

Visualiser vos données avec R

Nicolas Casajus, Kevin Cazelles

2020-05-14

Contents

Bienvenue	3
Qu'est-ce que ce livre ?	3
Future et Collaboration	4
Avant propos	4
Remerciements	5
1 Introduction	6
I Science visualisation et R	6
2 Science et graphiques	6
2.1 Une brève histoire du graphique	6
2.2 Qu'est-ce qu'un graphique ?	11
2.3 Un graphique pour quoi faire ?	11
2.4 Du bon usage des graphiques	11
3 À propos de R	11
3.1 De S à R - 50 ans d'histoire	11
3.2 R aujourd'hui	15
4 Les différents systèmes graphiques dont R dispose	28
4.1 Les étapes pour créer un graphique avec R	29
4.2 Le système graphique du package <i>graphics</i>	29
4.3 Le système graphique du package <i>grid</i>	29
4.4 Les graphiques interactives	30
4.5 Interfaces	30
II Utiliser graphics	30

5 Édition d'un graphique	30
5.1 Graphique vierge	30
5.2 Ajout de points	34
5.3 Ajout de lignes	35
5.4 Ajout de polygones	37
5.5 Ajout d'une flèche	39
5.6 Ajout d'un titre	40
5.7 Ajout de texte	40
5.8 Ajout d'une légende	42
5.9 Ajout d'un axe	44
5.10 Ajout d'une image	45
6 Graphiques classiques	47
6.1 Diagramme de dispersion	48
6.2 Boîte à moustaches	49
6.3 Diagramme en bâtons	51
6.4 Histogramme	52
6.5 Diagramme sectoriel	54
6.6 Fonctions mathématiques	55
7 Paramètres graphiques	57
7.1 La fonction <code>par()</code>	57
7.2 Fonte de caractères	58
7.3 Symboles ponctuels	62
7.4 Types de lignes	64
7.5 Modification des axes	64
7.6 Ajustement des marges	67
7.7 Les couleurs sous R	69
8 Périmètres et exportation	76
8.1 Types de périphériques	76
8.2 Les fonctions <code>dev.x()</code>	77
8.3 Exportation d'un graphe	78
9 Partitionnement et composition	80
9.1 Partitionnement basique	80
9.2 Partitionnement avancé	82
9.3 Graphe dans un graphe	85

10 Exercices	88
10.1 Partitionnement avancé	89
10.2 Superposition de graphes	89
10.3 Inclusion en médaillon	90
11 Solutions des exercices	91
11.1 Partitionnement avancé	91
11.2 Superposition de graphes	94
11.3 Inclusion en médaillon	94
III Utiliser ggplot2	95
12 Introduction au package grid	96
12.1 Principes	96
12.2 Examples	96
12.3 Application	96
13 Introduction au package ggplot2	96
14 ggplot2 in action	96
14.1 Principes	96
14.2 Reprendre ce qui a été fait dans la partie 1	96
15 Les extensions de ggplot2	96
16 Graphiques interactives	96
Annexe 1 - courte introduction à R	96
Première commande	97
Pour R, tout est objet	98
Annexe 2 - une revue des packages existant	104
17 Annexe 3 - quel package pour tel graphique ?	105

Bienvenue

Qu'est-ce que ce livre ?

Ce livre est un tour d'horizon des nombreuses possibilités qu'offre le langage de programmation R pour créer des graphiques. Les sources sont disponibles à l'adresse suivante: <https://github.com/inSileco/VisualiseR>.

Cet ouvrage est sous licence Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Ce site internet est déployé GitHub Actions

Future et Collaboration

La première version de ce livre est centré sur faire des graphiques avec les fonctionnalités de base de R mais le paysage R a bien changé au cours des dernières années et beaucoup de packages sont devenus très populaires notamment `ggplot2`. Nous souhaitons augmenter cette version et introduire ces paquets.

Pour d'autres tours d'horizons des capacités graphiques que R offrent, le lecteur pourra se reporter à la lecture de différents ouvrages:

- *Data Visualization* par Kieran Healy <https://socviz.co/>
- *Fundamentals of Data Visualization* par Claus O. Wilke <https://serialmentor.com/dataviz/>
- *R Graphics Cookbook: Practical Recipes for Visualizing Data* par Winston Chang <http://shop.oreilly.com/product/0636920023135.do>
- *ggplot2: Elegant Graphics for Data Analysis* par Hadley Wickham <https://www.springer.com/us/book/9780387981413>

Toutes les contributions sont les bienvenues :smile:. Si vous êtes en dehors du groupe inSileco et que vous avez un compte GitHub, vous pouvez créer une fork ce répertoire et puis créer un pull request à partir de votre *fork*. Vous pouvez aussi directement éditer les fichiers en ligne.

Un chapitre est un fichier `chapitre.Rmd`. Pour en éditer le contenu, l'étendre ou signaler des erreurs, le plus simple est d'ouvrir un *issue* en expliquant ce que vous souhaitez faire ou de détailler l'erreur. Si vous êtes collaborateurs sur le projet, vous pouvez vous assignez la tâche pour signaler au groupe ce sur quoi vous travailler. Deux projets ont été organisés dans le répertoire GitHub (voir *About project boards* pour plus de détails). Pour les utiliser, il suffit de les mentionner au moment de l'ouverture d'un nouvel *issue* est ouvert, on peut en suite les éditer et suivre leur avancement dans l'onglet *project*.

Autres remarques:

- ce livre est édité avec bookdown, la documentation est très utile <https://bookdown.org/yihui/bookdown/>
- `code/chapN/` pour les scripts pour le chapitre N;
- `img/chapN/` pour les figures/images non créés avec R pour le chapitre N;
- `extdata/` pour les données externes;
- ajouter les references dans `book.bib`;
- suivre les bonnes pratiques pour le code: voir *goodpractice*
- donner des noms au *code chunk* afin d'avoir des repères pour faciliter le débogage.

Avant propos

Le logiciel R est un environnement de statistiques *open-source* librement distribué sous les termes de la licence publique générale GNU (GPL2). Très puissant pour réaliser n'importe quel type d'analyses statistiques, il s'avère aussi extrêmement performant dans la visualisation des données. D'ailleurs, dès son apparition au milieu des années quatre-vingt-dix, R était déjà muni d'un module permettant de produire des graphiques.

Utiliser le logiciel R pour produire des graphiques de haute qualité présente un certain nombre d'avantages. Premièrement, chaque composant du graphique peut être modifié, ce qui offre beaucoup de souplesse à l'utilisateur. Deuxièmement, il permet de réaliser l'ensemble du flux de travail (importation de données, manipulation de données, analyses statistiques, représentation graphique et exportation) sur un même support

logiciel. Ce qui nous épargne l'apprentissage de différents outils à usage unique. Enfin, son utilisation va trouver toute sa justification lorsqu'une chaîne de traitements devra être répétée un grand nombre de fois (automatisation des tâches).

Au cours des dernières années, de nombreux packages ont été développés pour faciliter la production de graphiques sous R. Parmi eux, citons le package `lattice` implémenté par Deepayan Sarkar. Ce package s'intéresse spécifiquement à la visualisation de données multivariées. Plus récemment, le package `ggplot2` développé par Hadley Wickham a énormément gagné en popularité dans les laboratoires de recherches. Il repose sur la grammaire des graphiques (*The Grammar of Graphics*), ouvrage de référence écrit par Leland Wilkinson.

Bien que ces packages soient très intéressants, ils présentent l'inconvénient de dépendre d'un certain nombre de packages additionnels. De plus, leur prise en main peut s'avérer difficile puisqu'ils implémentent souvent des méthodes spécifiques, qui dans le cas de `ggplot2` peut s'apparenter à un sous-langage R à part entière. L'idée ici n'est pas de dénigrer de tels outils, qui s'avèrent être tout de même puissants et complets. Non, notre objectif est de fournir les clés nécessaires pour produire de graphiques de haute qualité ne nécessitant aucune retouche supplémentaire via des logiciels comme *GNU Image Manipulation Program (GIMP)*, *Adobe Illustrator* ou encore *Adobe Photoshop*. En d'autres termes, vous apprendrez à réaliser des graphiques prêts à être soumis à une revue scientifique.

Cet enseignement est basé sur l'utilisation du système graphique traditionnel de R : le package `graphics`. Il fait abstraction de tout autre package complémentaire. Le package `graphics` fait partie des packages de base de R. Sa philosophie est à la fois simple et très puissante : n'importe quel graphique peut être généré sans avoir recours à des packages additionnels. Cela a néanmoins un coût : tout est possible, certes, mais avec un nombre de lignes parfois important, nous le concéderons volontiers. Mais, c'est un très bon support pour découvrir l'univers des graphiques et faire connaissance avec leurs éléments constitutifs.

Ce document fait suite à une formation donnée en novembre 2014 à une trentaine d'étudiants gradués de l'Université du Québec à Rimouski. Il est structuré en sept parties. Alors que le premier chapitre illustre les grands types de graphiques réalisables sous R à l'aide des *High-level plotting functions*, le second vous permettra d'éditer un graphe en lui ajoutant des informations avec des *Low-level plotting functions*. Le troisième, probablement le plus long, passe en revue les différents paramètres graphiques. Ainsi, vous apprendrez à jouer avec les couleurs, modifier les marges, les axes, formater une fonte de caractères, etc. Les deux chapitres suivants, un peu plus avancés, vous permettront d'en savoir plus sur les périphériques graphiques et l'exportation de graphes, ainsi que sur la réalisation de graphiques composés (fonction `layout()`). Le chapitre six est constitué de trois exercices que nous vous invitons à essayer de réaliser avant de consulter le code source présent au chapitre suivant.

Malgré ce programme alléchant, ce document est loin d'être exhaustif, loin s'en faut. Mais, nous voulons croire qu'il répondra à certaines de vos interrogations sur les graphiques sous R. Écrire un document sur les graphiques sous-entend que ce-dit document soit richement illustré. Et c'est le cas. Cependant, les graphiques produits par l'ensemble des lignes de code recensées ici ne sont pas tous présentés. Ceci dans un soucis de clarté de lecture, mais aussi de taille de document. C'est pourquoi nous vous invitons à ouvrir une session R en parallèle de votre lecture, et à recopier les lignes de code. Amusez-vous également à modifier certains paramètres pour voir leurs impacts. La connaissance commence par la curiosité.

Nicolas Casajus, Kevin Cazelles

Remerciements

A ajouter.

1 Introduction

Part I

Science visualisation et R

2 Science et graphiques

Ce chapitre se veut une introduction générale au graphique scientifique. Il s'ouvre sur une brève histoire de la visualisation graphique des données. Le lecteur voyagera des premières cartes rudimentaires du Paléolithique jusqu'à l'époque moderne, marquée par la naissance du *dataviz* et de l'infographie. Il n'existe pas pour l'instant de théorie complètement satisfaisante sur le graphique (même si des efforts ont été entrepris depuis la seconde moitié du XX^e siècle), mais nous avons tout de même souhaité fournir au lecteur quelques règles de bon usage dans l'élaboration de graphiques. C'est l'objet de la dernière partie de ce chapitre.

2.1 Une brève histoire du graphique

Notre époque est marquée par une véritable révolution dans notre manière d'appréhender le monde et les choses qui nous entourent. Ce monde numérique dans lequel nous vivons nous permet de quantifier et d'archiver une quantité phénoménale d'informations. Et les récents développements technologiques fournissent des outils sophistiqués pour analyser et visualiser ces données. C'est dans ce contexte que sont apparues l'infographie et la *data visualisation* (également connue sous le terme de *dataviz*). Longtemps confinés dans des disciplines spécialisées, les graphiques ont envahi notre quotidien (presse écrite et en ligne, journaux télévisés, Internet, etc.) au point que certains abus et erreurs sont régulièrement commis, et les bonnes pratiques dans la conception de graphiques, héritées d'une longue évolution, sont purement écartées au profit de l'esthétisme ou de l'expérience visuelle. Ainsi, il est pertinent de se plonger dans l'histoire du graphique afin d'en dégager les principales innovations et règles de bon usage permettant d'atteindre le but principal de tout graphique : transmettre une information fidèle aux observations, de manière la plus simple sans déformer la réalité.

L'un des plus grands spécialistes de l'histoire du graphique est sans aucun doute Michael Friendly, professeur de psychologie à l'université York au Canada. Il est l'auteur de plusieurs ouvrages dédiés à l'histoire de la visualisation de l'information et des données, mais sa contribution la plus notable est son *Milestones in the History of Thematic Cartography, Statistical Graphics, and Data Visualization*¹ (dont l'histoire relatée ici s'est largement inspirée), une chronologie illustrée retracant la longue histoire de la visualisation des données et des principales innovations. Friendly reconnaît plusieurs époques successives (Friendly 2008), mais dans un soucis de synthèse, nous nous limiterons aux quatre époques suivantes :

- les premiers balbutiements, dont la période précède le XVIII^e siècle;
- l'âge d'or, s'étalant du XVIII^e siècle à la fin du XIX^e siècle;
- l'âge sombre, couvrant la première moitié du XX^e siècle;
- la renaissance du graphique, amorcée dès la seconde moitié du XX^e siècle.

2.1.1 Premiers balbutiements

Il serait tentant de croire que le graphique est un procédé récent, mais les premières tentatives sont très anciennes et trouvent leur origine dans la cartographie et l'astronomie, disciplines indispensables à l'exploration

¹<http://datavis.ca/milestones/>

et à la navigation. La plus ancienne carte connue à ce jour date de 13660 calBP² (période du Paléolithique supérieur) et a été découverte dans la grotte d'Abauntz en Navarre espagnole. Il s'agit d'une carte gravée sur une roche et représentant le parcours d'une chasse à venir (ou passée) placé dans le contexte du paysage environnant à la grotte (Utrilla et al., 2009).

Par la suite, on verra apparaître la première carte du monde connu (Anaximandre de Milet, 550 av. J.-C.), ainsi que la première représentation graphique de la longitude dans une carte figurant une Terre sphérique (Ptolémée, années 150 de notre ère). Au cours des siècles suivants, les découvertes successives dans le domaine de la cartographie conduiront à la réalisation du premier atlas moderne en 1570 par le cartographe néerlandais Abraham Ortelius : le *Teatrum Orbis Terrarum*, composé de 53 cartes (Figure ??).

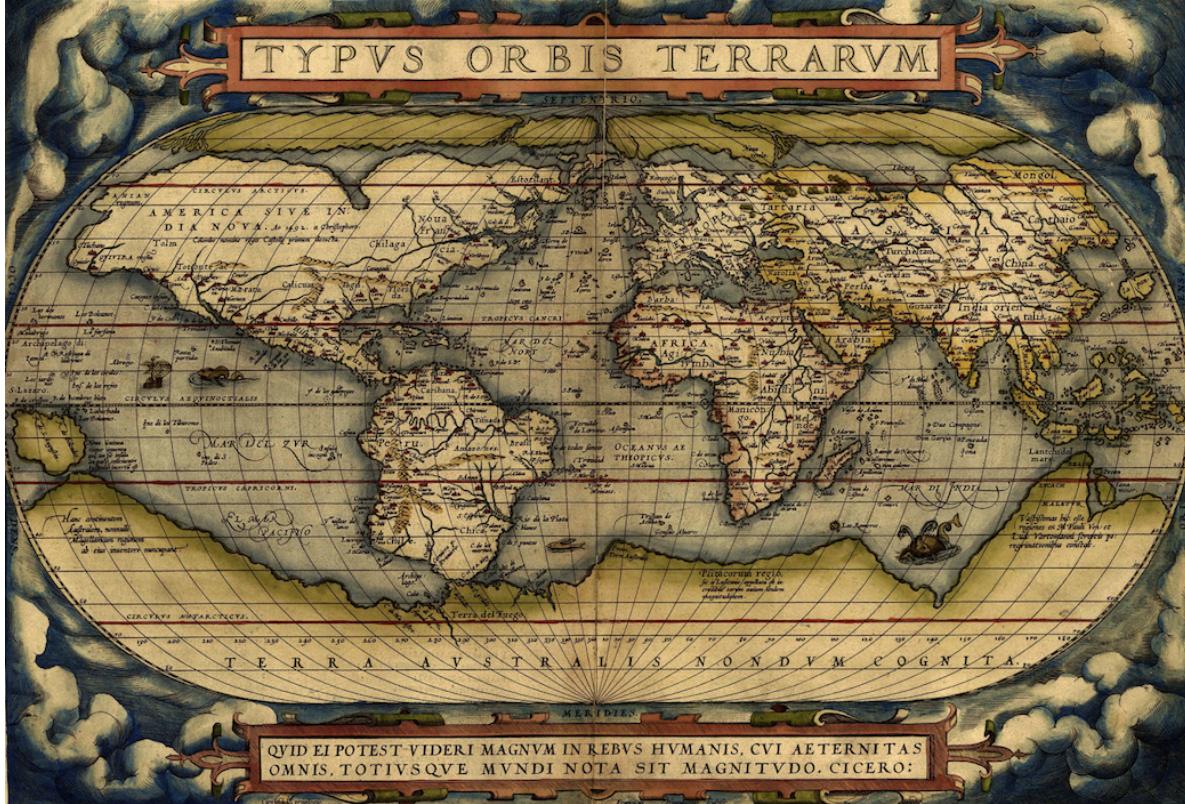


Figure 1: Le *Teatrum Orbis Terrarum* d'Abraham Ortelius (1570, domaine public)

C'est donc avec la cartographie que les premières représentations graphiques sont nées et ceci s'explique facilement : une carte ne requiert qu'un faible niveau d'abstraction (on représente l'espace dans l'espace). Cependant, des tentatives d'abstraction plus poussées ont été entreprises afin de représenter sous forme de graphiques des informations non spatiales. Ainsi, Nicolas Oresme, évêque de Lisieux aux nombreuses activités (économiste, physicien, mathématicien, astronome, philosophe, etc.) fut le premier à représenter sous forme graphique le rapport entre deux variables (vers 1350). Sans le savoir, il fut le précurseur du système de coordonnées, bien avant René Descartes et son repère cartésien (1637). Un siècle plus tard, Nicolas Krebs (également connu sous le patronyme de Nicolas de Cuse), cardinal et grand penseur allemand à qui l'on doit de nombreux écrits astronomiques, conçoit un graphique théorique dans lequel il représente la vitesse en fonction de la distance.

