. Dans les expressions régulières basiques, ce sont des caractères ordinaires et il n'existe aucun équivalent pour leur fonctionnalité respective.
- Dans les expressions régulières étendues, le symbole «`|`» est un opérateur de choix entre plusieurs possibilités. Dans les expressions régulières basiques, c'est un caractère ordinaire et il n'existe aucun équivalent pour sa fonctionnalité.
- Dans les expressions régulières étendues, les parenthèses «`()`» et les accolades «`{ }`» sont, respectivement, des opérateurs de regroupement et de quantificateur borné. Dans les expressions régulières basiques, ce sont des caractères ordinaires et il faut les précéder du caractère d'échappement «`\`» pour les convertir en opérateurs.

Dans la suite, lorsqu'il existe une différence entre les expressions régulières basiques et les expressions régulières étendues, nous utiliserons toujours ces dernières. Pour indiquer à `grep` et à `sed` d'utiliser la syntaxe des expressions régulières étendues, il suffit de leur ajouter l'option `-E` mentionnée à la [section 11.2.1](#).

Dans le premier exemple ci-dessous, le motif `«chien|chat»` est interprété par `grep` comme une expression régulière basique dans laquelle le symbole `«|»` est un caractère comme les autres. Comme le fichier `chanson.txt` ne contient pas la chaîne `chien|chat`, le motif ne correspond à aucune ligne. Dans le second exemple `grep` est invoqué avec l'option `-E` et, par conséquent, le motif est interprété comme une expression régulière étendue qui signifie plutôt : *chien ou chat*.

```
$ grep 'chien|chat' chanson.txt
```

```
$ grep -E 'chien|chat' chanson.txt
Rampent comme des chiens
```



Évitez-vous bien des maux de tête en utilisant toujours `grep` et `sed` avec l'option `-E`.

11.3.2 Syntaxe de base

Comme mentionné précédemment, une expression régulière est une suite de caractères typographiques qui décrit une chaîne de texte. Elle est composée de caractères littéraux (dont la « valeur » correspond au symbole lui-même) et de caractères spéciaux qui font office d'opérateurs⁶. La syntaxe obéit à quelques règles fondamentales en définitive assez simples :

1. Un caractère littéral correspond à la *première* occurrence de ce caractère dans la chaîne de texte. Par exemple, le motif `«a»` correspond à `Gave-moi de ton amour`. Que la lettre se trouve à l'intérieur d'un mot n'a pas d'importance.
2. Une expression régulière peut être formée par concaténation de deux ou plusieurs expressions régulières. Elle correspond alors à toute chaîne de texte qui correspond aux sous-expressions. Autrement dit, les composantes d'une expression régulière sont par défaut reliées par une clause « et ». Dans l'exemple précédent, le motif `«am»` correspond à `Gave-moi de ton amour`.
3. Une douzaine d'opérateurs permettent de rechercher dans une chaîne de texte autre chose que des caractères littéraux.

Vous pouvez visualiser le texte auquel un motif correspond à l'aide d'un testeur d'expressions régulières en ligne⁷, ou encore simplement en remplaçant les

6. Les opérateurs ou caractères spéciaux des expressions régulières sont aussi parfois appelés *métacaractères*.

7. Recherchez *regular expression tester* pour en obtenir une liste et choisissez votre favori.

TAB. 11.1 – Principaux opérateurs des expressions régulières étendues

Opérateur	Fonction
	alternance (« ou »)
^	début de la ligne ; négation (à l'intérieur d'une classe)
\$	fin de la ligne
.	caractère de remplacement
*	zéro, une ou plusieurs occurrences
+	une ou plusieurs occurrences
?	zéro ou une occurrence
()	groupe de caractères
[]	classe de caractères
{ }	quantificateur du nombre d'occurrences
\	caractère d'échappement

correspondances par du texte quelconque avec `sed`. Je vous laisse exécuter les deux commandes ci-dessous et en étudier attentivement les résultats.

```
$ sed 's/a!/!' chanson.txt
```

```
$ sed 's/am/--/' chanson.txt
```

Le [tableau 11.1](#) présente les opérateurs des expressions régulières étendues. Dans la suite, nous allons graduellement les intégrer dans les expressions régulières afin de sans cesse augmenter l'éventail des motifs disponibles.



L'espace est un « caractère » comme les autres dans les expressions régulières. Le motif `re` correspond donc à `Gave-moi de ton amour`, alors que le motif `re_` correspond plutôt à `Gave-moi de ton amour`.



Avant d'entrer véritablement dans le vif du sujet, prenez le temps d'étudier les quelques exemples d'expressions régulières aux lignes [134-208](#) du fichier de script `texte.sh` reproduit à la [section 11.6](#).

11.3.3 Remplacement et répétition

Nous avons établi que l'un des principes de base des expressions régulières est qu'un caractère littéral correspond à lui-même dans un motif. Ce qu'il nous manque pour augmenter considérablement l'éventail des motifs disponibles, c'est une façon de signifier « n'importe quel caractère » et « n'importe quelle suite de

caractères ». C'est le rôle du caractère de remplacement et des opérateurs de répétition.

Dans les expressions régulières, le point « . » est le caractère de remplacement (*wildcard character*). Par exemple, le motif «`le.`» correspond à `le` suivi d'un caractère quelconque, *y compris l'espace*. Par conséquent, tant `les` que `lent` ou `elle` `mange` correspondent à ce motif, mais pas `be` `le` placé en fin de ligne.

Effectuons une autre recherche dans le fichier `chanson.txt` : les lignes contenant un mot se terminant par `ou` et une lettre quelconque. Nous allons délimiter un mot en supposant que la chaîne recherchée est suivie d'une espace — ce qui a pour effet d'exclure les mots situés en fin de vers. Le motif approprié pour la recherche est «`ou.`».

```
$ grep 'ou.' chanson.txt
Dans ma tête il est pour toi
Nous avons l'outil
Pour shimmer l'Univers
Et nous l'habiterons avec
Et nous l'abîmerons avec
```

Affichons, à l'aide de l'option `-o` de `grep`, la portion de chaque ligne qui correspond au motif dans l'exemple précédent.

```
$ grep -o 'ou.' chanson.txt
our
ous
our
ous
ous
```

Les opérateurs de répétition (ou de quantification), quant à eux, permettent de décrire des suites de caractères. Comme le montre le [tableau 11.1](#), il existe quatre opérateurs de répétition dans les expressions régulières (étendues) : les caractères réservés «`*`», «`+`» et «`?`», ainsi que les accolades «`{ }`». Ces opérateurs s'appliquent à ce qui les précède immédiatement.

L'opérateur «`*`» signifie : « ce qui précède, zéro ou plusieurs fois ». Ainsi le motif «`a*b`» correspond, entre autres, aux chaînes `b`, `ab` ou `aaab`. C'est le seul opérateur de répétition aussi pris en charge par les expressions régulières basiques.

L'opérateur «`+`» signifie : « ce qui précède, une ou plusieurs fois ». Ainsi le motif «`a+b`» correspond aux chaînes `ab` ou `aaab`, mais pas à `b`.

Quant à l'opérateur «`?`», il signifie : « ce qui précède, zéro ou une fois ». En d'autres termes, «`?`» rend le caractère qui précède optionnel dans la chaîne recherchée. Par exemple, le motif «`ab?c`» correspond aux chaînes `ac` et `abc`.

Combinés avec le caractère de remplacement, les trois opérateurs de répétition ci-dessus permettent de décrire des chaînes de caractères très générales puisque l'opération « n'importe quel caractère » s'en trouve répétée autant de fois que

TAB. 11.2 – Comparaison des trois variantes des quantificateurs bornés

Motif	Correspondances					
<code>「ah{2}」</code>	ah	ahh	ahhh	ahhhh	ahhhhh	ahhhhhh
<code>「ah{2,}」</code>	ah	ahh	ahhh	ahhhh	ahhhhh	ahhhhhh
<code>「ah{2,4}」</code>	ah	ahh	ahhh	ahhhh	ahhhhh	ahhhhhh

nécessaire. Ce sont les combinaisons d'opérateurs les plus utilisées dans les expressions régulières.

`「.*」` correspond à une chaîne de caractères quelconques de longueur arbitraire, y compris zéro.

`「.+」` correspond à une chaîne de caractères quelconques longue d'au moins un caractère; équivalent à `「.*」`.

`「.?」` correspond à aucun ou un seul caractère quelconque.

Comparez attentivement les résultats des deux exemples ci-dessous.

```
$ grep -E 'Le .*e' chanson.txt
Le temps qui s'appelle hier
Le chemin du plus lâche
Le vent qui est bon
```

```
$ grep -E 'Le .?e' chanson.txt
Le temps qui s'appelle hier
Le vent qui est bon
```

Enfin, les accolades « `{ }` » font office de quantificateurs bornés, c'est-à-dire qu'elles permettent de spécifier le nombre minimal de répétitions *requises* et, de manière optionnelle, le nombre maximal de répétitions *permises*. Il existe trois manières de les utiliser. (Dans ce qui suit, n et m sont des entiers et $m > n$.)

`{n}` répète ce qui précède exactement n fois.

`{n,}` répète ce qui précède au moins n fois.

`{n,m}` répète ce qui précède au moins n fois et jusqu'à m fois.

Les opérateurs « `*` », « `+` » et « `?` » sont donc simplement des raccourcis pour, dans l'ordre, `{0,}`, `{1,}` et `{0,1}`. Le [tableau 11.2](#) fournit des exemples de correspondances pour chaque variante des accolades.



Les quantificateurs des expressions régulières sont par défaut gloutons (*greedy*), c'est-à-dire qu'ils trouvent la plus longue chaîne possible en accord avec un motif. Par exemple, le motif `「L.+c」` correspond à `Le chemin du plus lâche`, et non à seulement `Le chemin du plus lâche`. Il existe aussi des quantificateurs non gloutons; vous pourrez les rechercher lorsque vous souhaitez restreindre la portée d'un quantificateur à la plus petite chaîne possible. Vous trouverez également dans les exemples de la [section 11.6](#) une solution de contournement.

11.3.4 Caractère d'échappement

Si les caractères «`*`», «`+`» ou «`?`» — en fait, tous ceux du [tableau 11.1](#) — ont une signification particulière dans les expressions régulières, comment alors rechercher ces caractères dans du texte? C'est là qu'entre en jeu le caractère d'échappement «`\`». Lorsqu'il précède un symbole, le caractère d'échappement désactive la signification spéciale de ce symbole. Par exemple, le motif `「\.`」 trouve un accord avec le caractère «`.`». Pour rechercher un nom de fichier long d'un à huit caractères et suivi d'une extension `.txt`, un motif approprié⁸ serait donc : `「.{1,8}\.txt」`.

11.3.5 Marqueurs de position

Les expressions régulières comportent deux marqueurs de position qui permettent de fixer une condition pour qu'un motif trouve un accord dans une chaîne de caractères : le caret (ou accent circonflexe) «`^`» et le signe de dollar «`$`». Ils identifient respectivement le début et la fin d'une ligne de texte.

Par exemple, la première commande ci-dessous identifie uniquement les lignes qui contiennent `la` en début de ligne, alors que la seconde retient seulement les lignes qui contiennent `que` en fin de ligne. (Il est laissé en exercice de comparer avec les résultats sans les marqueurs de position.)

```
$ grep '^la' chanson.txt
la journée qui s'en vient est flambant neuve
la journée qui s'en vient est flambant neuve
la journée qui s'en vient est flambant neuve
la journée qui s'en vient est flambant neuve
la journée qui s'en vient est flambant neuve
la journée qui s'en vient est flambant neuve
```

8. Cet exemple très simplifié ne tient pas compte des éventuelles règles de validité d'un nom de fichier.

```
$ grep 'que$' chanson.txt
Je promets, je promets que
Je promets, je promets que
Je promets, je promets que
Je promets, je promets que
Je promets, je promets que
Je promets, je promets que
```

Combien de lignes de `chanson.txt` débutent par `la`? C'est un travail pour `grep` muni de l'option `-c`.

```
$ grep -c '^la' chanson.txt
6
```

Observez maintenant le motif particulier `^$` et prenez une minute pour déterminer à quoi il correspond.

Vous avez trouvé?


Le motif décrit un début de ligne suivi immédiatement d'une fin de ligne, soit... une ligne blanche! Ce motif est surtout utile lorsque combiné avec l'option `-v` de `grep` qui permet d'inverser les résultats. Ainsi, la commande suivante retourne toutes les lignes du fichier `chanson.txt` qui ne sont *pas* vides. Autrement dit, la commande supprime les lignes blanches du fichier. (Question d'économiser de l'espace en ces pages, je vous laisse le soin d'en faire la vérification.)

```
$ grep -v '^$' chanson.txt
```



Nous avons déjà couvert plusieurs éléments importants de la syntaxe des expressions régulières. Étudiez les lignes 210-334 du fichier de script `texte.sh` reproduit à la [section 11.6](#).

11.3.6 Classes de caractères

La rectification de l'orthographe du français — ou **nouvelle orthographe**  — a mené à la disparition de l'accent circonflexe sur les lettres *i* et *u* : *chaîne* plutôt que *chaîne*, *disparaître* plutôt que *disparaître*, *bruler* plutôt que *brûler*, etc. Si vous devez rechercher l'une ou l'autre graphie dans un texte, vous souhaitez pouvoir les décrire toutes les deux à l'aide d'une seule expression régulière. C'est le rôle des opérateurs « `[]` » qui définissent ce que l'on appelle une classe de caractères. Par exemple, `[iî]ne` correspond à la fois à `chaîne` et à `chaîne`.

Rappelez-vous la seconde règle fondamentale de la syntaxe des expressions régulières énoncée à la [section 11.3.2](#) : il y a un « et » implicite entre les composantes d'une expression régulière. À l'intérieur d'une classe de caractères, c'est l'inverse : les composantes de la classe sont reliées par un « ou ».

Voici un autre exemple pour lequel les classes de caractères sont souvent utilisées : permettre qu'un mot débute par une majuscule ou une minuscule.

```
$ grep '[Mm]ême' chanson.txt
Même si le mouvement
Même si la fatigue
Est le même qui arrache
```

Une classe peut contenir autant de caractères que vous le souhaitez. À titre d'exemple, `[123456]` trouve une correspondance avec n'importe lequel des entiers de 1 à 6. Question de simplifier la composition de longues listes de caractères, le tiret « - » a une signification particulière lorsqu'il est employé à l'intérieur d'une classe : il définit une plage de caractères. Ainsi, `[1-6]` est équivalent à l'exemple précédent. Les plages les plus fréquemment employées sont les suivantes.

`[0-9]` les entiers de 0 à 9.

`[a-z]` les lettres minuscules de a à z, sans les lettres accentuées.

`[A-Z]` les lettres majuscules de A à Z, sans les lettres accentuées.

Je dois immédiatement apporter une précision à ce qui précède : le tiret devient un opérateur à l'intérieur d'une classe de caractères seulement lorsqu'il est employé *ailleurs que comme premier caractère* de la classe. Comme il ne saurait définir une plage en se trouvant au début de la classe, il conserve alors son rôle de caractère littéral. Cela permet d'inclure le tiret lui-même dans une classe de caractères : le motif `[-A-F]` recherche le tiret en plus des lettres majuscules de A à F.

Lorsque le caret « ^ » suit immédiatement le crochet ouvrant d'une classe, la signification de celle-ci est renversée pour correspondre à tous les caractères qui ne se trouvent *pas* dans la classe. Par exemple, le motif `[^aeiouy]` correspond à toute consonne, ce qui est plus simple que de dresser la liste des vingt lettres. L'expression ci-dessous permet, quant à elle, d'identifier dans le fichier `chanson.txt` les occurrences de `ou` qui ne sont pas suivies de la lettre `r`, ni de la lettre `s`.

```
$ grep 'ou[^\rs]' chanson.txt
Même si le mouvement
Voudrait que tu deviennes
Nous avons l'outil
```

Les caractères spéciaux autres que le caret et le tiret perdent leurs significations particulières à l'intérieur des classes de caractères. Il est donc possible d'identifier les différentes marques de fin de phrase avec `[.?!]`.



Les classes de caractères comportent en quelque sorte leur propre syntaxe et leurs opérateurs spécifiques. Il s'avère simplement que certains symboles — en particulier « ^ » — jouent un rôle différent à l'intérieur et à l'extérieur des classes de caractères.

Il y aurait encore une foule de petits détails à couvrir au sujet des classes de

TAB. 11.3 – Classes de caractères prédéfinies de la norme POSIX

Nom	Signification
<code>[:alnum:]</code>	caractères alphabétiques et numériques
<code>[:alpha:]</code>	caractères alphabétiques selon les paramètres régionaux en vigueur
<code>[:blank:]</code>	espace et tabulation
<code>[:cntrl:]</code>	caractères de contrôle
<code>[:digit:]</code>	caractères numériques
<code>[:graph:]</code>	caractères visibles (tout sauf les espaces et les caractères de contrôle)
<code>[:lower:]</code>	minuscules
<code>[:print:]</code>	caractères et espaces visibles
<code>[:punct:]</code>	ponctuations et symboles
<code>[:space:]</code>	tous les caractères d'espacement, y compris les sauts de ligne
<code>[:upper:]</code>	majuscules
<code>[:xdigit:]</code>	caractères numériques hexadécimaux (0-9, a-f, A-F)

caractères, mais pour les fins de cet ouvrage je me contenterai de traiter brièvement des classes prédéfinies⁹. Celles-ci peuvent s'avérer particulièrement utiles pour travailler avec des lettres accentuées ou des caractères invisibles. La liste des classes prédéfinies varie beaucoup d'un système d'expressions régulières à un autre. Je fournis au [tableau 11.3](#) les classes prédéfinies de ce qui constitue généralement le dénominateur commun, soit la norme POSIX.



Les crochets font partie intégrante du nom des classes prédéfinies. Comme nous utilisons celles-ci dans une classe de caractères, il faut dédoubler les crochets. Par exemple, le motif pour rechercher un caractère numérique à l'aide d'une classe prédéfinie est `[[[:digit:]]]`, alors que celui pour exclure toute lettre majuscule est `[^[[:upper:]]]`.

11.3.7 Alternance

Nous arrivons à l'un des derniers opérateurs du [tableau 11.1](#) : le symbole « `|` » qui représente l'alternance, ou le choix, entre deux sous-expressions d'une expression régulière. Par exemple, tel que déjà mentionné à la [section 11.3.1](#), le motif

9. J'ai tenté de contourner une source possible de confusion dans la terminologie anglaise relative aux classes de caractères. Alors que plusieurs auteurs les nomment tout simplement *character class*, la norme POSIX — déjà abordée au [chapitre 5](#) lorsqu'il s'agissait de la représentation des dates — les nomme *bracket expression*. La norme POSIX réserve le terme *character class* aux classes prédéfinies. Prenez garde à cette nuance si vous consultez de la documentation en anglais.

«chien|chat» correspond à **chien** ou **chat**. Voici un autre exemple ayant recours à notre fichier `chanson.txt`¹⁰.

```
$ grep -E 'mouvement|fatigue' chanson.txt
Même si le mouvement
Même si la fatigue
```

Retournons à notre exemple «cha[iî]ne» de la [section 11.3.6](#). Nous pouvons ré-écrire le motif ainsi : «chaîne|chaîne», ou encore «cha(i|î)ne». Dans ce dernier cas, nous devons avoir recours aux parenthèses — sur lesquelles nous élaborerons à la section suivante — afin de limiter la portée de l’alternance. Attention, toutefois : «cha[i|î]ne» n’est pas ce que nous voulons, car «|» est un caractère littéral à l’intérieur d’une classe de caractères.

À ce stade, vous devez faire la distinction entre les concepts d’alternance et de classe de caractère. La classe de caractère ne trouve des correspondances qu’avec *un seul caractère* à l’intérieur de la classe. En revanche, chaque possibilité d’une alternance étant une expression régulière en soi, cette opération peut décrire une quantité de texte de longueur quelconque.

Terminons cette section avec un exemple de substitution de texte qui regroupe plusieurs concepts. Supposons que l’on souhaite remplacer les mots *mouvement* et *fatigue* dans *La journée qui s’en vient est flambant neuve* par *pendule* (un choix complètement arbitraire). Le mot étant masculin, il faut également modifier le déterminant. Voici quatre commandes `sed` pour effectuer le traitement. Analysez ces commandes attentivement et essayez d’identifier leurs différences, s’il y en a. Nous y reviendrons plus loin.

```
$ sed -E 's/le mouvement|la fatigue/le pendule/' \
chanson.txt
```

```
$ sed -E 's/l[ea] (mouvement|fatigue)/le pendule/' \
chanson.txt
```

```
$ sed -E 's/le|la mouvement|fatigue/le pendule/' \
chanson.txt
```

```
$ sed -E 's/(le|la) (mouvement|fatigue)/le pendule/' \
chanson.txt
```

11.3.8 Groupes

Les parenthèses () sont des opérateurs de regroupement dans les expressions régulières étendues. Elles ont trois rôles principaux. Nous avons vu le pre-

10. Rappelons que le caractère «|» n’est un opérateur que dans les expressions régulières étendues, ce qui nous oblige à utiliser l’option `-E` dans `grep` et `sed`.

mier à la section précédente, soit de limiter la portée de l'opérateur d'alternance « | ».

Nous pouvons également utiliser les parenthèses pour créer un groupe de caractères auquel sera appliqué un des opérateurs de répétition de la [section 11.3.3](#). Cela permet de répéter autre chose qu'un caractère seul. Par exemple, le motif « (co){2} » permet d'identifier dans un dictionnaire français¹¹ des mots comme **cocon**, **cocotier** ou **rococo**.

Enfin, les parenthèses permettent de créer des groupes de capture numérotés. C'est une fonctionnalité à laquelle on a principalement recours avec `sed`. Le texte qui correspond à un motif entre parenthèses est « capturé » — ou mémorisé — et devient disponible pour être réutilisé dans le texte de substitution sous un numéro : \1 pour le premier groupe, \2 pour le second groupe, etc.

Clarifions le tout par un exemple. Tous les vers du fichier `chanson.txt` qui débutent par `Le` comptent exactement quatre mots après ce déterminant.

```
$ grep '^Le' chanson.txt
Le temps qui s'appelle hier
Le chemin du plus lâche
Le vent qui est bon
```

Amusons-nous à créer des nouveaux vers en changeant l'ordre des mots. Pour ce faire, nous allons créer trois groupes : les deux mots qui suivent immédiatement `Le` ; l'avant-dernier mot du vers ; le dernier mot du vers. Le motif pour capturer ces trois groupes est « `Le(.) (.) (.)` ». Rappelez-vous : les opérateurs de répétition sont gloutons, donc le premier groupe capture deux mots pour ensuite laisser un mot à chacun des deux autres groupes. Confirmons d'abord le contenu de chacun des groupes.

```
$ grep '^Le' chanson.txt | \
sed -E 's/Le (.) (.) (.) /Premier groupe: \1/'
Premier groupe: temps qui
Premier groupe: chemin du
Premier groupe: vent qui
```

```
$ grep '^Le' chanson.txt | \
sed -E 's/Le (.) (.) (.) /Deuxième groupe: \2/'
Deuxième groupe: s'appelle
Deuxième groupe: plus
Deuxième groupe: est
```

```
$ grep '^Le' chanson.txt | \
sed -E 's/Le (.) (.) (.) /Troisième groupe: \3/'
Troisième groupe: hier
```

11. Comme, par exemple, les [dictionnaires français](#)  de la suite bureautique Libre Office.

```
Troisième groupe: lâche
Troisième groupe: bon
```

Nous sommes maintenant prêts à changer l'ordre des mots en réutilisant tous les groupes dans la chaîne de substitution.

```
$ grep '^Le' chanson.txt | \
sed -E 's/Le (.*) (.*) (.*)/Le \2 \3 \1/'
Le s'appelle hier temps qui
Le plus lâche chemin du
Le est bon vent qui
```

Terminons cette très longue section sur les expressions régulières en revenant, comme promis, sur les quatre commandes de substitution de mots de la fin de la [section 11.3.7](#).

- La première commande effectue le traitement désiré, de surcroît avec la syntaxe la plus simple. Le motif trouve des correspondances avec `le mouvement` ou avec `la fatigue` et le texte est remplacé par `le pendule`.
- Il faut lire le motif de la seconde commande ainsi : « `l` suivi de `a` ou de `e`, suivi d'une espace, suivi de `mouvement` ou de `fatigue` ». Cela correspond bien au texte que nous voulons remplacer par `le pendule`. Cependant, à la différence de la première variante, le motif correspond aussi à `la mouvement` et à `le fatigue`. C'est sans conséquence ici.
- La troisième commande n'effectue pas le traitement attendu. En l'absence de parenthèses pour limiter la portée de l'opérateur « `|` », le motif correspond à `le` ou à `la mouvement` ou à `fatigue`. Cette commande remplacerait toutes les occurrences de `le` dans le texte de la chanson — y compris dans `Frôle` — par la chaîne `le pendule`. Ce n'est certainement pas ce que nous voulons !
- Par l'utilisation de parenthèses dans le motif, la quatrième commande vient corriger la troisième. Elle est fonctionnellement équivalente à la seconde commande.




Étudiez les lignes [336-492](#) du fichier de script `texte.sh` reproduit à la [section 11.6](#).

11.4 Traitement de texte divisé en champs avec awk

AWK ([Aho et collab., 1988](#)) est un langage de programmation spécialisé dans le traitement de texte et l'extraction de données développé à partir de 1977 chez Bell Labs. Le nom du langage provient des initiales de ses trois auteurs : Alfred Aho, Peter Weinberger et Brian Kernighan. Ce dernier est l'un des auteurs du grand classique de l'informatique *The C Programming Language* ([Kernighan et Ritchie](#),

1978) que nous avons déjà mentionné au [chapitre 1](#). La syntaxe de AWK s'apparente donc beaucoup à celle du C.

L'interpréteur du langage AWK auquel nous allons nous intéresser est l'utilitaire `awk`. Comme `grep` et `sed`, `awk` est un utilitaire standard des systèmes Unix. Véritable couteau suisse, c'est un outil basé sur les expressions régulières qui est souvent plus simple à utiliser que `sed`, surtout s'il faut effectuer des opérations complexes ou sur plusieurs lignes du texte à traiter.

Je vise avec cette section à piquer votre curiosité pour `awk` et à vous montrer à effectuer certaines opérations simples, mais néanmoins fort utiles. Un large pan des détails d'utilisation du langage est toutefois laissé de côté. Si vous souhaitez en découvrir davantage — chose que je vous encourage à faire! — je vous recommande l'excellent [guide d'utilisation de GNU Awk](#)  (Robbins, 2023).

Pour vous convaincre d'entrée de jeu de la puissance et de l'utilité de `awk`, imaginez le contexte suivant. On vous fournit une base de données en format CSV qui compte un million d'enregistrements (lignes) et quelques dizaines de champs (colonnes). Vous devez produire une nouvelle base de données composée des deux premières colonnes et d'une nouvelle colonne contenant la somme de celles-ci. Votre premier réflexe consistera sans doute à utiliser un tableur pour effectuer les opérations suivantes :

1. Importer le fichier CSV;
2. Supprimer les colonnes autres que les deux premières;
3. Ajouter dans une troisième colonne une formule pour calculer la somme des deux premières;
4. Copier la formule dans toutes les lignes;
5. Exporter la base de données en format CSV.

Cette procédure est fastidieuse, lente — surtout avec une grande base de données — et désespérément manuelle. Sous peu, vous saurez remplacer de telles manipulations par un simple appel à `awk` qui effectuera le traitement requis *en une fraction de seconde* et sans même avoir à lancer un lourd tableur :

```
$ awk -F ',' '{ print $1 "," $2 "," $1 + $2 }' data.csv \  
> data-new.csv
```

(De gauche à droite, la commande ci-dessus indique à `awk` d'utiliser la virgule comme séparateur de champs; d'afficher à la sortie standard, séparés par des virgules, le contenu de la première colonne, le contenu de la deuxième colonne, puis la somme de ces deux colonnes; de lire les données dans un fichier `data.csv`. Ensuite, les résultats sont redirigés de la sortie standard vers un fichier `data-new.csv`.)

11.4.1 Concepts de base

AWK est conçu autour du concept de séparation du texte à traiter en *enregistrements* (*records*), puis en *champs* (*fields*). Le séparateur d'enregistrements par défaut est le retour à la ligne. Dans la suite, nous ne ferons plus de différence entre une ligne du texte et un enregistrement.

Lorsque awk a terminé de lire une ligne du texte fourni en entrée, il sépare automatiquement celle-ci en champs. Le séparateur de champs par défaut est l'espace, mais, comme nous le verrons à la [section 11.4.5](#) il est facile de le changer pour tout autre caractère.

Un programme (ou script) awk est constitué d'une série de *règles*. Une fois que awk a séparé un enregistrement en champs, il exécute les règles une par une. Les règles ont accès à des variables qui contiennent l'enregistrement complet et chacun des champs, ce qui facilite grandement les opérations sur une ligne et sur ses composantes. La variable \$0 contient l'enregistrement complet, alors que les champs se trouvent dans des variables nommées \$1, \$2, \$3,

Illustrons ces concepts de base à l'aide de quelques exemples.