Par la suite, le XVII^e siècle sera marqué par la profusion et l'amélioration des mesures de grandeurs physiques, l'apparition des premiers suivis socio-économiques et le développement de nombreuses théories (géométrie analytique, erreurs et estimations des mesures, statistiques démographiques). La correspondance entre Blaise

²calBP, pour années *Before Present* calibrées

Pascal et Pierre de Fermat sera à l'origine de la théorie des probabilités, dont le premier ouvrage sur le sujet (*De ratiociniis in ludo aleae*) sera publié par Christian Huygens en 1657.

Fait marquant de ce siècle, la première visualisation graphique de données statistiques en 1644. Celle-ci est l'œuvre de Michael Florent van Langren, mathématicien et astronome à la cours du roi Philippe IV d'Espagne ; cette visualisation représente la variation dans la détermination de la longitude entre les villes de Tolède en Espagne et de Rome en Italie.

Ainsi, à la fin du XVII^e siècle, les bases seront en place pour développer le début de la pensée graphique et les nouvelles formes graphiques qui conduiront à l'âge d'or du graphique.

2.1.2 L'âge d'or

Le XVIII^e siècle a vu apparaître de nouvelles formes graphiques afin de répondre aux besoins de l'époque. Les cartographes ont commencé à rajouter des informations non géographiques sur leurs cartes, inventant ainsi les isolignes (lignes de même valeur, Edmond Halley, 1701). Ils s'essayent également à la cartographie thématique et les premières cartes géologiques (Johann Friedrich von Charpentier, 1778), topographiques (Marcellin du Carla-Boniface, 1782) et économiques (August Friedrich Wilhelm Crome, 1782) voient le jour.

Le degré d'abstraction est poussé encore plus loin et, avec le développement des statistiques, la visualisation de fonctions théoriques commencent à se répandre (Thomas Bayes, 1763). De nouveaux suivis sont mis en place, et de nouvelles données sont récoltées conduisant à l'invention d'outils (interpolation, ajustement de courbes) et de nouvelles formes visuelles pour les analyser/représenter. C'est en 1765 que Joseph Priestley, théologien, pasteur, pédagogue et enseignant britannique, à qui l'on attribue la découverte de l'oxygène, inventa l'historiographie moderne au moyen de frises chronologiques (Figure ?). Pour la première fois, il représente le temps par l'espace.

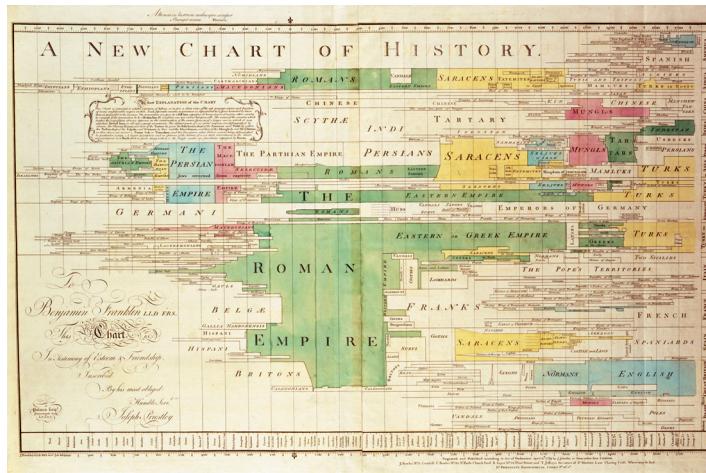


Figure 2: *A new chart of History* (Joseph Priestley, 1765, domaine public)

Un autre personnage majeur de ce XVIII^e siècle est le mathématicien-philosophe Jean-Henri Lambert. Il est le premier à utiliser les graphiques pour analyser des données empiriques et calculer les pentes de courbes. Il a également développé un système de couleurs pyramidal figurant pour la première fois la notion de saturation. Enfin, il est l'un des premiers à reconnaître l'utilité des graphiques : *a diagram does incomparably better service here than a table*.

Cependant, ces différentes innovations graphiques étaient restreintes à quelques publications éparses, et il faudra attendre la première moitié du XIX^e siècle pour que le graphique statistique et la cartographie thématique se démocratisent et connaissent un essor sans précédent. On estime que vers les années 1850, les principales formes graphiques avaient été inventées.

Un des pionniers de cette époque est William Playfair, ingénieur et économiste écossais, à qui l'on doit l'invention de trois graphiques majeurs constituant l'assise de la statistique graphique moderne : le diagramme en barres (1786), probablement inspiré des frises chronologiques de Priestley, le diagramme circulaire, ou camembert (1801), permettant de représenter les proportions relatives des parties au tout, et le graphique linéaire, ou série temporelle (1786), visualisant l'évolution temporelle d'une variable. Ces graphiques sont, encore aujourd'hui, considérés comme des modèles de clarté (Figure ??). En 1876, il publie *The Commercial and Political Atlas*, premier ouvrage illustré de graphiques statistiques. Ces graphiques, outre le fait d'introduire de nouvelles formes visuelles, superposent plusieurs informations permettant d'établir le rapport entre plusieurs variables. Vingt ans plus tard, Playfair publierà le *Statistical Breviary* (1801) qui fit découvrir les premiers diagrammes circulaires.

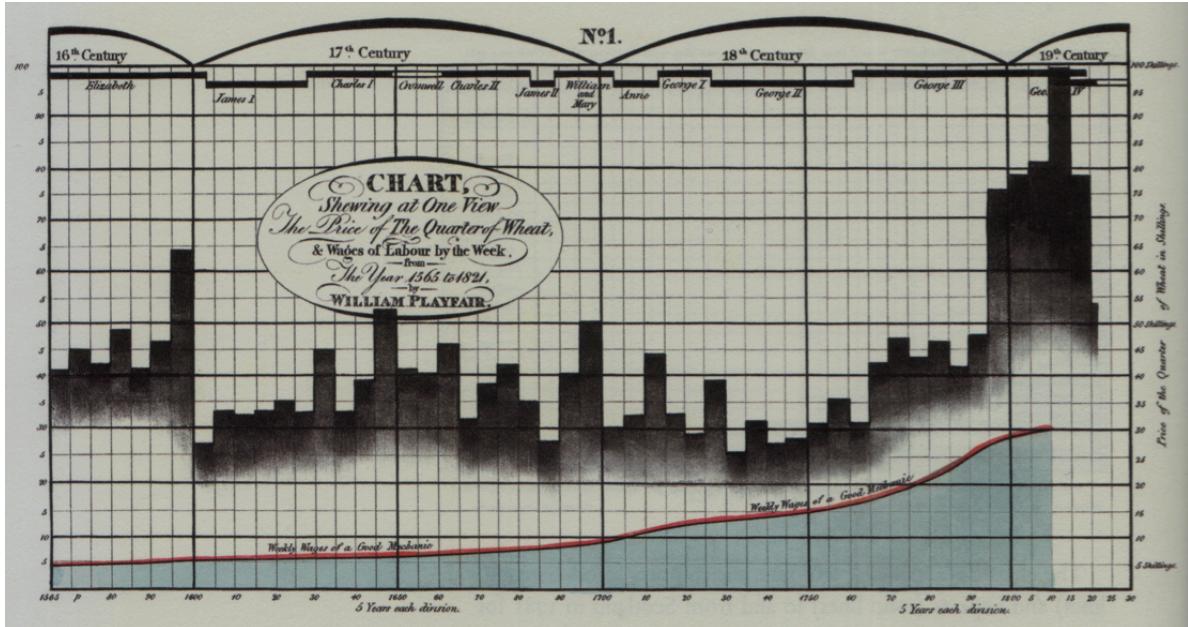


Figure 3: Graphique représentant le prix du blé et le salaire hebdomadaire de 1565 à 1821 (William Playfair, 1821, domaine public)

Le début de ce XIX^e siècle va également être marqué par les travaux du Baron Pierre Charles Dupin, à qui l'on doit la première carte chloroplète (1819) et ceux d'André-Michel Guerry, inventeur du diagramme polaire en 1829, précédant de trente ans les *coxcombs* de Florence Nightingale, souvent reconnue comme étant la première à avoir utiliser cette forme graphique. Guerry, avec Adolphe Quetelet, a énormément contribué à la statistique morale et ses travaux sur la criminologie, largement agrémentés de cartes chloroplètes comparatives, constituent le fondement des sciences sociales modernes.

Dès les années 1820, un nombre croissant de publications scientifiques commence à contenir des graphiques. Cependant, leur usage reste purement descriptif, et il faudra encore attendre plusieurs décennies avant de voir des analyses basées sur les représentations graphiques. Progressivement, la méthode graphique commence à être reconnue dans certains cercles officiels pour ses potentiels dans le domaine de l'économie, du commerce, de l'industrie, de la planification sociale, etc.

Figure de proue de cette seconde moitié du XIX^e siècle, l'ingénieur civil français Charles Joseph Minard qui a su faire preuve d'une grande inventivité, notamment dans la synthèse de multiples informations sur un même graphique. Ainsi, il est l'auteur d'une cinquantaine de cartes sur lesquelles sont ajoutés des symboles proportionnels, des diagrammes circulaires, des flux de marchandises, etc. Mais, il est surtout connu pour sa carte figurative des pertes successives en hommes de l'Armée française dans la campagne de Russie en 1812-1813 (Figure ??), dont Edward Tufte n'hésitera pas à dire qu'il consiste le "meilleur graphique statistique jamais produit". En effet, ce graphique peut être vu comme un modèle de perfection puisqu'il figure sur un même graphique plusieurs variables : l'emplacement et l'itinéraire de l'armée française indiquant les

points de séparation et de ralliement de plusieurs unités, les pertes humaines, la topographie des lieux et la température de l'air.

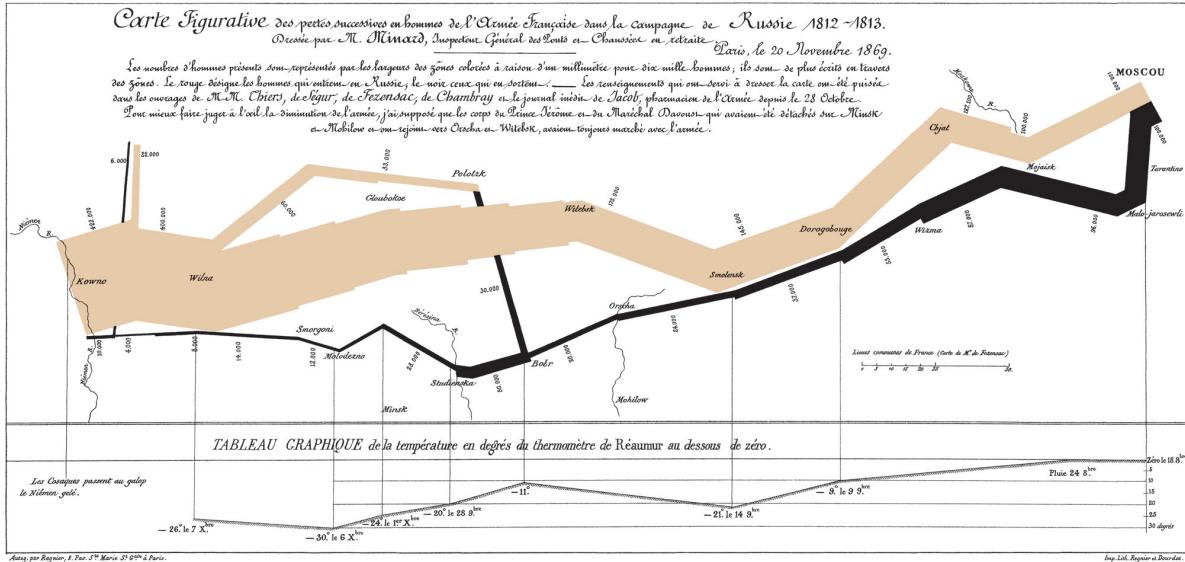


Figure 4: Carte figurative des pertes successives en hommes de l'Armée française dans la campagne de Russie en 1812–1813 (Charles Joseph Minard, 1869, domaine public)

Les innovations vont se poursuivre jusqu'à la fin du XIX^e siècle : carte épidémiologique (Snow, 1855), carte météorologique (Galton, 1861), cartogramme (Levasseur, 1868), diagramme illustrant la notion de corrélation (Galton, 1874), graphique en 3D (Perrozzi, 1880), carte anamorphique (Cheysson, 1888), etc. On assiste à la fin de ce siècle aux premiers inventaires des graphiques : écriture de la *Statistique graphique* par Émile Levasseur en 1885, publication annuelle des *Albums de statistiques graphiques* par le Ministère des travaux publics de 1879 à 1897 sous la direction d'Émile Cheysson, synthèse de l'ensemble des graphiques statistiques connus présentée à la *Statistical Society* de Londres, etc. La fin du XIX^e siècle sera donc caractérisée par une large diffusion des méthodes graphiques dans différents sphères publiques.

2.1.3 L'âge sombre

- L'entrée dans le XIX^e siècle sera...
- Malgré cette profusion d'inventions, certains acteurs restent méfiants face à cette nouvelle méthode de travail, notamment les statisticiens britanniques qui continuent à croire que seuls les tableaux de données permettent de décrire correctement la réalité des phénomènes naturels.
- Le XX^e siècle : Peu d'innovations, mais des tentatives de normalisation et standardisation...

2.1.4 Renaissance

- Époque moderne : transition avec la section “Du bon usage des graphiques”
- Ainsi, la visualisation graphique comme procédé de transfert d'informations et de connaissances a longtemps co-évolué avec d'autres disciplines telles les mathématiques, les statistiques, l'astronomie, la médecine, les sciences sociales, les sciences économiques, le commerce, l'industrie, etc. Et, les principales innovations en la matière ont été permises grâce au développement technologique et à des avancées dans la récolte de données.

2.2 Qu'est-ce qu'un graphique ?

- Une représentation graphique, en version courte graphique, est un résumé visuel de données mesurées (ou recensées) permettant de dégager des tendances ou toute autre information (cycles, agrégation, connexion, etc.) qu'il serait difficile d'observer dans collection de nombres (stockés le plus souvent dans des tableaux). Il existe de nombreux graphiques différents : c'est la nature des données et le type d'analyses souhaité qui orienteront le choix du graphique à utiliser.
- Simplification de la réalité
- À travers l'histoire du graphique brossée dans la section précédente, différents types de representations graphiques ont été mentionnée (carte, diagramme circulaire, en barres, ...{}).

2.3 Un graphique pour quoi faire ?

- exploration vs. présentation

2.4 Du bon usage des graphiques

- Dosage design / données
- Choisir le bon type de représentation (adapté aux données et à la question posée)
- Data-ink ratio; 3D; pie chart; line chart pour données continues
- Titre; repère (axe, étiquette); source des données; symbologie;
- Un graphique doit se suffire à lui-même (donc légende et clarté)

3 À propos de R

Étant donné l'engouement autour de R depuis les 10 dernières années, on oublie que le début de l'histoire remonte au années 70. Dans ce chapitre, nous revenons sur la naissance et le développement de R: de S depuis les années 70 jusqu'à la fin des années 90. Nous décrivons ensuite l'émergence de R et la relation qu'il existe entre S et R. Dans la deuxième partie de ce chapitre, nous dressons un portait de R en 2019.

3.1 De S à R - 50 ans d'histoire

La description qui suit est une synthèse des trois documents suivants:

- Un article d'un des principaux contributeurs de Richard A. Becker (Becker, 1994)
- Le chapitre 2 du dernier livre de John M. Chambers, l'un des principaux contributeurs de S et maintenant contributeur de R (Chambers, 2016)
- Le document *R genesis* par Ross Ihaka, l'un des auteurs de R disponible en ligne https://cran.r-project.org/doc/html/interface98-paper/paper_1.html consulté le 7 février 2019.

3.1.1 Le langage S

Pour bien comprendre l'émergence de R, nous revenons sur le contexte dans lequel le langage S a vu le jour et comprendre ce que R lui doit. S est le produit du travail de chercheurs qui en quête d'un outil performant pour la recherche en statistique dans un monde où l'informatique était en pleine effervescence.

Le résumé que nous faisons de cette émergence débute en 1976, à Murray Hill dans le New-Jersey (États-Unis), au département de recherche en statistique informatique des laboratoires Bell³, les calculs sont alors

³La lecture de la page Wikipédia dédiée au Laboratoires Bell permet de se faire une idée de l'importance des Laboratoires Bell en terme d'innovation, les laboratoires Google de l'époque (https://fr.wikipedia.org/wiki/Laboratoires_Bell, consulté le 31 janvier 2019).

réalisés à l'aide d'une bibliothèque FORTRAN⁴ SCS (*Statistical Computing Subroutines*). Un des problèmes posés par cet outil était le temps passé à gérer les entrées et les sortis au programme qui était long même pour des calculs simples (une régression linéaire par exemple). Suite à ce constat, les chercheurs John Chambers, Richard Becker, Doug Dunn et Paul Tukey, décidèrent de construire un environnement interactif et un nouveau langage afin de faciliter leur recherche⁵. Ce nouveau programme informatique faciliterait l'utilisation des la bibliothèque SCS grâce à un ensemble d'objets et de fonctions manipulés à l'aide d'une syntaxe intuitive et permettrait une exploration visuelle des données à l'aide d'un outil graphique performant.

Suite aux efforts de ces chercheurs, la première version de S (version 1.0) vu le jour en janvier 1977 et fut diffusée au sein des laboratoires Bell. Ainsi, se constitua la toute première communauté d'utilisateurs de S, celles des chercheurs des laboratoires Bell, communauté restreinte mais très active. La première version de S reflète les avancements en statistique informatique fait depuis les années 50 et met en application les principes que John Chambers développe en 1977 (Chambers, 1977) :

1. stockage, manipulation des données avant traitement (les trier par exemple);
2. utiliser des algorithmes efficaces pour traiter les données;
3. visualiser les données avec des bibliothèques adéquates.

L'objet de base de S était alors le vecteur (de nombres, de caractères ou bien de variables booléennes), ce qui reste valable avec R aujourd'hui. Une matrice était aussi un vecteur mais avec un vecteur auxiliaire **Dim** (ces vecteurs auxiliaires deviendront plus tard les attributs accessible avec la fonction **attributes** dans R). Cette version offrait également la possibilité de créer des structures dont les différentes composantes étaient appelées grâce à l'opérateur **\$** (abondamment utilisé par tous les utilisateurs de R aujourd'hui, /de même que la fonction **str** pour avoir un aperçu de l'organisation d'un objet dans R). Ces données étant facilement manipulable via un opérateur d'accès aux différents éléments **[]** (subscripting operator) et permet, entre autre, de sous-échantillonner les éléments à partir d'une variable booléenne (ce que les utilisateurs de R font très souvent). Enfin, les graphiques, sujet de ce livre, étaient réalisés grâce à un ensemble de fonctions implémentées dans une bibliothèque distincte appelée GR-Z dont les fonctionnalités sont encore aujourd'hui présentes dans le package R **graphics**.

L'année suivante, la version 2.0 introduisit notamment les objets de type **list** pour inclure une collection d'autre objets, amenant ainsi un nouveau type de relation entre les objets. En 1978, les progrès du langage de programmation C, développé dans le même bâtiment aux laboratoires Bell, donnèrent un nouvel élan au langage S. Ce langage développé par Dennis Ritchie et Ken Thompson à la fin des années 60 a été conçu pour être puissant, flexible et portable, c'est notamment le langage dans lequel est écrit le kernel Linux qui est un peu partout aujourd'hui⁶. La portabilité de C, c'est-à-dire le fait qu'il soit utilisable sur un très grand nombre d'ordinateurs d'architectures différentes, décida les créateurs du langage S (dont les bureaux étaient voisins de celui de Ritchie) d'opter pour une utilisation du langage C et du système UNIX. C'est ainsi qu'en octobre 1979, une version UNIX fut implémentée et devint la version de développement du langage S. R est lui-même écrit en C.

La suite du développement du langage est liée aux retours des premiers utilisateurs de S. Ces derniers étaient assez limités dans les tâches qu'ils pouvaient accomplir avec S. Afin d'augmenter les possibilités offertes à l'utilisateur, une boucle **for** très flexible fut introduite, dont la syntaxe a été conservé par R : **for (i in index) expression**. De même, une fonction **apply()** fut ajoutée pour répéter une opération donnée sur chaque colonne ou chaque ligne d'une matrice. Il apparut également une interface de langage pour créer des macros⁷. Toutes ces innovations sont contenues dans la version de 1984 décrite dans un livre surnommé le

⁴FORmula TRANslator est un langage apparu en 1954 développé par John Backus chez IBM (<https://fr.wikipedia.org/wiki/Fortran>, consulté le 31 janvier 2019).

⁵*It was the realization that routine data analysis should not require writing Fortran programs that really got S going.* (Becker and Chambers, 1984, p. 2) “Our primary goal was to bring interactive computing to bear on statistics and data analysis problems.”

⁶L'article de Wikipedia est un bon point de départ pour s'informer sur l'histoire de C: [https://fr.wikipedia.org/wiki/C_\(langage\)](https://fr.wikipedia.org/wiki/C_(langage)), consulté le 21 Décembre 2018.

⁷Une macro est formée d'un identificateur et d'un morceau de code. Avant l'exécution du programme, toutes les occurrences de l'identificateur dans le code source seront remplacées par le morceau de code auquel elles sont associées.

*Brown Book*⁸: *S: An Interactive Environment for Data Analysis and Graphics* (Becker and Chambers, 1984). L'année suivante, le livre *Extending the S system* (Becker and Chambers, 1985) décrit comment ajouter de nouvelles fonctions au langage S. C'est au même moment que le code source du langage S fut largement distribué sous licence commerciale et éducationnelle.

3.1.2 Le « new S »

Durant la période 1984–1988, S fut profondément transformé. Alors qu'il était à l'origine dédié à la recherche en statistique, S avait, de fait, de grandes capacités pour manipuler des données, les analyser et les visualiser. Un projet mené par John Chambers fut dédié à l'exploitation de ces atouts pour produire un langage de programmation générale dédié au traitement des données (le *Quantitative Programming Environment*, QPE). En 1986, suite aux avancés du projet, il se posa la question de savoir comment articuler S et QPE. De cette interrogation naquit une troisième version du langage S décrite dans le *Blue Book: The New S Language: A Programming Environment for Data Analysis and Graphics* (Becker et al., 1988). Les changements apportés étaient d'une telle ampleur qu'il fallut changer le nom : S devint le new S. Beaucoup de fonctions furent réécrites en C. Ainsi, la part de FORTRAN diminua au profit de C pour améliorer la portabilité du new S. A cela s'ajouta la fin de l'interface de langage au profit d'un ensemble très cohérent pour les fonctions. Toutes les fonctions devinrent des objets et les utilisateurs purent alors créer leurs propres fonctions avec la syntaxe que nous connaissons aujourd'hui dans R. Ces fonctions pouvaient avoir un nombre non prédéfini d'arguments et certains de ces arguments pouvaient être passés à d'autres fonctions grâce à l'argument Les fonctions écrites en C ou FORTRAN pouvaient être incorporées au new S grâce à des fonctions particulières qui existent toujours dans R: .C(), .Fortran(), etc. Les fonctions graphiques furent également retravaillées et le new S apporta de nouvelles fonctionnalités pour profiter des environnements graphiques, notamment le *X Window System* (X11), apparu en 1984.

En 1992, un nouveau livre, le *White Book: Statistical Models in S* (Chambers and Hastie, 1993), développa les nouvelles fonctionnalités du new S pour travailler sur les modèles statistiques. Les objets de classe **data.frame** furent introduits : des matrices dont les colonnes peuvent contenir des objets de différentes natures (p. ex. numérique, texte, booléen, etc.). C'est un objet qui, à notre sens, est vraiment pertinent pour les statistiques et son utilisation est une des forces de R. Avec les **data.frames** sont introduits les objets de classe **formula** (caractérisés par l'opérateur ~) qui permettent de décrire des modèles complexes en très peu de lignes de code. Enfin, la programmation orientée objet⁹ prend une place plus importante (il y en avait des prémisses en 1988 avec la notamment la fonction **print()**). Cette approche donne une cohérence et une certaine facilité d'utilisation au langage S et encore aujourd'hui au langage R : on peut ainsi afficher tous nos objets avec la fonction **print()** dont l'affichage dépendra du type de l'objet (leur classe) sans se poser de question. Le développement d'une approche orientée objet plus formelle est aussi le point central du dernier livre sur S, le *Green Book: Programming with Data: A guide to the S Language* qui formalise les objets dit S4 (Chambers, 1998).

La création du langage S a soulevé des enjeux techniques majeurs que les développeurs ont surmontés en piochant des idées tantôt dans d'autres langages (p. ex. l'opérateur assignment <- provient du langage Algol), tantôt en trouvant des solutions spécifiques à S (p. ex. l'argument ... de certaines fonctions). Le résultat de la vingtaine d'années de développement de S a produit un langage puissant pour la recherche en statistique mais plus généralement pour l'ensemble des opérations du traitement de données. C'est pour ces travaux que John Chambers fut récompensé en 1998 par le prix ACM (*Association for Computing Machinery*). Au début des années 1990, l'histoire de S se prolongea de deux manières: d'un côté la naissance du logiciel libre R et de l'autre, la création d'une version commercial baptisée S-PLUS proposée par l'entreprise américaine

⁸Les livres décrivant le langage S sont nommés par des noms de couleurs, en rapport avec la couleur de leur couverture voir <http://www.sumsar.net/blog/2014/11/tidbits-from-books-that-defined-s-and-r/> consulté le 31janvier 2018.

⁹La programmation orientée objet définit des objets et des méthodes tout en permettant de faire un lien entre les deux. Par exemple, si des objets de classe *point* et d'autres de classe *ligne* sont définis, une méthode, c.-à-d. une fonction, *plot()* peut ensuite être définie pour afficher ces deux objets dans un environnement graphique. La force de la programmation orientée objet sera de permettre un affichage adéquat : des points séparés à l'appel de la méthode *plot* sur l'objet de classe *point* (*plot(point)*) et des points reliés entre eux à l'appel de la méthode *plot* sur l'objet de classe *ligne* (*plot(ligne)*).

TIBCO, qui prit fin en 2009 quand l'entreprise, au vu de la popularité de R, opta pour le développement d'un outil basé sur R : TERR (*TIBCO Enterprise Runtime for R*)¹⁰.

3.1.3 De S à R

Au milieu des années 1990, au département de statistiques de l'université d'Auckland, Ross Ihaka et Robert Gentleman font à un constat similaire à celui des chercheurs des Bell Labs firent des années plus tôt : le besoin d'un outil logiciel performant pour le domaine. Bien que les premières versions du langage S étaient facilement accessible aux milieux universitaires, l'outil n'était pas libre et difficile à mettre en place sur les machines Macintosh avec lesquelles ces chercheurs travaillaient. Les deux chercheurs se lancèrent alors dans le développement d'un nouvel outil inspirés par le new S mais aussi le Scheme¹¹ comme ils expliquent dans leur article de 1996 (Ihaka and Gentleman, 1996). Ainsi naquit le langage R, fortement influencé par le new S mais avec un soupçonné de quelque principe de Scheme, notamment pour la gestion de la mémoire et de la portée des variables. Il faut ajouter à ces changements, des travaux sur l'interface graphique avec l'implémentation des couleurs, la texture des lignes, la possibilité d'introduire des formules mathématiques sur un graphique et la mise en page des graphiques. Ces deux derniers éléments sont le fait des travaux de thèse de Paul Murrel qui a travaillé sur la gestion des graphiques sous R¹².

En août 1993, les auteurs annoncent la mise à disposition de leur travail R sur la liste de diffusion S-news. Comme il a été fait pour S, R a mis en place le moyen de s'enrichir des retours des utilisateurs. L'un des utilisateurs particulièrement actif, Martin Mächler de l'Eidgenössische Technische Hochschule Zürich (ETH Zürich)¹³, devint un acteur important du projet et incita les chercheurs de Nouvelle-Zélande à rendre leur logiciel libre. C'est ainsi que R fut mis à disposition de tous sous la licence publique générale GNU-GPL dès 1995, d'où son surnom de *GNU S*. L'article de 1996 *R: A Language for Data Analysis and Graphics* qui met au grand jour l'existence de R (Ihaka and Gentleman, 1996). Ces deux événements combinés donnent une impulsion forte au langage R. De fait, ces dispositions allaient accroître le nombre d'utilisateur, et la distribution et la communication allaient en être profondément changées. En mars 1996, Mächler créa la première liste de diffusion spécifique à R : R-tester, qui fut divisée en trois l'année suivante : R-announcement, R-help et R-devel. Pour organiser le développement de R, il devint pressant d'avoir un système d'archivage dédié à R. Ce fut Kurt Hornik de l'université Technique de Vienne¹⁴ qui s'en chargea et mis au point un système qui évoluera pour donner le CRAN actuel. Le nombre de retours sur R devint tel que Ihaka, Gentleman et Mächler ne suffisaient plus pour assurer convenablement le développement du langage. Ils décidèrent alors de partager cette tâche avec Doug Bates, Peter Dalgaard, Kurt Hornik, Friedrich Leisch, Thomas Lumley, Paul Murrell, Heiner Schwarte et Luke Tierney : l'équipe de développement *R-Core*¹⁵ se formait.

Une manière simple et efficace pour prendre conscience de l'évolution de R de 1997 à nos jours est de fouiller dans les archives du site internet du CRAN. Par exemple, dans le fichier *README* de la version R.0.49, on peut lire qu'en avril 1997, une bonne partie des fonctions du *Blue book* (Becker et al., 1988) et du *White Book* (Chambers and Hastie, 1993) étaient déjà implémentées. La reconnaissance formelle du groupe R-Core est présente dans la version R.0.60 de décembre 1997. L'implémentation d'une grande partie des fonctionnalités de S conduisit à la version 1.0.0 de R, le 29 février 2000. Pour l'anecdote, le premier CD-ROM fut donné à John Chambers qui avait rejoint l'équipe du R-Core en 1998. En janvier 2001, c'est la première version d'un journal dédié à R (alors appelé R news) qui fut publiée (voir https://www.r-project.org/doc/Rnews/Rnews_2001-1.pdf, consulté le 12 décembre 2018). En décembre 2001, avec la version 1.4, ce sont les objets S4 qui sont implémentés. Ces derniers étaient décrits dans le *Green Book* (Chambers, 1998) et procurèrent à R un cadre beaucoup plus formel pour faire de la programmation orientée objet. En octobre 2004, la version 2.0 était distribuée, et introduit pour la première fois le *lazy*

¹⁰Voir <https://en.wikipedia.org/wiki/S-PLUS>, consulté le 7 février 2019.

¹¹Scheme est apparu dans les années 1970 au Massachusetts Institute of Technology (MIT) grâce au travail de Guy Lewis Steel et Geral Jay Sussman.

¹²Il est notamment l'auteur de l'ouvrage *R graphics*[ref].

¹³Avec Friedrich Leisch, de la même institution, ils s'étaient rapprochés du projet en apportant des corrections à des bugs.

¹⁴<https://stat.ethz.ch/mailman/listinfo>

¹⁵Ils étaient 11 initialement, 16 en 2002 et 21 en août 2015.

loading (chargement paresseux) permettant d'améliorer les performances de R: lorsqu'un package (extension) est chargé, il est rare que toutes ses fonctionnalités soient exploitées en même temps, R crée des bases de données pour stocker et répertorier le code qui ne sera alors chargé qu'au premier appel de l'utilisateur. En avril 2013, la gestion des valeurs d'indexation des vecteurs supérieurs à 2^{31} pour les systèmes exploitant des processeurs à 64 bits augmenta les performances de R en termes de mémoire et nous amena aux versions 3.x.x. Pour les lecteurs qui cherchent à obtenir des détails techniques relatifs aux fonctionnements internes de R, il est à noter qu'une documentation couvrant ces aspects est disponible en ligne <https://cran.r-project.org/doc/manuals/r-release/R-ints.html> (consulté le 21 décembre 2018). De plus, l'ensemble des changements associés aux différentes versions de R depuis Juin 2009 sont rapportés dans les différents numéros du *R Journal* dont les numéros sont visibles en ligne à l'adresse suivante <https://journal.r-project.org/> (consulté le 21 décembre 2018) et il est possible de remonter jusqu'en 2001 en consultant les différents numéros de *R News* disponible à <https://journal.r-project.org/archive/r-news.html> (consulté le 21 décembre 2018).

Table 1: Date de sortie des différentes version sur le CRAN (voir <https://cran.r-project.org/> pour une liste complète).

Version	Date de publication	Notes
R-0.49	23 avril 1997	première version accessible sur le CRAN
R-1.0.0	29 février 2000	pour l'année millénaire et donc bissextille
R-1.2.1	22 juin 2001	premier volume de R news
R-2.0.0	4 octobre 2004	introduit le <i>lazy loading</i>
R-3.0.0	3 avril 2013	optimization pour les processeurs à 64 bits
R-3.5.0	23 avril 2018	tous les packages sont
R-4.0.0	24 avril 2020	pour les 20 ans de R
R-4.0.0	24 avril 2020	version utilisée pour ce livre

3.2 R aujourd'hui

3.2.1 Mais R c'est quoi au juste?

Les premières phrases de l'article en français consacré à R sur Wikipedia définissent R ainsi:

R est un langage de programmation et un logiciel libre destiné aux statistiques et à la science des données soutenu par la R Foundation for Statistical Computing. R fait partie de la liste des paquets GNU3 et est écrit en C (langage), Fortran et R.¹⁶

De même sur la page du projet R, on peut lire:

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.¹⁷

[Traduction] R est un environnement logiciel pour la statistique computationnelle et les graphiques. Il peut être compilé et exécuté sur un grand nombre de plateforme UNIX, Windows et MacOS.

R est donc un langage: permet d'écrire du code (*en R*) mais aussi en logiciel qui permet d'exécuter du code R (*dans R*) pour faire des statistiques et des graphiques. R est un langage dit interprété.

Par opposition à un langage compilé comme C (le langage dans lequel R est écrit).

¹⁶[https://fr.wikipedia.org/wiki/R_\(langage\)](https://fr.wikipedia.org/wiki/R_(langage)) consulté le 22 décembre 2018.

¹⁷[https://fr.wikipedia.org/wiki/R_\(langage\)](https://fr.wikipedia.org/wiki/R_(langage)) consulté le 22 décembre 2018.