- ▶ En traitant le fichier `carburant.txt` présenté au haut de la [figure 9.2](#), awk rend disponible dans la variable \$0 le contenu complet d'une ligne et dans les variables \$1, \$2, \$3, \$4 et \$5 les données de chacune des colonnes. À la première ligne, ces cinq variables contiennent les noms des colonnes.
- ▶ Pour effectuer un traitement équivalent avec le fichier `carburant.csv` présenté au bas de la [figure 9.2](#), il faudrait spécifier à awk d'utiliser la virgule « , » comme séparateur de champs.
- ▶ Lorsque le texte du fichier `chanson.txt` de la [figure 11.1](#) est fourni à awk, chaque vers est séparé en mots. Le nombre de champs est donc variable d'une ligne à l'autre. Une règle ayant recours au motif `^M` permettrait d'effectuer un traitement uniquement sur les lignes qui débutent par « M ».

11.4.2 Commandes simples

L'interpréteur awk prend en argument un programme encadré par des guillemets simples¹². Comme `grep` et `sed`, awk lit ensuite l'entrée standard ou une liste de fichiers ligne par ligne. La syntaxe d'une commande est donc :

```
awk <options> ' <programme> ' <fichiers>
```

- ▶ `<options>` est une liste facultative d'options (ou drapeaux) qui modifient le comportement de awk.
- ▶ `<programme>` est un programme awk formé de règles.

12. Les guillemets doubles sont aussi permis, mais ils risquent fort d'entrer en conflit avec des guillemets dans le script. Je recommande de toujours utiliser des guillemets simples pour encadrer un programme awk.

- *⟨fichiers⟩* est le ou les fichiers à traiter ; si aucun fichier n'est spécifié, awk traite l'entrée standard.

Un programme awk décrit le traitement à effectuer sur le texte en entrée. Tel que mentionné précédemment, il est constitué d'une série de règles, souvent séparées par des retours à la ligne. Chaque règle spécifie un motif à rechercher dans l'enregistrement courant et une action à exécuter lorsque le texte correspond au motif.

Plus précisément, une règle est formée d'un *motif* suivi d'une *action*. L'action est placée entre accolades « { } » pour la séparer du motif. La forme générale d'un programme awk est donc la suivante :

```
⟨motif⟩ { ⟨action⟩ }  
⟨motif⟩ { ⟨action⟩ }  
...
```

Étudions quelques exemples pour mieux comprendre ces principes de la programmation awk. Nous reviendrons plus en détail sur la construction de motifs et d'actions dans les sections suivantes.

Le premier exemple effectue avec awk une opération que nous avons déjà réalisée avec grep à la [section 11.2.1](#) : afficher les lignes du fichier `chanson.txt` qui débutent par la lettre « M ». Le motif à rechercher est l'expression régulière `^M`. La syntaxe de awk exige que les expressions régulières soient placées entre barres obliques « / ». Nous avons déjà vu à la [section 11.4.1](#) qu'à l'intérieur d'une règle, la variable `$0` contient l'intégralité de l'enregistrement courant.

```
$ awk '/^M/ { print $0 }' chanson.txt  
Mes bras désolés  
Même si le mouvement  
Meurt mieux qu'il ne se charge  
Même si la fatigue
```

Vous aurez compris que la commande `print` sert à afficher du texte en sortie. L'instruction `print` sans argument est équivalente à `print $0`. Nous pourrions donc simplifier légèrement la commande ci-dessus en remplaçant « `print $0` » par « `print` ».

Cela dit, il est permis d'omettre le motif ou l'action dans une règle awk — mais pas les deux. Si le motif est absent, l'action est exécutée pour chaque enregistrement. Si c'est l'action qui est absente, l'instruction `print` est exécutée implicitement. Nous pouvons donc aller encore plus loin dans la simplification de la commande précédente en omettant l'action.

```
$ awk '/^M/' chanson.txt  
Mes bras désolés  
Même si le mouvement
```

```
Meurt mieux qu'il ne se charge  
Même si la fatigue
```

Voici maintenant une variante qui est compliquée à réaliser avec `grep`, mais très simple avec `awk` : afficher uniquement le deuxième mot des lignes qui débutent par « M ».

```
$ awk '/^M/ { print $2 }' chanson.txt  
bras  
si  
mieux  
si
```

Parce que la règle ne comporte pas de motif, la commande ci-dessous affiche plutôt le deuxième mot de chaque ligne du fichier `chanson.txt`. (Vérifiez par vous-même.)

```
$ awk '{ print $2 }' chanson.txt  
comme  
temps  
ce  
ma  
promets,  
journée  
...
```

Pour limiter le traitement ci-dessus aux cinq premières lignes du fichier, nous pouvons utiliser comme motif une expression relationnelle et la variable `NR` qui contient le numéro de l'enregistrement courant. Celle-ci est une des variables fournies automatiquement par `awk` durant l'exécution des règles.

```
$ awk 'NR <= 5 { print $2 }' chanson.txt  
comme  
temps  
ce  
ma  
promets,
```

L'instruction `print` accepte également en argument une liste d'éléments à afficher en sortie, séparés par des virgules. Ils sont séparés par une espace dans les résultats de la commande.

```
$ awk 'NR <= 5 { print NR, ":", $1, $2 }' chanson.txt  
1 : Oh comme  
2 : Le temps  
3 : Prends-le ce  
4 : Dans ma
```

```
5 : Je promets,
```

Une autre variable fournie automatiquement par `awk` est `NF`, le nombre de champs dans l'enregistrement courant. Nous pouvons utiliser cette variable pour extraire les vers de la chanson qui comptent moins de quatre mots.

```
$ awk 'NF < 4' chanson.txt
Prends-le ce diamant

Mes bras désolés

Nous avons l'outil

Frôle-moi davantage
Pour shimmer l'Univers
```

Vous aurez remarqué que la commande précédente a aussi extrait les lignes blanches du fichier. Pour ne pas en tenir compte, il suffit d'exiger aussi qu'un enregistrement compte au moins un champ.

```
$ awk 'NF > 0 && NF < 4' chanson.txt
Prends-le ce diamant
Mes bras désolés
Nous avons l'outil
Frôle-moi davantage
Pour shimmer l'Univers
```

Le prochain exemple est plus élaboré. La commande `awk` permet de calculer la longueur — en caractères — du plus court vers de `chanson.txt`. Cela nécessite trois règles :

- ▶ la première initialise une variable `min` à la longueur du premier enregistrement du fichier;
- ▶ la seconde met à jour la valeur de `min` si l'enregistrement courant n'est pas une ligne blanche et si sa longueur est inférieure à la valeur courante de `min`;
- ▶ la troisième utilise le motif spécial `END` qui permet d'exécuter des instructions après la lecture de tous les enregistrements, ici l'affichage de la valeur de `min`.

En plus d'étudier la commande, analysez attentivement l'algorithme.

```
$ awk 'NR == 1 { min = length($0) }
      NF > 0 { if (length($0) < min) min = length($0) }
      END { print min }' chanson.txt
```

```
18
```

Terminons avec deux exemples basés sur des données divisées naturellement en enregistrements et en champs. Comme les résultats de ces deux exemples occupent plusieurs lignes, la validation des commandes vous est laissée en exercice.

Le fichier `carburant.dat` livré avec le présent ouvrage contient des données de consommation de carburant pour divers types de voitures¹³. Examinez le contenu du fichier dans votre éditeur de texte avant d'aller plus loin.

En premier lieu, nous pouvons facilement créer une nouvelle base de données ne contenant que les modèles de voitures 4 cylindres sans passer par un tableur ou par R. La commande suivante conserve uniquement les commentaires et les noms de colonnes en entête du fichier, ainsi que les enregistrements dont le second champ est égal à 4.

```
$ awk '/^[#a-z]/ || $2 == 4' carburant.dat
```

En second lieu, l'ajout d'une action à la commande précédente permet de conserver uniquement les deux premiers champs des données. La sélection doit évidemment s'effectuer uniquement sur les titres de colonnes et sur les données. Examinez d'abord la commande, les explications détaillées suivent.

```
$ awk '/^[#a-z]/ || $2 == 4 {
    print ($0 ~ /^#/ ) ? $0 : $1 " " $2
}' carburant.dat
```

L'action fait appel à deux opérateurs de awk que nous n'avons pas rencontrés auparavant, ainsi qu'à la concaténation.

~ retourne vrai lorsque la chaîne de caractères du côté gauche de l'opérateur (`$0` par défaut) correspond à l'expression régulière du côté droit¹⁴. La version négative « !~ » existe aussi.

```
<chaîne> ~ /<motif>/
<chaîne> !~ /<motif>/
```

? : retourne le résultat de l'expression en second ou en troisième argument selon que la condition en premier argument est, respectivement, vraie ou fausse¹⁵.

```
<condition> ? <expression_si_vrai> : <expression_si_faux>
```

Pour ce qui est de la concaténation de chaînes de caractères, aussi étrange que cela puisse paraître, il n'existe pas d'opérateur désigné dans awk. Il suffit de placer les chaînes de caractères les unes à la suite des autres pour en créer une nouvelle. L'action dans la commande ci-dessus affiche donc le contenu complet de l'enre-

13. Comme la présence marquée des antédiluviens moteurs 8 cylindres l'indique, il s'agit d'un très vieux jeu de données!

14. Non, l'opérateur `==` n'est pas valide entre une chaîne de caractères et une expression régulière.

15. Souvent appelé *opérateur ternaire*, cet opérateur existe dans plusieurs langages de programmation. Dans R, ce qui s'en approche le plus est la fonction `ifelse`.

gistement (\$0) lorsque celui-ci débute par le symbole « # », et les deux premiers champs séparés par une espace (la chaîne « " " ») autrement.

À ce stade, vous disposez déjà de connaissances suffisantes pour réaliser plusieurs tâches de manipulation de texte avec `awk`. Les sections suivantes ne font qu'ajouter des détails sur l'écriture des règles et la configuration de l'outil. Si vous souhaitez étudier des exemples additionnels, consultez [Robbins \(2023, sections 1.3 à 1.5\)](#).

11.4.3 Motifs

Comme nous l'avons déjà vu, les motifs dans `awk` contrôlent l'exécution des règles : l'action d'une règle est exécutée lorsque le motif correspond à l'enregistrement courant. Il existe six types de motifs dans `awk`, dont quatre que nous avons déjà rencontrés dans les exemples de la section précédente.

<code>/⟨expression régulière⟩/</code>	correspond lorsque l'expression régulière correspond à l'enregistrement courant.
<code>⟨expression⟩</code>	correspond lorsque la valeur de l'expression <code>awk</code> est non nulle (s'il s'agit d'une valeur numérique) ou non vide (s'il s'agit d'une chaîne de caractères).
<code>⟨motif_début⟩, ⟨motif_fin⟩</code>	correspond lorsque l'enregistrement courant se trouve dans l'étendue définie par <code>⟨motif_début⟩</code> et <code>⟨motif_fin⟩</code> , inclusivement (attention : la virgule « , » entre les deux motifs fait partie intégrante de l'instruction).
<code>BEGIN</code> <code>END</code>	motifs spéciaux qui permettent d'exécuter des actions avant la lecture du premier enregistrement (<code>BEGIN</code>) et après la lecture du dernier (<code>END</code>).
<code>BEGINFILE</code> <code>ENDFILE</code>	motifs spéciaux qui permettent d'exécuter des actions au début et à la fin de la lecture d'un fichier, lorsqu'il y a plusieurs fichiers à traiter.
<code>⟨vide⟩</code>	un motif vide correspond à tous les enregistrements.

Nous n'étudierons pas davantage les étendues d'enregistrements ni les motifs spéciaux `BEGINFILE` et `ENDFILE`.

Toute expression `awk` est valide en tant que motif. La définition d'expression `awk` est en tous points équivalente à celle d'expression R donnée à la [section 4.2.1](#). L'expression est réévaluée à chaque fois que la règle est testée pour un nouvel enregistrement. C'est d'ailleurs cette réévaluation des motifs à chaque enregistrement qui rend simple avec `awk` d'effectuer un traitement sur plusieurs lignes du texte en entrée. Nous y reviendrons dans l'exemple ci-dessous.

Si l'expression du motif fait appel à la valeur d'un champ — \$1 par exemple —, c'est celle de l'enregistrement courant qui prévaut. Autrement, la valeur de l'expression dépend de l'état du programme au moment de son évaluation.

Les expressions logiques sont évidemment les plus souvent utilisées en tant que motifs. Les opérateurs arithmétiques et logiques de awk sont, pour l'essentiel, les mêmes que les opérateurs de R figurant dans le [tableau 4.1](#). Principale différence : « = » est l'opérateur d'affectation d'une valeur à une variable dans awk. Il faut aussi ajouter à cette liste les opérateurs « ~ » et « !~ » mentionnés à la [section 11.4.2](#). Vous trouverez une liste exhaustive des opérateurs de awk dans [Robbins \(2023, section 6.2\)](#).



Puisque la valeur d'une expression awk dépend de l'état du programme, l'ordre dans lequel les règles apparaissent dans le programme est important.

Illustrons certains des concepts ci-dessus à l'aide d'un exemple, notamment l'importance de l'ordre des règles dans un programme. La commande suivante permet d'extraire les deux premiers couplets de *La journée qui s'en vient est flambant neuve*. Ceux-ci sont délimités par des lignes blanches dans le fichier. La tâche consiste donc à extraire le texte jusqu'à la seconde ligne blanche, exclusivement. C'est donc un traitement sur plusieurs lignes.

```
$ awk '$0 ~ /\$/ && state == 1 { exit }  
      $0 ~ /\$/ { state = 1 }  
      { print }' chanson.txt  
Oh comme il est lourd  
Le temps qui s'appelle hier  
Prends-le ce diamant  
Dans ma tête il est pour toi  
Je promets, je promets que  
la journée qui s'en vient est flambant neuve  
  
Je le connais bien  
Le chemin du plus lâche  
Mes bras désolés  
Rampent comme des chiens  
Je promets, je promets que
```

La première règle provoque la sortie du programme lorsque awk rencontre une ligne blanche *et* qu'une variable d'« état » — d'où le nom, *state*, que j'ai choisi de lui donner — est égale à 1. Ici, vous devez savoir que tant qu'une valeur n'a pas été explicitement assignée à une variable, la valeur de celle-ci est 0. Le programme ne peut donc quitter avant qu'une valeur de 1 n'ait été explicitement affectée à la variable d'état.

La seconde règle du programme affecte justement la valeur 1 à la variable d'état lorsque awk rencontre une ligne blanche. Dans le déroulement du programme, ce sera à la première ligne blanche.

La troisième règle extrait simplement les enregistrements au fur et à mesure qu'ils sont lus.

En combinant les règles dans cet ordre précis, le programme se termine à la *seconde* ligne blanche du fichier, donc après avoir extrait les deux premiers couplets. Si l'ordre des deux premières règles était inversé, la commande n'extraitait que le premier couplet. (Déterminez pourquoi.)

Question de parfaire notre style de programmation awk, apportons deux simplifications à la commande ci-dessus :

- ▶ l'expression `/^$/` est équivalente à `$0 ~ /^$/` puisque `$0` est la valeur par défaut de l'argument gauche de l'opérateur « `~` » ;
- ▶ la troisième règle peut être remplacée par un motif identité qui est toujours vrai (« 1 ») et aucune action, l'instruction `print` étant implicite. C'est la règle « affiche le contenu de l'enregistrement courant » que l'on retrouve souvent à la fin des programmes awk.

```
$ awk '/^$/ && state == 1 { exit }  
      /^$/ { state = 1 }  
      1' chanson.txt  
Oh comme il est lourd  
Le temps qui s'appelle hier  
Prends-le ce diamant  
Dans ma tête il est pour toi  
Je promets, je promets que  
la journée qui s'en vient est flambant neuve  
  
Je le connais bien  
Le chemin du plus lâche  
Mes bras désolés  
Rampent comme des chiens  
Je promets, je promets que
```

11.4.4 Actions

L'action d'une règle dans un programme awk indique à l'interpréteur quoi faire lorsqu'un enregistrement correspond à un motif. Une action est constituée d'une ou plusieurs instructions awk placées entre accolades « `{ }` » et séparées les unes des autres par un retour à la ligne ou par un point-virgule « `;` ». Chaque instruction d'une action indique une chose à faire.

Une action absente est équivalente à « `{ print $0 }` ». Cependant, une action vide « `{ }` » ne fait rien lorsque l'enregistrement correspond au motif. Autrement

dit, la règle « /foo/ { } » ne fait rien lorsque l'enregistrement correspond à «foo», alors que la règle « /foo/ » affiche le contenu de l'enregistrement.

Les principaux types d'instructions que l'on retrouve dans les actions sont : les expressions (au premier chef l'affectation de variables); les structures de contrôle (if, for, while et do); les instructions d'affichage (comme print et printf). Les expressions peuvent faire appel à plusieurs fonctions awk internes, notamment pour la manipulation de chaînes de caractères. Consultez au besoin [Robbins \(2023, chapitres 6, 7 et 9\)](#).

Les données du fichier `carburant.dat` sont entièrement en unités de mesure impériales. La commande awk suivante permet de convertir le poids des voitures, actuellement en milliers de livres, vers des kilogrammes. L'action de la première règle remplace le poids du véhicule — le cinquième champ — par le poids en kilogrammes (2,20462 étant le facteur de conversion entre les deux unités). Remarquez que cette action n'affiche rien, elle ne fait que modifier un champ de l'enregistrement courant. C'est la seconde règle « affiche le contenu de l'enregistrement courant » qui affiche les résultats. Sans celle-ci, le programme ne retournerait aucun résultat.

```
$ awk '/^[^#a-z]/ { $5 = $5*1000/2.20462 } 1' carburant.dat
### Données de consommation de carburant pour divers types
### de voitures.
###
### mpg: consommation de carburant en milles par gallon
### nb cyl: nombre de cylindres
### cylindree: cylindrée du moteur, en pouces cubes;
### cv: puissance en chevaux vapeurs;
### poids: poids de la voiture en milliers de livres.
###
### Source: Abraham & Ledolter (1983)
mpg  nb cyl  cylindree  cv  poids
16.9  8  350  155  1977.67
15.5  8  351  142  1838.87
19.2  8  267  125  1635.2
18.5  8  360  150  1787.16
...
```

11.4.5 Changer le séparateur d'enregistrements et de champs

Les séparateurs d'enregistrements et de champs par défaut sont, respectivement, le retour à la ligne et l'espace. Ces symboles sont stockés dans les variables prédéfinies `RS` (*record separator*) et `FS` (*field separator*). Pour changer l'un ou l'autre de ces séparateurs, il suffit d'affecter un nouveau symbole ou une expression régulière dans la variable correspondante, habituellement dans une règle spé-

ciale BEGIN.

Pour des raisons historiques, il est également possible de changer le séparateur de champs — et uniquement celui-ci — à la ligne de commande avec l'option `-F`. C'est l'approche que j'ai retenue dans le tout premier exemple de la section, à la [page 268](#).

Dans le premier exemple, ci-dessous, la règle modifie le séparateur de champs pour la virgule. Dans le second, le recours à un motif d'expression régulière permet de définir comme séparateur de champs l'espace *ou* le tiret.

```
BEGIN { FS = "," }
```

```
BEGIN { FS = "[ -]" }
```

Chose intéressante, il est également possible de modifier les séparateurs d'enregistrements et de champs que `awk` utilise lors de la sortie des résultats. Ceux-ci sont stockés dans les variables prédéfinies `ORS` (*output record separator*) et `OFS` (*output field separator*). Les séparateurs de sortie par défaut sont aussi le retour à la ligne et l'espace.

Par exemple, pour séparer les champs par un point-virgule dans les résultats, il suffit d'inclure la règle ci-dessous dans son programme `awk`.

```
BEGIN { OFS = ";" }
```



Les opérations de composition du présent ouvrage, de publication d'une nouvelle version et de mise à jour de la page web sont entièrement automatisées avec une combinaison des utilitaires de ce chapitre et de `make` [🔗](#), un autre outil standard des systèmes Unix. Je vous invite à jeter un coup d'œil au fichier de configuration `Makefile` qui se trouve dans le [code source du projet](#) [🔗](#).



Le fichier de script `texte.sh` reproduit à la [section 11.6](#) fournit des exemples additionnels d'utilisation de `awk`. Étudiez attentivement les lignes [495-601](#).

11.5 Fonctions internes utiles

Cette section présente quelques unes seulement des fonctions internes de `R` pour rechercher et remplacer du texte à l'aide d'expressions régulières. Consultez la rubrique d'aide (commune, sauf pour `regmatches`) de n'importe laquelle de celles-ci pour connaître l'ensemble des fonctions disponibles et les détails concernant leur fonctionnement. Vous trouverez également dans la rubrique d'aide de nombreux exemples d'utilisation additionnels.



Rappelons qu'en R un vecteur de mode `character` peut contenir plusieurs chaînes de caractères. Les fonctions ci-dessous effectuent leur traitement sur chacune des chaînes d'un vecteur.

Le vecteur `state.name` fourni avec R contient les noms des cinquante États américains. Les exemples de cette section utilisent seulement les dix premiers.

```
> (state.name.10 <- head(state.name, 10))
[1] "Alabama"      "Alaska"      "Arizona"      "Arkansas"
[5] "California"    "Colorado"    "Connecticut"  "Delaware"
[9] "Florida"      "Georgia"
```

grep recherche d'une expression régulière dans un vecteur de chaînes de caractères.

```
> grep("ia$", state.name.10)
[1] 5 10
> grep("ia$", state.name.10, value = TRUE)
[1] "California" "Georgia"
```

regexpr comme `grep`, mais avec des informations additionnelles sur la position et la longueur du premier appariement dans les résultats. Pour identifier tous les appariements plutôt que seulement le premier, utiliser la fonction `gregexpr`.

```
> regexpr("ia$", state.name.10)
[1] -1 -1 -1 -1 9 -1 -1 -1 -1 6
attr("match.length")
[1] -1 -1 -1 -1 2 -1 -1 -1 -1 2
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

regmatches extrait le texte des appariements à partir des informations fournies par `regexpr` ou `gregexpr`.

```
> m <- regexpr("ia$", state.name.10)
> regmatches(state.name.10, m)
[1] "ia" "ia"
```

sub recherche une expression régulière dans un vecteur de chaînes de caractères et remplace la première occurrence dans chaque chaîne par une autre chaîne.

```
> sub("a", "*", state.name.10)
[1] "Al*bama"      "Al*ska"      "Arizon*"
[4] "Ark*nsas"     "C*ifornia"   "Color*do"
[7] "Connecticut" "Del*ware"    "Florid*"
[10] "Georgi*"
```

`gsub` comme `sub`, mais pour toutes les occurrences dans chaque chaîne.

```
> gsub("a", "*", state.name.10)
[1] "Al*b*m*"      "Al*sk*"      "Arizon*"
[4] "Ark*ns*s"     "C*iforni*"   "Color*do"
[7] "Connecticut" "Del*w*re"    "Florid*"
[10] "Georgi*"
```



Plusieurs langages de programmation contiennent des utilitaires `grep`, `sub` et `gsub`. Par exemple, dans `awk`, des fonctions `sub` et `gsub` permettent d'effectuer des remplacements à l'intérieur des champs.

Les fonctions ci-dessus prennent toutes le motif de l'expression régulière en argument sous forme d'une chaîne de caractères. Lorsque le motif contient un caractère d'échappement « `\` », il faut doubler celui-ci dans la chaîne de caractère de R. Par exemple, pour fournir en argument à une fonction R le motif `^.{1,8}\.txt` rencontré à la [section 11.3.4](#), il faut l'entrer sous la forme `^.{1,8}\\\.txt`.

Depuis la version 4.0.0, R offre une alternative pour entrer plus aisément des chaînes de caractères contenant, entre autres, des caractères d'échappement ou des guillemets. Il s'agit de chaînes de caractères brutes (*raw strings* ou *raw character constants*). La syntaxe pour créer une chaîne brute est la suivante :

```
r"(<...>)"
```

Ici, `<...>` représente n'importe quelle suite de caractères qui ne se termine pas par « `)` ».

Ainsi, pour passer le motif ci-dessus à une fonction R sans avoir à se soucier de doubler les caractères d'échappement, il est possible de l'entrer plutôt sous la forme `r"(.{1,8}\.txt)"`.



Le fichier de script `texte.R` reproduit à la [section 11.7](#) contient des exemples d'utilisation de fonctions R utilisant les expressions régulières, de même que des exemples additionnels de chaînes de caractères brutes. Étudiez l'entièreté du fichier.

11.6 Exemples pour la ligne de commande Unix

Exceptionnellement, ce chapitre propose des exemples d'utilisation des outils `grep`, `sed` et `awk` que vous devez exécuter depuis une ligne de commande Unix.

📍 Fichier d'accompagnement `texte.sh`

```

11  ## Les exemples de ce fichier utilisent les fichiers suivants
12  ## distribués avec le présent ouvrage:
13  ##
14  ## - fichiers de script *.R
15  ## - chanson.txt
16  ## - 100metres.dat
17  ## - carburant.dat
18  ## - NEWS
19  ##
20  ## Assurez-vous d'exécuter les exemples dans un répertoire contenant
21  ## ces fichiers.
22  ##
23  ## Vous pouvez exécuter les commandes ci-dessous telles quelles à une
24  ## ligne de commande Unix, ou encore dans le terminal RStudio avec
25  ## Bash (macOS) ou Git Bash (Windows) comme interpréteur de commandes.
26
27  ###
28  ### OUTILS D'ANALYSE ET DE CONTRÔLE DE TEXTE
29  ###
30
31  ## EXTRACTION DE LIGNES AVEC grep
32
33  ## L'utilitaire 'grep' permet d'identifier les lignes d'un fichier (ou
34  ## du texte en entrée standard) qui contiennent des correspondances
35  ## avec une expression régulière.
36  ##
37  ## L'expression régulière la plus simple est une chaîne de caractères
38  ## standard.
39  ##
40  ## Effectuons d'abord quelques recherches simples dans le texte de la
41  ## chanson «La journée qui s'en vient est flambant neuve».
42  grep 'temps' chanson.txt    # vers contenant le mot «temps»
43  grep 'chat' chanson.txt     # vers contenant le mot «chat»
44
45  ## Remarquez que 'grep' est sensible à la casse.
46  grep 'Même' chanson.txt     # vers contenant le mot «Même»
47  grep 'même' chanson.txt     # vers contenant le mot «même»
48
49  ## Dressons maintenant la liste de toutes les utilisations de la
50  ## fonction 'matrix' dans les fichiers d'exemples.
51  ##
52  ## Lorsque 'grep' reçoit plusieurs fichiers en entrée, il effectue la

```

```
53 ## recherche dans tous les fichiers tour à tour.
54 ##
55 ## La syntaxe '*.R' signifie: tous les fichiers dont le nom se termine
56 ## par l'extension «.R». (Attention: il s'agit d'une syntaxe de
57 ## l'interpréteur de commande Bash, et non d'une expression
58 ## régulière.)
59 grep 'matrix' *.R
60
61 ## L'option '-l' de 'grep' permet de limiter les résultats à la liste
62 ## des fichiers contenant au moins une utilisation de 'matrix'.
63 grep -l 'matrix' *.R
64
65 ## Recherchons maintenant les fichiers d'exemples dans lesquels
66 ## apparaissent les objets 'letters' et 'LETTERS'. Nous devons
67 ## rechercher sans égard à la casse pour attraper les deux cas.
68 ## L'option '-i' de 'grep' permet d'ignorer la casse.
69 grep -i 'letters' *.R
70
71 ## RECHERCHE ET REMPLACEMENT DE TEXTE AVEC sed
72
73 ## L'utilitaire 'sed' est un éditeur de texte en ligne très puissant,
74 ## mais dont le fonctionnement et la syntaxe des commandes laissent
75 ## souvent perplexe. Dans le cadre de cet ouvrage, nous nous
76 ## pencherons uniquement sur les opérations de recherche et de
77 ## remplacement de texte de 'sed'.
78 ##
79 ## La syntaxe de la commande de recherche et remplacement est la
80 ## suivante:
81 ##
82 ## s/<motif>/<remplacement>/[g]
83 ##
84 ## - <motif> est une expression régulière qui définit le texte à
85 ## rechercher;
86 ## - <remplacement> est le texte qui doit remplacer le texte identifié
87 ## par <motif>;
88 ## - «/» est en fait un symbole quelconque (mais «/» est celui le plus
89 ## souvent utilisé) qui sert à délimiter les parties de la commande;
90 ## - g est un modificateur optionnel qui indique à 'sed' d'effectuer
91 ## le remplacement pour toutes les correspondances sur une même
92 ## ligne ('sed' effectue le changement uniquement pour la première
93 ## correspondance par défaut).
94 ##
95 ## Reprenons d'abord les exemples du chapitre basés sur le fichier
96 ## chanson.txt.
97 sed 's/Oh/Ah/' chanson.txt # remplacer «Oh» par «Ah»
98 sed 's~Oh~Ah~' chanson.txt # idem avec un autre séparateur
99 sed 's|Oh|Ah|' chanson.txt # idem avec un autre séparateur
100 sed 's/mm/MM/' chanson.txt # remplacer «mm» par «MM»;
```