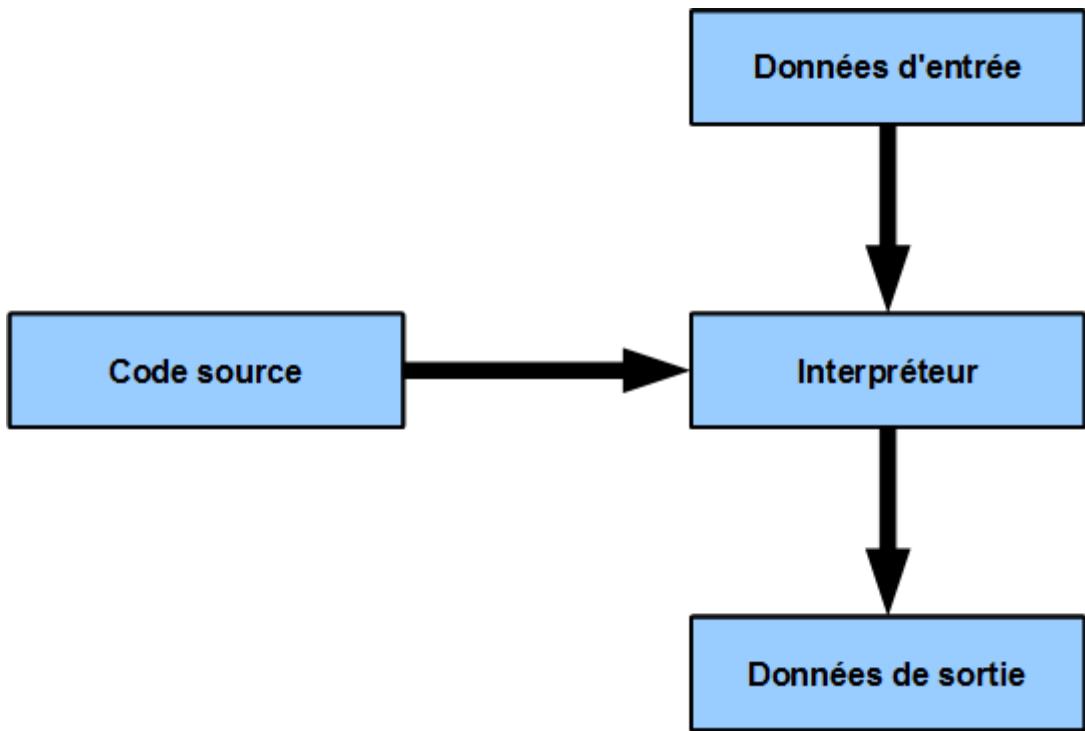


Figure 5: Langage interprété (voir <http://www.france-ioi.org/algo/course.php?idChapter=561&idCourse=2368>)

Concrètement R est un ensemble de fichiers. Dans la version 3.5.2, il y, entre autre, 559 fichiers .c, 289 fichiers .po (pour la traduction en différente langues), 196 fichiers .h (headers utilisé avec C), 86 fichiers .f (FORTRAN) et 2346 fichiers .R. À l'installation de R, tous ces fichiers sont compilés de sorte que vous pouvez utiliser R dans un terminal.

```

> R [09:01:31]

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>> KevCaz - 22/12/18 09h01min

```

Un terminal avec R permet d'utiliser toutes les fonctionnalités de R mais n'est suffisant pour programmer de manière efficace. C'est pourquoi on utilise une interface de développement, R propose une telle interface

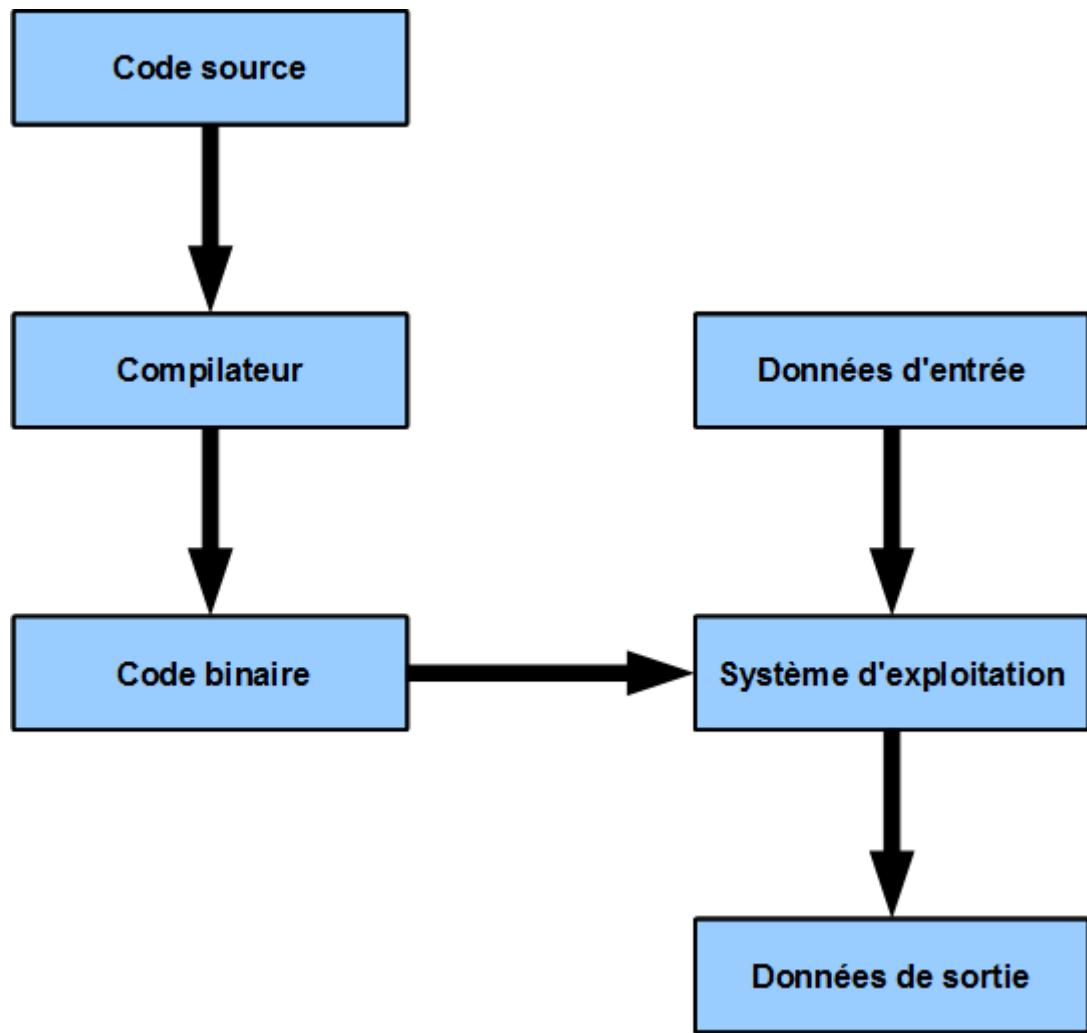


Figure 6: Langage compilé (voir <http://www.france-oi.org/algo/course.php?idChapter=561&idCourse=2368>)

mais la plus populaire de nos jours est RStudio¹⁸, nous y reviendrons.

3.2.2 Un ensemble d'acteurs

Il y a aujourd'hui une myriade d'acteurs qui gravitent autour de R de sorte qu'il est difficile de s'y retrouver et de comprendre qui fait quoi. Ce qui suit est une tentative de mettre un peu d'ordre dans toutes ces informations.

3.2.2.1 La R fondation et la *R Development Core Team*

Le plus simple pour se repérer est de consulter sur la page du projet R: <https://www.r-project.org/> (consulté le 23 décembre 2018). Sur cette page, on trouve notamment un lien vers la Fondation R¹⁹. Il s'agit d'une organisation à but non lucratif qui a été créée par les membres de la *R Development Core Team* pour pérenniser le développement de R et c'est finalement une distinction. Les membres de la fondation sont des personnes reconnus pour les efforts dans le développement de R, c'est une distinction honorifique. Quand aux membres de la *R Development Core Team*, ce sont les développeurs qui ont un accès d'écriture au code source de R. Ce sont les premiers membres listés quand vous tapez `contributors()` dans une console R. Ils fixent les bogues dans les fonctionnalités de base de R²⁰ et rendent disponibles les nouvelles versions de R sur le CRAN.

3.2.2.2 Le CRAN

Toujours sur la page du projet R, il y a un lien vers le CRAN (*Comprehensive R Archive Network*) qui est le site Internet où sont archivées les différentes versions de R ainsi que les packages qui ont été soumis au CRAN et qui ont été acceptés. C'est également une source privilégiée de ressources avec de nombreux manuels et ouvrages pour se familiariser avec R (voir les sections *Manuals* et *Contributed* sur <https://cran.r-project.org/>). Le CRAN est maintenu par une équipe de volontaires, la "CRAN team", qui met à jour le site et s'occupent des soumissions des packages. Pour être plus exact, le CRAN est une collection de sites miroirs dont les contenus sont synchronisés avec le site du CRAN maintenu à la Wirtschaftsuniversität à Vienne. En 2002, il existait 12 sites miroirs, et en décembre 2018, 164 sites miroirs se répartissent les téléchargements de R et de ses packages, ainsi que l'archivage des différentes versions. Comment avoir accès à ces chiffres? Avec R bien sur:

```
# lit le code source de la page qui référence les miroirs
mir <- readLines("https://cran.r-project.org/mirrors.html")
# compte le nombre de ligne qui contiennent des liens internet avec
# http dans le lien (il y a d'autres liens sur la page mais sans http)
sum(
  unlist(
    lapply(mir, grepl, pattern = "<a href=\"http\"")
  )
)
```

3.2.2.3 Le consortium R

Une autre entité qui a vu le jour récemment est le consortium R : <https://www.r-consortium.org/> (consulté le 24 décembre 2018). Comme indiqué par le site Internet dédié

The central mission of the R Consortium is to work with and provide support to the R Foundation and to the key organizations developing, maintaining, distributing and using R software through the identification, development and implementation of infrastructure projects.

¹⁸<https://www.rstudio.com/products/rstudio/download/> consulté le 24 décembre 2018.

¹⁹<https://www.r-project.org/foundation/> consulté le 23 décembre 2018

²⁰Les bogues peuvent être rapportés par les utilisateurs sur bugzilla <https://bugs.r-project.org/bugzilla3/>.

[Traduction] La mission centrale du consortium R est de collaborer avec la Fondation R et les organisations majeures qui développent, maintiennent, distribuent et utilisent le logiciel R à travers l'identification, le développement et l'implémentation de projets infrastructure.

Ce projet est l'un des projets Open Source soutenus par la Fondation Linux: <https://www.linuxfoundation.org/projects/> (consulté le 22 décembre 2018). Pour voir les soutiens financiers du consortium R rendez-vous sur cette page <https://www.r-consortium.org/members>.

3.2.2.4 RStudio

Il s'agit d'un acteur tellement connu qu'il y a une parfois une confusion entre R et RStudio. RStudio, Inc.²¹ une entreprise fondée par Joseph J. Allaire qui développe un environnement de développement²² pour R et propose des services aux entreprises basés sur R (voir la section *Products* sur leur site Internet). En décembre 2018, RStudio comptait près d'une centaine de salariés <https://www.rstudio.com/about/> dont de nombreux programmeurs parmi les plus actifs au sein de la communauté des utilisateurs de R (Hadley Wickham et Yihui Xie par exemple). RStudio est connu pour son interface de développement du même nom et pour la qualité des packages R que les salariés de RStudio développent. C'est un modèle économique classique dans le monde de l'open source²³: une entreprise propose des services autour d'un logiciel qu'elle contribue activement à améliorer et dont elle fait une promotion active (étant donné que c'est la qualité du logiciel qui va déterminer son nombre de clients). La contribution de RStudio à R est considérable, les packages qu'elles contribuent à développer sont parmi les plus utilisés²⁴.

3.2.2.5 rOpenSci

Sur le site Internet de rOpenSci²⁵, on peut lire:

rOpenSci fosters a culture that values open and reproducible research using shared data and reusable software.

[Traduction] rOpenSci encourage une culture qui promeut la recherche ouverte et réproductible qui utilise des données partagées et des logiciels réutilisables.

Pour cela l'équipe de rOpenSci crée et révise des packages²⁶ qui facilitent la recherche ouverte et réproductible. Il peut s'agir de packages qui interrogent des bases de données accessibles en ligne via des interfaces spécifiques, un exemple emblématique de cette catégorie est **taxize**²⁷, véritable couteau suisse pour obtenir des informations taxonomiques ou encore de packages qui améliorent le flux de travail comme **drake**²⁸. rOpenSci anime également un blog²⁹ sur lequel on trouve des posts présentant les différents packages révisés par cette organisation et des tutoriels pour apprendre à réaliser certaines manipulations de données avec R. Enfin, rOpenSci anime une communauté³⁰ dynamique et propose des *community calls*, c'est-à-dire des présentations sur certains aspects de R que l'on peut joindre en direct ou consulté l'archive associée³¹.

²¹<https://www.rstudio.com/> consulté le 24 décembre 2018.

²²https://fr.wikipedia.org/wiki/Environnement_de_developpement consulté le 24 décembre 2018.

²³<https://medium.com/france/business-model-de-l-open-source-a2d8e53181f7> consulté le 24 décembre 2018

²⁴voir le compte GitHub de RStudio <https://github.com/rstudio> (consulté le 24 décembre 2018)

²⁵<https://ropensci.org/about/> consulté le 26 décembre 2018 ou on apprend aussi que rOpenSci est sponsorisé par NUMFOCUS (<https://numfocus.org/>).

²⁶<https://ropensci.org/packages/> consulté le 26 décembre 2018.

²⁷voir <https://github.com/ropensci/taxize> et https://ropensci.org/tutorials/taxize_tutorial/ consultés le 26 décembre 2018.

²⁸<https://github.com/ropensci/drake> consulté le 26 décembre 2018.

²⁹<https://ropensci.org/blog/> consulté le 26 décembre 2018.

³⁰<https://ropensci.org/community/> consulté le 26 décembre 2018.

³¹<https://ropensci.org/commcalls/> consulté le 4 janvier 2019.

3.2.2.6 La communauté des utilisateurs de R

La myriade des utilisateurs de R est sans aucun doute l'acteur le plus important car cette communauté :

1. augmente les fonctionnalités de R en créant des packages;
2. échanges des informations relative à R par la création et l'édition de manuels, de livres, de blogs, etc.;
3. s'auto-organise à travers des communautés locales d'utilisateurs;
4. s'auto-supporte: les utilisateurs répondent rapidement aux questions d'autres utilisateurs.

Tout le monde peut écrire des packages ou partager son expérience avec R et de fait, beaucoup d'utilisateurs le font. Il existe un grand nombre de bloggers qui produisent une multitude de posts sur différents aspects de R³². C'est une ressource riche et diversifiée et il existe une plateforme très utile qui les rassemble³³: R-bloggers³⁴. De nombreux indices attestent que la communauté des utilisateurs de R est en croissance. Par exemple, le nombre de groupes d'utilisateurs de R est en très forte hausse comme le montre la couverture du dernier livre de John Chambers qui compare le nombre de groupe d'utilisateurs de R entre 2010 et 2015.

On peut aussi regarder la croissance du nombre de packages qui dépassait 10,000 en 2016 (voir <https://blog.revolutionanalytics.com/2017/01/cran-10000.html> consulté le 26 décembre 2018). L'explosion du nombre de packages R hébergés sur GitHub³⁵ (près de 45000 à la fin décembre 2018 d'après <https://rdrr.io/>). Un autre indice est la popularité du tag "R" sur la plateforme d'échange *stack overflow* où les utilisateurs posent des questions auxquelles répondent d'autres utilisateurs. C'est d'ailleurs la popularité sur Github et sur stack overflow que le site internet redmonk évaluent la popularité des langages de programmation et en janvier 2018, R était classé 12^{ème}.

Un dernier indice de l'accroissement de la popularité de R chez les écologistes que nous sommes est la forte demande en formation de nos collègues. R figure incontestablement parmi les outils computationnels les plus utilisés pour la statistique et la science des données de manière générale. De plus en plus d'écologues apprennent R avec un niveau de formation de de plus en plus avancé.

3.2.3 Une documentation abondante et facilement accessible

Les acteurs mentionnés ci-dessus précédemment produisent une documentation abondante aussi bien pour débuter avec R, réaliser des tâches spécifiques avec R (au moyen de packages dédiés) que pour programmer efficacement avec R et développer ses propres packages³⁶.

Pour les débutants, un moyen rapide de mettre le pied à l'étrier est de consulter les sections *Manuals*, *Books* de la page Internet du projet R (<https://www.r-project.org>) et la section *Contributed* qui propose des introductions à R dans différentes langues. Cette page n'est malheureusement plus mise à jour mais on y trouve, par exemple, l'*Introduction à R* de Julien Barnier (voir <http://alea.fr.eu.org/pages/intro-R>). On peut aussi citer des acteurs de formation très populaires qui offrent des formations dédiés à R. On peut citer la formation d'introduction à R de DataCamp (en anglais, <https://www.datacamp.com/courses/free-introduction-to-r>, consulté le 28 décembre 2018) mais aussi celle de OpenClassrooms (en français, malgré le nom, <https://openclassrooms.com/fr/courses/1393696-effectuez-vos-etudes-statistiques-avec-r>, consulté le 28 décembre 2018).

De la documentation en ligne, gratuite et de qualité est aussi disponible pour se former à des tâches plus spécifiques. Le site bookdown.org³⁷ propose des livres disponibles, par exemple *Advanced R* de Hadley Wickham y est répertorié (voir <https://adv-r.hadley.nz/>). Notons que RStudio référence aussi certains ouvrage de leur membre <https://www.rstudio.com/resources/training/books/>. Le site RStudio propose aussi

³²Nous avons notre propre blog qui traite souvent de R <https://insileco.github.io/>.

³³https://blog.feedspot.com/r_programming_blogs/ pour une classification subjective.

³⁴<https://www.r-bloggers.com/> consulté le 26 décembre 2018.

³⁵un site Internet qui peut être décrit comme une forge logiciel basée sur le logiciel git qui rend la programmation collaborative particulièrement efficace.

³⁶nous maintenons une liste de resource à l'adresse suivante: <https://insileco.github.io/wiki/usefulr/>.

³⁷<https://bookdown.org/> consulté le 28 décembre 2018.

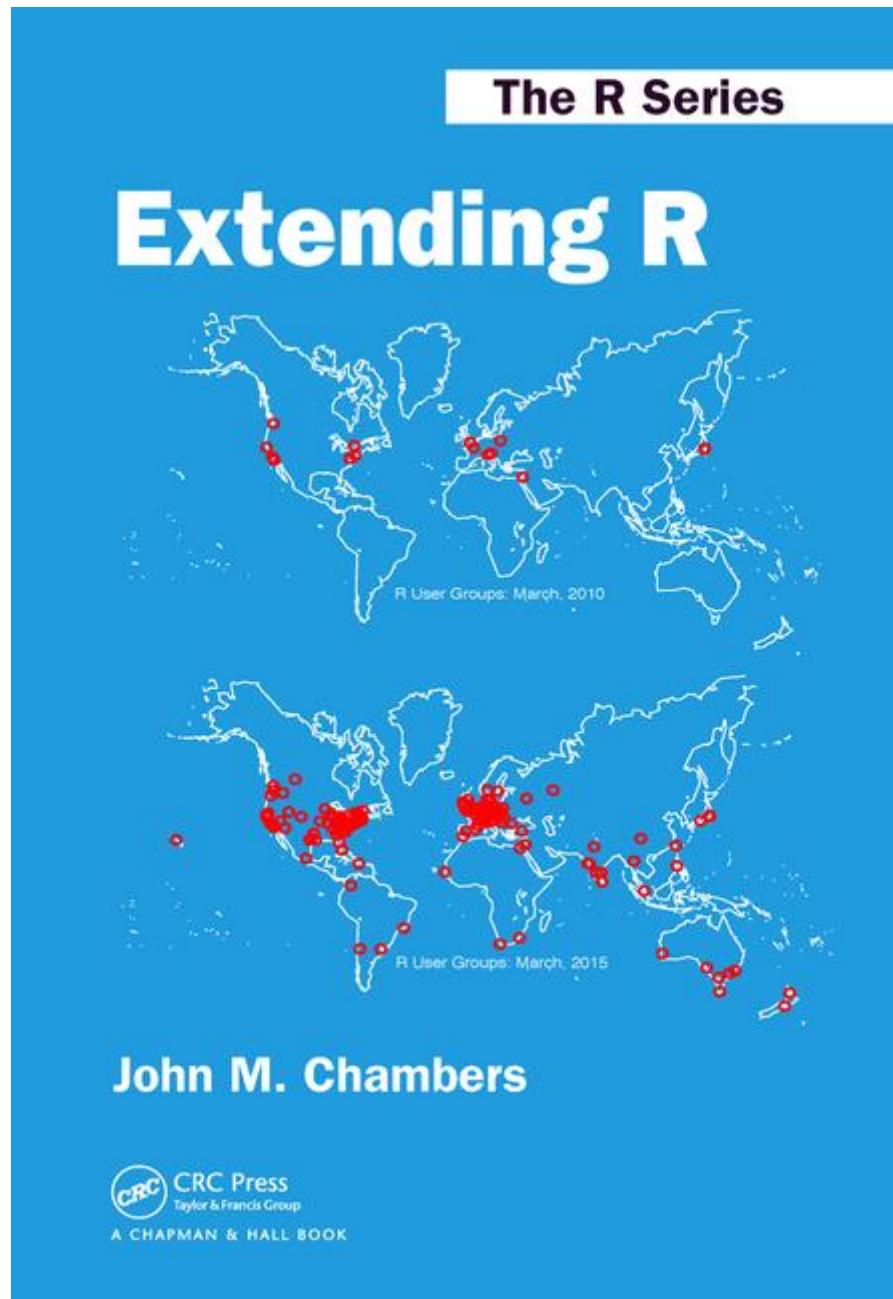


Figure 7: Couverture de *Extending R* (Chambers, 2016). Pour une liste qui recense ces communautés, voir <https://jumpingrivers.github.io/meetingsR/index.html> (consulté le 26 décembre 2018)

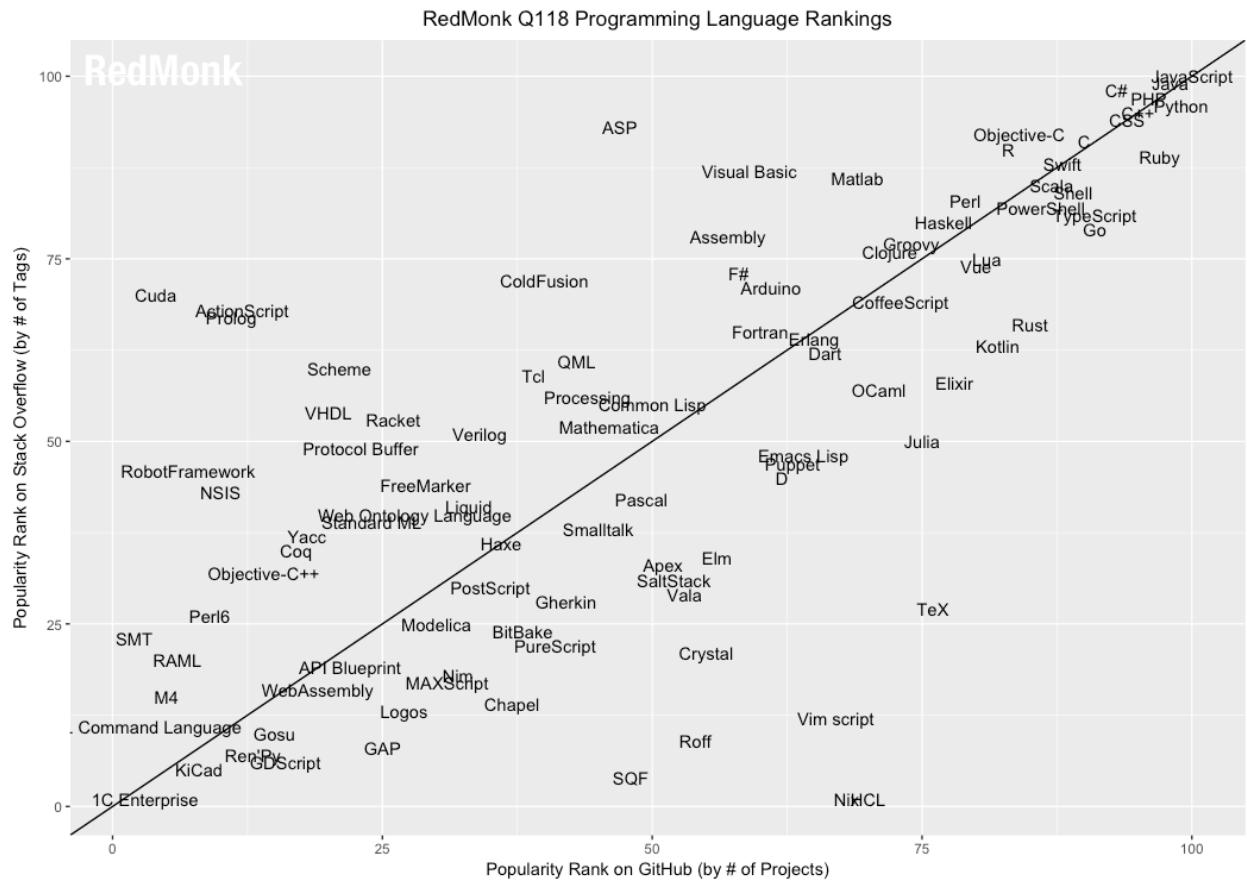


Figure 8: Le résultat du sondage redmonk en janvier 2018, voir <https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/> consulté le 26 décembre 2018.

des *cheatsheet* (des fiches récapitulatives, voir <https://www.rstudio.com/resources/cheatsheets/>). A cela, il faut ajouter la longue listes des post de différents blogues et les réponses aux questions sur les forums³⁸ sur *stack overflow*.

Les utilisateurs anglophones ont facilement accès à une masse considérable d'informations grâce à différents ouvrages, notamment ceux regroupés dans la collection *Use R!*, publiée par le groupe éditorial Springer³⁹ et éditée par R. Gentleman, K. Hornik Kurt et G. Parmigiani. Il y a aussi un certain nombre d'ouvrage publié chez CRC Press⁴⁰ and chez O'Reilly Media⁴¹. Des ouvrages en français sont également disponible mais en nombre plus restreint. Gaël Millot a ainsi écrit un ouvrage très complet (plus de 800 pages dans la troisième édition), *Comprendre et réaliser les tests statistiques à l'aide de R* édité chez de Boeck (Millot, 2018). Springer a édité durant un temps des ouvrages dans la série *Pratique R*. Il existe aussi trois ouvrages qui abordent la statistique avec R aux Presses Universitaires de Rennes et *R et espace* édité chez Framabook (ElementR, 2014).

The R journal, que nous avons mentionnés plus haut⁴², est une revue bisannuelle dédiée à l'actualité de R. Un numéro est un ensemble d'articles qui présentent les possibilités offertes par certains packages ou qui propose une synthèse des différentes packages sur un thème précis. On y trouve également un rapport exhaustif détaillant les modifications apportées aux nouvelles versions de R ainsi que les changements sur le CRAN.

L'ensemble des listes de diffusion relatives à R est aussi une mine d'information. En 1997, il y avait trois listes de diffusion : *R-announcement* pour annoncer les nouveautés de R, *R-help* dédiée à l'entraide entre utilisateurs R et *R-devel* pour les développeurs. Depuis, ont été ajoutées les listes *R-package-devel*, pour les développeurs de packages et *R-packages* pour annoncer la sortie de nouveaux packages. À côté de ces listes principales, on trouve également pas moins de 20 autres listes de diffusion couvrant les différentes utilisations de R : les listes *R-SIG* (*Special Interest Group*) qui traitent aussi bien de la manipulation des données spatiales que de l'écologie ou encore de la finance (<https://www.r-project.org/mail.html> consulté le 28 janvier 2018).

Utiliser Twitter est une manière efficace de découvrir les possibilités qu'offrent R et de trouver de nouvelles ressources. Les tweets relatifs à R contiennent généralement le hashtag **#rstats**. Il y a de nombreux membres actifs de la communauté R qui partagent quotidiennement des informations. Voici quelques comptes twitter avec des informations et des astuces bien utiles :

- @_R_Foundation tweets de la fondation R;
- @RConsortium tweets du consortium R;
- @rstudio compte principal de RStudio;
- @rOpenSci tweets de rOpenSci;
- @RLangPackage tweet à propos de 1 package par jour;
- @CRANberriesFeed un compte qui rapporte les mise à jour de packages;
- @RLangTip donne des astuces quotidiennement;
- @daily_r_sheets tweet une fiche récapitulative sur un package par jour
- @dataandme compte de Mara Averick, employée apr RStudio, qui partage de nombreuses resources.

Enfin, une autre manière d'enrichir ses connaissances relatives à R est de participer à l'une des conférences et autre rencontre sur R où se retrouvent les différents acteurs. La R foundation soutient activement deux conférences *Use R!* et *DSC* (Directions in Statistical Computing)⁴³. La première est présentée comme un forum pour la communauté des utilisateurs de R et la seconde est orientée sur la recherche en statistique computationnelle et donc ne concerne pas exclusivement R et est plutôt destinée à un public de chercheurs.

³⁸Tel que celui maintenu par le Centre de coopération internationale en recherche agronomique pour le développement (CIRAD) <http://forums.cirad.fr/logiciel-R/index.phpm> consulté le 28 janvier 2019.

³⁹<https://www.springer.com/series/6991> consulté le 28 janvier 2019.

⁴⁰<https://www.crcpress.com/> consulté le 28 janvier 2019.

⁴¹<https://www.safaribooksonline.com/library/publisher/oreilly-media-inc/> consulté le 28 janvier 2019.

⁴²<https://journal.r-project.org/> consulté le 30 janvier 2019.

⁴³<https://www.r-project.org/conferences.html> consulté le 30 janvier 2019.

A ces deux conférences, il faut ajouter deux autres rencontres annuelles: la *rstudio::conf* de RStudio⁴⁴, et la *unconference* (littéralement la non conférence) rOpenSci⁴⁵. Enfin, il y a des conférences qui à l'initiative de membres actifs de la communauté R comme par exemple *R à Québec*⁴⁶ et aussi de nombreuses rencontres de groupe d'utilisateurs⁴⁷.

3.2.4 Les packages

Un package (ou librairie) est un ensemble de fichiers que l'on ajoute à R pour en étendre les fonctionnalités. Quand on installe R, par défaut, on installe l'ensemble des fichiers qui permettent d'écrire en R et d'interagir avec, mais également des fonctionnalités de base regroupées en deux ensembles de packages: 15 packages dits de base (p. ex. le package **graphics**) et 15 packages recommandés (dont le package **nlme** qui permet de faire des modèles mixtes)⁴⁸. L'utilisateur peut par la suite ajouter des fonctionnalités supplémentaires pour étendre les possibilités offertes par R en téléchargeant depuis un serveur qui héberge des packages R, par exemple depuis le CRAN.

Un package valide est un ensemble de fichiers structuré et qui passe à un certain nombre de tests⁴⁹. En général, un package regroupe un ensemble de fonctions réunies sous une même thématique. Par exemple, on trouvera des packages pour faire des modèles mixtes, de la géostatistique, des graphiques, etc. Les packages *jpeg* et *png* proposés par Simon Urbanek sont chacun composé de deux fonctions permettant d'importer et d'exporter des fichiers au format jpeg et png. Le package *plotrix*, maintenu par Jim Lemon, est une collection d'environ 160 fonctions (dans sa version 3.7-4) qui étendent les fonctionnalités graphiques du package *graphics*. Le package *ggplot2*, maintenu par Hadley Wickham, est une implémentation de *La Grammaire des Graphiques* développée par Leland Wilkinson. Ce package est d'une certaine manière plus qu'une simple collection de fonctions, il développe des outils et des méthodes qui changent profondément l'approche de la création de graphique et d'ailleurs de nombreux packages l'utilisent pour ajouter de nouveaux types graphiques. Comment trouver un package sur le CRAN ? En générale c'est en lisant des articles ou des livres sur R qu'on apprend l'existence de tel ou tel packages. Cela étant dit, sur le CRAN sont disponibles des *task view* (<https://cran.r-project.org/web/views/> consulté le 5 février 2018) qui dresse un paysage des packages disponibles pour un thème donné⁵⁰. Il existe aussi des initiatives individuelles sur GitHub qui rassemblent et classifient une grande quantité de packages sous forme de listes⁵¹. Enfin, il y a le site METACRAN qui permet une recherche efficace parmi les packages disponible sur le CRAN.

Combien de packages existent-t-il actuellement ? Dans le numéro de décembre 2009 du *The R journal*, John Fox relate la croissance exponentielle du nombre de packages R (Fox, 2009). D'une centaine de packages au début des années 2000, on arrive à plus de 7000 en août 2015; la barre des 10000 packages a été franchie en 2017 et en février 2019, plus de 13600 packages sont disponibles sur le CRAN⁵².

À ce nombre de packages viennent s'ajouter près de 1700 packages disponibles sur Bioconductor et des dizaines de milliers disponibles sur différentes forges logiciel : R-forge, omegahat, GitHub, bitbucket. Le site R Package Documentation rassemble la documentation des packages disponibles sous différentes plateformes.

Comme le soulignait déjà Kurt Hortsik en 2009 (?), cette abondance de packages pose un certain nombre de questions relatives à la facilité de trouver la fonctionnalité désirée, la redondance des fonctions et la qualité du code. Les tests auxquels sont soumis les packages soumis au CRAN ne couvrent pas complètement ces deux derniers aspects. Selon Kurt Hortsik, en dépit des défis posés par le nombre croissant de packages développés, une politique plus contraignante n'est pas souhaitable, et quand bien même celle-ci serait mise en

⁴⁴<https://www.rstudio.com/conference/> consulté le 30 janvier 2019.

⁴⁵<http://unconf18.ropensci.org/> consulté le 31 janvier 2019.

⁴⁶<http://ra Quebec.ulaval.ca/2019/> consulté le 30 janvier 2019.

⁴⁷Voir <https://jumpingrivers.github.io/meetingsR/index.html>, consulté le 31 janvier 2019.

⁴⁸Voir <https://stackoverflow.com/questions/9700799> et <https://cran.r-project.org/src/contrib/3.5.2/Recommended/> consultés le 31 janvier 2019.

⁴⁹Pour les détails techniques voir *Writing R extensions* <https://cran.r-project.org/doc/manuals/r-release/R-exts.html>, consulté le 31 janvier 2018.

⁵⁰Le package *ctv* permet d'installer tous packages d'un thème particulier.

⁵¹Voir par exemple <https://github.com/qinwf/awesome-R>, consulté le 7 février 2019.

⁵²Le nombre exact est indiqué en haut de la page <https://cran.r-project.org/web/packages/> consulté le 5 février.

METACRAN Packages Authors Services About Search for packages

METACRAN: Search and browse all CRAN/R packages

15,674 active packages

8,835 package maintainers

344 updates last week

42,013,105 downloads last week

Featured packages

xgboost

Extreme Gradient Boosting
1.0.0.2, published 2 months ago, by [Tong He](#)

h2o

R Interface for the 'H2O' Scalable Machine Learning Platform
3.30.0.1, published a month ago, by [Erin LeDell](#)

dplyr

A Grammar of Data Manipulation
0.8.5, published 2 months ago, by [Hadley Wickham](#)

prophet

Automatic Forecasting Procedure
0.6.1, published 13 days ago, by [Sean Taylor](#)

ggplot2

Create Elegant Data Visualisations Using the Grammar of Graphics
3.3.0, published 2 months ago, by [Hadley Wickham](#)

feather

R Bindings to the Feather 'API'
0.3.5, published 8 months ago, by [Hadley Wickham](#)

mlflow

Interface to 'MLflow'
1.8.0, published 20 days ago, by [Matei Zaharia](#)

shiny

Web Application Framework for R
1.4.0.2, published 2 months ago, by [Winston Chang](#)

data.table

Extension of `data.frame`
1.12.8, published 5 months ago, by [Matt Dowle](#)

Figure 9: Haut de la page principale du *METACRAN* <https://www.r-pkg.org/> consultée le

Number of R packages ever published on CRAN

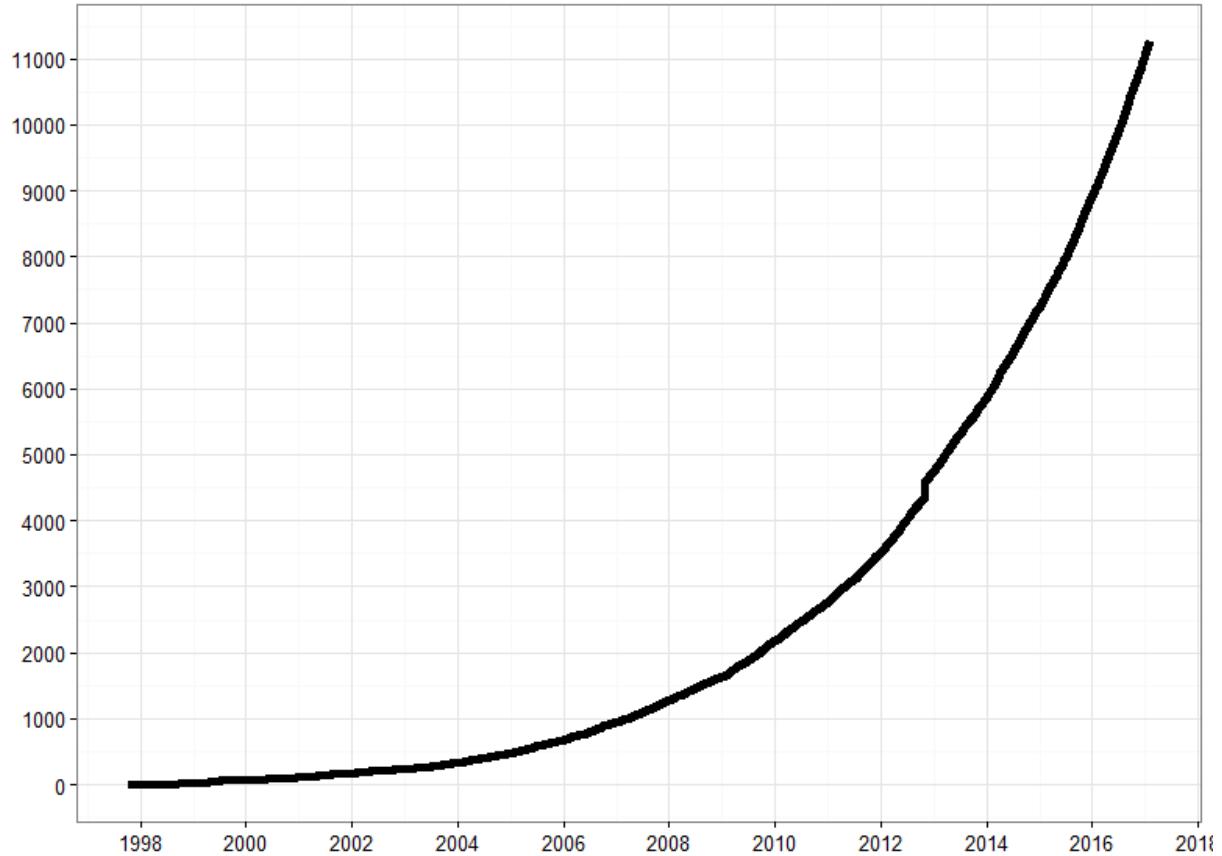


Figure 10: Croissance des packages R disponibles sur le CRAN - article de publié par David Smith sur le blog *Revolutions* <https://blog.revolutionanalytics.com/2017/01/cran-10000.html>, consulté le 5 février 2019

R Package Documentation

A comprehensive index of R packages and documentation from CRAN, Bioconductor, GitHub and R-Forge.