```
101
102 ## Observez maintenant l'effet du modificateur 'g' dans le
103 ## remplacement de «promets» par «jure». Sans lui, seul le premier mot
104 ## de la ligne est remplacé. Pour remplacer toutes les occurrences de
105 ## «promets» sur une ligne par «jure», il faut ajouter le modificateur
106 ## 'g' à la commande.
107 sed 's/promets/jure/' chanson.txt
108 sed 's/promets/jure/g' chanson.txt
109
110 ## Remplaçons maintenant les symboles de commentaires triples '###' en
111 ## début de ligne (à la Emacs) dans le fichier 'pratiques.R' par un
112 ## symbole unique '#' (à la RStudio).
113 ##
114 ## Comme nous le verrons dans la suite du chapitre, le symbole «^»
115 ## identifie le début d'une ligne.
116 sed 's/^###/#!/' pratiques.R
117
118 ## Le texte modifié par 'sed' est affiché à la sortie standard sans
119 ## modifier le fichier d'origine.
120 ##
121 ## Pour sauvegarder le texte modifié, il faut rediriger la sortie
122 ## standard vers un fichier avec l'opérateur de redirection «>». Le
123 ## nom de ce fichier doit être différent de celui que 'sed' est occupé
124 ## à traiter.
125 ##
126 ## Reprenons l'exemple précédent en sauvegardant le résultat dans un
127 ## nouveau fichier 'pratiques-new.R'.
128 sed 's/^###/#!/' pratiques.R > pratiques-new.R
129
130 ###
131 ### EXPRESSIONS RÉGULIÈRES
132 ###
133
134 ## SYNTAXE DE BASE
135
136 ## Les règles de base des expressions régulières sont simples et peu
137 ## nombreuses:
138 ##
139 ## 1. un caractère littéral correspond à lui-même (plus précisément: à
140 ## sa première occurrence dans le texte examiné);
141 ## 2. les composantes d'une expression régulière sont reliées par «et»
142 ## par défaut;
143 ## 3. des opérateurs permettent de définir des motifs contenant autre
144 ## chose que des caractères littéraux.
145 ##
146 ## Examinons quelques exemples simples de recherches dans le mot
147 ## «chatons». Exécutez aussi ces exemples dans un testeur
148 ## 'dexpressions régulières en ligne afin de mieux voir à quel texte
```

```

149 ## les motifs correspondent.
150 ##
151 ## Le texte est composé avec la commande 'echo' et passé ensuite à
152 ## 'grep' avec l'opérateur de transfert de données (tuyau).
153 echo "chatons" | grep 'c'
154 echo "chatons" | grep 't'
155 echo "chatons" | grep 'b'
156 echo "chatons" | grep 'chat'
157 echo "chatons" | grep 'chaton'
158 echo "chatons" | grep 'ton'
159 echo "chatons" | grep 'chats'
160
161 ## Il existe des différences importantes dans les définitions des
162 ## opérateurs entre les expressions régulières basiques (BRE) et les
163 ## expressions régulières étendues (ERE).
164 ##
165 ## Dès lors que votre motif contient un opérateur, je vous recommande
166 ## d'utiliser les expressions régulières étendues. On les active dans
167 ## 'grep' et dans 'sed' avec l'option '-E'.
168 ##
169 ## Par exemple, le symbole «|» est l'opérateur d'alternance («ou»)
170 ## dans les ERE et un caractère littéral dans les BRE.
171 grep 'chat|chien' chanson.txt # recherche «chat|chien»
172 grep -E 'chat|chien' chanson.txt # recherche «chat» ou «chien»
173
174 ## Illustrons quelques-uns des principaux opérateurs des expressions
175 ## régulières étendues. Nous reviendrons sur ceux-ci plus en détail
176 ## plus loin.
177 ##
178 ## Débutons avec un autre exemple d'alternance («ou»).
179 grep -E 'amour' chanson.txt # «amour» sur une ligne
180 grep -E 'mieux' chanson.txt # «mieux» sur une ligne
181 grep -E 'amour|mieux' chanson.txt # l'un OU l'autre
182
183 ## Début et fin de ligne.
184 grep -E 'la' chanson.txt # «la» sur une ligne
185 grep -E '^la' chanson.txt # «la» en début de ligne
186 grep -E 'que' chanson.txt # «que» sur une ligne
187 grep -E 'que$' chanson.txt # «que» en fin de ligne
188
189 ## Le point «.» a un rôle très important dans les expressions
190 ## régulières: il représente «n'importe quel caractère».
191 ##
192 ## Puisque l'espace constitue également un caractère littéral, le
193 ## motif ' .' permet de rechercher un mot d'une lettre (non placé en
194 ## début ou en fin de ligne puisqu'il doit être précédé et suivi d'une
195 ## espace).
196 grep -E ' .' chanson.txt

```



```
197
198 ## Voici deux manières de rechercher un mot d'exactly deux lettres
199 ## (toujours placé ailleurs qu'en début ou en fin de ligne):
200 grep -E ' .. ' chanson.txt      # doublement de l'opérateur
201 grep -E ' .{2} ' chanson.txt    # opérateur + quantificateur
202 grep -E -o ' .{2} ' chanson.txt # afficher les correspondances
203
204 ## Une autre manière simple de visualiser les correspondances d'un
205 ## motif dans du texte consiste à remplacer celles-ci avec 'sed' par
206 ## du texte facile à repérer.
207 sed 's/a/!/' chanson.txt
208 sed 's/am/--/' chanson.txt
209
210 ## REMPLACEMENT ET RÉPÉTITION
211
212 ## Comme mentionné ci-dessus, le caractère «.» est le caractère de
213 ## remplacement dans les expressions régulières. Il signifie
214 ## «n'importe quel caractère», y compris l'espace.
215 ##
216 ## Le motif 'ou. ' permet d'identifier les mots se terminant par «ou»
217 ## et un caractère quelconque. L'espace dans le motif permet de
218 ## délimiter un mot. Les mots en fin de ligne sont donc exclus de
219 ## l'expression régulière.
220 grep -E 'ou. ' chanson.txt
221
222 ## L'option '-o' de 'grep' permet d'afficher la partie de la ligne qui
223 ## correspond au motif.
224 grep -E -o 'ou. ' chanson.txt
225
226 ## Les opérateurs de répétition permettent de décrire des suites de
227 ## caractères. Il existe quatre opérateurs de répétition dans les
228 ## expressions régulières étendues:
229 ##
230 ## * : ce qui précède 0 ou plusieurs fois;
231 ## ? : ce qui précède 0 ou 1 fois;
232 ## + : ce qui précède 1 ou plusieurs fois;
233 ## { } : quantificateurs bornés.
234 ##
235 ## Pour illustrer le fonctionnement des opérateurs de répétition, nous
236 ## allons créer un petit fichier 'ah.txt' contenant des onomatopées
237 ## "ah!" de diverses longueurs, à raison d'une par ligne.
238 s="h"
239 for i in {1..6}; do echo "a${s}"'!'; s+="h"; done > ah.txt
240
241 ## Vérification du contenu du fichier.
242 cat ah.txt
243
244 ## Comparez les résultats des trois commandes suivantes.
```

```

245 grep -E 'ahh?!' ah.txt      # un ou deux «h»
246 grep -E 'ahh*!' ah.txt     # un «h» et plus
247 grep -E 'ahh+!' ah.txt     # deux «h» et plus
248
249 ## Les quantificateurs bornés s'utilisent de trois façons:
250 ##
251 ## {n} : ce qui précède exactement 'n' fois
252 ## {n,m} : ce qui précède entre 'n' et 'm' fois
253 ## {n,} : ce qui précède au moins 'n' fois
254 ##
255 ## Nous devons avoir recours aux quantificateurs bornés lorsque les
256 ## cas les plus fréquents couverts par '*', '+' et '?' ne suffisent
257 ## plus ou seraient trop longs à écrire (et moins lisibles).
258 grep -E 'ah{3}!' ah.txt     # exactement trois «h»
259 grep -E 'ah{3,5}!' ah.txt   # entre trois et cinq «h»
260 grep -E 'ah{3,}!' ah.txt    # trois «h» et plus
261
262 ## Les opérateurs de répétition sont souvent utilisés avec le
263 ## caractère de remplacement pour décrire des chaînes de caractères
264 ## arbitraires.
265 ##
266 ## La commande suivante permet d'extraire du fichier 'chanson.txt' les
267 ## vers qui débutent par «L» (les majuscules se trouvant uniquement en
268 ## début de ligne) et qui contiennent un «i» quelque part sur la
269 ## ligne.
270 grep -E 'L.*i' chanson.txt
271
272 ## Remplacer le motif 'L.*i' dans la commande précédente par 'L.*e'
273 ## permet d'illustrer le caractère glouton des opérateurs de
274 ## répétition.
275 ##
276 ## Au premier abord, il semble que 'L.*e' devrait correspondre au mot
277 ## «Le» que l'on retrouve au début de plusieurs vers. Or, la chaîne
278 ## correspondante se prolonge plutôt jusqu'au dernier «e» de la ligne,
279 ## comme l'option '-o' de 'grep' permet de le constater.
280 grep -E 'L.*e' chanson.txt
281 grep -E -o 'L.*e' chanson.txt
282
283 ## La commande suivante permet d'extraire les lignes de commentaires
284 ## contenant du texte du fichier 'bases.R'. Celles-ci sont constituées
285 ## de deux ou trois symboles «#» successifs suivis d'une espace et
286 ## d'au moins un autre caractère
287 grep -E '^###? .+' bases.R
288
289 ## Nettoyage: supprimer le fichier 'ah.txt'
290 rm ah.txt
291
292 ## CARACTÈRE D'ÉCHAPPEMENT

```

```
293
294 ## Y a-t-il un point final quelque part dans le fichier 'chanson.txt'?
295 ## Demandons à 'grep' de nous trouver ça.
296 grep '.' chanson.txt
297
298 ## La commande ci-dessus n'a pas donné le résultat escompté, hein?
299 ## Puisque le symbole «.» est un opérateur signifiant «n'importe quel
300 ## caractère» dans les expressions régulières, le motif '.' correspond
301 ## à toute ligne contenant au moins un caractère. C'est pourquoi la
302 ## commande précédente retourne l'intégralité du fichier... sauf les
303 ## lignes blanches.
304 ##
305 ## Pour rechercher le caractère «.», il faut enlever au symbole sa
306 ## signification particulière en le précédant d'une barre oblique
307 ## inversée «\». (Il s'agit là d'une convention très répandue dans
308 ## plusieurs langages informatiques.)
309 grep '\.' chanson.txt      # aucun point final dans le fichier
310
311 ## Dans le même esprit, recherchons dans le fichier 'bases.R' toutes
312 ## les occurrences du caractère «?». Dans les expressions régulières
313 ## étendues, c'est un symbole spécial qu'il faut désactiver avec le
314 ## caractère d'échappement.
315 grep -E '\?' bases.R
316
317 ## MARQUEURS DE POSITION
318
319 ## Le symbole «^» identifie le début d'une ligne. Comparer les
320 ## résultats des deux commandes ci-dessous.
321 grep 'la' chanson.txt      # «la» n'importe où sur la ligne
322 grep '^la' chanson.txt    # «la» en début de ligne
323
324 ## Le symbole «$» identifie la fin d'une ligne. Comparer les résultats
325 ## des deux commandes ci-dessous.
326 grep 'que' chanson.txt    # «que» n'importe où sur la ligne
327 grep 'que$' chanson.txt   # «que» en fin de ligne
328
329 ## Le motif '^$' signifie: un début de ligne immédiatement suivi d'une
330 ## fin de ligne. Il s'agit du motif robuste pour rechercher une ligne
331 ## blanche (ou vide) dans du texte. (Le motif '.' mentionné ci-dessous
332 ## retiendrait une ligne comptant au moins une espace.)
333 grep -c '^$' chanson.txt  # nombre de lignes blanches
334 grep -v '^$' chanson.txt  # lignes blanches supprimées
335
336 ## CLASSES DE CARACTÈRES
337
338 ## Les crochets '[' ]' définissent une classe de caractères dans les
339 ## expressions régulières. Les classes de caractères sont
340 ## particulièrement utiles pour:
```

```

341 ##
342 ## - définir un choix entre plusieurs caractères, les composantes
343 ## d'une classe étant reliées par «ou»;
344 ## - définir facilement une longue liste de symboles (toutes les
345 ## lettres minuscules, par exemple);
346 ## - définir un motif par la négative («n'importe quoi sauf...»).
347 ##
348 ## (D'ailleurs, les crochets sont souvent utilisés dans la
349 ## documentation en informatique pour représenter un choix entre
350 ## plusieurs éléments ou quelque chose d'optionnel.)
351 ##
352 ## Premier exemple: rechercher un mot écrit avec une majuscule ou une
353 ## minuscule initiale.
354 grep '[Mm]ême' chanson.txt # fonctionne toujours
355 grep -i 'même' chanson.txt # idem, mais dépend de 'grep'
356
357 ## La commande suivante identifie les utilisations de «ou» dans
358 ## 'chanson.txt' qui ne sont PAS suivies de «r» ni de «s».
359 grep 'ou[^\rs]' chanson.txt
360
361 ## La commande suivante extrait toutes les mentions des fonction
362 ## 'lapply', 'sapply' et 'tapply' du fichier 'donnees.R'.
363 grep '[\slt]apply' donnees.R
364
365 ## Pour exclure les mentions dans les commentaires, ajoutons des
366 ## pièces au motif pour exiger que la ligne débute par tout caractère
367 ## AUTRE que '#'. Permettons également, avec l'opérateur '*' que ce
368 ## caractère apparaisse 0 ou plusieurs fois, ce qui permet
369 ## d'identifier les utilisations des fonctions d'application ailleurs
370 ## qu'au début d'une ligne.
371 grep '^[^#]*[\slt]apply' donnees.R
372
373 ## La classe de caractère renversée (qui débute par «^») est
374 ## particulièrement utile pour restreindre la portée des
375 ## quantificateurs gloutons.
376 ##
377 ## Par exemple, tel qu'expliqué dans le chapitre, le motif 'L.+c' ---
378 ## que l'on voudrait signifier «L suivi d'au moins un caractère, puis
379 ## un c» --- trouve un accord avec les caractères jusqu'au «c» dans
380 ## «lâche» lorsque appliqué à «Le chemin du plus lâche»
381 echo 'Le chemin du plus lâche' | grep -E -o 'L.+c'
382
383 ## Le truc pour limiter la portée du quantificateur («+» dans le
384 ## présent exemple) consiste à modifier la recherche
385 ##
386 ## L suivi d'au moins un caractère, puis un c
387 ##
388 ## par

```

```
389 ##
390 ## L suivi de n'importe quoi d'autre qu'un c, puis un c.
391 ##
392 ## De cette manière, le moteur d'expressions régulière devra cesser sa
393 ## recherche au tout premier «c» rencontré!
394 echo 'Le chemin du plus lâche' | grep -E -o 'L[^\c]+c'
395
396 ## Utilisons ce truc pour identifier dans le fichier 'bases.R' les
397 ## utilisations de la fonction de concaténation 'c' avec exactement
398 ## trois arguments. Les arguments sont séparés par des virgules.
399 ##
400 ## Les parenthèses étant des opérateurs dans les expressions
401 ## régulières étendues, il faut les désactiver dans le motif avec le
402 ## caractère d'échappement pour rechercher les caractères eux-mêmes.
403 grep -E 'c\([^,]*, \[^,]*, \[^,]*\)\' bases.R
404
405 ## Si vous avez étudié attentivement le résultat de la commande
406 ## précédente, vous aurez identifié quelques anomalies.
407 ##
408 ## Pourquoi retrouve-t-on quelques appels de fonction avec seulement
409 ## deux arguments?
410 ##
411 ## Réponse: une autre virgule suivie de texte et d'une parenthèse
412 ## fermante se trouve plus loin sur la ligne, au-delà de la véritable
413 ## fin de l'appel de fonction.
414 ##
415 ## Améliorons un peu notre motif pour empêcher de trouver un accord
416 ## avec une parenthèse fermante avant le troisième argument.
417 grep -E 'c\([^,)]*, \[^,)]*, \[^,]*\)\' bases.R
418
419 ## Tournons-nous brièvement vers les plages de caractères et les
420 ## classes prédéfinies.
421 ##
422 ## Le fichier '100metres.dat' contient la liste des 31 meilleurs temps
423 ## enregistrés au 100 mètres homme entre 1964 et 2005. Identifions,
424 ## parmi les records effectués dans les mois de mai, juin et juillet.
425 ##
426 ## (J'ai ajouté un «-» à la fin du motif pour m'assurer de ne pas
427 ## identifier une date entre le 6e et le 8e jour du mois. Vous
428 ## remarquerez aussi que le motif utilisé ne résisterait pas à l'ajout
429 ## de dates entre 2006 et 2008, inclusivement.)
430 grep '0[6-8]-' 100metres.dat
431
432 ## Le fichier 'NEWS' contient l'historique des versions de l'ouvrage.
433 ##
434 ## Les nouvelles versions sont toujours identifiées par un symbole '#'
435 ## unique en début de ligne suivi d'un chiffre (le premier du numéro
436 ## de version).
```

```

437 ##
438 ## La commande suivante dresse la liste des numéros de versions de
439 ## l'ouvrage au fil des années.
440 grep -E '^# [[:digit:]]' NEWS
441
442 ## ALTERNANCE
443
444 ## Le caractère «|» permet d'effectuer un choix entre deux expressions
445 ## régulières étendues. Autrement dit, c'est l'opérateur «ou» comme
446 ## dans plusieurs langages de programmation (dont R).
447 grep -E 'mouvement|fatigue' chanson.txt
448
449 ## L'alternance se fait entre tout ce qui se trouve à gauche de
450 ## l'opérateur et tout ce qui se trouve à droite.
451 ##
452 ## Conséquemment, il faut souvent utiliser l'opérateur d'alternance
453 ## avec les parenthèses pour en limiter la portée.
454 ##
455 ## La comparaison des quatre commandes de substitution ci-dessous
456 ## permet d'observer l'effet des parenthèses.
457 sed -E 's/le mouvement|la fatigue/le pendule/' chanson.txt
458 sed -E 's/l[ea] (mouvement|fatigue)/le pendule/' chanson.txt
459 sed -E 's/le|la mouvement|fatigue/le pendule/' chanson.txt
460 sed -E 's/(le|la) (mouvement|fatigue)/le pendule/' chanson.txt
461
462 ## Allons-y maintenant d'une expression régulière qui permet de
463 ## trouver toutes les utilisations de la fonction 'g' dans les
464 ## fichiers d'exemples, c'est-à-dire la chaîne 'g(' précédée d'au
465 ## moins une espace ou en début de ligne.
466 grep -E '( |^)g\(' *.R
467
468 ## GROUPES
469
470 ## Outre limiter la portée de l'opérateur d'alternance, les
471 ## parenthèses servent, un peu comme en mathématiques, à définir des
472 ## groupes dans les expressions régulières étendues. Nous pouvons
473 ## ensuite appliquer des opérateurs à ces groupes.
474 ##
475 ## Penchons-nous de nouveau sur l'exemple d'identification des
476 ## utilisations de la fonction de concaténation 'c' avec exactement
477 ## trois arguments dans 'donnees.R'.
478 ##
479 ## Puisque le premier et le second bloc du motif sont identiques, nous
480 ## pouvons réécrire le motif comme ci-dessous. (Ça devient moins
481 ## lisible, hein?).
482 grep -E 'c\(([^\,]*)\, ){2}([^\,]*)' bases.R
483
484 ## Les parenthèses servent aussi à créer des groupes de capture. C'est

```

```
485 ## surtout utile avec 'sed' pour réutiliser dans le texte de
486 ## remplacement du texte trouvé lors de la recherche.
487 sed -E 's/Le (.*) (.*) (.*)/Le \2 \3 \1/'
488
489 ## La commande ci-dessous remplace toutes les occurrences de la chaîne
490 ## «foo», mais pas de «foob» (comme «foobar»), dans 'bases.R' par
491 ## «abc» et le caractère qui suit «foo».
492 sed -E 's/foo([^\b])/abc\1/g' bases.R
493
494 ###
495 ### TRAITEMENT DE TEXTE DIVISÉ EN CHAMPS AVEC AWK
496 ###
497
498 ## Cette section regroupe certains exemples d'utilisation de 'awk'
499 ## tirés du texte du chapitre, ainsi que des exemples additionnels.
500 ##
501 ## Effectuons d'abord l'extraction des temps des records dans le
502 ## fichier '100metres.dat'. C'est le second champ du fichier. Voilà un
503 ## travail tout désigné pour 'awk'.
504 awk '{ print $2 }' 100metres.dat
505
506 ## Nous pouvons aussi aisément avec 'awk' inverser les deux colonnes
507 ## du fichier.
508 awk '{ print $2, $1 }' 100metres.dat
509
510 ## Les colonnes du fichier '100metres.dat' sont séparées par une
511 ## espace. Remplaçons ces espaces par des virgules pour convertir le
512 ## fichier en format CSV. Je fournis deux solutions, avec 'awk' (la
513 ## plus simple) et avec 'sed'.
514 awk '{ print $1 "," $2 }' 100metres.dat
515 sed 's/ /,/' 100metres.dat
516
517 ## Changeons maintenant le format de la date du format ISO 8601
518 ## 'aaaa-mm-qq' vers le format américain 'qq/mm/aa'. D'abord une
519 ## solution avec 'awk'.
520 ##
521 ## En premier lieu, nous devons changer le séparateur de champs de
522 ## 'awk' pour l'espace et le tiret. Ainsi, 'awk' va non seulement
523 ## séparer les deux colonnes du fichier, mais aussi les champs de la
524 ## date.
525 ##
526 ## Il suffit ensuite de replacer les champs dans l'ordre voulu avec
527 ## les bons séparateurs. La fonction 'substr' de awk permet de
528 ## sélectionner une partie d'une chaîne de caractères (voir la
529 ## section~9.1.3 du guide d'utilisation de GNU Awk).
530 awk 'BEGIN { FS = "[ -]" }
531       { print $2/"$3"/"substr($1, 3), $4 }' \
532       100metres.dat
```

```

533
534 ## La solution avec 'sed' maintenant, qui repose non pas sur des
535 ## champs détectés automatiquement, mais plutôt sur la recherche et le
536 ## remplacement d'expressions régulières.
537 ##
538 ## Le motif recherche, en capturant chaque élément trouvé (sauf les
539 ## tirets):
540 ##
541 ## - les nombres 19 ou 20;
542 ## - deux chiffres;
543 ## - un tiret;
544 ## - deux chiffres;
545 ## - un tiret;
546 ## - deux chiffres;
547 ## - tout le reste de la ligne.
548 ##
549 ## Il s'agit ensuite de replacer les éléments capturés dans l'ordre
550 ## souhaité avec des barres obliques '/' entre les éléments de dates.
551 ##
552 ## Comme le symbole '/' est utilisé dans la chaîne de sortie, il vaut
553 ## mieux utiliser un autre symbole pour séparer les champs de la
554 ## commande 'sed'. J'ai utilisé le symbole '~'.
555 sed -E \
556 's~(19|20)([0-9]{2})-([0-9]{2})-([0-9]{2})(.*)~\3/\4/\2\5~'\
557 100metres.dat
558
559 ## Les deux commandes suivantes sont reprises du texte du chapitre.
560 ## Elles utilisent le fichier 'carburant.txt'.
561 ##
562 ## La première permet de créer une nouvelle base de données ne
563 ## contenant que les modèles de voitures 4 cylindres sans passer par
564 ## un tableur ou par R.
565 awk '/^[#a-z]/ || $2 == 4' carburant.dat
566
567 ## L'ajout d'une action à la commande précédente permet de conserver
568 ## uniquement les deux premiers champs des données. La sélection doit
569 ## évidemment s'effectuer uniquement sur les titres de colonnes et sur
570 ## les données.
571 awk '/^[#a-z]/ || $2 == 4 {
572     print ($0 ~ /\^#/ ) ? $0 : $1 " " $2
573 }' carburant.dat
574
575 ## La commande suivante, aussi reprise du texte du chapitre, extrait
576 ## les deux premiers couplets de «La journée qui s'en vient est
577 ## flambant neuve».
578 awk '/^$/ && state == 1 { exit }
579     /^$/ { state = 1 }
580     1' chanson.txt

```



```

581
582 ## Dans la même veine, nous souhaitons extraire du fichier 'NEWS' les
583 ## notes de mise à jour de la plus récente version de l'ouvrage.
584 ##
585 ## Les notes relatives à une version donnée débutent toujours à une
586 ## ligne marqué par '# '.
587 ##
588 ## Cette opération requiert une variable d'état pour identifier les
589 ## lignes que nous souhaitons extraire.
590 ##
591 ## De la manière dont les règles sont écrites dans la commande, leur
592 ## ordre n'a pas d'importance. (Déterminez pourquoi en comparant avec
593 ## l'exemple similaire ci-dessus.)
594 ##
595 ## La dernière règle sert à afficher les lignes que nous voulons
596 ## extraire (grâce à l'action implicite). Le motif est équivalent à
597 ## 'state == 1' ou 'state != 0' puisque 'awk' évalue l'expression et
598 ## détermine que le motif correspond si la valeur est non nulle.
599 awk '(state == 0) && /^# / { state = 1; next }
600       (state == 1) && /^# / { exit }
601 state' NEWS

```

11.7 Exemples pour les fonctions R

📍 Fichier d'accompagnement texte.R

```


11 ## Reprenons d'abord certains des exemples pour la ligne de commande
12 ## Unix effectués sur le fichier chanson.txt.
13 ##
14 ## Pour ce faire, il faut importer le contenu du fichier dans R. La
15 ## fonction 'readLines' lit un fichier et retourne un vecteur de
16 ## chaînes de caractères, à raison d'une chaîne par ligne du fichier.
17 ## Il faut spécifier le type de codage du fichier lorsque l'UTF-8
18 ## n'est pas le type par défaut de votre système d'exploitation (bref,
19 ## sous Windows).
20 ##
21 ## Vous remarquerez que cela correspond au mode de fonctionnement de
22 ## 'grep', 'sed' et 'awk': une ligne à la fois.
23 chanson <- readLines("chanson.txt",
24                       encoding = "UTF-8") # importer le fichier
25 chanson                                     # contenu du vecteur
26
27 ## Effectuons maintenant quelques recherches dans le vecteur avec la
28 ## fonction 'grep' de R. Le résultat est l'indice de l'élément du
29 ## vecteur contenant le motif.
30 grep("temps", chanson)      # vers contenant le mot «temps»
31 grep("chat", chanson)       # vers contenant le mot «chat»
32

```

```
33 ## Pour obtenir le texte de l'élément plutôt que sa position dans le
34 ## vecteur, il faut utiliser 'value = TRUE'.
35 grep("temps", chanson, value = TRUE)
36
37 ## R utilise par défaut des expressions régulières étendues (donc le
38 ## symbole | est un opérateur d'alternance).
39 grep("amour|mieux", chanson, value = TRUE)
40
41 ## Changeons un peu de jeu de données.
42 ##
43 ## La fonction 'data' retourne la liste des jeux de données disponible
44 ## dans la session R courante. Le résultat est une liste de plusieurs
45 ## éléments.
46 data()
47
48 ## L'expression ci-dessus permet de ne conserver que les noms
49 ## d'objets.
50 (nm <- data()$results[, "Item"])
51
52 ## Pour extraire de la liste les objets dont le nom comporte un point
53 ## «.», il faut désactiver le rôle spécial de ce caractère dans
54 ## l'expression régulière à l'aide du caractère d'échappement.
55 grep("\\.", nm) # erreur!
56
57 ## L'expression ci-dessus cause une erreur parce que R interprète le
58 ## caractère d'échappement à l'intérieur du motif.
59 ##
60 ## Il existe deux solutions pour régler ce problème.
61 ##
62 ## La première consiste à doubler le caractère d'échappement. Ça
63 ## devient rapidement pénible s'il y a plusieurs caractères à doubler.
64 grep("\\\\.", nm, value = TRUE)
65
66 ## La seconde est disponible depuis la version 4.0.0 de R. Il s'agit
67 ## des chaînes de caractères brutes. En gros, une chaîne de caractères
68 ## brute se charge d'ajouter les caractères d'échappement aux bons
69 ## endroits et retourne une chaîne de caractères standard.
70 ##
71 ## On crée une chaîne brute avec 'r"(...)"'.
72 r"(\.)" # syntaxe d'une chaîne brute
73 r"(g.*\.*\())" # autre exemple
74 grep(r"(\.)", nm, value = TRUE) # utilisation dans 'grep'
75
76 ## L'expression suivante identifie les éléments (ou vers) de 'chanson'
77 ## qui débutent par «L» et qui contiennent un «i» quelque part dans la
78 ## chaîne. Le résultat est un vecteur des positions des éléments qui
79 ## correspondent au motif.
80 grep("^L.*i", chanson) # positions des éléments
```