Search for anything R related

Find an R package by name, find package documentation, find R documentation, find R functions, search R source code...

gaussian

Search

14506

CRAN PACKAGES

1706

BIOCONDUCTOR PACKAGES

2064

R-FORGE PACKAGES

46052

GITHUB PACKAGES

Find an R package

Browse R language docs

Run R code online

Over 9,000 packages are preinstalled!

Create an R Notebook

Figure 11: Haut de la page principale du *R Package Documentation* <https://rdrr.io/> consultée le 5 février 2019

place, elle n'affecterait pas tant la courbe de croissance. À son avis, il faudrait repenser les relations entre les packages afin de diminuer la similarité entre les packages au profit d'interdépendances. Une première mesure pour maintenir une augmentation de la qualité des packages sur le CRAN est de permettre au utilisateur de donner un retour de leur utilisation (rapporté des bogues notamment). Les forges logiciel telles que *Github*, *Bitbucket* ou *R-Forge* sont fait pour cela et leur utilisation est de plus en plus fréquente. Une autre mesure est d'inciter les créateurs de packages à faire évaluer leur travail par les pairs, en soumettant leur package à des journaux/organisations spécialisée comme *Journal of Open Source Software* ou rOpenSci (voir plus haut).

3.2.5 Les interfaces utilisateurs

Travailler avec R, c'est écrire des lignes de code dans le langage R qui sont interprétées par R et pour cela, l'utilisateur utilise le programme R qui peut être ouvert de différentes manière. Par exemple en tapant R dans un émulateur de Terminal⁵³.

À chaque utilisation de R, l'utilisateur écrit une suite plus ou moins longue de commandes et utiliser simplement la console pour travailler son code est finalement assez peu efficace. C'est pour cela qu'on a recourt à un éditeur de code avec lequel l'utilisateur crée des fichier .R dans lequel il rédige les suites de commandes qu'il envoie vers la console R. Donc, ce qu'il fait pour travailler efficacement avec R c'est une console R et un éditeur de code. Il existe des logiciels qui offrent les deux fonctionnalités. Par exemple, lorsque R est téléchargé depuis le CRAN, il est fourni avec une interface utilisateur assez minimalistique avec laquelle l'utilisateur peut facilement créer des scripts et une console R est ouverte à l'ouverture du logiciel.

Comme nous l'avons mentionné précédemment, une des interfaces utilisateurs les plus utilisée actuellement est RStudio. RStudio est un logiciel multi-plateforme et gratuit (pour la version *Desktop* qui nous intéresse ici) qui permet de visualiser facilement la console où le code est exécuté, l'éditeur de lignes de code, les figures réalisées, la liste des packages chargés, la documentation, etc. Cette interface est à recommander lorsqu'on débute avec R et pour la suite si vous utiliser principalement R.

⁵³https://fr.wikipedia.org/wiki/%C3%89mulateur_de_terminal, consulté le 6 Février 2019.

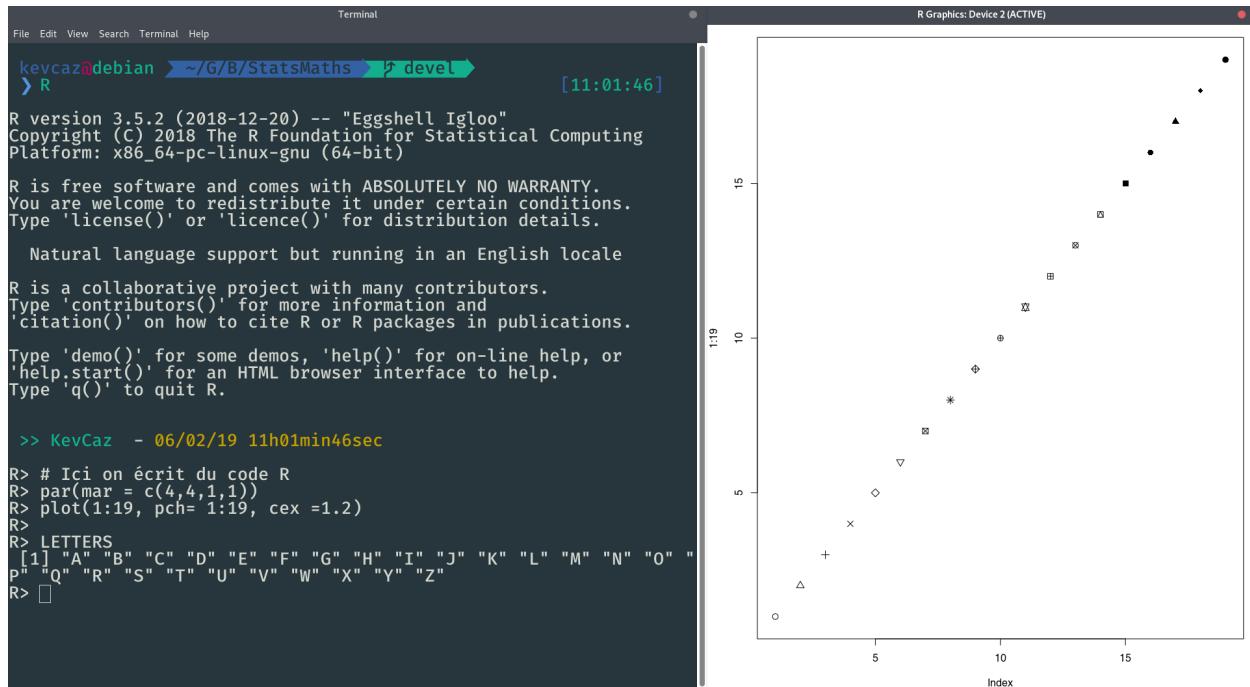


Figure 12: R lancé dans GNOME Terminal 3.30.2

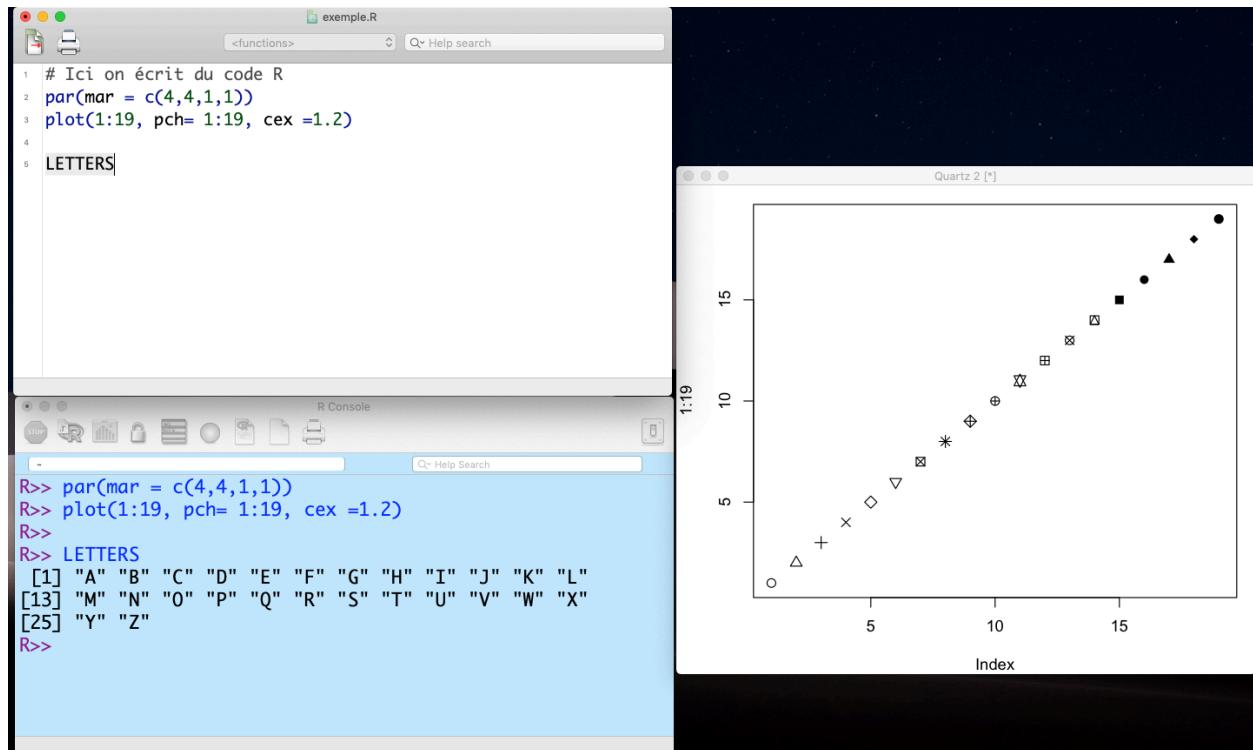


Figure 13: R utilisé avec l'interface utilisateur téléchargée depuis le CRAN

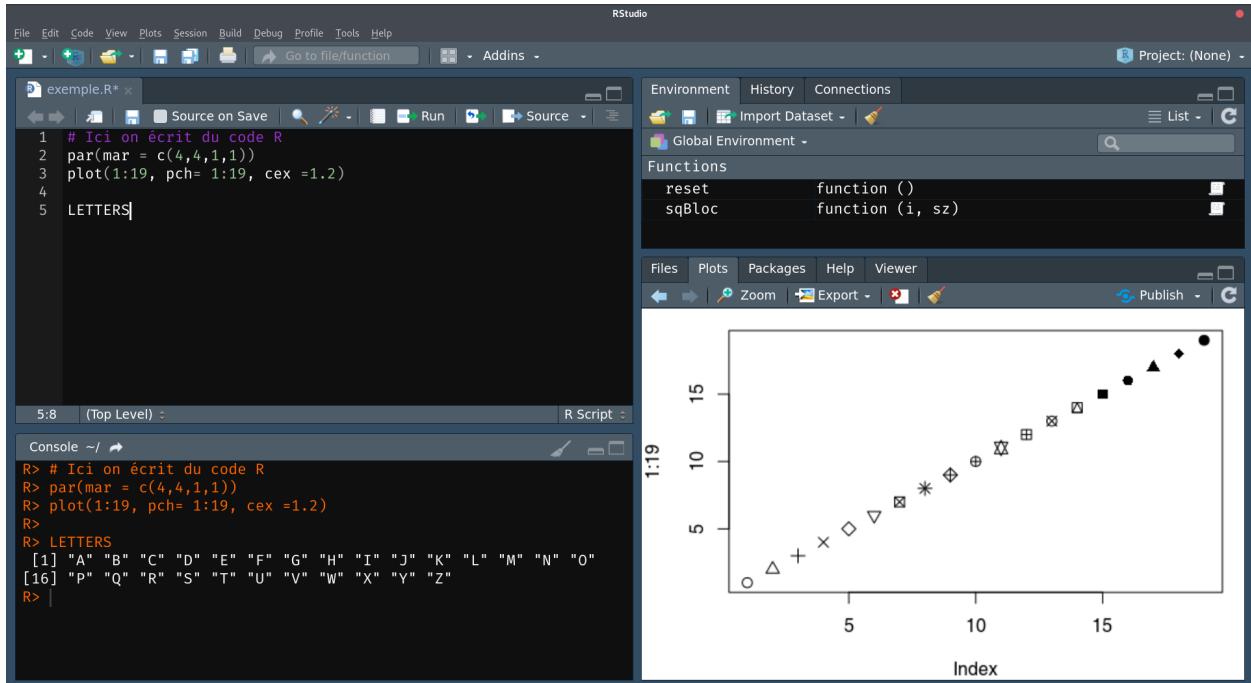


Figure 14: RStudio

Enfin, il est aussi possible d'utiliser R avec des éditeurs de codes généralistes, citons en quelques uns:

- Emacs
- Vim
- Visual Studio Code
- Sublime Text
- Atom

Chacun de ces logiciels offre de multiples fonctionnalités et viennent avec des extensions pratiques pour l'utilisation d'un très grande quantité de langages de programmation dont R. L'avantage de ces éditeurs de code est la multitude des fonctionnalités qui sont empruntées à différentes communautés de programmeurs. Avec ces logiciels, au prix de quelques heures de prise en main, il est possible d'avoir un environnement de travail multi-langage, personnalisé d'une très grande efficacité.

Cette section donne un aperçu des possibilités qui existent pour travailler avec R. Si vous débutez avec R sans grande expérience en programmation, nous recommandons l'utilisation de R Studio. Si vous êtes un programmeur chevronné et que vous utilisez déjà un éditeur de code performant et généraliste, il est probable que vous trouverez des extensions pour R pour cet éditeur de code pour cet éditeur de code en particulier.

4 Les différents systèmes graphiques dont R dispose

R dispose de nombreux outils pour faire un travail approfondi de visualisation des données. Lorsque R est installé, nous avons à notre disposition de multiple fonctionnalités pour créer, éditer et exporter des figures. L'édition de graphique sous R peut être divisé en trois branches : d'un côté l'approche développée dans le package *graphics*, celle du package *grid* (qui inclus ggplot2) et une troisième, hétérogène, qui est l'ensemble des libraries qui utilisent d'autres langages, par exemple, javascript (et ainsi créer des figures dans un navigateur web).

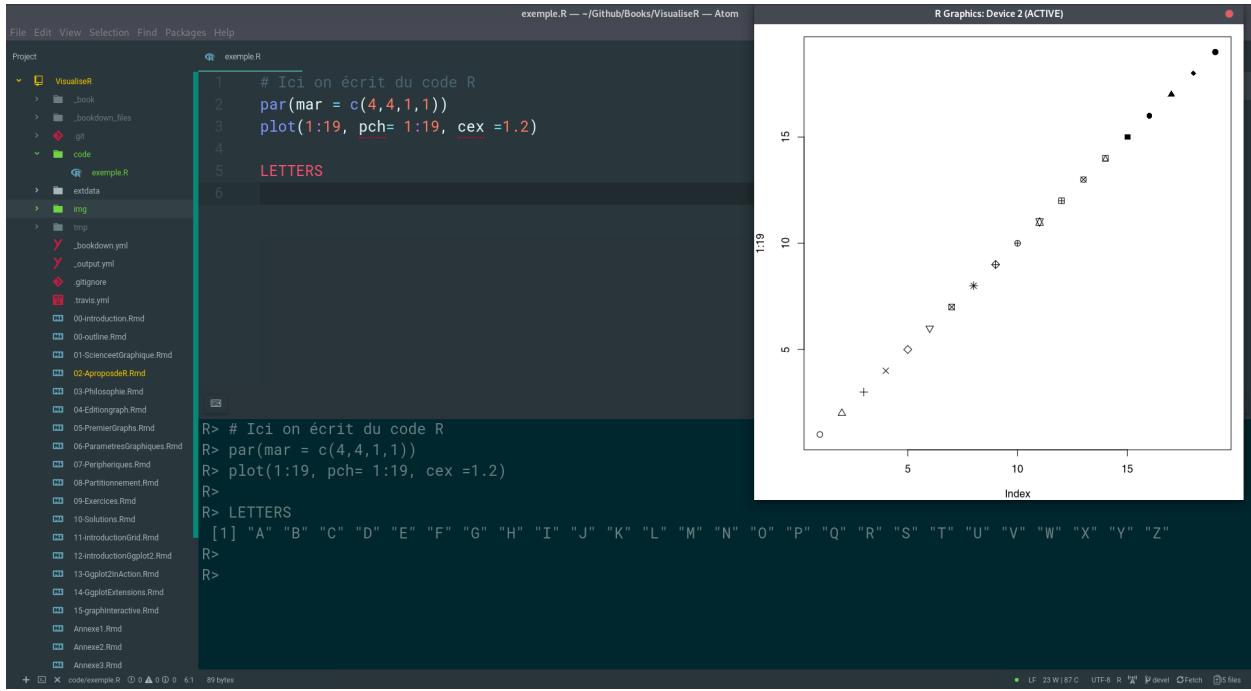


Figure 15: Utilisé R avec atom

4.1 Les étapes pour créer un graphique avec R

4.2 Le système graphique du package *graphics*

Le package *graphics* est une implémentation du système graphique historique de *S*. Il s'agit, en effet, de la ré-écriture des fonctionnalités de la bibliothèque GRZ pour les utiliser avec R. Ce travail a été en grande partie mené par Ross Ihaka, l'un des deux initiateurs de R. Avec ce package, nous pouvons obtenir en très peu de lignes de commandes des graphiques tout à fait convenables (grâce, par exemple, à la fonction *plot()*). Nous pouvons également travailler par couches successives pour ajouter un à un les éléments du graphiques et obtenir un rendu graphique plus aboutit. L'utilisation de *graphics* est répandue, simple, intuitive et sa puissance est souvent mésestimée. C'est pour ces raisons que nous avons choisi de la détailler dans ce livre. De manière générale, pour faire un graphique avec ce système on procède en trois étapes:

1. On définit les paramètres graphiques globaux (fonction *par()* voir chapitre)
2. On trace le plot et ajoute la fonction que nous souhaitons (*plot()*, *boxplot()*, ...)
3. On ajoute les différents éléments supplémentaires : points, lignes, textes, légendes.

Dans la suite, nous reprenons point par point ces différentes étapes. Une des principales faiblesses de ce système réside dans la difficulté à changer les éléments déjà placés. La plupart du temps, il est nécessaire de ré-exécuter tout le code modifier. Ce système est riche en fonctionnalités et plusieurs packages de R utilisent ce système pour implémenter des graphiques plus spécifiques. Le package *plotrix*, développé par Jim Lemon, utilise *graphics* et ajoute à celui-ci de nombreux types de graphiques supplémentaires dont les graphiques en coordonnées polaires.