```
81
82 ## La fonction 'regexpr' retourne davantage d'informations sur la
83 ## position et la longueur des appariements du motif dans la chaine de
84 ## caractères.
85 ##
86 ## Le vecteur des résultats contient une valeur positive correspondant
87 ## à la position dans la chaine où débute chacun des appariements, ou
88 ## -1 si la chaine ne correspond pas au motif. (Dans notre exemple,
89 ## les nombres positifs sont tous des 1 puisque nous recherchons
90 ## précisément des appariements dès le début de la ligne.)
91 ##
92 ## Le vecteur des résultats est également muni d'un attribut qui
93 ## contient la longueur des appariements.
94 regexpr("^L.*i", chanson) # plus d'informations
95
96 ## À partir des résultats de 'regexpr', il est possible d'extraire
97 ## uniquement les portions de texte qui correspondent au motif, un peu
98 ## comme avec l'option '-o' de 'grep'.
99 ##
100 ## Il suffit, pour chaque appariement, d'extraire le texte de la
101 ## chaine à partir de la position initiale de l'appariement et pour la
102 ## longueur de celui-ci.
103 ##
104 ## C'est ce travail qu'effectue la fonction 'regmatches'.
105 regmatches(chanson, regexpr("^L.*i", chanson))
106
107 ## Les fonctions 'sub' et 'gsub' jouent en partie le rôle de 'sed'
108 ## dans R, soit remplacer des chaines de caractères.
109 ##
110 ## La fonction 'sub' remplace uniquement la première occurrence dans
111 ## une chaine de caractères, alors que 'gsub' remplace toutes les
112 ## occurrences (comme 'sed' lorsque muni du modificateur 'g').
113 sub("Oh", "Ah", chanson)
114 sub("promets", "jure", chanson)
115 gsub("promets", "jure", chanson)
116
117 ## Plusieurs autres fonctions de R acceptent des expressions
118 ## régulières en argument.
119 ##
120 ## Par exemple, la fonction 'list.files' permet d'obtenir la liste des
121 ## fichiers (du répertoire de travail par défaut) dont le nom
122 ## correspond à un certain motif.
123 ##
124 ## Les expressions ci-dessous affichent la liste des fichiers dont le
125 ## nom se termine par l'extension ".R".
126 list.files(pattern = "\\..R$") # avec chaine standard
127 list.files(pattern = r"\\.R$") # avec chaine brute
```

11.8 Exercices

11.1 Voici l'exercice — une série d'exercices, en fait — la plus ludique du document : les mots croisés d'expressions régulières de **Regex Crossword** . Compléter les casse-têtes jusqu'au niveau *Experienced...* ou au-delà si le cœur vous en dit!

11.2 Déterminer à laquelle ou auxquelles des chaînes de caractères correspondent les motifs d'expression régulière étendue ci-dessous ¹⁶.

a) `「a(ab)*a」`

- | | | |
|------------|-------------|-----------|
| 1. abababa | 2. aaba | 3. aabbaa |
| 4. aba | 5. aabababa | |

b) `「ab+c?」`

- | | | |
|--------|---------|---------|
| 1. abc | 2. ac | 3. abbb |
| 4. bbc | 5. abcc | |

c) `「a.[bc]+」`

- | | | |
|---------------|----------------|---------------------|
| 1. abc | 2. abbbbbbbbbb | 3. azc |
| 4. abcbcbcbcb | 5. ac | 6. asccbbbbbcbccccc |

d) `「abc|xyz」`

- | | | |
|--------|--------|------------|
| 1. abc | 2. xyz | 3. abc xyz |
|--------|--------|------------|

e) `「[a-z]+[.?!]」`

- | | | |
|--------------|-------------|--------------|
| 1. battle! | 2. Hot | 3. green |
| 4. swamping. | 5. jump up. | 6. undulate? |
| 7. is.? | | |

f) `「[a-zA-Z]*[^(,)=]」`

- | | | |
|--------------|--------------|------------|
| 1. Butt= | 2. BotHEr,= | 3. Ample |
| 4. FIdDlE7h= | 5. Brittle = | 6. Other.= |

g) `「[a-z][.?!][[:blank:]]+[A-Z]」`

- | | | |
|---------|---------|--------|
| 1. A. B | 2. c! d | 3. e f |
| 4. g. H | 5. i? J | 6. k L |

h) `「<[^>]+>」`

16. Exercice adapté de <https://regex.sketchengine.eu/basic-exercises.html>.

1. `<an xml tag>`
2. `<opentag> <closetag>`
3. `</closetag>`
4. `<>`
5. `<with attribute="77">`

11.3 Composer un motif d'expression régulière qui correspond à tous les mots de la liste de gauche, ci-dessous, mais à aucun des mots de la liste de droite¹⁷.

pit	pt
spot	Pot
spate	peat
slap two	part
respice	

11.4 Composer un motif d'expression régulière qui permet de vérifier la validité d'un code postal canadien dans un formulaire électronique. Ne pas tenir compte des règles précises de composition d'un code postal, mais bien seulement qu'il s'agit d'une suite de six caractères alternant entre une lettre et un chiffre. Les lettres peuvent être fournies en majuscule ou en minuscule et l'espace entre le troisième et le quatrième symbole peut être présente ou non.

11.5 Écrire une commande `sed` permettant de changer le séparateur décimal dans les temps du fichier `100metres.dat` pour une virgule.

11.6 Utiliser les outils étudiés dans ce chapitre pour extraire les informations suivantes du fichier `100metres.dat`.

- a) Les informations des temps réalisés au mois d'août.
- b) Les informations des temps de moins de 10 secondes.
- c) Les informations des temps de moins de 10 secondes réalisés au mois d'août. Vous pouvez avoir recours au tuyau Unix « `|` » pour combiner les commandes des parties a) et b).

11.7 Pour tous les appels à une fonction `f` comportant deux arguments dans le fichier `environnement.R` livré avec le présent ouvrage, changer le nom de la fonction pour `fun` et inverser l'ordre des deux arguments.

11.8 Résoudre le présent exercice dans R en utilisant le vecteur interne de données `state.name`.

- a) Déterminer les positions, dans l'ordre alphabétique, des États dont le nom comporte deux « s » de suite.
- b) Extraire les États dont le nom comporte au moins trois fois la lettre « s », qu'elles soient successives ou non.

17. Exercice adapté de <https://regex.sketchengine.co.uk/cgi/ex1.cgi>.

- c) Extraire les États dont le nom comporte un double « s » parmi les quatre premiers caractères, puis abrégier leur nom après ce double « s » et faire suivre d'un point. Par exemple : « Massachusetts » devient « Mass. ».

11.9 Vous remarquez que plusieurs de vos collègues ont tendance à oublier d'inclure la section « Exemples » dans la documentation de leurs fonctions R. Vous souhaitez donc concevoir un petit outil qui permettra de déterminer si la section est présente ou non dans la documentation d'une fonction rédigée selon le format du gabarit fourni avec le présent ouvrage. (Il faut uniquement valider que la section est présente, et non que l'information qu'elle contient est juste ou pertinente.)¹⁸

- a) Concevoir un motif d'expression régulière le plus général possible pour trouver le nom de la section dans la documentation d'une fonction. Saisir le motif sous forme de chaîne de caractères brute (*raw string*) dans R.
- b) Créer une fonction `is.examplesPresent` dans R qui retourne une valeur booléenne selon que la section est présente ou non. L'argument de la fonction est un vecteur de chaînes de caractères. Vous pouvez ensuite tester la fonction avec les fichiers `sqrt.R` et `sqrt-sans-exemples.R` livrés avec l'ouvrage. Utilisez la fonction `readLines` avec l'option `encoding = "UTF-8"` pour importer le contenu de ces fichiers dans R sous forme de vecteurs de chaînes de caractères.

11.10 Les commandes R suivantes créent une base de données `data.csv` d'un million d'enregistrements et 20 champs.

```
> x <- sample(1:100, 20 * 1e6, replace = TRUE)
> write.csv(matrix(x, ncol = 20), file = "data.csv")
```

Produire une nouvelle base de données composée des deux premières colonnes et d'une colonne contenant la somme des deux autres, d'abord avec un tableur, puis avec `awk`. (Vous aurez reconnu le contexte de l'exemple au début de la [section 11.4.](#))

18. Exercice conçu par Jean-Christophe Langlois.

12 Environnement et règles d'évaluation

Objectifs du chapitre

- ▶ Décrire les concepts d'environnement de fonction et d'environnement d'évaluation de R.
- ▶ Comprendre comment R associe des symboles à des valeurs.
- ▶ Utiliser la portée lexicale et l'évaluation paresseuse pour simplifier les fonctions R.

Ce chapitre vise à expliquer plusieurs règles d'évaluation et de portée des variables que nous avons jusqu'ici un peu laissées dans l'ombre pour en appeler plutôt à votre bonne intuition. Pour illustrer, reprenons l'un de nos tous premiers exemples de fonction au [chapitre 4](#), celui du calcul du carré d'un nombre avec la fonction `square`.

```
> square <- function(x) x * x
```

Nous n'avons jamais remis en doute que la valeur de `x` à l'intérieur de la fonction `square` doit être celle passée en argument, et non celle d'un objet `x` qui existerait déjà dans l'espace de travail. De plus, si un tel objet `x` devait exister dans l'espace de travail, l'exécution de la fonction `square` ne devrait pas modifier sa valeur.

```
> x <- 5
> square(10)
[1] 100
> x
[1] 5
```

Examinons maintenant cet autre exemple quelque peu artificiel, mais néanmoins instructif, tiré de [Ihaka et Gentleman \(1996\)](#). Une fonction `f` calcule le carré d'un nombre, puis affiche le résultat à l'écran¹ par le biais d'une autre fonction `g`.

1. Oui, je vous ai dit à la [section 4.4.3](#) de ne jamais utiliser `print` ainsi. Permettons aux créateurs

```
> f <- function(x)
+ {
+   y <- x * x
+   g <- function() print(y)
+   g()
+ }
```

On remarquera que l'objet `y` n'est pas un argument de `f`, mais qu'il est défini dans la fonction. De plus, la fonction `g` utilise `y` sans que cet objet ne soit un de ses arguments. Dans les circonstances, quels seront, selon vous, les résultats des expressions suivantes et, surtout, pourquoi ? Allez les essayer, je vous attends ici.

```
> y <- 123
> f(10)
> y
```

Les questions ci-dessus trouvent leurs réponses dans les concepts d'environnement et de portée lexicale de R. Maîtriser ces concepts permet à la fois de simplifier notre code et de réduire significativement les sources de bogues. Le chapitre traite également du concept d'évaluation paresseuse auquel les programmeurs sont moins souvent directement confrontés, mais qui permet aussi de coder plus efficacement.



Avant d'aller plus loin, passez en revue les lignes 12-27 du fichier de script `environnement.R` reproduit à la [section 12.5](#). Je vous invite particulièrement à tenter de répondre à la colle de la ligne 27.

12.1 Environnement

Un *environnement* dans R est un objet spécial formé de deux composantes : un dictionnaire² constitué de couples symbole-valeur ; un pointeur vers l'environnement englobant. La [figure 12.1](#) propose une représentation schématique de l'environnement.

La notion, plutôt abstraite, devient plus concrète dans le contexte de la création et des appels de fonctions, où les environnements jouent un rôle important. Nous nous concentrons dans cette section sur le cas de la création des fonctions ; le rôle des environnements dans les appels de fonctions est étudié à la section suivante.

En termes techniques, une fonction définie avec `function` dans R constitue une *fermeture*, ou *clôture* (du Lisp *closure*). Cette fermeture est formée de trois éléments, dont deux que nous connaissons déjà bien : les arguments formels de

de R cette entorse à la règle en les soupçonnant de savoir ce qu'ils font.

2. La traduction littérale du terme *frame* utilisé dans la documentation de R serait *cadre*, mais ce terme est moins explicite que « dictionnaire ». Oui, il s'agit bien du même *frame* que l'on retrouve dans *data frame*.

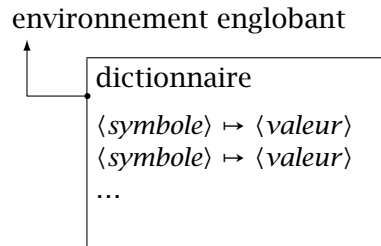


FIG. 12.1 – Représentation schématique d'un environnement dans R

la fonction et le corps de la fonction. Le troisième élément est l'environnement de la fonction. Les fonctions `formals`, `body` et `environment` permettent d'extraire ces trois éléments, dans l'ordre.

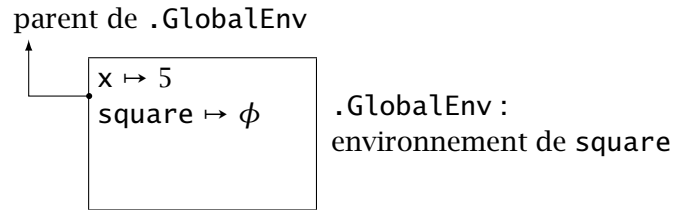
L'environnement d'une fonction est l'environnement actif au moment de la définition de la fonction (à moins qu'il ne soit changé) et le dictionnaire « capture » tous les objets (ou toutes les paires symbole-valeur) de ce même environnement actif. L'environnement englobant de la fonction est donc le même que celui de l'environnement actif.

Autrement dit, dès que nous définissons une fonction dans R, l'interpréteur lui attribue un environnement. Lorsque la fonction est créée dans l'espace de travail, son environnement est l'*environnement global*, représenté par le symbole `.GlobalEnv`, et son dictionnaire contient tous les objets se trouvant dans l'espace de travail. (Les expressions ci-dessous sont exécutées dans un espace de travail vide au préalable.)

```
> x <- 5
> square <- function(x) x * x
> formals(square)
$x
> body(square)
x * x
> environment(square)
<environment: R_GlobalEnv>
> ls(envir = environment(square))
[1] "square" "x"
> get("x", envir = environment(square))
[1] 5
```

La [figure 12.2](#) propose une représentation schématique de l'environnement de la fonction `square`.

Quelques fonctions permettent de manipuler les environnements de fonctions. Nous en avons déjà utilisé trois ci-dessus.

FIG. 12.2 – Environnement de la fonction `square`

<code>environment</code>	extraction ou modification de l'environnement.
<code>ls</code>	affichage de la liste des éléments de l'environnement.
<code>get</code>	extraction de la valeur associée à un symbole dans l'environnement.
<code>assign</code>	affectation d'une nouvelle paire symbole-valeur dans l'environnement.
<code>new.env</code>	création d'un nouvel environnement (vide par défaut).



La fonction `ls.str` permet d'afficher en une seule commande la liste des objets d'un environnement et les valeurs de ceux-ci.

12.2 Environnement d'évaluation

Dans R, tout appel de fonction entraîne implicitement la création d'un *environnement d'évaluation*, distinct de l'environnement de la fonction, et dans lequel, comme le nom l'indique, la fonction sera évaluée. Cet environnement d'évaluation ne vit que durant l'exécution de la fonction. Son environnement englobant est l'environnement de la fonction et son dictionnaire comprend toutes les variables définies à l'intérieur de la fonction, y compris les arguments.

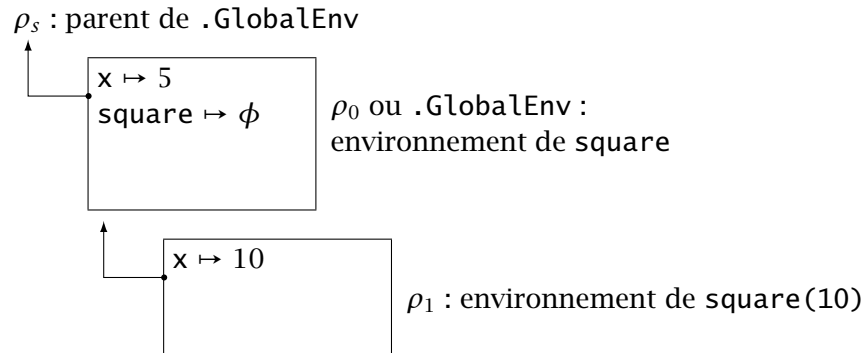
Pour bien illustrer tout cela, nous aurons recours à l'ingénieuse notation de [Ihaka et Gentleman \(1996\)](#) par laquelle un environnement d'évaluation ρ est représenté sous la forme :

$$\rho = \{ \langle \text{symbole} \rangle \mapsto \langle \text{valeur} \rangle, \langle \text{symbole} \rangle \mapsto \langle \text{valeur} \rangle, \dots; \rho_e \},$$

où ρ_e est l'environnement englobant. Nous noterons ρ_0 l'environnement global et ρ_s son environnement englobant³, l'environnement contenant les fonctions et les constantes du système. Reprenons l'exemple de la section précédente.

```
> x <- 5
> square <- function(x) x * x
```

3. Oui, l'environnement dit « global » est lui-même englobé d'un environnement !

FIG. 12.3 – Environnement d'évaluation de la fonction `square`

Après l'exécution des deux commandes ci-dessus (toujours dans un espace de travail vide au préalable), l'environnement global contient les objets `x` et `square`. Nous avons donc :

$$\rho_0 = \{x \mapsto 5, \text{square} \mapsto \phi; \rho_s\},$$

où ϕ représente symboliquement la définition de la fonction `square`.

Un appel à la fonction `square` crée un nouvel environnement d'évaluation ρ_1 .

```
> square(10)
```

Conformément à ce qui a été établi plus haut, l'environnement englobant de ρ_1 est celui de la fonction `square`, soit ρ_0 puisque la fonction a été définie dans l'environnement global. De plus, le dictionnaire contient la paire symbole-valeur `x ↦ 10` correspondant à l'argument de la fonction. La fonction `square` est donc évaluée dans l'environnement

$$\rho_1 = \{x \mapsto 10; \rho_0\}.$$

Or, dans cet environnement, la valeur de `x` est 10. C'est pourquoi l'expression `x * x` retourne la valeur 100. La [figure 12.3](#) offre une représentation plus schématique de ce qui précède.

C'est ainsi, à l'aide des environnements d'évaluation, que R ne confond pas la valeur de `x` dans l'espace de travail et celle à l'intérieur de la fonction `square`, pas plus que la seconde n'écrase la première durant l'exécution de la fonction.

Il y a toutefois plus à connaître à ce sujet. C'est l'objet de la prochaine section.



La mécanique de création d'environnements d'évaluation pour tous les appels de fonctions fait en sorte que ceux-ci coûtent cher en temps et en ressources système. C'est pour cette raison que le calcul récursif est plutôt inefficace en R.



Étudiez les exemples additionnels des lignes 30-105 du fichier de script `environnement.R` reproduit à la [section 12.5](#).

12.3 Portée lexicale

Qu'arrive-t-il lorsqu'une variable n'est pas définie dans l'environnement d'évaluation d'une fonction ? C'est là qu'entrent en jeu les très importantes règles de portée lexicale (*lexical scoping*) de R. Celles-ci établissent comment R recherche un symbole.

- ▶ En premier lieu, R recherche le symbole dans le dictionnaire de l'environnement courant. Si le symbole est trouvé, sa valeur est retournée et la recherche est terminée.
- ▶ Autrement, R recherche le symbole dans le dictionnaire de l'environnement englobant. Cette procédure se répète autant de fois que nécessaire. En d'autres termes, R recherche dans l'environnement englobant, dans l'environnement englobant de l'environnement englobant, et ainsi de suite jusqu'à ce que le symbole soit trouvé.
- ▶ Lorsque R atteint l'environnement global `.GlobalEnv`, la recherche se poursuit le long du chemin de recherche (*search path*) qui contient les paquets chargés dans la session de travail :

```
> search()
[1] ".GlobalEnv"          "package:RweaveExtra"
[3] "package:stats"       "package:graphics"
[5] "package:grDevices"   "package:utils"
[7] "package:datasets"    "package:methods"
[9] "AutoLoads"           "package:base"
```

- ▶ Si R n'a toujours pas trouvé le symbole dans l'environnement du package de base (`package:base` ci-dessus), il passe à l'environnement vide et, par conséquent, la recherche du symbole échoue.

Les règles de portée lexicale méritent une bonne illustration. Reprenons le second exemple de l'introduction du chapitre.

```
> y <- 123
> f <- function(x)
+ {
+   y <- x * x
+   g <- function() print(y)
+   g()
+ }
> f(10)
```

En supposant, comme précédemment, un espace de travail vide avant l'exécution des commandes ci-dessus, l'environnement global est maintenant :

$$\rho_0 = \{y \mapsto 123, f \mapsto \phi; \rho_s\}.$$

La fonction f étant définie dans l'espace de travail, son environnement est ρ_0 .

L'évaluation de f en une certaine valeur entraîne la création d'une cascade d'environnements. Tout d'abord, l'appel de la fonction crée, comme dans l'exemple de la [section 12.2](#), l'environnement d'évaluation

$$\rho_1 = \{x \mapsto 10; \rho_0\}.$$

Le corps de la fonction f est ensuite évalué dans ρ_1 , ce qui entraîne l'ajout de deux objets dans l'environnement :

$$\rho_1 = \{x \mapsto 10, y \mapsto 100, g \mapsto \gamma; \rho_0\},$$

où γ est une représentation symbolique de la fonction g . Celle-ci se voit automatiquement munie d'un environnement ρ_2 dont l'environnement englobant est ρ_1 :

$$\rho_2 = \{x \mapsto 10, y \mapsto 100, g \mapsto \gamma; \rho_1\}.$$

Toujours à l'intérieur de la fonction f , l'appel $g()$ crée un nouvel environnement ρ_3 pour l'évaluation de la fonction g . L'environnement englobant est ρ_2 et, comme cette fonction n'a aucun argument, son dictionnaire est vide :

$$\rho_3 = \{; \rho_2\}.$$

L'expression `print(y)` est ensuite évaluée dans l'environnement ρ_3 . Le symbole y n'existe pas dans cet environnement. En vertu des règles de portée lexicale, R va alors rechercher le symbole dans l'environnement englobant ρ_2 . À cet endroit, la valeur 100 correspond à y . C'est donc cette valeur qui est affichée à l'écran. La valeur 123 dans l'espace de travail n'entre jamais en jeu.

```
> f(10)
[1] 100
```

La [figure 12.4](#) illustre l'arbre des environnements créés par l'appel de fonction dans l'exemple ci-dessus.



Vous aurez compris que R trouvera éventuellement dans l'espace de travail une variable qui n'est pas définie à l'intérieur d'une fonction. Il est extrêmement dangereux de compter sur cette fonctionnalité puisque la valeur dans l'espace de travail peut changer à tout moment ! Prenez l'habitude de passer en argument toutes les valeurs dont une fonction qui est appelée depuis l'espace de travail peut avoir besoin.

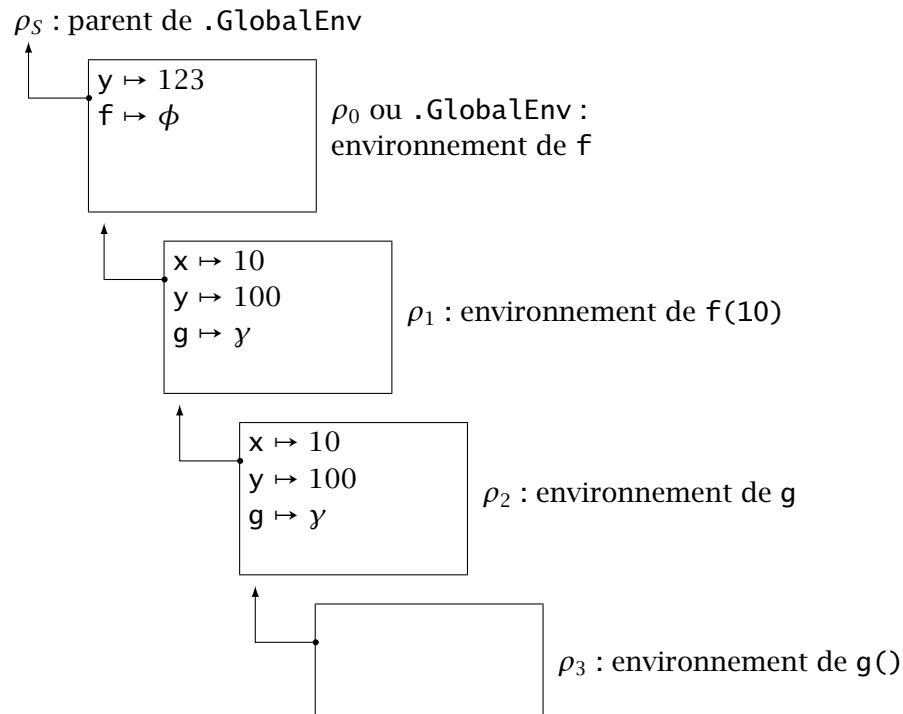


FIG. 12.4 – Arbre des environnements créés par un appel de fonction

La portée lexicale de R est un outil puissant et, comme tous les outils puissants, il peut s'avérer dangereux. La mise en garde ci-dessus enjoint de ne pas se fier à la portée lexicale dans les fonctions conçues pour être appelées depuis la ligne de commande. Par contre, la portée lexicale est très utile pour simplifier le code des fonctions créées à l'intérieur d'autres fonctions puisque nous pouvons faire l'économie de passer tous les arguments d'une fonction à l'autre.



Vous trouverez un exemple complet de portée lexicale aux lignes 108–171 du fichier de script `environnement.R` reproduit à la section 12.5.

12.4 Évaluation paresseuse

R utilise par défaut une technique d'évaluation des arguments des fonctions appelée *évaluation paresseuse* (*lazy evaluation*). Ce que cela signifie, c'est que toute expression passée en argument à une fonction ne sera évaluée qu'au moment où la valeur de l'argument sera utilisée dans la fonction, et pas avant. Ainsi, une expression en argument pourrait ne jamais être évaluée. Ou encore, la valeur par défaut d'un argument peut dépendre de la valeur d'un autre argument puisque celui-ci sera connu au moment de l'évaluation.

Par exemple, la fonction `sumsq` ci-dessous calcule la somme des écarts au carré entre les éléments d'un vecteur `y` et une valeur pivot `about`, par défaut la moyenne des éléments de `y`. On remarque que la valeur par défaut de l'argument `about` dépend de celle de l'argument `y`.