4.3 Le système graphique du package *grid*

Le package *grid*, que nous devons à Paul Murrell, est un système graphique puissant qui donne un cadre performant pour développer des graphiques précis. Il y a plusieurs éléments importants sur lesquels nous

revenons en annexe. Le premier est le concept de *viewPort* : il s'agit de zone rectangulaire qui sont définis dans l'espace et avec un système de coordonnées. Il est possible d'en utiliser autant que souhaiter, elles peuvent se recouper ou non et posséder des paramètres. C'est en quelques sortes un ensemble très flexibles de calques. Il est possible de naviguer à travers pour placer à tout moment n'importe quel élément du future graphique. Le package *grid* définit ces propres fonction primitive pour dessiner points lignes et polygones. En plus d'un affichage et contrairement à ce que nous avons dans le package *graphics*, dans *grid*, les objets affichés sont aussi des objets que l'on peut modifier à sa guise. Cela donne au package *grid* une grande puissance. Il serait bien long de redéveloppé chaque graphique avec *grid* et comparer *graphics* et *grid* est un peu bancal en se sens qu'il ne sont pas réellement implémenté et qu'il faut les faire à la main. Néanmoins c'est grâce à sa puissance que sont nés deux autres systèmes de graphique à succès. Le premier est le système développé par Deepayan Sarkar dans le package *lattice*. Le second est une implémentation de la grammaire des graphiques, *ggplot2* que nous devons à Hadley Wickham. Notez que depuis peu, *grid* et *graphics* ont été reliés partiellement grâce au package *gridGraphics* qui permet d'écrire les fonctions graphiques en *grid*. Ces packages doivent beaucoup au travail de Paul Murrel dont le livre *R graphics* [ref] et aussi un chapitre sur *lattice* demeure une référence pour approfondir l'édition de graphiques avec R.

Lorsque les graphiques sont produits que ce soit avec *grid* ou *graphics*, pour les exporter dans différent format, on fait appel à un troisième package : *grDevices*.

4.4 Les graphiques interactives

Les graphiques qui font appel à des bibliothèques JS.

4.5 Interfcaes

Part II

Utiliser graphics

5 Édition d'un graphique

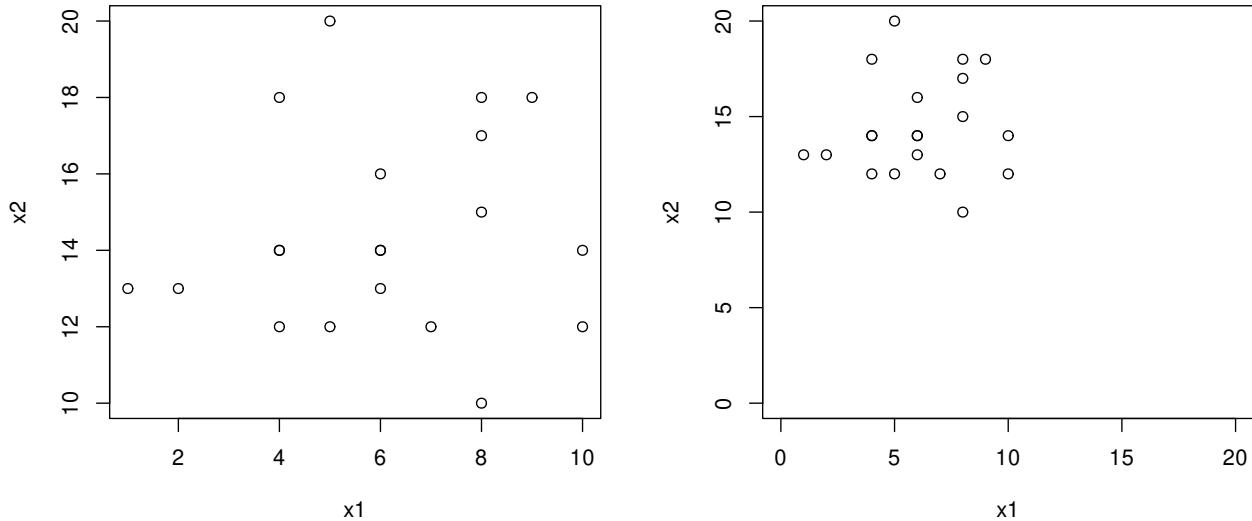
5.1 Graphique vierge

La philosophie des auteurs en termes de graphiques sous R est simple : ajouter les éléments un à un, en commençant par ouvrir une fenêtre graphique avec des dimensions (axes et marges) choisies, mais sans que rien ne s'affiche à l'écran. Regardons donc comment créer un graphe vide. Lorsqu'on fait appel à la fonction *plot()*, les axes sont déterminés automatiquement par R en fonction de l'étendue des valeurs des données que l'on souhaite représenter. Nous allons regarder comment modifier l'étendue des axes à l'aide de deux arguments : *xlim* et *ylim*. Créons tout d'abord deux variables continues.

```
(x1 <- sample(x = 0:10, size = 20, replace = TRUE))
#> [1] 8 7 4 5 4 2 4 10 6 9 8 10 8 6 6 5 6 8 1 4
(x2 <- sample(x = 10:20, size = 20, replace = TRUE))
#> [1] 18 12 14 20 14 13 12 12 14 18 17 14 15 13 16 12 14 10 13 18
```

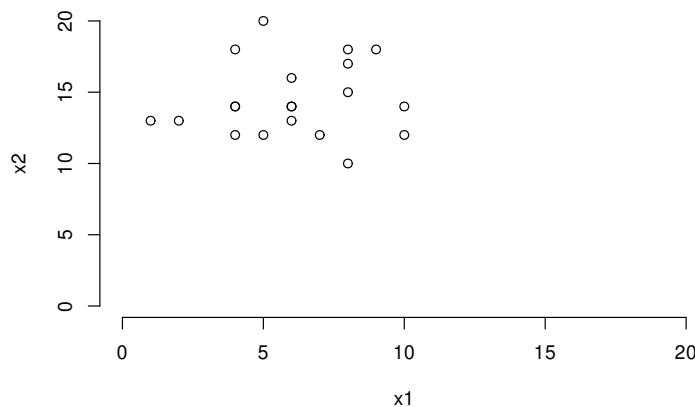
Ces deux variables ne varient pas de la même manière. Nous allons faire deux graphes. Le premier utilisera les paramètres par défaut de R. Dans le second, nous allons fixer les axes de manière à ce qu'ils soient bornés entre 0 et 20.

```
par(mfrow = c(1, 2))
plot(x1, x2)
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20))
```



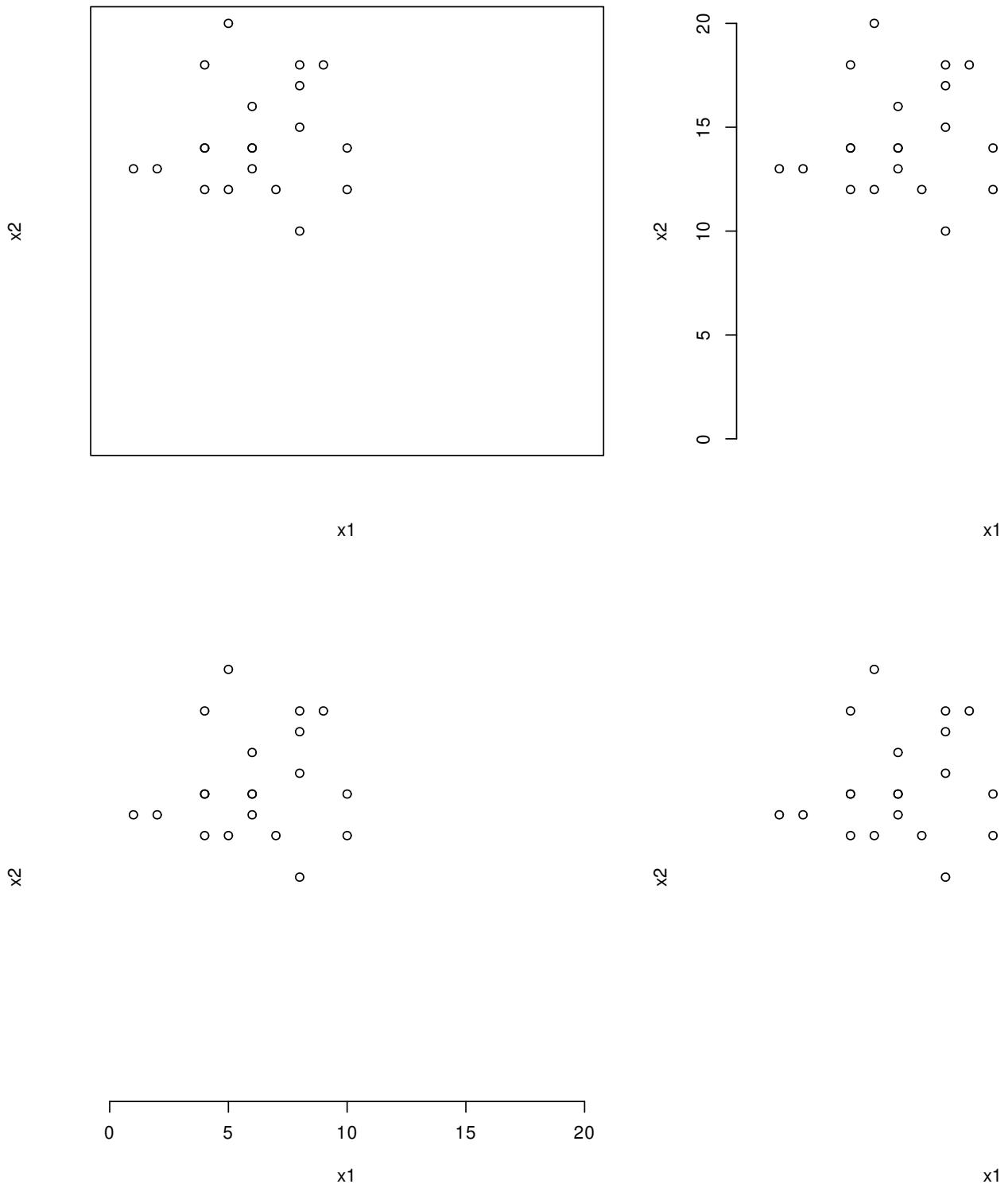
Par défaut, la fonction `plot()` affiche une boîte autour de la région graphique. C'est l'argument `bty` qui définit cela. Par défaut, il prend la valeur “`o`”. Pour supprimer ce cadre, on peut lui attribuer la valeur “`n`”.

```
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), bty = "n")
```



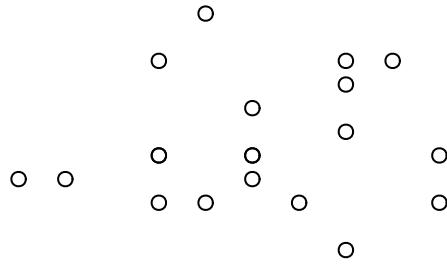
Maintenant que nous avons fixé les axes, nous allons les supprimer. Ceci n'aura aucune incidence sur l'étendue du graphe. Pour ce faire, nous allons utiliser les arguments `xaxt` et `yaxt` qui contrôlent l'affichage des axes. L'argument `axes` permet quant à lui de supprimer à la fois les axes, mais aussi le cadre. Comparez ces graphes suivants

```
par(mfrow = c(2, 2))
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), xaxt = "n", yaxt = "n")
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), bty = "n", xaxt = "n")
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), bty = "n", yaxt = "n")
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), axes = FALSE)
```



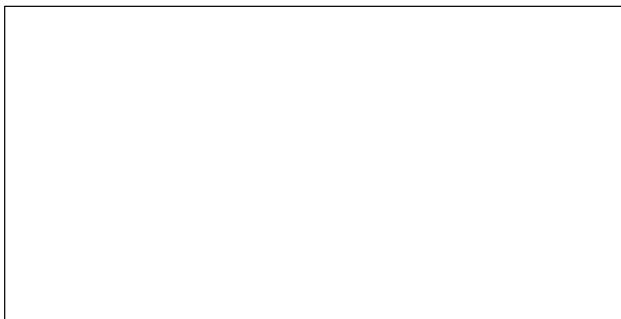
Poursuivons notre destruction graphique, et supprimons le nom des axes avec les arguments `xlab` et `ylab`. Par défaut, le nom des axes correspond au nom des variables. L'astuce ici consiste à leur attribuer la valeur `" "`. Cependant, une alternative consisterait à utiliser l'argument `ann` qui va supprimer toute annotation dans le graphe (nom des axes, mais aussi titre et sous-titre).

```
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), axes = FALSE, ann = FALSE)
```



Il ne nous reste plus qu'à supprimer les points avec l'argument `type`. Nous allons tout de même laisser le cadre afin de délimiter notre graphe.

```
plot(x1, x2, xlim = c(0, 20), ylim = c(0, 20), xaxt = "n", yaxt = "n", ann = FALSE, type = "n")
```



Étant donné que nous fixons les bornes des axes, et que nous supprimons l'affichage des données, nous pourrions éviter de spécifier les données en x et en y, et simplement demander de (ne pas) représenter la valeur **0** (ou autre chose). Ainsi, l'écriture précédente pourrait se résumer à ceci (sans le cadre) :

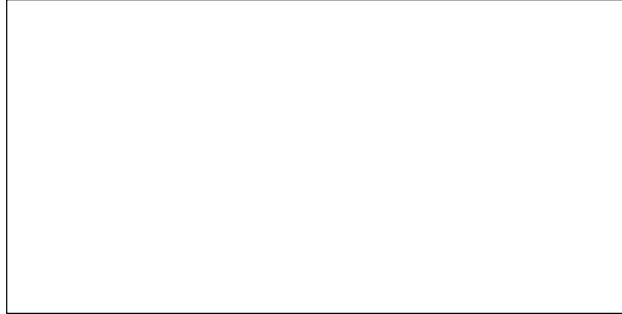
```
plot(0, xlim = c(0, 20), ylim = c(0, 20), axes = FALSE, ann = FALSE, type = "n")
```

Une particularité des programmeurs, c'est qu'il sont fainéants et s'ils peuvent économiser des lignes de code, alors ils le feront. Ainsi, nous allons créer une fonction qui implémentera un graphe vierge.

```
plot0 <- function(y = 0, x = y, type = "n", axes = FALSE, ann = FALSE, ...){  
  plot(x, y, axes = axes, type = type, ann = ann, ...)  
}
```

Par la suite, il suffira de faire appel à cette fonction pour ouvrir un nouveau périphérique graphique n'affichant rien, mais dont les dimensions auront été spécifiées.

```
plot0(xlim = c(0, 20), ylim = c(0, 20))  
box("plot")
```



Remarque : la fonction `box` permet d'afficher un cadre autour de la région du plot ("plot") ou de la figure ("figure"). Autre remarque : on pourrait utiliser cette fonction comme la fonction `plot()` et afficher, par ex. les données en indiquant `type = "p"`.

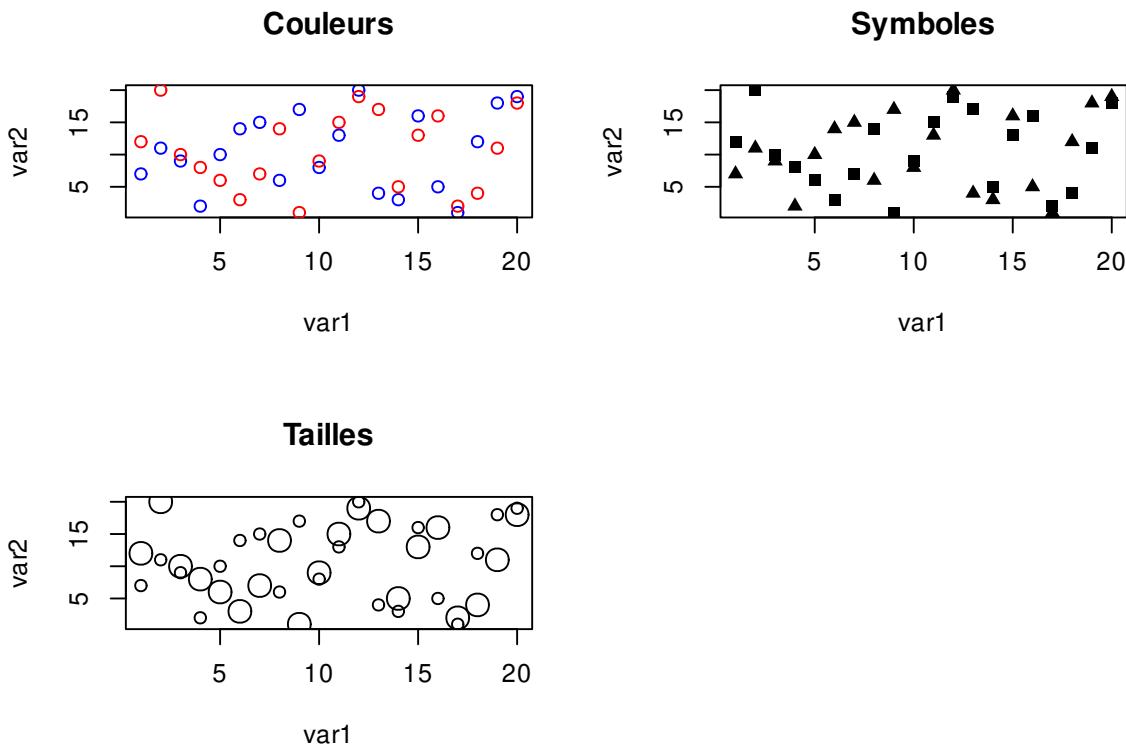
5.2 Ajout de points

Pour insérer des points sur un graphe, rien de plus simple : il suffit d'utiliser la fonction `points()`. Celle-ci partage un très grand nombre d'arguments avec la fonction `plot()`. Créons trois nouvelles variables.

```
var1 <- seq(1, 20)
var2 <- sample(var1, 20, replace = FALSE)
var3 <- sample(var1, 20, replace = FALSE)
```

Nous allons maintenant représenter sur le même graphe `var2` en fonction de `var1`, puis dans un second temps `var3` en fonction de `var1`. Nous alloir voir trois exemples pour distinguer les deux séries de valeurs.

```
par(mfrow = c(2, 2))
plot(var1, var2, col = "blue", main = "Couleurs")
points(var1, var3, col = "red")
plot(var1, var2, pch = 17, main = "Symboles")
points(var1, var3, pch = 15)
plot(var1, var2, cex = 1, main = "Tailles")
points(var1, var3, cex = 2)
```



Dans ces exemples, les trois variables présentaient la même gamme de valeurs (de 1 à 20). Si vous souhaitez superposer sur un même graphe des séries de points qui n'ont pas la même étendue de valeurs, il faudra convenablement définir les bornes des axes dans la fonction `plot()` avec les arguments `xlim` et `ylim` de manière à ce que toutes les séries de points s'affichent correctement. Introduisons maintenant une commande intéressante sous R : la fonction `locator()`. Celle-ci permet de récupérer les coordonnées d'un (ou de plusieurs) clic(s) sur un graphique. Voici comment l'utiliser pour deux clics :

```
locator(n = 2)
```

Cette fonction permet également de rajouter simultanément des points sur le graphe au fur et à mesure des clics.

```
locator(n = 3, type = "p")
```

Nous aurions également pu écrire :

```
points(locator(n = 3))
```

Remarque : la sélection peut être interrompue par un clic droit. La valeur par défaut de l'argument `n` est de 512 (clics).