```
> sumsq <- function(y, about = mean(y))
+   sum((y - about)^2)
```

Avec l'évaluation paresseuse, le calcul de la moyenne sera effectué uniquement lorsque l'argument `about` n'est pas spécifié dans l'appel de `sumsq`, et ce, seulement au moment d'effectuer le calcul `y - about`. Le cas échéant, la valeur de `y` sera alors connue au moment d'effectuer le calcul. Joli, non ?



Étudiez les lignes 174-212 du fichier de script `environnement.R`.

12.5 Exemples

📍 Fichier d'accompagnement `environnement.R`

```
11 ###
12 ### CONTEXTE
13 ###
14
15 ## La portée (domaine où un objet existe et comporte une valeur) des
16 ## arguments d'une fonction et de tout objet défini à l'intérieur de
17 ## celle-ci se limite à la fonction.
18 ##
19 ## Ceci signifie que l'interpréteur R fait bien la distinction entre
20 ## un objet dans l'espace de travail et un objet utilisé dans une
21 ## fonction (fort heureusement!).
22 square <- function(x) x * x # fonction de bases.R
23 x <- 5                      # objet dans l'espace de travail
24 square(10)                 # dans 'square' x vaut 10
25 x                          # valeur inchangée
26 square(x)                  # passer valeur de 'x' à 'square'
27 square(x = x)              # colle... signification?
28
29 ###
30 ### ENVIRONNEMENT ET ENVIRONNEMENT D'ÉVALUATION
31 ###
32
33 ## Un environnement est un objet spécial qui contient un lien vers un
34 ## environnement englobant (ou «parent») et un dictionnaire de
35 ## symboles auxquels sont associés des valeurs (bref, des objets avec
36 ## des valeurs).
37 ##
38 ## La création d'une fonction R (une «fermeture» en termes techniques)
```

```
39 ## entraine automatiquement la création d'un environnement pour cette
40 ## fonction:
41 ##
42 ## - l'environnement englobant est l'environnement actif (dans lequel
43 ##   la fonction est créée);
44 ## - le dictionnaire «capture» les objets de l'environnement actif.
45 ##
46 ## Le cas le plus simple est celui des fonctions créées dans l'espace
47 ## de travail .GlobalEnv.
48 f <- function(x) x + 2      # création d'une fonction
49 formals(f)                 # arguments formels
50 body(f)                    # corps de la fonction
51 environment(f)             # environnement de la fonction
52
53 ## Lorsqu'une fonction est créée à l'intérieur d'une autre fonction,
54 ## son environnement hérite automatiquement de toutes les variables
55 ## définies dans la première fonction.
56 ##
57 ## L'exemple suivant devrait permettre de clarifier tout cela. La
58 ## fonction 'f', qui a un argument 'x', retourne une fonction
59 ## (anonyme) qui a un argument, 'y'.
60 f <- function(x)
61   function(y) x + y
62
63 ## L'appel de la fonction 'f' donne une valeur à son argument 'x'. À
64 ## l'intérieur de cet appel, la fonction anonyme est créée. La valeur
65 ## de 'x' est alors «capturée» dans l'environnement de la fonction
66 ## anonyme.
67 ##
68 ## En affectant le résultat (une fonction, ne l'oublions pas) de
69 ## l'appel de 'f' à un objet, celui-ci devient une fonction avec un
70 ## environnement qui contient la fameuse valeur de 'x'.
71 g <- f(2)
72 ls(envir = environment(g))
73 get("x", envir = environment(g))
74 ls.str(envir = environment(g)) # plus simple
75
76 ## Par conséquent, lorsque nous appelons la fonction 'g', l'expression
77 ## 'x + y' utilise la valeur de 'x' dans l'environnement de la
78 ## fonction et la valeur de 'y' passée en argument.
79 g(10)
80
81 ## Cette capture d'objets est parfois problématique: si plusieurs
82 ## objets existent au moment où une fonction est créée à l'intérieur
83 ## d'une autre, tous les objets se trouvent transférés dans
84 ## l'environnement de la nouvelle fonction.
85 ##
86 ## Dans de tels cas, nous pouvons utiliser la fonction 'new.env' pour
```



```
87  ## créer un environnement vide et, par la suite, affecter des objets
88  ## dans l'environnement avec la fonction 'assign'.
89  ##
90  ## Reprenons l'exemple ci-dessus en supposant que la valeur de 'x' que
91  ## nous voulons placer dans l'environnement est non pas celle passée
92  ## en argument, mais plutôt toujours 42. Parce que 42, on le sait,
93  ## c'est la réponse à tout.
94  ##
95  ## Voici comment procéder.
96  f <- function(x)
97  {
98      fun <- function(y) x + y
99      environment(fun) <- new.env()
100      assign("x", 42, envir = environment(fun))
101      fun
102  }
103  g <- f(2)
104  ls.str(envir = environment(g))
105  g(10)
106
107  ###
108  ### PORTÉE LEXICALE
109  ###
110
111  ## Tout appel de fonction dans R crée un environnement d'évaluation
112  ## dans lequel sont définies les variables locales de la fonction, y
113  ## compris les arguments. Cet environnement d'évaluation a comme
114  ## environnement englobant celui de la fonction.
115  ##
116  ## Le concept de portée lexicale signifie que lorsqu'un objet n'existe
117  ## pas dans l'environnement d'évaluation, R va le rechercher dans les
118  ## environnements englobants jusqu'à ce que soit trouvée une valeur ou
119  ## jusqu'à ce que l'environnement vide soit atteint (auquel cas
120  ## l'objet n'existe pas).
121  ##
122  ## En pratique, cela signifie qu'il n'est pas toujours nécessaire de
123  ## passer des objets en argument. Il suffit de compter sur la portée
124  ## lexicale.
125  ##
126  ## Tel que promis dans le fichier bases.R, revenons en détail sur la
127  ## construction d'une fonction pour calculer
128  ##
129  ##  $f(x, y) = x (1 + xy)^2 + y (1 - y) + (1 + xy)(1 - y)$ .
130  ##
131  ## Deux termes sont répétés dans cette expression. Nous avons donc
132  ##
133  ##  $a = 1 + xy$ 
134  ##  $b = 1 - y$ 
```

```

135 ##
136 ## et  $f(x, y) = x a^2 + y b + a b$ .
137 ##
138 ## Nous pourrions décomposer le problème en deux fonctions: une
139 ## première pour calculer ' $x a^2 + y b + a b$ ' pour des valeurs de ' $x$ ',
140 ## ' $y$ ', ' $a$ ' et ' $b$ ' données, et une seconde pour calculer ' $a = 1 + xy$ '
141 ## et ' $b = 1 - y$ ' et les fournir à la première fonction.
142 g <- function(x, y, a, b)
143   x * a^2 + y * b + a * b
144 f <- function(x, y)
145   g(x, y, 1 + x * y, 1 - y)
146 f(2, 3)
147 f(2, 4)
148
149 ## Cependant, la fonction 'g' ne sert pas à grand chose ici en dehors
150 ## de 'f'. Nous pouvons donc plutôt la définir à l'intérieur de cette
151 ## dernière.
152 f <- function(x, y)
153 {
154   g <- function(x, y, a, b)
155     x * a^2 + y * b + a * b
156   g(x, y, 1 + x * y, 1 - y)
157 }
158 f(2, 3)
159 f(2, 4)
160
161 ## La portée lexicale de R nous permet de simplifier le code: inutile
162 ## de passer les valeurs de 'x' et 'y' à la fonction 'g' puisque R les
163 ## trouvera automatiquement dans l'environnement d'évaluation de 'f'.
164 f <- function(x, y)
165 {
166   g <- function(a, b)
167     x * a^2 + y * b + a * b
168   g(1 + x * y, 1 - y)
169 }
170 f(2, 3)
171 f(2, 4)
172
173 ###
174 ### ÉVALUATION PARESSEUSE
175 ###
176
177 ## L'évaluation paresseuse est une technique par laquelle un argument
178 ## d'une fonction est évalué uniquement au moment où sa valeur est
179 ## requise, et jamais avant.
180 ##
181 ## Il est sans doute raisonnable de supposer que l'objet
182 ## 'does_not_exist' n'existe pas dans votre espace de travail. Avec

```

```

183 ## l'évaluation paresseuse, nous pourrions passer cet objet inexistant
184 ## en argument à une fonction sans que cela ne cause d'erreur... à
185 ## condition que l'argument ne soit jamais utilisé dans la fonction.
186 f <- function(x, y)
187   if (x > 0) x * x else y
188 f(5, does_not_exist)      # argument 'y' jamais utilisé
189 f(0, does_not_exist)      # argument 'y' utilisé
190 f(0, 1/0)                 # argument 'y' utilisé
191
192 ## Voici un autre exemple plus intéressant tiré de Ihaka et Gentleman
193 ## (1996).
194 ##
195 ## Vous savez que tout calcul avec des données manquantes retourne
196 ## 'NA'.
197 ##
198 ## Nous pouvons généraliser la fonction 'sumsq' présentée dans le
199 ## texte du chapitre pour faire en sorte qu'elle puisse retirer les
200 ## éventuelles données manquantes d'un vecteur en argument avant de
201 ## faire la somme des écarts au carré.
202 ##
203 ## Grâce à l'évaluation paresseuse, les données manquantes seront
204 ## également supprimées *avant* le calcul de la moyenne dans le cas où
205 ## c'est la valeur par défaut de l'argument 'about' qui est utilisée.
206 sumsq <- function(y, about = mean(y), na.rm = FALSE)
207 {
208   if (na.rm)
209     y <- y[!is.na(y)]
210   sum((y - about)^2)
211 }
212 sumsq(c(10, 0, NA, -10), na.rm = TRUE)

```

12.6 Exercices

12.1 La fonction `ecdf` de R retourne une fonction pour calculer la fonction de répartition empirique d'un vecteur de données (x_1, \dots, x_n) :

$$F_n(x) = \frac{\#x_i \leq x}{n}.$$

```

> Fn <- ecdf(c(1, 1, 3))
> mode(Fn)
[1] "function"
> Fn(0:3)
[1] 0.0000000 0.6666667 0.6666667 1.0000000

```

Expliquer pourquoi il n'est pas nécessaire de fournir les valeurs du vecteur x à F_n lorsque l'on veut calculer $F_n(x)$. *Indice* : fouiller dans l'environnement de la fonction F_n .

12.2 Justifier le résultat du bloc d'expressions ci-dessous.

```
> y <- 123
> f <- function(x)
+ {
+   z <- x * x
+   g <- function() print(y)
+   g()
+ }
> f(10)
[1] 123
```

12.3 Déterminer le résultat du bloc d'expressions ci-dessous⁴.

```
> x <- 20
> f <- function()
+ {
+   x <- 10
+   function() x
+ }
> g <- f()
> g()
```

12.4 Déterminer le résultat du bloc d'expressions ci-dessous⁵.

```
> x <- 0
> y <- 10
> f <- function()
+ {
+   x <- 1
+   g()
+ }
> g <- function()
+ {
+   x <- 2
+   h()
+ }
> h <- function()
+ {
```

4. Exercice tiré de [Wickham \(2024\)](#).

5. *Idem*.

```
+ x <- 3  
+ x + y  
+ }  
> f()
```

- 12.5** Écrire une fonction R qui calcule le produit de deux nombres passés en arguments. Si un seul nombre est donné en argument, la fonction calcule le produit de ce nombre avec son logarithme. *Indice* : tirer profit de l'évaluation paresseuse.

A 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 paquetages 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.

A.1 Installation

RStudio est offert à l'identique pour les plateformes Windows, macOS et Linux. Pour une utilisation locale sur votre poste de travail, installez la version libre (*Open Source*) de **RStudio Desktop** [↗](#).

A.2 Description sommaire

La fenêtre de RStudio se divise toujours en quatre sous-fenêtres² — sauf au lancement, alors que la sous-fenêtre d'édition de code source n'est pas visible ; voir la [figure A.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 ;

1. À ce compte, GNU Emacs constitue un environnement de développement intégré.

2. Les sous-fenêtres sont appelées *panes* (en anglais) dans l'application.

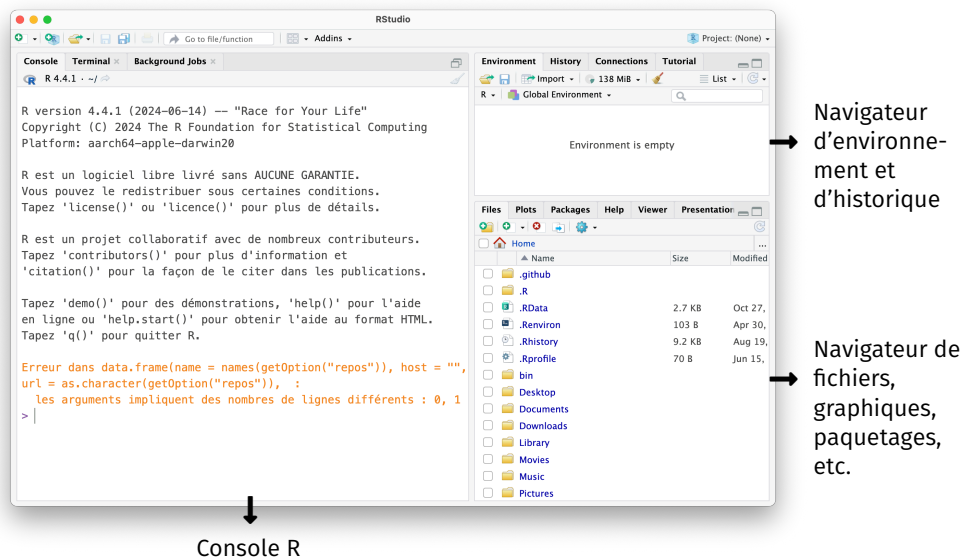


FIG. A.1 – Fenêtre de RStudio et trois de ses sous-fenêtres au lancement de l'application sous macOS. Sous Windows et Linux, la fenêtre comporte également une barre de menu.

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 A.7](#).

A.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 3.6](#)).
- 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 dont le répertoire de travail est celui 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.
- L'utilisation d'un projet permet également d'effectuer la gestion des versions du code avec Git ou Subversion directement depuis RStudio.

On trouvera plus d'information sur les projets dans l'aide en ligne de RStudio.

A.4 Commandes de base

Comme l'interface de RStudio respecte les standards modernes, je ne souligne 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, diffèrent légèrement. Je fournis ci-dessous les deux jeux, séparés par le symbole •.

- | | |
|-------------------------------|---|
| <code>Alt+- • \-</code> | insérer le symbole d'affectation <code><-</code> . |
| <code>Ctrl+Retour • ⌘↵</code> | 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.

↑ ↓	commande précédente suivante dans l'historique.
Ctrl+↑ • ⌘↑	afficher la fenêtre d'historique des commandes.



Dans les versions de RStudio antérieures à 1.3, le raccourci-clavier `⌘-` ne fonctionnait pas correctement sur les Mac munis d'un clavier canadien-français, et ce, à la console R, dans les fichiers de script ou les deux. La solution : associer un autre raccourci-clavier au symbole d'affectation. La description de la procédure de configuration est disponible en [vidéo](#)

A.5 Anatomie d'une session de travail (bis)

On reprend ici la description de la session de travail type présentée à la [section 3.9](#), mais en expliquant comment compléter chaque étape dans RStudio. Sont intercalés dans les instructions les raccourcis-clavier des commandes RStudio et les accès par les menus.

1. Lancer RStudio et ouvrir un nouveau fichier de script ou un fichier de script existant.

Ctrl+Shift+N • ⌘N	File New File R Script...
Ctrl+O • ⌘O	File Open File...

2. Changer le répertoire de travail de R pour le répertoire contenant le fichier de script.

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 console R afin d'essayer diverses expressions. On exécutera également des parties seulement du code se trouvant dans le fichier de script.

Ctrl+Retour • ⌘↵	Code Run Selected Line(s)
------------------	---------------------------

Ctrl+1 • ⌘1	View Move Focus to Source
Ctrl+2 • ⌘2	View Move Focus to Console

4. Sauvegarder le fichier de script.

Ctrl+S • ⌘S	File Save
-------------	-----------

(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.

Session|Save Workspace As...

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 A.7](#) explique comment configurer RStudio afin d'éviter de se faire poser la question à chaque fermeture de l'application.



La session de travail type avec RStudio ci-dessus fait aussi l'objet d'une [vidéo](#).

A.6 Terminal RStudio

Le terminal RStudio permet d'exécuter une ligne de commande du système d'exploitation à l'intérieur de l'interface familière de RStudio. C'est très utile pour exécuter des commande ou des scripts Bash.

Le terminal se présente dans un onglet à côté de celui de la console R. Si aucun onglet n'apparaît dans votre interface, vous pouvez démarrer un terminal depuis le menu **Tools|Terminal**.

Lorsqu'un fichier de script avec l'extension `.sh` est ouvert dans RStudio, le raccourci clavier **Ctrl+Retour • ⌘↵** évalue la ligne sous le curseur dans le terminal plutôt que dans la console R.



Pour utiliser une ligne de commande Bash dans RStudio sous Windows, il faut au préalable installer l'interpréteur Git Bash de [Git for Windows](#) sur le système et configurer l'éditeur tel qu'expliqué à la [section A.7](#).

A.7 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**

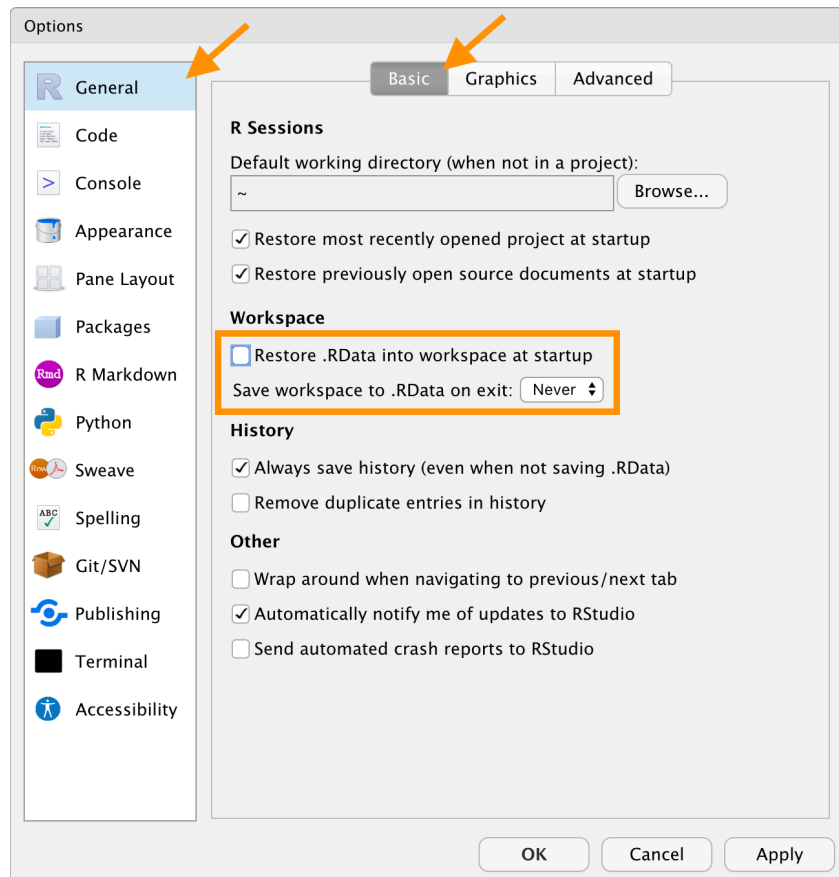


FIG. A.2 – Réglage de RStudio pour ne pas sauvegarder l'espace de travail de R au moment de quitter l'application

Options... sous Windows et Linux, et par le menu standard RStudio | Préférences (⌘,) sous macOS.

Je recommande d'effectuer les configurations suivantes :

1. Dans la catégorie General, désactiver l'option `Restore .RData into workspace at startup` et régler l'option `Save workspace to .RData on exit` à `Never`, tel qu'illustré à la figure A.2. Avec ces réglages, RStudio démarre toujours avec une séance de travail vierge et ne sauvegarde pas l'espace de travail de R à la fermeture de l'application.
2. Dans la catégorie Code et la sous-catégorie Editing, régler l'option `Tab width` à 4, tel qu'illustré à la figure A.3. Avec ce réglage, RStudio indentera par défaut le code de quatre caractères.
3. Toujours dans la catégorie Code, mais dans la sous-catégorie Saving, cocher les options `Ensure that source files end with newline` et `Strip trailing`

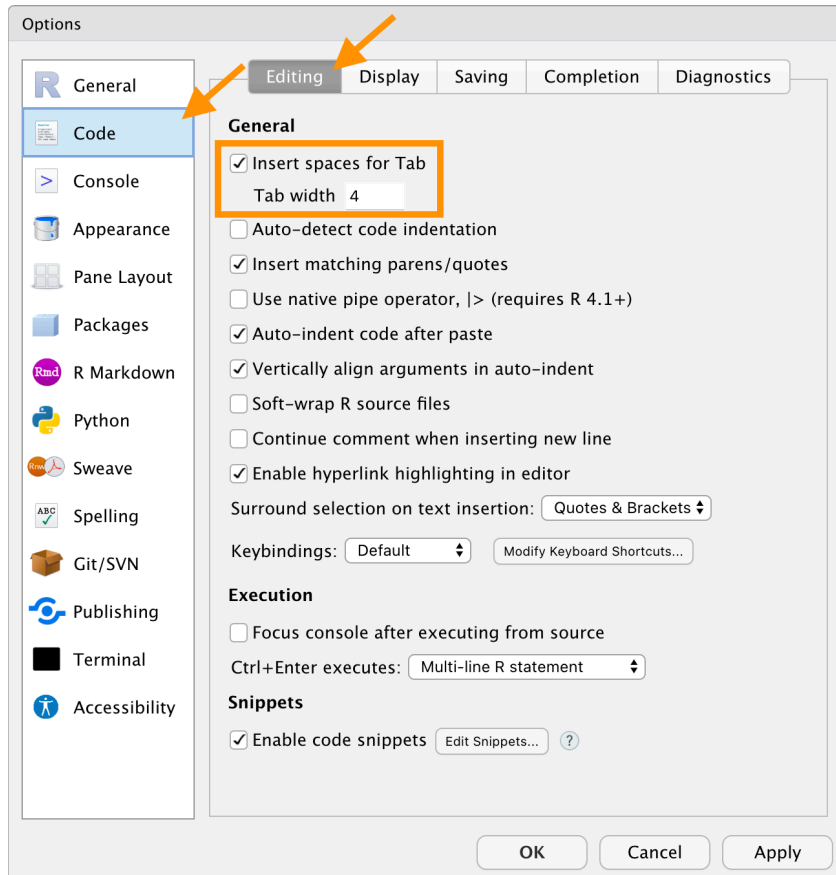


FIG. A.3 – Réglage de RStudio pour une indentation du code de quatre caractères

horizontal whitespace when saving, puis régler l'option Default text encoding à UTF-8, tel qu'illustré à la [figure A.4](#). RStudio sauvegardera ensuite les fichiers de script dans un format portable sur toutes les plateformes informatiques et supprimera automatiquement les espaces inutiles en fin de ligne.

4. (Windows seulement) Dans la catégorie Terminal, sélectionner Git Bash dans le menu déroulant de l'option New Terminals open with.

A.8 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.

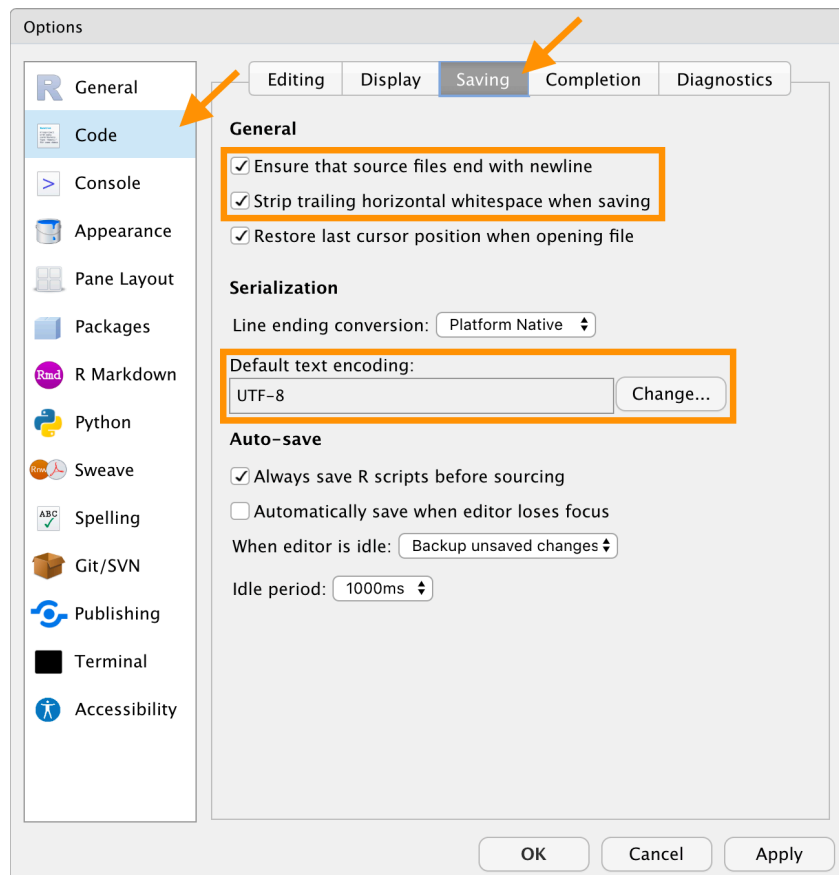


FIG. A.4 - Réglage de RStudio pour la sauvegarde des fichiers dans un format portable et sans espaces en fin de ligne

B 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 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.

B.1 Mise en contexte

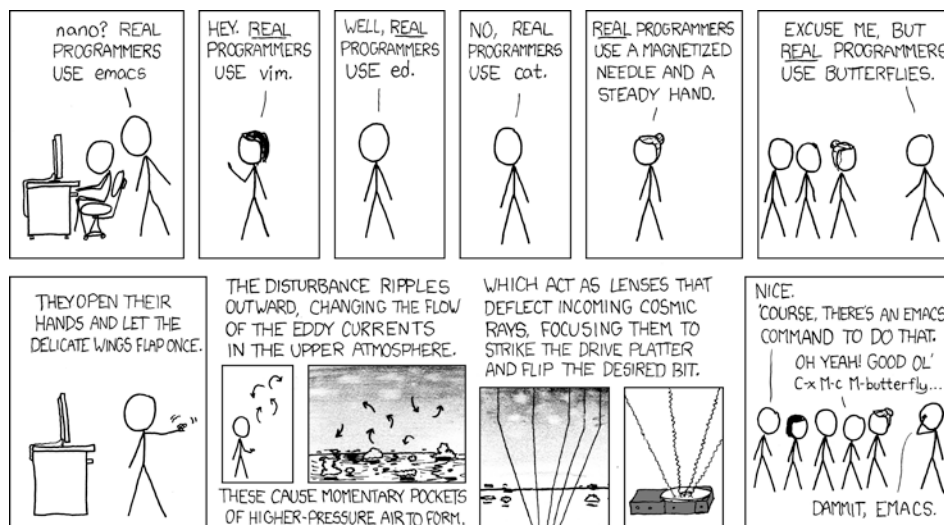
Emacs est le logiciel étendard du projet GNU (« *GNU is not Unix* »), dont le principal commanditaire est la *Free Software Foundation* (FSF) à l'origine de tout le mouvement du logiciel libre. Richard M. Stallman, président de la FSF et grand apôtre du libre, a écrit la première version de Emacs et 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 transparait 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.

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*, [Rossini et collab., 2024](#)). 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



Tiré de XKCD.com. La commande `M-x butterfly` existe vraiment dans Emacs... en référence à cette bande dessinée!

ESS est installé, le mode est activé automatiquement en ouvrant dans Emacs un fichier dont le nom se termine par l'extension `.R`.

B.2 Installation

Utilisateur invétéré de GNU Emacs, je prépare des **distributions**  pour Windows et macOS qui intègrent le mode ESS. Sous Linux, GNU Emacs et le mode ESS font normalement partie de toutes les distributions.

B.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 zones principales (voir la [figure B.1](#)) :

1. Tout au haut de la fenêtre (ou de l'écran sous macOS) se trouve l'habituelle barre des menus 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.

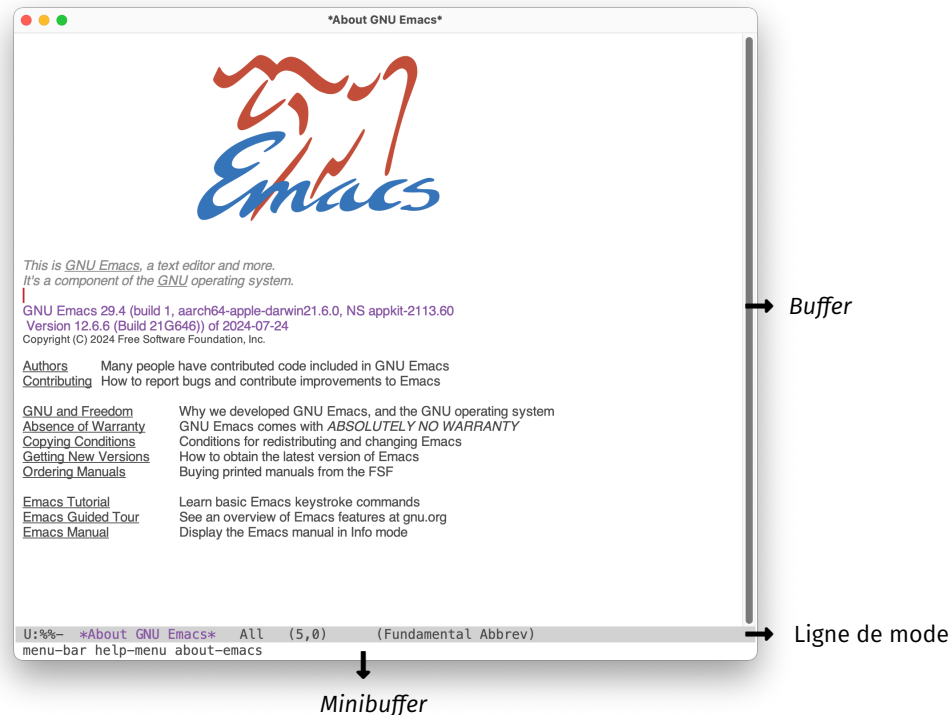


FIG. B.1 – Fenêtre de GNU Emacs et ses différentes parties au lancement de l'application sous macOS. La barre des menus se trouve au haut de l'écran dans macOS et à l'intérieur de la fenêtre de Emacs sous Windows et Linux.

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*.

B.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-clavier :

- 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 le bloc signalétique de la [page 326](#));
- ESC est la touche Échap (⌫) et est équivalente à Meta;



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 B.7](#)) :

```
(setq-default ns-command-modifier 'meta)
(setq-default ns-option-modifier 'none)
```

- SPC est la barre d'espace ;
- 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 qui peut être tapée dans le *minibuffer*. M-x démarre l'invite de commande.
- ▶ 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.

B.5 Commandes de base

Emacs compte une pléthore de commandes. Je me contenterai de mentionner les commandes les plus importantes regroupées par tâche.

B.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.

B.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.

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*).



Il n'existe pas de commande « nouveau fichier » dans Emacs. Pour créer un nouveau fichier, il suffit d'ouvrir un fichier qui n'existe pas déjà.

B.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.

B.5.4 Annuler, rechercher, remplacer

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*).

B.5.5 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.

B.5.6 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`).

B.5.7 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-= insérer le symbole d'affectation `=`, ou un autre symbole pour chaque pression additionnelle sur `=` (`ess-cycle-assign`).
- 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`).

B.5.8 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`).

B.5.9 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*).
- 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`).

B.6 Anatomie d'une session de travail (ter)

On reprend ici la description de la type présentée à la [section 3.9](#), mais en expliquant comment compléter chaque étape dans Emacs avec le mode ESS. Sont intercalés dans les instructions les raccourcis-clavier des commandes Emacs et les accès par les menus, le cas échéant.

1. Lancer Emacs et ouvrir un fichier de script.

`C-x C-f` `File|Open File...`

En spécifiant un nom de fichier qui n'existe pas déjà, on crée 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 de Emacs.

`M-x R` ↵

Emacs demandera de spécifier de répertoire de travail (*starting data directory*). Accepter la valeur par défaut 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` `ESS|Eval region | line`

`C-c C-c` `ESS|Eval region | func | para & step`

`C-c C-z` `ESS|Process|Switch to process buffer`

4. Sauvegarder le fichier de script.

`C-x C-s` `File|Save`

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.

`C-c C-q` `iESS|Quit`

Cette commande de ESS se charge de fermer tous les fichiers associés au processus R. On peut ensuite quitter Emacs en fermant l'application de la manière usuelle.



La session de travail type avec GNU Emacs ci-dessus fait aussi l'objet d'une [vidéo](#).

B.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 repose sur des commandes réunies dans un 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.



Visionnez la vidéo portant sur la [création de fichiers de configuration](#) pour Emacs et R.

B.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 `He lp`.

Autrement, on trouvera dans les sites respectifs des deux projets les manuels de [Emacs](#) et de [ESS](#).

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 `He lp` !

C Solutions des exercices

Chapitre 1

- 1.2 a) Une opération est formée d'un opérateur et de deux opérandes. En notation préfixée, les opérateurs s'entassent dans une pile jusqu'à ce qu'ils reçoivent deux opérandes. Lorsque c'est le cas, l'opérateur est retiré de la pile et le résultat de l'opération devient un opérande de l'opérateur plus bas dans la pile. En notation suffixée, ce sont plutôt les opérandes qui sont placés dans la pile.
- b) $\div \times A + B C D$ en notation préfixée; $A B C + \times D \div$ en notation suffixée.
- c) $A \times (B + C \div D)$

Chapitre 2

- 2.1 Voici des versions en langage naturel et en pseudocode.

Algorithme. Calculer les racines du polynôme $ax^2 + bx + c$ à partir des coefficients réels a , b et c .

1. Poser $d \leftarrow b^2 - 4ac$.
2. Si $d < 0$, retourner un résultat vide.
3. Si $d = 0$, retourner $-b/(2a)$.
4. Retourner $(-b - \sqrt{d})/(2a)$ et $(-b + \sqrt{d})/(2a)$. □

Algorithme. Calculer les racines du polynôme $ax^2 + bx + c$ à partir des coefficients réels a , b et c .

```
quadroots(réel a, réel b, réel c)
  d ← b^2 - 4 × a × c
  Si (d < 0)
    Retourner NULL
  Si (d = 0)
    Retourner -b/(2 × a)
  x ← (-b - sqrt(d))/(2 × a)
  y ← (-b + sqrt(d))/(2 × a)
```

Retourner x et y
Fin quadroots

□

- 2.2** Si la $\langle condition \rangle$ d'une instruction conditionnelle est vraie, la fonction se termine avec l'instruction « Retourner ». Toutes les instructions qui suivent la clause « Si $\langle condition \rangle$, alors ... » ne peuvent donc être évaluées que lorsque la $\langle condition \rangle$ est fausse. Inutile de les protéger par une clause « sinon ».

Par exemple, supposons qu'une fonction contient les instructions suivantes :

Si $\langle condition \rangle$
 $\langle conséquence \rangle$
sinon
 $\langle alternative \rangle$
 $\langle instruction \rangle$

Avec une telle structure, l' $\langle instruction \rangle$ sera évaluée en tout temps, que la $\langle condition \rangle$ soit vraie ou fausse. Or, si la $\langle conséquence \rangle$ contient « Retourner », tant l' $\langle alternative \rangle$ que l' $\langle instruction \rangle$ ne seront évaluées que lorsque la $\langle condition \rangle$ est fausse. La formulation ci-dessous est donc tout à fait équivalente.

Si $\langle condition \rangle$
 $\langle conséquence \rangle$
 $\langle alternative \rangle$
 $\langle instruction \rangle$

- 2.3** a) Les étapes à suivre sont, dans l'ordre : $t \leftarrow m, m \leftarrow n, n \leftarrow t$.
b) Une solution qui ne requiert que cinq remplacements est : $t \leftarrow a, a \leftarrow b, b \leftarrow c, c \leftarrow d, d \leftarrow t$.

- 2.4** Nous pouvons compléter les tables de vérité directement à partir des définitions des opérateurs ET et OU fournies à la [section 2.3](#) :

	p	q	$p \text{ ET } q$
	V	V	V
a)	V	F	F
	F	V	F
	F	F	F

	p	q	$p \text{ OU } q$
	V	V	V
b)	V	F	V
	F	V	V
	F	F	F

- 2.5** Tel que proposé dans l'énoncé, tous les résultats se démontrent à l'aide de tables de vérité.

	p	q	r	$p \wedge q$	$p \wedge r$	$q \vee r$	$p \wedge (q \vee r)$	$(p \wedge q) \vee (p \wedge r)$
a)	V	V	V	V	V	V	V	V
	V	V	F	V	F	V	V	V
	V	F	V	F	V	V	V	V
	V	F	F	F	F	F	F	F
	F	V	V	F	F	V	F	F
	F	V	F	F	F	V	F	F
	F	F	V	F	F	V	F	F
	F	F	F	F	F	F	F	F

	p	q	r	$p \vee q$	$p \vee r$	$q \wedge r$	$p \vee (q \wedge r)$	$(p \vee q) \wedge (p \vee r)$
b)	V	V	V	V	V	V	V	V
	V	V	F	V	V	F	V	V
	V	F	V	V	V	F	V	V
	V	F	F	V	V	F	V	V
	F	V	V	V	V	V	V	V
	F	V	F	V	F	F	F	F
	F	F	V	F	V	F	F	F
	F	F	F	F	F	F	F	F

	p	q	\bar{p}	\bar{q}	$\overline{p \wedge q}$	$\overline{p \vee q}$
c)	V	V	F	F	F	F
	V	F	F	V	V	V
	F	V	V	F	V	V
	F	F	V	V	V	V

	p	q	\bar{p}	\bar{q}	$\overline{p \vee q}$	$\overline{p \wedge q}$
d)	V	V	F	F	F	F
	V	F	F	V	F	F
	F	V	V	F	F	F
	F	F	V	V	V	V

	p	q	r	$p \wedge q$	$\bar{p} \wedge r$	$q \wedge r$	$(p \wedge q) \vee (\bar{p} \wedge r)$	$(p \wedge q) \vee (\bar{p} \wedge r) \vee (q \wedge r)$
e)	V	V	V	V	F	V	V	V
	V	V	F	V	F	F	V	V
	V	F	V	F	F	F	F	F
	V	F	F	F	F	F	F	F
	F	V	V	F	V	V	V	V
	F	V	F	F	F	F	F	F
	F	F	V	F	V	F	V	V
	F	F	F	F	F	F	F	F

- 2.6** Tel que mentionné à la [section 2.4](#), les algorithmes récursifs s'expriment généralement mieux en pseudocode. L'algorithme d'Euclide est particulièrement simple exprimé ainsi.

Algorithme. Calculer le plus grand commun diviseur de deux entiers positifs m et n avec, sans perte de généralité, $n < m$.

```
PGCD(entier m, entier n)
  Si (n = 0) retourner m
  Retourner PGCD(n, m mod n)
Fin PGCD
```

□

- 2.7** a) L'algorithme ci-dessous permet de calculer les nombres de Fibonacci de manière récursive.

Algorithme (Suite de Fibonacci). Calculer la valeur $f(n)$ de la suite de Fibonacci pour un entier non négatif n .

```
Fib(entier n)
  Si (n = 0), retourner 0
  Si (n = 1), retourner 1
  Retourner Fib(n - 2) + Fib(n - 1)
Fin Fib
```

□

- b) Les 10 premiers nombres de Fibonacci sont : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.
 c) En décomposant la méthode de calcul des nombres de Fibonacci employée par l'algorithme de la partie a), on obtient :

$$\begin{aligned}
 f(5) &= f(4) + f(3) \\
 &= (f(3) + f(2)) + (f(2) + f(1)) \\
 &= ((f(2) + f(1)) + (f(1) + f(0))) \\
 &\quad + ((f(1) + f(0)) + f(1))
 \end{aligned}$$

$$\begin{aligned}
 &= A(0, A(0, 256)) \\
 &= A(0, 512) \\
 &= 1\,024.
 \end{aligned}$$

- 2.9 a) Les trois algorithmes sont itératifs. L'algorithme A effectue sa boucle n fois ; l'algorithme B, $n/2$ fois ; l'algorithme C, $\lfloor \sqrt{n} \rfloor$ fois. La performance de chacun est donc $O(n)$, $O(n/2)$ et $O(\sqrt{n})$, dans l'ordre.
- b) Si l'on suit à la lettre les étapes des algorithmes, on obtient les affichages suivants :

Algorithme A

$n = 12$: 1 2 3 4 6 12

$n = 16$: 1 2 4 8 16

Algorithmes B et C

$n = 12$: 1 12 2 6 3 4

$n = 16$: 1 16 2 8 4 4

- c) Les algorithmes B et C comportent deux erreurs (ou anomalies) : les diviseurs ne s'affichent pas en ordre croissant ; la valeur de \sqrt{n} s'affiche deux fois lorsque n est un carré.

Corrigeons d'abord l'affichage en ordre croissant. Une solution simple consiste à afficher uniquement les diviseurs i sans leur contrepartie n/i , à entreposer les contreparties dans un « vecteur », puis, avant de quitter la procédure, à afficher le contenu du vecteur de la dernière valeur à la première.

Algorithme C'. Afficher les diviseurs d'un nombre naturel n .

1. Poser $i \leftarrow 1$ et v un vecteur vide.
2. Si $n \bmod i = 0$, afficher la valeur i et ajouter n/i au vecteur v : $v \leftarrow v, n/i$.
3. Incrémenter i : $i \leftarrow i + 1$.
4. Si $i \leq \lfloor \sqrt{n} \rfloor$, retourner à l'étape 2.
5. Afficher les valeurs de v de la dernière à la première. □

Avec $n = 12$, l'algorithme révisé affichera les valeurs

1 2 3,

entreposera les valeurs (12, 6, 4) dans v , puis affichera les valeurs

4 6 12.

Avec $n = 16$, la valeur 4 s'affichera cependant toujours deux fois. Le second correctif à apporter à l'algorithme consiste à entreposer la valeur n/i seulement lorsqu'elle n'est pas égale à i .

Algorithme C''. Afficher les diviseurs d'un nombre naturel n .

1. Poser $i \leftarrow 1$ et v un vecteur vide.
 2. Si $n \bmod i = 0$:
 - 2.1 afficher la valeur i ;
 - 2.2 si $n/i \neq i$, ajouter n/i au vecteur v : $v \leftarrow v, n/i$.
 3. Incrémenter i : $i \leftarrow i + 1$.
 4. Si $i \leq \lfloor \sqrt{n} \rfloor$, retourner à l'étape 2.
 5. Afficher les valeurs de v de la dernière à la première. □
- 2.10** a) Si le cube a n unités de côté, sa surface — ou le nombre de carrés — est $6n^2$. La performance de l'algorithme est donc $O(n^2)$. Nous pouvons obtenir cette réponse simplement en remarquant que l'algorithme doit calculer une aire.
- b) Chaque face d'un cube $n \times n \times n$ compte 4 coins et 4 côtés chacun d'une longueur $n - 2$ (en excluant les coins). Par conséquent, le nombre total de carrés sur une face est $4 + 4(n - 2) = 4n - 4$ et le nombre total de carrés est $6(4n - 4) = 24n - 24$. La performance de l'algorithme est donc $O(n)$. Nous pouvons aussi obtenir cette réponse en remarquant que l'algorithme doit en définitive calculer une longueur.
- 2.11** L'étage k d'une pyramide de n étage compte $k(k + 1)/2$ cubes. Le nombre total de cubes est donc égal à

$$\sum_{k=1}^n \frac{k(k+1)}{2}.$$

À l'aide des identités

$$\begin{aligned} \sum_{k=1}^n k &= \frac{n(n+1)}{2} \\ \sum_{k=1}^n k^2 &= \frac{n(n+1)(2n+1)}{6}, \end{aligned}$$

on obtient avec un peu d'algèbre que

$$\sum_{k=1}^n \frac{k(k+1)}{2} = \frac{n^3 + 3n^2 + 2n}{6}.$$

La performance de l'algorithme est donc $O(n^3)$, une réponse intuitivement évidente dans la mesure où l'algorithme calcule un volume.

- 2.12** Vous pouvez tracer un graphique regroupant toutes les fonctions dans R avec les expressions suivantes.

```
> plot(0:20, 0:20, type = "n")
> curve(log(x), add = TRUE, col = 1)
> curve(sqrt(x), add = TRUE, col = 2)
> curve(I(x), add = TRUE, col = 3)
> curve(x * log(x), add = TRUE, col = 4)
> curve(x^2, add = TRUE, col = 5)
> curve(factorial(x), add = TRUE, col = 6)
```

Chapitre 4

4.1 a) Produit élément par élément entre deux vecteurs.

```
[1] 0 2 0 4 0
```

b) Comparaison élément par élément, le second vecteur étant utilisé quatre fois.

```
[1] FALSE TRUE FALSE TRUE
```

c) Comparaison élément par élément, le second vecteur étant utilisé deux fois.

```
[1] TRUE FALSE TRUE TRUE
```

d) Élévation à une puissance élément par élément, le premier vecteur étant utilisé deux fois.

```
[1] -1.0000 4.0000 4.0000 1.0000 0.5000 0.0625
```

e) « Et » logique élément par élément.

```
[1] TRUE FALSE FALSE
```

f) Attention à la priorité des opérations! En vertu des priorités énoncées au [tableau 4.1](#), les opérations sont effectuées dans cet ordre : inégalité (<), négation logique (!), « ou » logique (|).

```
[1] TRUE FALSE FALSE
```

g) Encore ici, la génération d'une suite a priorité sur la division.

```
[1] -Inf NaN Inf
```

4.2 On crée d'abord le vecteur avec

```
> x <- c(18, 11, 10, 2, 19, 9, 12, 15, 13, 12, 1, 6)
```

a) Indicage avec un seul entier positif pour extraire une donnée.


```
| > x[2]
```

- b) Indixage avec un vecteur d'entiers positifs pour extraire plusieurs données.

```
| > x[1:5]
```

- c) Indixage avec un vecteur booléen pour extraire par critère.

```
| > x[x > 14]
```

- d) Indixage avec un vecteur d'entiers négatifs pour laisser tomber des données.

```
| > x[-c(6, 7, 12)]
```

- 4.3** a) La somme de deux vecteurs — qui s'effectue élément par élément — fait partie intégrante du langage R.

```
| > 3 * (x + y)
```

- b) La fonction `sum` additionne tous les éléments d'un vecteur.

```
| > sum(x)/length(x)
```

- c) La fonction `prod` multiplie entre eux tous les éléments d'un vecteur.

```
| > prod(x)^(1/length(x))
```

- d) Le calcul de la moyenne harmonique repose sur la fonction `sum`.

```
| > length(x)/sum(1/x)
```

- e) Le produit scalaire est la somme des produits élément par élément de deux vecteurs.

```
| > sum(x * y)
```

- f) La norme 1 est la somme des différences en valeurs absolues élément par élément de deux vecteurs.

```
| > sum(abs(x - y))
```

- g) La norme « infini » est la plus grande des différences en valeurs absolues élément par élément de deux vecteurs.

```
| > max(abs(x - y))
```

- 4.4** b) On a

```
| > amean <- function(x) sum(x)/length(x)
```

- c) On a

```
| > gmean <- function(x) prod(x)^(1/length(x))
```

d) On a

```
| > hmean <- function(x) length(x)/sum(1/x)
```

e) On a

```
| > pscal <- function(x, y) sum(x * y)
```

f) On a

```
| > norm1 <- function(x, y) sum(abs(x - y))
```

g) On a

```
| > normINF <- function(x, y) max(abs(x - y))
```

4.5 La solution repose sur l'indigage pour effectuer la première étape de l'[algorithme 2.6a](#) et sur l'utilisation de la fonction interne `mean` pour la seconde étape.

```
| > emrl <- function(x, d) mean(x[x > d]) - d
```

4.6 a) Les fonctions utilisent la structure `if ... else ...` qui n'est pas vectorielle.

b) Dans les mises en œuvre des algorithmes [2.3](#) et [2.3a](#), le résultat de la fonction est directement le résultat d'une expression conditionnelle. La mise en œuvre de l'[algorithme 2.3b](#), de son côté, contient une expression après la clause `if`. Si nous ne forçons pas la sortie de la fonction avec `return` à l'intérieur de la condition, l'évaluation de la fonction se poursuivrait et le résultat ne serait pas celui attendu.

```
| > abs <- function(x)
+ {
+   if (x < 0)
+     -x
+   x
+ }
| > abs(-5)
| [1] -5
```

4.7 Les deux algorithmes étant récursifs, les fonctions le sont aussi. Tel qu'expliqué à la [section 4.6](#), il est préférable d'utiliser `Recall` pour invoquer la fonction à l'intérieur d'elle-même sans avoir à se soucier de son nom.

Tout d'abord, la mise en œuvre de l'[algorithme 2.7](#).

```
> pow <- function(b, n)
+ {
+   if (n == 0)
+     return(1)
+   b * Recall(b, n - 1)
+ }
```

Ensuite, celle de l'[algorithme 2.7a](#).

```
> pow2 <- function(b, n)
+ {
+   if (n == 0)
+     return(1)
+   if (n %% 2)
+     b * Recall(b, n - 1)
+   else
+     Recall(b, n/2)^2
+ }
```

4.8 a) L'algorithme demandé est une variante de l'algorithme d'Euclide.

1. Poser d égal au plus grand commun diviseur de m et n .
2. Retourner mn/d .

b) L'algorithme nécessite de calculer le PGCD, chose que l'on peut faire de manière récursive avec l'algorithme d'Euclide. C'est là une bonne occasion d'effectuer une mise en œuvre PGCD de l'[algorithme 2.1](#). Débutons avec cette fonction.

```
> PGCD <- function(m, n)
+ {
+   if (n == 0) return(m)
+   Recall(n, m %% n)
+ }
```

La fonction PPCM est ensuite très simple à programmer. L'essentiel du code est consacré à la vérification des cas triviaux pour éviter les calculs inutiles.

```
> PPCM <- function(m, n)
+ {
+   if (any(c(m, n) == 0))
+     return(0)
+   if (any(c(m, n) == 1))
+     return(max(c(m, n)))
+   if (m == n)
```

```
+      return(m)
+      (m * n)/PGCD(m, n)
+ }
```

- 4.9** a) $x = 2, y = 3, z = 4$
 b) $x = 2, y = 4, z = 3$
 c) $x = 2, y = \text{NULL}, z = 3$
 d) $x = 4, y = 3, z = 2$
 e) $x = 0, y = \text{NULL}, z = 2$
 f) $x = 3, y = 4, z = 2$
 g) **C'est un piège!** ☒ Cet appel n'est pas valide puisque l'argument z est manquant.

4.10 Cet exercice fait quelque peu appel à votre intuition pour déterminer les valeurs des objets x et y à l'intérieur de la fonction g . Sans doute avez-vous utilisé les valeurs déjà définies dans la fonction f , ce qui est juste en vertu des règles de portée lexicale de R. Celles-ci font l'objet de la [section 12.3](#). Dans certains langages de programmation, les objets seraient indéfinis et les appels de fonction résulteraient tous en une erreur.

- a) Dans l'appel $f(1, 1)$, les valeurs de x et y sont égales à 1 dans la fonction f et l'argument a de la fonction g est égal à 2.

```
| [1] 2
```

- b) Dans l'appel $f(f(1, 1), 1)$, l'argument x de la fonction f externe est égal au résultat en a), soit 2, et l'argument y est égal à 1. Le reste des calculs s'effectue comme en a).

```
| [1] 9
```

- c) Cette fois, l'argument x de la fonction f externe est égal à 1 et l'argument y est égal au résultat en b).

```
| [1] -70
```

4.11 a) Vecteur des nombres 0 et 6, répété trois fois.

```
| > rep(c(0, 6), 3)
```

- b) Suite de 1 à 10 par bonds de 3.

```
| > seq(1, 10, by = 3)
```

- c) Suite 1, 2, 3 répétée quatre fois.

```
| > rep(1:3, 4)
```

- d) Suite 1, 2, 3 dont les éléments sont répétés une, deux et trois fois, dans l'ordre.

```
| > rep(1:3, 1:3)
```

- e) Suite 1, 2, 3 dont les éléments sont répétés trois, deux et une fois, dans l'ordre.

```
| > rep(1:3, 3:1)
```

- f) Suite de trois nombres équidistants entre 1 à 10, inclusivement.

```
| > seq(1, 10, length = 3)
```

- g) Suite 1, 2, 3 dont les éléments sont chacun répétés quatre fois.

```
| > rep(1:3, rep(4,3))
```

- 4.12** a) Générer les entiers de 11 à 20 et diviser ensuite par 10.

```
| > 11:20/10
```

- b) Générer la suite 0, 2, 4, ..., 18 en multipliant par 2 la suite 0, 1, 2, ..., 9, puis ajouter 1.

```
| > 2 * 0:9 + 1
```

- c) Répéter la suite -2, -1, 0, 1, 2 deux fois.

```
| > rep(-2:2, 2)
```

- d) Répéter deux fois chaque élément de la suite -2, -1, 0, 1, 2.

```
| > rep(-2:2, each = 2)
```

- e) Générer la suite 1, 2, ..., 10 et multiplier par 10.

```
| > 10 * 1:10
```

- 4.13** Il s'agit de calculer le produit cumulatif des nombres de 1 à 10.

```
| > cumprod(1:10)
```

- 4.14** $x == (x \% y) + y * (x \% \% y)$

- 4.15** a) Extraction par position des cinq premiers éléments du vecteur.

```
| > x[1:5]  
| > head(x, 5)
```

- b) La fonction max retourne la plus grande valeur d'un vecteur.

```
| > max(x)
```

- c) Il s'agit d'extraire les cinq premiers éléments du vecteur et de calculer la moyenne des valeurs obtenues.

```
> mean(x[1:5])
> mean(head(x, 5))
```

- d) Il s'agit d'extraire les cinq derniers éléments du vecteur et de calculer la moyenne des valeurs obtenues. Deux solutions : une qui implique de connaître exactement la longueur du vecteur et l'autre, préférable, qui demeure valable peu importe la longueur du vecteur.

```
> mean(x[16:20])
> mean(tail(x, 5))
```

Chapitre 5

- 5.1 a) Vecteur de trois chaînes de caractères, peu importe leur longueur.

```
[1] 3
```

- b) Longueur du vecteur.

```
[1] 3
```

- c) Concaténer NULL à un vecteur n'ajoute rien (ou du vide, qui n'a pas de longueur).

```
[1] 4
```

- d) La valeur booléenne est convertie en nombre réel.

```
[1] "numeric"
```

- e) Le nombre réel est converti en chaîne de caractères.

```
[1] "character"
```

- f) Nombre de données manquantes dans le vecteur.

```
[1] 2
```

- g) Deuxième colonne de la matrice $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$.

```
[1] 3 4
```

- h) Jour du mois de la date.

```
[1] 9
```

- i) Liste trois éléments dont le troisième élément est une sous-liste de deux éléments.

```
[[1]]
[1] NA

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

[[3]]
[[3]][[1]]
[1] NA

[[3]][[2]]
[1]  1  2  3  4  5  6  7  8  9 10
```

- j) Première colonne du tableau de données, sous forme de vecteur.

```
[1] 1 2 3
```

- 5.2 a) Extraction d'un élément unique à l'aide de d'un indice de ligne et d'un indice de colonne.

```
> x[4, 3]
```

- b) Extraction d'une ligne complète à l'aide d'un indice de ligne.

```
> x[6, ]
```

- c) Extraction de deux colonnes complètes à l'aide de deux indices de colonne.

```
> x[, c(1, 4)]
```

- d) Extraction de lignes par critère, critère lui-même obtenu par un calcul sur les éléments de la première colonne.

```
> x[x[, 1] > 50, ]
```

- 5.3 a) $(j - 1) * I + i$

- b) $((k - 1) * J + j - 1) * I + i$

- 5.4 a) Sommes par ligne.

```
> rowSums(m)
```

- b) Moyennes par colonne.

```
> colMeans(m)
```

- c) Indichage de la matrice d'origine et calcul du maximum.

```
> max(m[1:3, 1:3])
```

- d) Extraction par critère, celui-ci étant obtenu en calculant les moyennes par ligne.

```
> m[rowMeans(m) > 7, ]
```

- 5.5 La solution repose sur l'utilisation judicieuse de la fonction `order`.

```
> sort.matrix <- function(m, decreasing = FALSE)
+   m[order(m[, 1], decreasing = decreasing), ]
```

- 5.6 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) La fonction `names` permet d'extraire les étiquettes d'un objet.

```
> names(x)
```

- c) Extraire le quatrième élément de la liste avant d'utiliser `mode` et `length` pour obtenir son mode et sa longueur.

```
> mode(x$test)
> length(x$test)
```

- d) Extraire le deuxième élément de la liste avant d'utiliser `dim` pour obtenir ses dimensions.

```
> dim(x$data)
```

- e) Le truc ici consiste à indiquer deux fois : d'abord avec des crochets doubles pour extraire le second élément de la liste, puis avec des crochets simples (puisque le résultat de la première extraction est un vecteur) pour extraire les deuxième et troisième éléments.

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

- f) Il faut utiliser des crochets doubles.

```
> x[[3]] <- 3:8
```

- 5.7 Calculer une quantité pour toutes les combinaisons de deux vecteurs : c'est un travail pour `outer`. La première solution, ci-dessous, n'a recours à `outer` que pour le calcul de $(1 + i)^{-n}$ pour toutes les combinaisons de i et de n . Les calculs de $1 - (1 + i)^{-n}$, puis de $(1 - (1 + i)^{-n})/i$, suivent par arithmétique vectorielle et recyclage. Le division de la matrice par un vecteur donne le bon résultat, ici, car les taux d'intérêt varient par colonne.


```
> n <- 1:10
> i <- seq(0.05, 0.1, by = 0.01)
> (1 - outer((1 + i), -n, "^"))/i
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.9523810	1.859410	2.723248	3.545951	4.329477
[2,]	0.9433962	1.833393	2.673012	3.465106	4.212364
[3,]	0.9345794	1.808018	2.624316	3.387211	4.100197
[4,]	0.9259259	1.783265	2.577097	3.312127	3.992710
[5,]	0.9174312	1.759111	2.531295	3.239720	3.889651
[6,]	0.9090909	1.735537	2.486852	3.169865	3.790787

	[,6]	[,7]	[,8]	[,9]	[,10]
[1,]	5.075692	5.786373	6.463213	7.107822	7.721735
[2,]	4.917324	5.582381	6.209794	6.801692	7.360087
[3,]	4.766540	5.389289	5.971299	6.515232	7.023582
[4,]	4.622880	5.206370	5.746639	6.246888	6.710081
[5,]	4.485919	5.032953	5.534819	5.995247	6.417658
[6,]	4.355261	4.868419	5.334926	5.759024	6.144567

Une seconde version tire moins partie des propriétés vectorielles de R, mais elle est sans doute plus intuitive : tout le calcul est effectué par une fonction (anonyme) passée à `outer`. Le choix inusité de l'ordre des arguments vise simplement à obtenir le même résultat que ci-dessus.

```
> outer(i, n, function(i, n) (1 - (1 + i)^(-n))/i)
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.9523810	1.859410	2.723248	3.545951	4.329477
[2,]	0.9433962	1.833393	2.673012	3.465106	4.212364
[3,]	0.9345794	1.808018	2.624316	3.387211	4.100197
[4,]	0.9259259	1.783265	2.577097	3.312127	3.992710
[5,]	0.9174312	1.759111	2.531295	3.239720	3.889651
[6,]	0.9090909	1.735537	2.486852	3.169865	3.790787

	[,6]	[,7]	[,8]	[,9]	[,10]
[1,]	5.075692	5.786373	6.463213	7.107822	7.721735
[2,]	4.917324	5.582381	6.209794	6.801692	7.360087
[3,]	4.766540	5.389289	5.971299	6.515232	7.023582
[4,]	4.622880	5.206370	5.746639	6.246888	6.710081
[5,]	4.485919	5.032953	5.534819	5.995247	6.417658
[6,]	4.355261	4.868419	5.334926	5.759024	6.144567

5.8 On a

```
> 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)
[1] 11.26506
```

5.9 Soit X_{ij} et w_{ij} des matrices, et X_{ijk} et w_{ijk} des tableaux à trois dimensions.

a) Il s'agit de calculer les moyennes pondérées par lignes.

```
> apply(Xij * wij, 1, sum)/apply(wij, 1, sum)
```

ou

```
> rowSums(Xij * wij)/rowSums(wij)
```

b) Il s'agit de calculer les moyennes pondérées par colonnes.

```
> apply(Xij * wij, 2, sum)/apply(wij, 2, sum)
```

ou

```
> colSums(Xij * wij)/colSums(wij)
```

c) Ici, le résultat est la moyenne pondérée de toutes les données.

```
> sum(Xij * wij)/sum(wij)
```

d) Il s'agit de calculer les moyennes pondérées par carottes horizontales.

```
> apply(Xijk * wijk, c(1, 2), sum) /
+   apply(wijk, c(1, 2), sum)
```

e) Il s'agit de calculer les moyennes pondérées par tranches horizontales.

```
> apply(Xijk * wijk, 1, sum)/apply(wijk, 1, sum)
```

f) Il s'agit de calculer les moyennes pondérées par tranches verticales.

```
> apply(Xijk * wijk, 2, sum)/apply(wijk, 2, sum)
```

g) Il s'agit à nouveau de la moyenne pondérée de toutes les données.

```
> sum(Xijk * wijk)/sum(wijk)
```

5.10 a) Générer séparément les suites de 0 jusqu'à 1, 2, ..., 10, puis les placer bout à bout.

```
> unlist(lapply(0:10, seq, from = 0))
[1] 0 0 1 0 1 2 0 1 2 3 0 1 2 3 4 0 1
[18] 2 3 4 5 0 1 2 3 4 5 6 0 1 2 3 4 5
[35] 6 7 0 1 2 3 4 5 6 7 8 0 1 2 3 4 5
[52] 6 7 8 9 0 1 2 3 4 5 6 7 8 9 10
```

b) Générer séparément les suites de 1, 2, ..., 10 0 jusqu'à 10, puis les placer bout à bout.

```
> unlist(lapply(1:10, seq, from = 10))
[1] 10 9 8 7 6 5 4 3 2 1 10 9 8 7 6 5 4
[18] 3 2 10 9 8 7 6 5 4 3 10 9 8 7 6 5 4
[35] 10 9 8 7 6 5 10 9 8 7 6 10 9 8 7 10 9
[52] 8 10 9 10
```

- c) Générer séparément les suites de 10, 9, ..., 1 0 jusqu'à 1, puis les placer bout à bout.

```
> unlist(lapply(10:1, seq, to = 1))
[1] 10 9 8 7 6 5 4 3 2 1 9 8 7 6 5 4 3
[18] 2 1 8 7 6 5 4 3 2 1 7 6 5 4 3 2 1
[35] 6 5 4 3 2 1 5 4 3 2 1 4 3 2 1 3 2
[52] 1 2 1 1
```

- 5.11 a) La fonction `rpareto` n'est pas vectorielle pour son argument `n`. Il faut avoir recours à une fonction d'application pour simuler plusieurs échantillons de taille différentes. La fonction à utiliser est `lapply` puisqu'elle permet d'itérer sur un vecteur de tailles d'échantillons et qu'elle retourne son résultat dans une liste, tel que demandé.

```
> x <- lapply(seq(100, 300, by = 50), rpareto,
+           shape = 2, scale = 5000)
```

- b) On attribue des étiquettes à la liste avec `names`.

```
> names(x) <- paste("sample", 1:5, sep = "")
```

- c) Il faut encore utiliser une fonction d'application. Pour retourner le résultat directement sous forme de vecteur, c'est `sapply` qu'il nous faut.

```
> sapply(x, mean)
```

- d) Définissons d'abord une fonction pour calculer la fonction de répartition de la loi de Pareto. Celle-ci est vectorielle pour son argument `q`.

```
> ppareto <- function(q, shape, scale)
+   1 - (scale/(q + scale))^shape
```

Nous pouvons ensuite effectuer le calcul demandé, mais nous avons besoin d'une fonction anonyme pour aussi trier les résultats.

```
> lapply(x, function(x) sort(ppareto(x, 2, 5000)))
```

- e) Pour utiliser un opérateur arithmétique comme fonction dans une fonction d'application, il faut le placer entre guillemets.

```
> lapply(x, "+", 1000)
```

- 5.12** La solution la plus simple consiste à placer l'expression qui effectue le calcul pour un seuil à l'intérieur d'une fonction d'application `sapply` qui, elle, se chargera de faire le calcul pour chacun des seuils. Il s'agit d'une belle occasion pour utiliser une fonction anonyme.

```
> emrl <- function(x, d)
+   sapply(d, function(y) mean(x[x > y])) - d
```

Chapitre 6

- 6.1** La présentation correcte comporte des espaces autour de tous les opérateurs, une indentation de quatre (4) caractères et des accolades ouvrante et fermante placées sur leur propre ligne.

```
f <- function(x)
{
  if (all(x >= 0) || all(x <= 0))
  {
    stop("all x are the same sign")
  }
  if (sum(diff(sign(x[x != 0]))) != 0) > 1)
    warning("more than one sign change")
  r <- polyroot(x)
  i <- 1/Re(r)[abs(Im(r)) < 1.5e-8] - 1
  i[i > -1]
}
```

- 6.2** Nous devons effectuer les vérifications suivantes :

1. erreur si tous les éléments du vecteur en argument sont positifs;
2. erreur si tous les éléments du vecteur en argument sont négatifs;
3. avertissement s'il y a plus d'un changement de signe;
4. réponses adéquates pour les deux cas fournis dans l'énoncé.

Le code ci-dessous permet d'effectuer les tests.

```
tools::assertError(f(c(4, 3, 0, 6, 21)))
tools::assertError(f(-c(4, 3, 0, 6, 21)))
tools::assertWarning(f(c(4, -3, 0, 21, -6)))
stopifnot(exprs = {
  all.equal(0.128564783114, f(c(-10, -7, 12, 0, 11)))
  all.equal(0.785366742540, f(c(-10, 12, 7, 0, 11)))
})
```

Chapitre 7

Nous allons tester nos fonctions avec des vecteurs aléatoires obtenus avec la fonction `sample` qui tire un échantillon aléatoire parmi des valeurs avec ou sans remise :

```
sample(x, size, replace = FALSE)
```

- `x` est un vecteur parmi lequel sera tiré l'échantillon;
- `size` est la taille de l'échantillon;
- `replace` est une valeur booléenne qui indique si l'échantillonnage doit s'effectuer avec ou sans remise.

7.1 La version de l'[algorithme 7.2](#) du tri par sélection déplace graduellement vers la droite la plus grande valeur du vecteur, la deuxième plus grande valeur, etc. Deux boucles sont donc nécessaires : une pour passer à travers les indices $n, n-1, \dots, 2$ du vecteur et une autre pour, à chaque itération de la première boucle, trouver le maximum.

Dans la majorité des langages de programmation, l'échange de deux éléments d'un vecteur nécessite une variable tampon, comme ceci :

```
> tmp <- x[i]
> x[i] <- x[j]
> x[j] <- tmp
```

Vous remarquerez comment, dans la mise en œuvre en R ci-dessous, l'échange est simple à effectuer avec l'indilage (le langage se charge des variables tampons). Remarquez également que la génération de la suite $j, j-1, \dots, 1$ avec `j:1` dans la deuxième boucle est sécuritaire car, dans le contexte, la valeur du compteur `j` est toujours supérieure à zéro.

```
> ssort <- function(x)
+ {
+   for (j in seq.int(to = 2, by = -1,
+                     length.out = length(x) - 1))
+   {
+     i.max <- j
+     for (i in j:1)
+     {
+       if (x[i] > x[i.max])
+         i.max <- i
+     }
+     x[c(j, i.max)] <- x[c(i.max, j)]
+   }
+   x
+ }
```

```
+ }
> (x <- sample(0:10, 7, replace = TRUE))
[1] 0 0 6 1 0 9 1
> ssort(x)
[1] 0 0 0 1 1 6 9
```

7.2 L'algorithme de tri à bulles fait graduellement « remonter à la surface » les plus grandes valeurs du vecteur. Il faut comparer deux valeurs adjacentes tant que le vecteur n'est pas trié. Par nature, l'algorithme requiert deux boucles : une pour poursuivre le travail tant et aussi longtemps que le vecteur n'est pas trié et une pour comparer toutes les paires de valeurs. Puisque la première boucle doit être exécutée au moins une fois (pour vérifier si le vecteur ne serait pas déjà trié), mais que l'on ignore le nombre de répétitions, le meilleur choix de boucle est `repeat`.

```
> bsort <- function(x)
+ {
+   max.unsorted <- length(x)
+   repeat
+   {
+     t <- 0
+     for (j in seq_len(max.unsorted - 1))
+     {
+       if (x[j] > x[j + 1])
+       {
+         x[c(j, j + 1)] <- x[c(j + 1, j)]
+         t <- j
+       }
+     }
+     if (t > 0)
+       max.unsorted <- t
+     else
+       break
+   }
+   x
+ }
> (x <- sample(0:10, 7, replace = TRUE))
[1] 5 1 4 1 9 1 7
> bsort(x)
[1] 1 1 1 4 5 7 9
```

7.3 L'algorithme de tri par dénombrement retourne le vecteur du nombre de va-

leurs inférieures à chacune des valeurs du vecteur à trier. Deux boucles à dénombrement sont nécessaires. La mise en œuvre ci-dessous retourne un vecteur d'entiers (d'où l'utilisation de l'entier 1L dans les additions).

```
> csort <- function(x)
+ {
+   count <- integer(length(x))
+   for (i in seq.int(to = 2, by = -1,
+                     length.out = length(x) - 1))
+   {
+     for (j in (i - 1):1)
+     {
+       if (x[i] < x[j])
+         count[j] <- count[j] + 1L
+       else
+         count[i] <- count[i] + 1L
+     }
+   }
+   count
+ }
> (x <- sample(0:10, 7, replace = TRUE))
[1] 5 7 10 0 9 2 10
> csort(x)
[1] 2 3 5 0 4 1 6
```

- 7.4 a) Tel que mentionné dans l'[algorithme 7.4](#), les résultats du tri par dénombrement sont les indices des éléments du vecteur à trier une fois qu'on leur additionne 1. C'est précisément le résultat de la fonction `rank`. Lorsque des valeurs sont répétées dans le vecteur, l'algorithme trie les éléments dans l'ordre où ils apparaissent dans le vecteur. Pour obtenir ce comportement avec la fonction `rank`, il faut préciser `ties.method = "first"`.

```
> (x <- sample(0:10, 7, replace = TRUE))
[1] 1 10 4 1 9 9 7
> csort(x) + 1
[1] 1 7 3 2 5 6 4
> rank(x, ties.method = "first")
[1] 1 7 3 2 5 6 4
```

- b) Le code informatique de la [section 7.9](#) montre que l'expression

```
> x <- x[order(x)]
```

trie le vecteur x en ordre croissant. Pour effectuer un traitement équivalent avec la fonction `rank`, il faut plutôt utiliser

```
> x[rank(x)] <- x
```

ou

```
> x[rank(x, ties.method = "first")] <- x
```

si des valeurs sont répétées dans le vecteur.

```
> (x <- sample(0:10, 7, replace = TRUE))
[1] 2 1 7 2 10 9 9
> x[rank(x, ties.method = "first")] <- x
> x
[1] 1 2 2 7 9 9 10
```

- 7.5** a) On pose $u = 1$ et $v = 4$. Le déroulement pas à pas de l'algorithme se trouve au [tableau C.1](#).
 b) On pose $u = 0$ et $v = 9$. Le déroulement pas à pas de l'algorithme se trouve au [tableau C.2](#).
- 7.6** La fonction `qlinsearch` ci-dessous est une mise en œuvre en R de l'algorithme de recherche séquentielle rapide.

```
> qlinsearch <- function(x, table)
+ {
+   n <- length(table)
+   i <- 1
+   table <- c(table, x)
+   while (x != table[i])
+     i <- i + 1
+   if (i <= n)
+     i
+   else
+     NA
+ }
> x <- c(513, 780, 321, 828, 623, 80, 596, 423,
+       380, 707, 693, 766, 139, 161, 316, 71)
> qlinsearch(139, x)
[1] 13
> qlinsearch(513, x)
[1] 1
```


Étape 1	K :	0	0	0	0	0	0	0	0	0									
Étape 2	K :	2	2	1	0	1	3	3	2	1	1								
Étape 4	K :	2	4	5	5	6	9	12	14	15	16								
Étape 6																			
j = 16	K :	2	4	5	5	6	9	12	13	15	16								
	S :	-	-	-	-	-	-	-	-	-	-	-	-	-	7N	-	-		
j = 15	K :	2	4	5	5	6	9	12	12	15	16								
	S :	-	-	-	-	-	-	-	-	-	-	-	-	-	7O	7N	-	-	
j = 14	K :	2	4	5	5	6	9	11	12	15	16								
	S :	-	-	-	-	-	-	-	-	-	-	-	-	6I	7O	7N	-	-	
j = 13	K :	2	4	5	5	6	9	10	12	15	16								
	S :	-	-	-	-	-	-	-	-	-	-	6T	6I	7O	7N	-	-		
j = 12	K :	2	4	5	5	6	8	10	12	15	16								
	S :	-	-	-	-	-	-	-	-	5L	-	6T	6I	7O	7N	-	-		
j = 11	K :	2	3	5	5	6	8	10	12	15	16								
	S :	-	-	-	1G	-	-	-	-	5L	-	6T	6I	7O	7N	-	-		
j = 10	K :	2	3	5	5	5	8	10	12	15	16								
	S :	-	-	-	1G	-	4A	-	-	5L	-	6T	6I	7O	7N	-	-		
j = 9	K :	2	3	5	5	5	8	9	12	15	16								
	S :	-	-	-	1G	-	4A	-	-	5L	6A	6T	6I	7O	7N	-	-		
j = 8	K :	2	3	4	5	4	8	9	12	15	16								
	S :	-	-	-	1G	2R	4A	-	-	5L	6A	6T	6I	7O	7N	-	-		
j = 7	K :	2	3	4	5	5	8	9	12	14	16								
	S :	-	-	-	1G	2R	4A	-	-	5L	6A	6T	6I	7O	7N	8S	-		
j = 6	K :	2	2	4	5	5	8	9	12	14	16								
	S :	-	-	1N	1G	2R	4A	-	-	5L	6A	6T	6I	7O	7N	8S	-		
j = 5	K :	2	2	4	5	5	8	9	12	14	15								
	S :	-	-	1N	1G	2R	4A	-	-	5L	6A	6T	6I	7O	7N	8S	9.		
j = 4	K :	1	2	4	5	5	8	9	12	14	15								
	S :	-	0O	1N	1G	2R	4A	-	-	5L	6A	6T	6I	7O	7N	8S	9.		
j = 3	K :	1	2	4	5	5	7	9	12	14	15								
	S :	-	0O	1N	1G	2R	4A	-	5U	5L	6A	6T	6I	7O	7N	8S	9.		
j = 2	K :	0	2	4	5	5	7	9	12	14	15								
	S :	0C	0O	1N	1G	2R	4A	-	5U	5L	6A	6T	6I	7O	7N	8S	9.		
j = 1	K :	0	2	4	5	5	6	9	12	14	15								
	S :	0C	0O	1N	1G	2R	4A	5T	5U	5L</									

TAB. C.3 - Déroulement de l'algorithme de recherche séquentielle pour données triées pour l'exercice 7.7

étape	i	C_i	conséquence
1	1	-	-
2	1	071	→ 3
3	2	-	→ 2
2	2	080	→ 3
3	3	-	→ 2
2	3	139	→ 3
3	4	-	→ 2
2	4	161	→ 3
3	5	-	→ 2
2	5	316	→ 4
4	-	316	retourner 5

```

utilisateur    système    écoulé
      0.020         0.000        0.019
> system.time(qlinsearch(y, x))
utilisateur    système    écoulé
      0.015         0.000        0.016
> system.time(match(y, x))
utilisateur    système    écoulé
      0.001         0.001        0.000

```

7.7 Il faut identifier la position de la valeur $C = 316$ dans la liste. Le déroulement pas à pas de l'algorithme se trouve au [tableau C.3](#).

7.8 La clé, ici, consiste à utiliser la fonction `cummin` de calcul des minimums cumulatifs pour identifier les records du monde. La fonction `unique` permet ensuite d'en obtenir la liste en conservant une seule occurrence de chaque temps.

```

> cummin(x)
1964-10-15 1968-06-20 1968-10-13 1968-10-14 1968-10-14
      10.06      10.03      10.02       9.95       9.95
1968-10-14 1968-10-14 1975-08-20 1977-08-11 1978-07-30
       9.95       9.95       9.95       9.95       9.95
1979-09-04 1981-05-16 1983-05-14 1983-07-03 1984-05-05
       9.95       9.95       9.95       9.93       9.93
1984-05-06 1988-09-24 1989-06-16 1991-06-14 1991-08-25
       9.93       9.92       9.92       9.90       9.86
1991-08-25 1993-08-15 1994-07-06 1994-08-23 1996-07-27

```

```

      9.86      9.86      9.85      9.85      9.84
1996-07-27 1999-06-16 1999-08-22 2001-08-05 2002-09-14
      9.84      9.79      9.79      9.79      9.78
2005-06-14
      9.77
> unique(cummin(x))
[1] 10.06 10.03 10.02  9.95  9.93  9.92  9.90  9.86
[9]  9.85  9.84  9.79  9.78  9.77

```

Reste ensuite à trouver à quelle date sont survenus ces records. La fonction `match` identifie les indices dans le vecteur.

```

> match(unique(cummin(x)), x)
[1]  1  2  3  4 14 17 19 20 23 25 27 30 31

```

Il ne reste qu'à indiquer les étiquettes du vecteur pour obtenir les dates.

```

> names(x)[match(unique(cummin(x)), x)]
[1] "1964-10-15" "1968-06-20" "1968-10-13"
[4] "1968-10-14" "1983-07-03" "1988-09-24"
[7] "1991-06-14" "1991-08-25" "1994-07-06"
[10] "1996-07-27" "1999-06-16" "2002-09-14"
[13] "2005-06-14"

```

Chapitre 8

8.2 Il manque des accolades aux clauses `if`.

```

param <- function (moyenne, variance, loi)
{
  loi <- tolower(lois)
  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 être une de ",
      "\"normale\", \"gamma\" ou \"pareto\"")
}

```

Chapitre 9

9.1 Avec les données réparties sur deux lignes, le plus simple demeure d'importer les données avec `scan` et les replacer dans le bon format avec `matrix`. Remarquez comment l'on peut importer directement depuis Internet en spécifiant une adresse URL comme nom de fichier.

```

> x <- matrix(
+   scan("http://lib.stat.cmu.edu/datasets/boston",
+       skip = 22),
+   ncol = 14, byrow = TRUE)

```

Juste pour le plaisir, amusons-nous à extraire les noms des variables (colonnes) dans l'entête du fichier. Ils forment le premier champ de mode caractère des lignes 8 à 21, inclusivement.

```

> (xnames <- scan("boston", skip = 7, nlines = 14,
+   what = "character", flush = TRUE))
[1] "CRIM"  "ZN"    "INDUS" "CHAS"  "NOX"
[6] "RM"    "AGE"   "DIS"   "RAD"   "TAX"
[11] "PTRATIO" "B"     "LSTAT" "MEDV"

```

Nous pouvons ensuite ajouter les noms de colonne à notre matrice de données, puis convertir en tableau de données pour un traitement statistique ultérieur. La fonction `head` permet d'afficher les six premières lignes de l'objet.

```

> colnames(x) <- xnames
> x <- as.data.frame(x)
> head(x)
      CRIM  ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD
1 0.00632 18   2.31    0 0.538 6.575 65.2 4.0900   1
2 0.02731  0   7.07    0 0.469 6.421 78.9 4.9671   2

```

3	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2
4	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3
5	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3
6	0.02985	0	2.18	0	0.458	6.430	58.7	6.0622	3
	TAX	PTRATIO		B	LSTAT	MEDV			
1	296	15.3	396.90	4.98	24.0				
2	242	17.8	396.90	9.14	21.6				
3	242	17.8	392.83	4.03	34.7				
4	222	18.7	394.63	2.94	33.4				
5	222	18.7	396.90	5.33	36.2				
6	222	18.7	394.12	5.21	28.7				

Chapitre 11

- 11.2** a) Le motif identifie la lettre minuscule « a », suivie de zéro, une ou plusieurs occurrences des lettres minuscules « ab », suivies de la lettre minuscule « a ». Les choix 2 et 5 satisfont ces conditions.
- b) Le motif identifie la lettre « a », suivie d'au moins une occurrence de la lettre minuscule « b », suivie de zéro ou une occurrence de la lettre minuscule « c ». Les choix 1 et 3 satisfont ces conditions.
- c) Le motif identifie la lettre minuscule « a », suivie de n'importe quel caractère (incluant les lettres « b » et « c »), suivi d'au moins une occurrence des lettres minuscules « b » ou « c ». Les choix 1, 2, 3, 4 et 6 satisfont ces conditions.
- d) Le motif identifie les suites de lettres minuscules « abc » ou « xyz ». Les choix 1 et 2 satisfont la condition. Le troisième choix n'est pas valide puisque l'expression n'identifie pas le symbole « | ».
- e) Le motif identifie au moins une lettre minuscule (à l'exclusion de tout autre symbole), suivie de l'un ou l'autre des symboles « . », « ? », « ! » (sans répétition). Les choix 1, 4 et 6 satisfont ces conditions.
- f) Le motif identifie zéro, une ou plusieurs lettres minuscules ou majuscules (mais aucun autre symbole), suivies de tout caractère autre qu'une virgule, suivi du symbole « = ». Les choix 1, 5 et 6 satisfont ces conditions.
- g) Le motif identifie une lettre minuscule, suivie de l'un ou l'autre des symboles « . », « ? », « ! » (sans répétition), suivi d'au moins une espace, suivie d'une lettre majuscule. Les choix 4 et 5 satisfont ces conditions.
- h) Le motif identifie le symbole « < », suivi d'au moins un symbole autre que « > », suivi du symbole « > ». Les choix 1, 3 et 5 satisfont ces conditions. Dans le choix 2, tant <opentag> que <closetag> satisfont la condition, mais pas les deux ensemble.

- 11.3** Le motif doit identifier la lettre minuscule « p » précédée ou non d'une ou de plusieurs lettres (ou symboles, ce n'est pas spécifié), suivie d'une (et une seule) lettre ou d'une espace, de la lettre « t » et, enfin, de zéro ou de plusieurs lettres. Il y a assurément plusieurs réponses valides. En voici une : `「.*p[a-z]t.*」`.

- 11.4** Le motif

`「[a-zA-Z][0-9][a-zA-Z][[:blank:]]?[0-9][a-zA-Z][0-9]」`

convient si l'on ne permet que zéro ou une espace entre les deux blocs. S'il n'y a pas de limite au nombre d'espaces, remplacer « ? » par « * ».

- 11.5** `sed` est l'outil idéal pour de tels traitements simples à effectuer ligne par ligne.

```
| $ sed 's/\./,/ ' 100metres.dat
```

Pour placer le fichier modifié dans, disons, `100metres-dec.data`, utiliser

```
| $ sed 's/\./,/ ' 100metres.dat > 100metres-dec.data
```

- 11.6** a) C'est un travail pour `grep`. La solution la plus simple serait :

```
| $ grep '-08-' 100metres.dat
```

Cependant, `grep` n'aime pas cette commande puisque le symbole `-` est le préfixe des options. Dans de tels cas, il faut utiliser l'option « `-e` » pour déclarer explicitement à `grep` que ce qui suit est le motif à rechercher :

```
| $ grep -e '-08-' 100metres.dat
```

Autrement, simplement ajouter quelque chose à chercher avant le tiret :

```
| $ grep '.*-08-' 100metres.dat
```

- b) Le plus simple, ici, consiste à utiliser `awk` puisque les seconds champs — les temps — seront automatiquement disponibles.

```
| $ awk '$2 < 10' 100metres.dat
```

Malheureusement, cette commande risque de ne pas fonctionner sur certains systèmes qui utilisent non pas le point, mais la virgule comme séparateur décimal. C'est le cas, notamment, des Mac en configuration française. Dans ce cas, essayer plutôt :

```
| $ LC_ALL="en_US.UTF-8" awk '$2 < 10' 100metres.dat
```

Une solution avec `grep` consisterait à rechercher « 9. » après l'espace sur chaque ligne :

```
| $ grep ' 9\.' 100metres.dat
```

- c) Nous pouvons combiner les deux commandes ainsi :

```
| $ grep -e '-08-' 100metres.dat | awk '$2 < 10'
```

ou

```
| $ grep -e '-08-' 100metres.dat | grep ' 9\.'
```

C'est aussi possible de tout faire en un seul appel à awk :

```
| $ awk '/-08-/ && $2 < 10' 100metres.dat
```

- 11.7** Ce traitement s'effectue bien avec sed :

```
| $ sed -E 's/( |^ )f\((.*) , (.*)\) /fun(\3, \2)/' \
environnement.R
```

Le premier groupe « (|^) » sert à capturer une espace avant f ou le début de la ligne.

- 11.8** a) Utilisons la fonction `grep`. Le motif de l'expression régulière est simple dans ce cas. Avec l'argument `value` à `FALSE` (la valeur par défaut), la fonction retourne les positions dans le vecteur des chaînes de caractères qui correspondent à l'expression régulière. Comme le vecteur de noms est trié, il s'agit des valeurs demandées.

```
| > grep("ss", state.name)
```

- b) C'est toujours un travail pour `grep`, mais l'expression régulière est cette fois un peu plus complexe. Deux solutions sont proposées ci-dessous. Il faut utiliser l'argument `value = TRUE` pour obtenir les noms des États, et non seulement leurs positions dans `state.name`.

```
| > grep("(.*s){3,}", state.name, value = TRUE)
| > grep(".*s.*s.*s", state.name, value = TRUE)
```

- c) Nous devons extraire les États avec `grep`, puis utiliser `sub` pour remplacer les caractères après le quatrième par un point. Ceci requiert de sauvegarder les caractères jusqu'aux deux « s » successifs pour les réutiliser dans la chaîne de remplacement.

```
| > sub("(^.{0,2}ss).*", r"(\1.)",
| +   grep("(^.{0,2}ss)", state.name, value = TRUE))
```

- 11.9** a) Une ligne satisfait les critères si, minimalement :

- i) elle débute par un ou plusieurs symboles « # » suivi ou non d'une ou plusieurs espaces;
- ii) le mot « exemple » y apparaît avec ou sans majuscule;

iii) le mot « exemple » est écrit au singulier ou au pluriel.

Un motif adéquat serait donc `#+ *[Ee]xemples?`. Saisi sous forme de chaîne de caractères brute dans R : `r"#+ *[Ee]xemples?"`.

b) Voici une mise en œuvre possible de cette fonction.

```
> is.examplesPresent <- function(x)
+   any(grepl(r"#+ *[Ee]xemples?", x))
```

Ça fonctionne ?

```
> is.examplesPresent(readLines("sqrt.R",
+                               encoding = "UTF-8"))
[1] TRUE
> is.examplesPresent(readLines("sqrt-sans-exemples.R",
+                               encoding = "UTF-8"))
[1] FALSE
```

Chapitre 12

12.1 La réponse au mystère se cache dans l'environnement de la fonction `Fn` :

```
> ls.str(envir = environment(Fn))
f :   num 0
method : int 2
na.rm : logi TRUE
nobs : int 3
x :   num [1:2] 1 3
y :   num [1:2] 0.667 1
yleft : num 0
yright : num 1
```

L'environnement de la fonction contient donc les éléments requis pour calculer la fonction de répartition empirique en un point quelconque, soit, entre autres, les valeurs distinctes du vecteur de données (vecteur `x`) et les valeurs correspondantes de la fonction de répartition empirique (vecteur `y`).

12.2 Le résultat de `f(10)` est 123 puisque `y` n'est pas défini dans la fonction `f`. Par conséquent, R va remonter jusqu'à l'environnement global pour trouver le symbole et la valeur qui lui est associée.

12.3 La fonction `f` retourne une fonction (anonyme) qui elle-même retourne la valeur d'un objet `x` qui ne fait pas partie de ses arguments, mais qui se trouve dans son environnement d'évaluation. La valeur de `x` dans cet environnement est 10. C'est donc cette valeur que retourne l'appel `g()`.

- 12.4** Si l'on suit la chaîne de fonctions, le calcul qui est effectué par `f()` est en définitive $x + y$. Reste à déterminer les valeurs de x et de y au moment du calcul. C'est facile pour y : l'objet n'est défini que dans l'espace de travail et sa valeur est 10. Quant à x , même si sa valeur change à plusieurs reprises (plus précisément : l'objet est défini dans plusieurs environnements successifs), R utilisera celle qui a cours lorsque l'addition est effectuée, soit celle à l'intérieur de la fonction `h`. Le résultat cherché est donc 13.
- 12.5** Grâce à l'évaluation paresseuse, il suffit de spécifier comme valeur par défaut pour le second argument le logarithme du premier argument.

```
> f <- function(x, y = log(x)) x * y
```

Bibliographie

- Abelson, H., G. J. Sussman et J. Sussman. 1996, *Structure and Interpretation of Computer Programs*, 2^e éd., MIT Press, ISBN 0-26201153-0. Cité aux pages 20, 28, 36, 40 et 57.
- Aho, A. V., B. W. Kernighan et P. J. Weinberger. 1988, *The AWK Programming Language*, Pearson, ISBN 978-020107981-4. Cité à la page 267.
- Apple. 2014, *Shell Scripting Primer*, Apple, <https://developer.apple.com/library/content/documentation/OpenSource/Conceptual/ShellScripting>. Cité à la page 256.
- Becker, R. A. 1994, *A Brief History of S*, rapport technique, AT&T Bell Laboratories, <http://cm.bell-labs.com/cm/ms/departments/sia/doc/94.11.ps>. Cité à la page 39.
- Becker, R. A. et J. M. Chambers. 1984, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, ISBN 0-53403313-X. Cité à la page 39.
- 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. Cité à la page 39.
- Braun, W. J. et D. J. Murdoch. 2007, *A First Course in Statistical Programming with R*, Cambridge University Press, ISBN 978-0-52169424-7. Cité à la page 51.
- Braun, W. J. et D. J. Murdoch. 2016, *A First Course in Statistical Programming with R*, 2^e éd., Cambridge University Press, ISBN 978-1-10757646-9. Cité à la page 51.
- Burger, M., K. Juenemann et T. Koenig. 2024, *RUnit: R Unit Test Framework*, <https://cran.r-project.org/package=RUnit>. R package version 0.4.33. Cité à la page 170.
- Cameron, D., J. Elliott, M. Loy, E. S. Raymond et B. Rosenblatt. 2004, *Leaning GNU Emacs*, 3^e éd., O'Reilly, Sebastopol, CA, ISBN 0-59600648-9. Cité à la page 323.
- Chambers, J. M. 1998, *Programming with Data: A Guide to the S Language*, Springer, ISBN 0-38798503-4. Cité à la page 39.

- Chambers, J. M. 2000, « Stages in the evolution of S », <https://9p.io/cm/ms/departments/sia/S/history.html>. Cité à la page 39.
- Chambers, J. M. 2008, *Software for Data Analysis: Programming with R*, Springer, ISBN 978-0-38775935-7. Cité à la page 40.
- Chambers, J. M. et T. J. Hastie. 1992, *Statistical Models in S*, Wadsworth & Brooks/Cole, ISBN 0-53416765-9. Cité à la page 39.
- Friedl, J. 2006, *Mastering Regular Expressions*, 3^e éd., O'Reilly Media, ISBN 0-59652812-4. Cité aux pages 249 et 256.
- Goulet, V. 2024a, *Gestion de versions avec Git*, <https://gitlab.com/vigou3/gestion-versions-avec-git>. Cité à la page vi.
- Goulet, V. 2024b, *Ligne de commande Unix*, <https://gitlab.com/vigou3/ligne-commande-unix>. Cité aux pages vi et 251.
- Hebenstreit, J. 2024, « Informatique — Principes », *Encyclopædia Universalis*, <https://www.universalis.fr/encyclopedie/informatique-principes/>. Consulté le 9 décembre 2024. Cité à la page 3.
- Hoare, C. A. R. 1962, « Quicksort », *The Computer Journal*, vol. 5, n° 1, p. 10-16, doi : 10.1093/comjnl/5.1.10. Cité à la page 189.
- Hoare, C. A. R. 1973, *Hints on Programming Language Design*, Computer Science Department Report No. CS-403, Stanford University, https://web.eecs.umich.edu/~bchandra/courses/papers/Hoare_Hints.pdf. Cité à la page 8.
- Hornik, K. et R Core Team. 2024, « The R FAQ », <https://cran.r-project.org/doc/FAQ/R-FAQ.html>. Cité aux pages 46 et 171.
- Hunt, A. et D. Thomas. 1999, *The Pragmatic Programmer : From Journeyman To Master*, Addison-Wesley, ISBN 978-020161622-4. Cité aux pages 47, 160, 169, 213 et 220.
- Iacus, S. M., S. Urbanek, R. J. Goedman et B. D. Ripley. 2023, « R for macOS FAQ », <https://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html>. Cité à la page 48.
- IEEE. 2003, 754-1985 *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, Piscataway, NJ. Cité à la page 111.
- Ihaka, R. et R. Gentleman. 1996, « R: A language for data analysis and graphics », *Journal of Computational and Graphical Statistics*, vol. 5, n° 3, p. 299-314. Cité aux pages 40, 299 et 302.

- Kernighan, B. W. et R. Pike. 1999, *The Practice of Programming*, Addison-Wesley, ISBN 0-20161586-X. Cité aux pages 160, 166 et 181.
- Kernighan, B. W. et P. J. Plauger. 1978, *The Elements of Programming Style*, 2^e éd., McGraw-Hill, ISBN 0-07034207-5. Cité à la page 160.
- Kernighan, B. W. et D. M. Ritchie. 1978, *The C Programming Language*, Prentice Hall, ISBN 0-13110163-3. Cité aux pages 9 et 267.
- Knuth, D. E. 1997a, *The Art of Computer Programming*, vol. 1, Fundamental Algorithms, Addison-Wesley, ISBN 0-20189683-4. Cité aux pages 20 et 21.
- Knuth, D. E. 1997b, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, Addison-Wesley, ISBN 0-20189684-2. Cité à la page 20.
- Knuth, D. E. 1997c, *The Art of Computer Programming*, vol. 3, Sorting and Searching, Addison-Wesley, ISBN 0-20189685-0. Cité aux pages 181, 182, 187, 188, 192, 193 et 208.
- Ligges, U. 2003, « R-WinEdt », dans *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, sous la direction de K. Hornik, F. Leisch et A. Zeileis, TU Wien, Vienna, Austria, <https://www.r-project.org/conferences/DSC-2003/Proceedings>. Cité à la page 47.
- Loo, M. P. J. v. d. 2021, « A method for deriving information from running R code », *The R Journal*, vol. 13, n° 1. Cité à la page 170.
- Martin, R. C. 2009, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, ISBN 978-013235088-4. Cité à la page 160.
- Oualline, S. 1997, *Practical C Programming*, 3^e éd., O'Reilly, ISBN 978-156592306-5. Cité aux pages 5, 10, 160, 162 et 166.
- Oualline, S. 2003, *Practical C++ Programming*, 2^e éd., O'Reilly, ISBN 978-059600419-4. Cité à la page 160.
- R Core Team. 2020, *R Language Definition*, R Foundation for Statistical Computing, Vienna, Austria, <https://cran.r-project.org/doc/manuals/R-lang.html>. Cité à la page 73.
- R Core Team. 2024, *R Data Import/Export*, R Foundation for Statistical Computing, Vienna, Austria, <https://cran.r-project.org/doc/manuals/R-data.html>. Cité à la page 231.
- Redd, A. 2010, « Introducing NppToR: R interaction for Notepad++ », *R Journal*, vol. 2, n° 1, p. 62-63, https://journal.r-project.org/archive/2010-1/RJournal_2010-1.pdf. Cité à la page 47.

- Ripley, B. D. et D. J. Murdoch. 2024, « R for Windows FAQ », <https://cran.r-project.org/bin/windows/base/rw-FAQ.html>. Cité à la page 48.
- Robbins, A. D. 2023, *Gawk: Effective AWK Programming: A User's Guide for GNU Awk*, Free Software Foundation, 5^e éd., <https://www.gnu.org/software/gawk/manual>. Cité aux pages 268, 274, 275 et 277.
- Rossini, A. J., R. M. Heiberger, K. Hornik, M. Maechler, R. A. Sparapani, S. J. Eglen, S. P. Luque, H. Redestig, V. Spinu, L. Henry et J. A. Branham. 2024, *ESS — Emacs Speaks Statistics*, ESS Developers, <https://ess.r-project.org>. Cité à la page 323.
- Stephens, R. 2013, *Essential Algorithms, A Practical Approach to Computer Algorithms*, Wiley, ISBN 978-111861210-1. Cité aux pages 20, 34, 37 et 188.
- Swinnen, G. 2012, *Apprendre à programmer avec Python 3*, <https://inforef.be/swi/python.htm>. Cité à la page 2.
- Tierney, L. 1990, *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley, ISBN 0-47150916-7. Cité à la page 40.
- Venables, W. N. et B. D. Ripley. 2000, *S Programming*, Springer, New York, ISBN 0-38798966-8. Cité à la page 51.
- Venables, W. N. et B. D. Ripley. 2002, *Modern Applied Statistics with S*, 4^e éd., Springer, New York, ISBN 0-38795457-0. Cité à la page 51.
- Venables, W. N., D. M. Smith et R Core Team. 2024, *An Introduction to R*, R Foundation for Statistical Computing, <https://cran.r-project.org/doc/manuals/R-intro.html>. Cité à la page 54.
- Wall, L., T. Christiansen et R. L. Schwartz. 1996, *Programming Perl*, 2^e éd., O'Reilly & Associates, ISBN 1-56592149-6. Cité à la page 2.
- Wickham, H. 2011, « testthat: Get started with testing », *The R Journal*, vol. 3, p. 5-10, https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf. Cité à la page 170.
- Wickham, H. 2024, *Advanced R*, <https://adv-r.had.co.nz>. Consulté le 9 décembre 2024. Cité à la page 312.
- Wikipédia. 2023, « Nombre magique — Wikipédia, l'encyclopédie libre », [https://fr.wikipedia.org/wiki/Nombre_magique_\(programmation\)](https://fr.wikipedia.org/wiki/Nombre_magique_(programmation)). Page disponible le 18 novembre 2023. Cité à la page 162.
- Wikipédia. 2024a, « Algorithme — Wikipédia, l'encyclopédie libre », <https://fr.wikipedia.org/wiki/Algorithme>. Page disponible le 9 décembre 2024. Cité à la page 21.

Wikipédia. 2024b, « Expression régulière — Wikipédia, l'encyclopédie libre », https://fr.wikipedia.org/wiki/Expression_régulière. Page disponible le 18 avril 2024. Cité à la page 247.

Wikipédia. 2024c, « Récursion terminale — Wikipédia, l'encyclopédie libre », https://fr.wikipedia.org/wiki/Récursion_terminale. Page disponible le 29 novembre 2024. Cité à la page 28.

Index

Symboles

!, 59
!=, 59, 132
!~ (awk), 273, 275
() (regex), 256, 258, 265
*, 59
* (regex), 258, 259, 259, 260
+, 59, 132
+ (regex), 256, 258, 259, 259, 260
, (awk), 271
-, 59, 132
- (regex), 263, 263
->, 59, 62
->>, 59
. (regex), 258, 259
..., 70, 71, 109, 119, 120
/, 59
/ (awk), 270
:, 59
:, 105
;, 63
<, 59, 132
<-, 59, 61, 164
<<-, 59
<=, 59, 132
=, 62
= (awk), 275
==, 59, 132, 171
== (awk), 273
>, 59, 132
>=, 59, 132
?, 51
? (regex), 256, 258, 259, 259, 260

?: (awk), 273
[, 59, 67, 68, 115, 124
[<-, 67, 68
[[, 59, 124
[] (regex), 258, 262
\$, 59, 125
%*%, 59
%/%, 59
%%, 59
%in%, 199, 207
%o%, 59, 137, 153
&, 59
&&, 59
\$ (regex), 258, 261
\ (Bash), 251
\\ (regex), 258, 261, 280
^, 59
^ (regex), 258, 261, 263
~ (awk), 273, 275, 276
~ (Unix), 15
{ } (awk), 270, 276
{ } (regex), 256, 258, 259, 260
||, 59
|, 59
| (regex), 256, 258, 264, 266, 267
| (tuyau Unix), 251, 297

A

abs, 72, 77, 96, 97, 99, 103, 104,
172-176, 203
Ackermann (fonction), 36
acos, 77
action (awk), 270, 276

action (awk), [270](#)
 affectation, [59](#), [61](#)
 Algol, [7](#)
 algorithme, [3](#), [19](#), [20](#)
 algorithmique, [3](#), [19](#)
 all, [80](#)
 all, [74](#), [80](#), [95](#), [101](#), [102](#)
 all.equal, [170](#)
 any, [80](#)
 any, [74](#), [80](#), [95](#), [101](#), [102](#)
 APL, [8](#), [40](#), [57](#)
 append, [237](#), [245](#)
 application, [108](#)
 apply, [119](#), [122](#), [125](#), [143](#), [144](#)
 args, [94](#)
 arithmétique vectorielle, [66](#), [69](#), [72](#),
 [108](#), [128](#)
 array, [114](#), [142](#), [143](#)
 arrondi, [79](#)
 as.character, [65](#)
 as.data.frame, [129](#)
 as.Date, [132](#)
 as.logical, [65](#)
 as.numeric, [65](#)
 as.POSIXct, [132](#)
 as.POSIXlt, [132](#)
 assembleur, [5](#)
 assertError, [170](#)
 assertWarning, [170](#)
 assign, [302](#)
 attr, [113](#), [139](#), [140](#)
 attribut, [112](#)
 attributes, [112](#), [139](#), [140](#)
 avertissement, [72](#)
 AWK, [267](#), [269](#)
 awk, [251](#), [255](#), [268](#), [268-278](#), [280](#), [281](#),
 [291-293](#), [298](#), [363](#), [364](#)

B

Bash, [319](#)
 BEGIN (awk), [274](#), [278](#)
 BEGINFILE (awk), [274](#)

biscuits, *voir* Syndrome de la plaque
 à biscuits
 body, [301](#), [308](#)
 boucle, [23](#)
 BRE, [256](#)
 break, [197](#), [197](#), [203](#), [204](#)
 browser, [218](#), [218](#), [227](#)
 by, [53](#), [99](#), [100](#), [150-152](#), [201](#)

C

C, [9](#), [57](#), [268](#)
 C, [87](#)
 c, [64](#)
 C#, [9](#)
 C++, [9](#), [10](#)
 caractère d'échappement, [261](#), [280](#)
 caractère de remplacement, [259](#)
 caret, *voir* ^ (regex)
 carotte, [116](#), [121](#), [122](#)
 cat, [235](#), [235](#), [237](#), [245](#), [285](#)
 cbind, [85](#), [116](#), [117](#), [130](#), [141](#), [142](#), [153](#)
 ceiling, [79](#), [101](#)
 chaîne de caractères
 brute, [280](#)
 chaîne de caractères
 concaténation, [82](#)
 nombre de caractères, [82](#)
 sous-chaîne, [82](#)
 séparation, [83](#)
 character, [64](#), [87](#), [110](#), [279](#)
 chemin d'accès, [15](#), [233](#)
 absolu, [15](#), [49](#)
 relatif, [16](#), [49](#)
 class, [112](#), [149](#), [152](#)
 classe de caractères, [262](#)
 classe prédéfinie, [264](#)
 clé (tri), [184](#)
 COBOL, [7](#)
 col, [94](#)
 colMeans, [120](#), [135](#), [143](#)
 colSums, [120](#), [135](#), [153](#)
 comment.char, [232](#)
 commentaire, [44](#)

commentaires, 166
compilateur, 4
compilé (langage), 4
complex, 87, 110
compteur, 30
concaténation, 64, 82, 257
contraction, 28
conversion forcée, 65, 67
cos, 52, 69, 77, 86, 93, 99
CSV, 233, 268
cummax, 82, 102
cummin, 82, 102, 359
cumprod, 81, 102
cumsum, 81, 102
curve, 94
cut, 12

D

data, 95, 139, 149
data frame, *voir* tableau de données
data.frame, 129
Date, 132, 132
density, 53
det, 135
diag, 136, 141, 153
dictionnaire, 62, 300
diff, 80, 102
différences, 80
digamma, 77
dim, 112-114, 140-143, 149, 153, 239, 348
dimension, 112, 155
dimnames, 112
do (awk), 277
done, 285
dossier de départ, *voir* répertoire personnel
dossier de travail, *voir* répertoire de travail

E

écart type, 80
ecdf, 311

echo, 284, 285, 288, 289
ed, 252, 253
else, 73, 96-98, 221-227, 311
Emacs, 46, 51, 164, 165, 167, 213, 220, 323-331
 annuler, 327
 déplacement du curseur, 327
 mode ESS, 46, 165, 328-331
 nouveau fichier, 327
 rechercher et remplacer, 327
 sauvegarder, 327
 sauvegarder sous, 327
 sélection, 328
END (awk), 272, 274
ENDFILE (awk), 274
enregistrement (tri), 182
environment, 301, 302, 308, 309
environnement, 62, 300
 d'évaluation, 302, 304
 englobant, 300, 302, 304
 global, 301, 302, 304
 vide, 304
ERE, 256
erreur, 72
espace de travail, 43, 47, 50, 61
étendue, 81
étiquette, 64, 65, 68, 112, 123, 155
Euclide (algorithme), 20, 20, 22, 28, 35, 336, 343
évaluation paresseuse, 306
Excel, 41
exp, 52, 69, 76, 86, 99, 221-227
expansion, 28
Explorateur Windows, 14, 15, 17
exponentielle, 76
expression, 41, 59
expression, 110, 138
expression régulière, 247, 251-253, 256, 257, 270, 273, 274, 277, 278
 basique, 256
 Perl, 256
 étendue, 256, 257

F

F, *voir* FALSE
 factor, [130](#), [130](#), [149](#), [150](#)
 factorial, [77](#), [105](#)
 factorielle, [28](#), [30](#), [77](#)
 FALSE, [58](#), [109](#)
 fermeture, [300](#)
 Fibonacci, [35](#), [336](#), [337](#)
 fichier (tri), [184](#)
 file, [237](#), [245](#)
 Finder, [13](#), [14](#), [17](#)
 floor, [79](#), [101](#)
 fonction
 anonyme, [70](#), [352](#)
 appel, [72](#)
 argument formel, [70](#)
 définition, [70](#)
 résultat, [71](#)
 for, [163](#), [195](#), [195](#), [197](#), [200](#), [201](#),
 [204-206](#), [285](#)
 for (awk), [277](#)
 formals, [301](#), [308](#)
 Fortran, [5](#)
 FS (awk), [277](#)
 function, [70](#), [70](#), [90-94](#), [96-99](#), [110](#),
 [143](#), [146-148](#), [153](#), [172-176](#),
 [202-206](#), [221-228](#), [300](#),
 [307-311](#)

G

gamma, [77](#)
 gamma, [52](#), [77](#), [86](#), [99](#), [105](#)
 get, [302](#), [308](#)
 getwd, [47](#), [52](#)
 Git, [vi](#), [49](#)
 Git Bash, [243](#), [251](#), [255](#), [319](#), [321](#)
 .GlobalEnv, [301](#), [304](#)
 Go, [16](#)
 gregexpr, [279](#), [279](#)
 grep, [12](#), [248](#), [251](#), [252](#), [252](#), [253](#),
 [255-257](#), [259](#), [262](#), [265](#),
 [268-271](#), [281](#), [282](#), [284-290](#),
 [293](#), [294](#), [363](#)

grep (R), [279](#), [364](#)
 groupe, *voir* () (regex)
 groupe de capture, [266](#)
 groupe de caractères, [266](#)
 gsub, [280](#)

H

head, [78](#), [101](#), [361](#)
 help, [51](#)
 help.search, [51](#)

I

identical, [171](#), [171](#)
 if, [73](#), [93](#), [95-98](#), [172-176](#), [203-206](#),
 [221-227](#), [311](#)
 if (awk), [277](#)
 ifelse, [75](#), [75](#), [99](#), [273](#)
 in, [285](#)
 indentation, [23](#), [164](#)
 indiciage, [64](#)
 liste, [123](#), [155](#)
 matrice, [115](#), [154](#)
 tableau, [115](#)
 vecteur, [67](#), [103](#)
 Inf, [58](#), [111](#)
 install.packages, [244](#)
 interpréteur, [4](#), [41](#), [268](#), [269](#)
 interprété (langage), [4](#)
 is.character, [65](#)
 is.finite, [111](#)
 is.infinite, [111](#)
 is.logical, [65](#)
 is.na, [85](#), [111](#), [139](#), [311](#)
 is.nan, [111](#)
 is.null, [112](#)
 is.numeric, [65](#)
 isFALSE, [74](#)
 isTRUE, [74](#)
 itération, [26](#), [28](#), [194](#)

J

J, [8](#)
 Java, [9](#), [10](#)

L

lapply, 90, **125**, 128, 146, 148, 351
length, 53, 87, 89, 93, 99, 100, **110**,
138-140, 142, 144, 145, 149,
204, 205, 348
LETTERS, 52, 86
letters, 52, 86, 87, 140, 178
levels, **130**, 149, 150
lfactorial, 77, 105
lgamma, 77, 105
.libPaths, 243, 244
.libPaths, 243
library, 171, 241, **242**, 243-246
Linux, 9, 11, 13, 15
Lisp, 6, 40, 57, 108
list, **110**, **123**, 123, 138, 144-146
liste, **123**
log, **76**, 99, 221-227
log10, 99
logarithme, 76
logical, **64**, 87, 110, 111
longueur, **110**, 155
ls, 54, 137, 302, 308
ls.str, **302**

M

macOS, 11, 13-15, 17, 327
mapply, **128**, 148
match, **199**, 207, 208, 360
match.arg, 93
matrice, **113**
 diagonale, 136
 déterminant, 135
 identité, 136
 inverse, 135
 moyennes par colonne, 135
 moyennes par ligne, 135
 nombre de colonnes, 134
 nombre de lignes, 134
 produit, 135
 sommes par colonne, 135
 sommes par ligne, 134
 transposée, 135

matrix, 53, 86, 94, 95, 101, **114**, 114,
140-143, 145, 238
max, 53, **81**, 93, 102, 103, 143, 345
maximum
 cumulatif, 82
 indice, 200
 parallèle, 82
 valeur, 81
mean, **80**, 99, 102, 143, 146-148, 150,
172-176, 311, 342
median, **81**, 102
médiane, 81
min, 53, **81**, 102
MinGW, 12
minimum
 cumulatif, 82
 indice, 200
 parallèle, 82
 valeur, 81
mode, 64, **109**, 155
mode, 90, **109**, 138, 139, 144, 149, 348
modulo, 23
motif, **247**, 248
motif (awk), 270, 274
motif (awk), **270**
moyenne, 80
MSYS, 243, 251
MSYS2, 243, 251, 255

N

NA, **58**, 74
na.rm, 311
names, **65**, 87, 89, 112, 140, 149, 348,
351
NaN, **58**, 111
nchar, **82**, 103, **110**, 138
ncol, 53, 95, **134**, 140, 141, 143, 153,
238
new.env, **302**
next, **197**
NF (awk), 272
nombre magique, **162**
notation

infixée, 16
 préfixée, 7, 16
 suffixée, 16
 Notepad++, 47
 NR (awk), 271
 nrow, 53, 95, 134, 140, 141, 143, 153, 238
 NULL, 58, 112, 113
 numeric, 64, 87, 89, 99, 100, 110, 138, 201, 206, 221-227

O

OFS (awk), 278
 opérateur de répétition, 259
 opérateurs, 58
 order, 92, 199, 199, 207, 348
 ordered, 149
 ordre, 199
 ORS (awk), 278
 outer, 136
 outer, 137, 153, 348

P

paquetage, 170, 241
 paradigme, 4
 déclaratif, 4
 fonctionnel, 4
 impératif, 4
 orienté objet, 5
 paste, 82, 82, 98, 103, 157, 224
 paste0, 82
 PCRE, 256
 Perl, 1, 248
 pile, 28
 plot, 52
 pmax, 82, 102
 pmin, 82, 102
 portée lexicale, 300, 304, 306, 344
 POSIX, 132, 264
 POSIXct, 132, 132, 133
 POSIXlt, 132, 133
 print, 71, 95, 200-202, 214, 224, 299
 print (awk), 270, 271, 276, 277

printf (awk), 277
 prod, 80, 102, 103, 143, 341
 produit, 80
 cumulatif, 81
 extérieur, 136
 pseudocode, 22
 Python, 10, 40

Q

q, 50
 qlinsearch, 356
 quantile, 81
 quantile, 81, 102
 quicksort, 189, 190

R

racine carrée, 76
 rang, 199
 range, 81, 102
 rank, 199, 199, 207, 208
 rbind, 116, 130, 141, 142
 .rds (extension), 236
 read.csv, 233, 234, 235
 read.csv2, 234, 235
 read.delim, 234
 read.delim2, 234
 read.table, 233, 234, 235
 readLines, 298
 readRDS, 236, 239
 Recall, 75, 195, 342
 recherche
 bisection, 193
 dichotomique, 193
 interpolation, 193
 linéaire, 193
 séquentielle, 193
 séquentielle pour données triées, 194, 208
 séquentielle rapide, 193, 208
 récursion, 26, 28, 60, 194
 terminale, 28
 récursivité, voir récursion
 recyclage, 66, 66, 67

regexpr, [279](#), [279](#)
règle (awk), [269](#), [270](#)
regmatches, [278](#), [279](#)
renverser, [199](#)
rep, [53](#), [77](#), [90](#), [100](#), [105](#), [143](#), [148](#)
rep.int, [78](#)
rep_len, [78](#), [101](#)
repeat, [196](#), [197](#), [202](#), [203](#)
répertoire, [13](#)
répertoire de travail, [47](#), [49](#), [54](#), [233](#)
répertoire personnel, [12](#), [13](#), [15](#), [17](#)
répétition, [77](#)
replace, [99](#), [146](#), [148](#), [152](#)
return, [71](#), [75](#), [93](#), [96](#), [104](#), [163](#),
 [172-176](#), [205](#), [206](#), [342](#)
rev, [102](#), [199](#), [207](#)
rgamma, [221-224](#), [226-228](#)
rm, [54](#), [97](#), [180](#), [204](#)
rnorm, [52](#)
round, [53](#), [54](#), [79](#), [101](#)
row.names, [149](#)
rowMeans, [120](#), [135](#)
rowSums, [120](#), [134](#), [143](#), [153](#)
rpois, [90](#)
RS (awk), [277](#)
RStudio, [46](#), [49](#), [51](#), [55](#), [164](#), [168](#), [213](#),
 [220](#), [315-321](#)
 configuration, [319](#)
 projets, [316](#)
 sous-fenêtres, [315](#)
 symbole d'affectation, [317](#)
runif, [53](#), [221-227](#)

S

S, [39](#), [40](#), [57](#)
S+, [40](#)
S-PLUS, [39](#)
sample, [99](#), [102](#), [126](#), [143](#), [146](#), [148](#),
 [152](#), [237](#), [353](#)
sapply, [127](#), [128](#), [146-148](#), [163](#), [351](#),
 [352](#)
save.image, [43](#), [50](#), [319](#), [330](#)
saveRDS, [236](#), [239](#)

scale, [221-227](#)
scan, [232](#), [238](#)
Scheme, [28](#), [40](#)
sd, [80](#), [102](#)
search, [241](#), [245](#)
sed, [248](#), [251](#), [253](#), [253-258](#), [265](#), [266](#),
 [268](#), [269](#), [281-283](#), [285](#),
 [290-292](#), [297](#), [364](#)
sémantique, [3](#), [162](#)
séparateur
 d'enregistrements, [269](#), [277](#)
 de champs, [269](#), [277](#)
seq, [53](#), [77](#), [78](#), [90](#), [99](#), [100](#), [105](#), [132](#),
 [138](#), [145](#), [146](#), [151](#), [201](#)
seq.int, [204](#)
seq_along, [77](#), [78](#), [100](#), [201](#)
seq_len, [77](#), [78](#), [100](#), [201](#), [205](#), [206](#)
setwd, [49](#), [49](#)
signature, [23](#), [167](#), [169](#)
sin, [52](#), [77](#), [86](#), [93](#), [99](#)
solve, [53](#), [54](#), [135](#), [153](#)
somme, [80](#)
 cumulative, [81](#)
sort, [198](#), [199](#), [206](#)
source, [170](#), [171](#), [177](#), [180](#)
spécification, [169](#)
sqrt, [76](#), [93](#), [99](#), [172-176](#), [179](#), [202](#),
 [203](#)
Startup, [243](#)
state.name, [279](#), [297](#)
stop, [72](#), [75](#), [93](#), [172-176](#), [203](#), [205](#),
 [206](#), [221-227](#)
stopifnot, [170](#)
str, [171](#)
strsplit, [83](#), [103](#)
structure, [140](#)
sub, [279](#), [280](#), [364](#)
subset, [149](#)
substr, [82](#), [103](#)
suite, [77](#)
sum, [72](#), [80](#), [92](#), [93](#), [102](#), [103](#), [143-146](#),
 [150](#), [311](#), [341](#)
summary, [81](#), [102](#)

Swift, 16
 switch, 24, 74
 Syndrome de la plaque à biscuits, 198
 syntaxe, 3
 system.time, 99, 206, 208
 système d'exploitation, 11
 système de fichiers, 13
 sérialisation, 236

T

T, *voir* TRUE
 t, 53, 135, 153
 table, 205
 tableau, 114
 tableau de données, 129, 129
 tail, 78, 101
 tan, 77, 100
 tapply, 131, 150
 Terminal, 243, 251
 terminal (RStudio), 319
 tests unitaires, 171
 tests unitaires, 169, 170
 tr, 12
 traceback, 218, 226
 tranche, 116, 121, 122
 tri
 bulles, 188, 208
 dénombrement, 182, 183, 189, 191, 208
 dénombrement rapide, 191, 208
 insertion, 182-185
 sélection, 182, 183, 186, 187, 207
 échange, 182, 183, 187
 tri (R), 198
 Trig, 100
 trigamma, 77
 trigonométrie, 77
 TRUE, 58, 109
 trunc, 79, 101
 tube, *voir* carotte
 typeof, 110, 138

U

unclass, 152, 201
 unique, 199, 207, 359
 Unix, 11, 12, 12, 15, 16, 132, 251, 281
 philosophie, 12
 unlink, 239
 unlist, 125, 145, 146
 update.packages, 244

V

valeur absolue, 77
 var, 80, 102
 variance, 80
 VBA, 10
 vecteur, 63, 64
 atomique, 64
 concaténation, 64
 nommé, 65
 récursif, 109, 123
 simple, 64, 65, 109, 123
 étiqueté, 64
 vector, 87, 144
 Visual Basic, 10

W

warning, 72, 75
 which, 199, 207
 which.max, 200, 207
 which.min, 200, 207
 while, 172-176, 196, 196, 197, 204, 205, 221-224, 226-228
 while (awk), 277
 wildcard character, *voir* caractère de remplacement
 Windows, 9, 11-14, 16, 17
 WinEdt, 47
 working directory, *voir* répertoire de travail
 workspace, *voir* espace de travail
 write, 235, 237
 write.csv, 235
 write.csv2, 235
 write.table, 235

Ce document a été produit avec le système de mise en page Xe_{La}TeX. Le texte principal est composé 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 Fira Sans. Des icônes proviennent de la police Font Awesome.