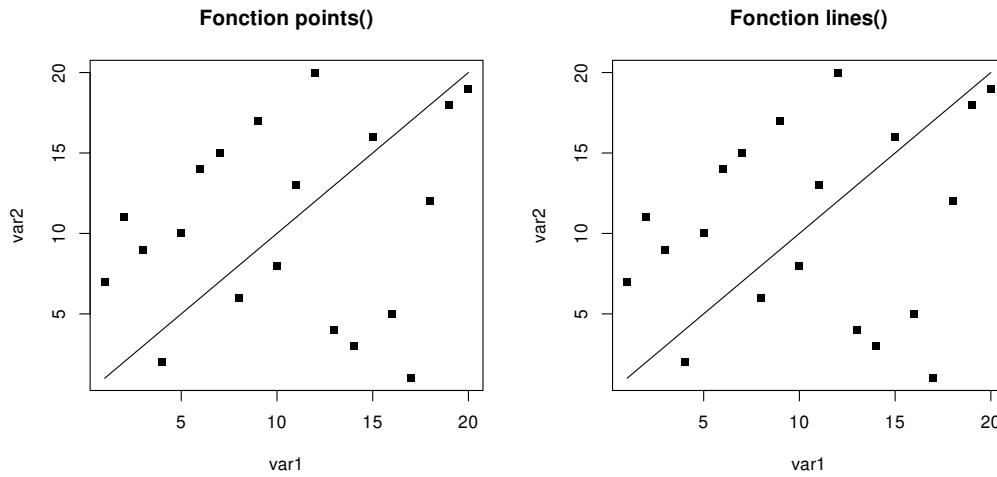
5.3 Ajout de lignes

Sous R, plusieurs fonctions permettent de tracer une ligne. Tout dépend de l'information de départ. Si on dispose des coordonnées des deux points extrêmes, nous pouvons utiliser à nouveau la fonction `points()`. La fonction `lines()` s'écrira de la même manière.

```

par(mfrow = c(1, 2))
plot(var1, var2, pch = 15, main = "Fonction points()")
points(x = c(1, 20), y = c(1, 20), type = "l")
plot(var1, var2, pch = 15, main = "Fonction lines()")
lines(x = c(1, 20), y = c(1, 20))

```

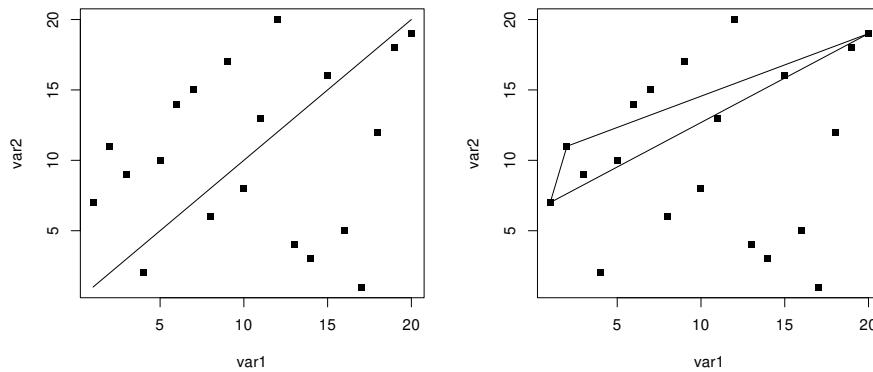


La fonction `segments()` s'utilise à peu de choses prêt de la même manière.

```

par(mfrow = c(1, 2))
plot(var1, var2, pch = 15)
segments(x0 = 1, y0 = 1, x1 = 20, y1 = 20)
plot(var1, var2, pch = 15)
segments(var1[1], var2[1], var1[2], var2[2])
segments(var1[1], var2[1], var1[20], var2[20])
segments(var1[20], var2[20], var1[2], var2[2])

```

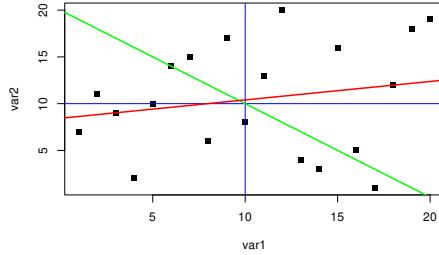


Parlons maintenant de la fonction `abline()`. Celle-ci offre plusieurs possibilités : elle permet entre autres de tracer des lignes horizontales et verticales ainsi que des droites de régression. Mais, contrairement aux trois fonctions précédentes, qui étaient bornées aux coordonnées fournies, les droites tracées avec cette fonction s'étendentront d'un bord à l'autre de la région graphique.

```

plot(var1, var2, pch = 15)
abline(h = 10, v = 10, col = "blue")
abline(a = 20, b = -1, col = "green", lwd = 2)
abline(reg = lm(var2 ~ var1), col = "red", lwd = 2)

```



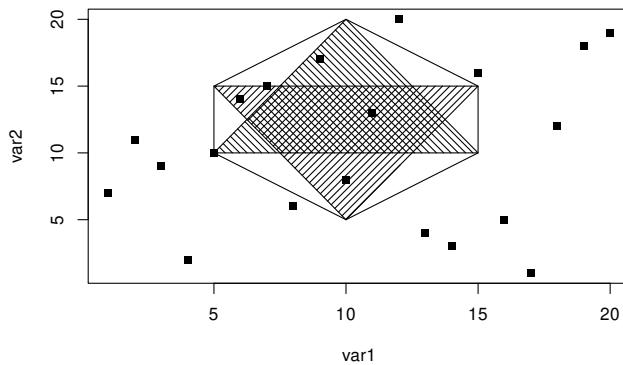
Finalement, mentionnons que la fonction `locator()` pourra être utilisée pour tracer des lignes de manière interactive.

```
locator(n = 2, type = "l", col = "blue", lty = 3)
lines(locator(4), col = "red")
```

5.4 Ajout de polygones

Insérer une forme polygonale sur un graphique se fera à l'aide de la fonction `polygon()`.

```
plot(var1, var2, pch = 15)
polygon(x = c(10, 5, 5, 10, 15, 15), y = c(5, 10, 15, 20, 15, 10))
polygon(x = c(10, 5, 15), y = c(5, 15, 15), density = 20, angle = 45)
polygon(x = c(5, 10, 15), y = c(10, 20, 10), density = 20, angle = 135)
```

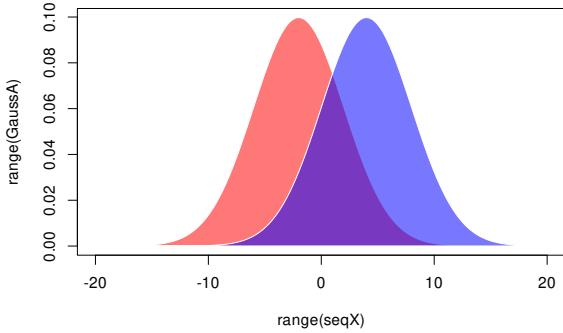


Nous retrouvons ici des arguments vu précédemment dans d'autres fonctions. Voyons maintenant un exemple en couleurs. Nous allons générer trois variables, dont deux correspondant à deux distributions normales.

```
seqX <- seq(-20, 20, 0.01)
GaussA <- dnorm(seqX, mean = -2, sd = 4)
GaussB <- dnorm(seqX, mean = 4, sd = 4)
```

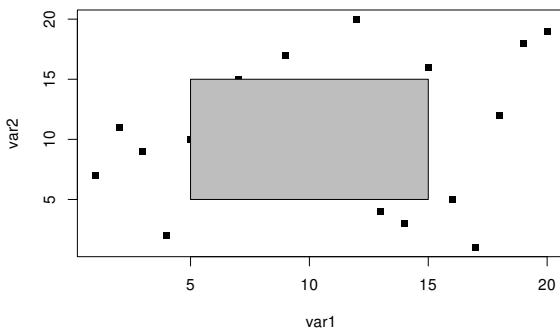
Traçons ces distributions avec la fonction `polygon()`.

```
plot(range(seqX), range(GaussA), type = "n")
polygon(x = seqX, y = GaussA, border = 0, col = "#FF000088")
polygon(x = seqX, y = GaussB, border = 0, col = "#0000FF88")
```



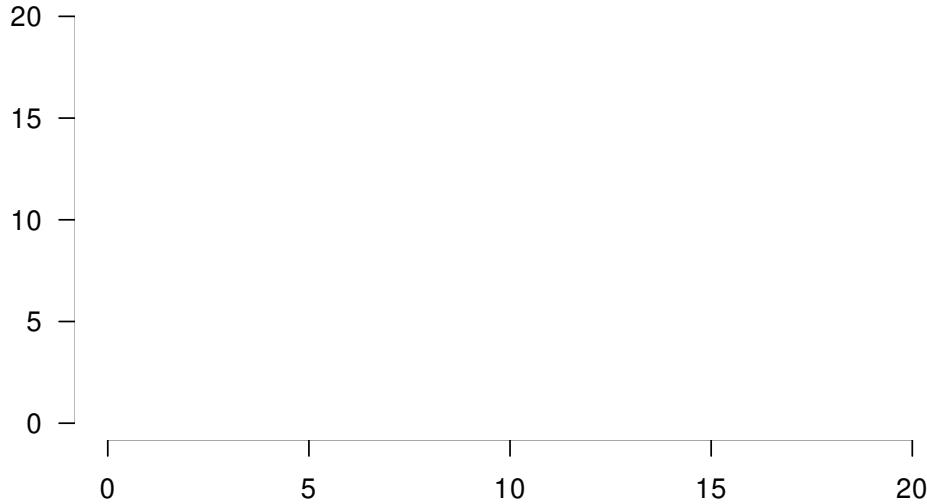
Ici, une remarque s'impose : nous venons d'attribuer deux couleurs dans un format un peu spécial : il s'agit du format hexadécimal. Celui-ci comprend, dans sa forme minimale, six caractères précédés du symbole *dièse* : les deux premiers symboles reflètent la quantité de rouge, les deux suivants celle de vert, et les deux derniers la quantité de bleu. À ceci, on peut rajouter (comme nous l'avons fait) deux autres caractères représentant le degré de transparence. Nous reviendrons sur ce point dans le chapitre suivant. Intéressons-nous maintenant à la fonction `rect()` qui permet de tracer un rectangle.

```
plot(var1, var2, pch = 15)
rect(xleft = 5, ybottom = 5, xright = 15, ytop = 15, col = "gray")
```



Remarque : la fonction `locator()` peut également être utilisée avec `polygon()`, mais pas avec la fonction `rect()`. Mettons à profit ce que nous venons d'apprendre avec les fonctions `rect()` et `abline()` pour personnaliser la zone de plot.

```
plot(0, type = "n", xlim = c(0, 20), ylim = c(0, 20), ann = FALSE, las = 1, bty = "n")
par()$usr
#> [1] -0.8 20.8 -0.8 20.8
rect(xleft = par()$usr[1], ybottom = par()$usr[3], ytop = par()$usr[4], xright = par()$usr[2], col = "lightblue")
abline(h = seq(0, 20, by = 5), col = "white")
abline(v = seq(0, 20, by = 5), col = "white")
abline(h = seq(2.5, 17.5, by = 5), col = "white", lty = 3)
abline(v = seq(2.5, 17.5, by = 5), col = "white", lty = 3)
rect(xleft = par()$usr[1], ybottom = par()$usr[3], ytop = par()$usr[4], xright = par()$usr[2], col = 0)
```



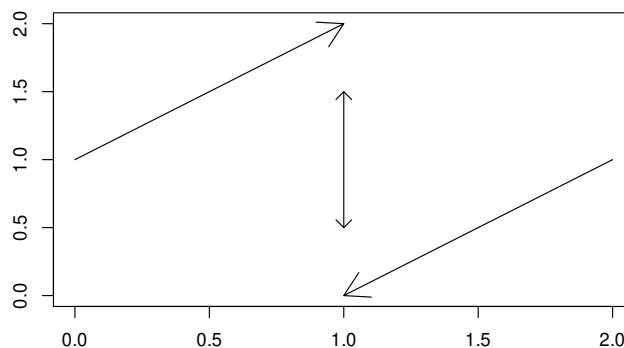
Ici, nous avons utilisé de nouveaux arguments. L'argument `lty` contrôle le type de ligne (`1` étant un trait plein, et `3` un trait en pointillés). L'argument `border = 0` indique que la couleur du contour du rectangle sera transparente. Enfin, l'argument `las = 1` spécifie que les valeurs portées sur les axes seront écrites horizontalement. Quelques précisions maintenant sur la commande `par()\$usr`. Nous avons défini nous-même l'étendue des axes x et y (0, 20). Cependant, par défaut, R laisse un peu de marge avant et après chaque axe (4% pour être précis). Ainsi, cette commande nous permet de récupérer les dimensions exactes (après ajout de ces 4%) de la région du plot. En attribuant la valeur “`i`” aux paramètres graphiques `xaxs` et `yaxs`, nous aurions supprimé cet ajout de 4%. Pour preuve :

```
plot(0, xlim = c(0, 20), ylim = c(0, 20), xaxs = "i", yaxs = "i")
par()\$usr
#> [1] 0 20 0 20
```

5.5 Ajout d'une flèche

Bien que peu fréquent, nous pouvons aussi tracer des flèches. Ceci se fera avec la fonction `arrows()`. Nous pouvons spécifier si la flèche sera en début ou en fin de ligne (ou les deux) avec l'argument `code`. On peut aussi définir la longueur de la flèche et son angle par rapport au trait. Un dessin vaut mille mots, donc allons-y.

```
plot(0, xlim = c(0, 2), ylim = c(0, 2), type = "n", ann = FALSE)
arrows(x0 = 0, y0 = 1, x1 = 1, y1 = 2)
arrows(1, 0, 2, 1, length = 0.25, angle = 30, code = 1)
arrows(1, 0.5, 1, 1.5, length = 0.10, angle = 45, code = 3)
```



Nous n'en dirons pas plus sur les flèches.

5.6 Ajout d'un titre

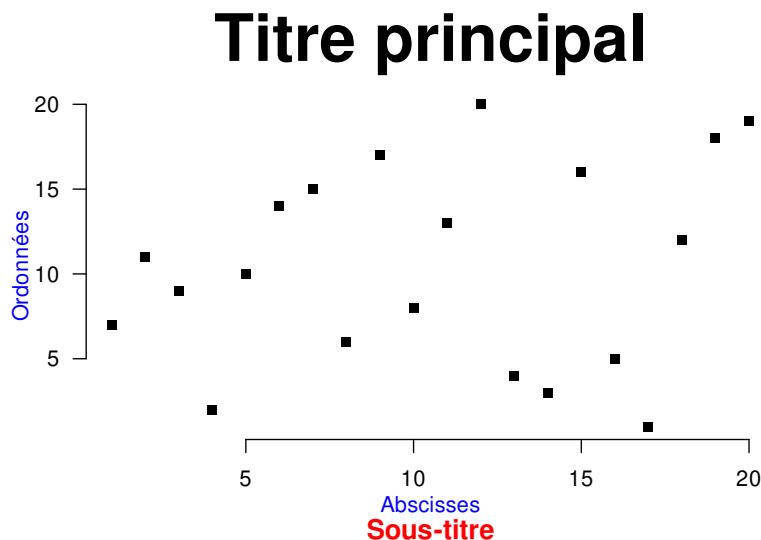
Précédemment, nous avons vu qu'il était possible de définir un titre directement dans les *High-level plotting functions*. Mais, il existe aussi la fonction `title()` en alternative. Voici les principaux arguments de cette fonction (qui sont aussi valables pour les fonctions `plot()` and co.). {ll}

Argument & Signification:

- `main` & Titre principal
- `sub` & Sous-titre
- `xlab` & Nom de l'axe des x
- `ylab` & Nom de l'axe des y
- `cex.main` & Taille du titre
- `cex.sub` & Taille du sous-titre

Utilisons ces arguments.

```
par(mgp = c(2, 1, 0))
plot(var1, var2, pch = 15, ann = FALSE, las = 1, bty = "n")
title(main = "Titre principal", cex.main = 3)
title(xlab = "Abscisses", ylab = "Ordonnées", col.lab = "blue")
title(sub = "Sous-titre", font.sub = 2, col.sub = "red", cex.sub = 1.25)
```



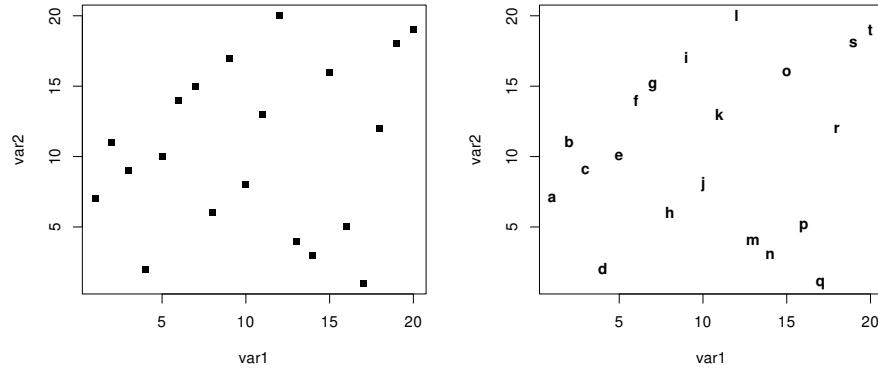
Nous verrons dans la section suivante qu'il est aussi possible d'ajouter des annotations dans les marges de la figure, et nous aurions pu utiliser à la place, la fonction `mtext()`.

5.7 Ajout de texte

Intéressons-nous maintenant aux annotations textuelles. Nous distinguerons les expressions textuelles insérées dans la zone de plot de celles destinées à apparaître en dehors de cette zone, c.-à-d. dans les marges de la figure. En effet, les fonctions correspondantes ne sont pas les mêmes et ne s'écrivent pas de la même manière.

La fonction `text()` permet d'insérer du texte dans la région du plot. Il faudra lui fournir les coordonnées en x et en y, ainsi que l'expression textuelle à ajouter. Regardons cela au travers d'un exemple.

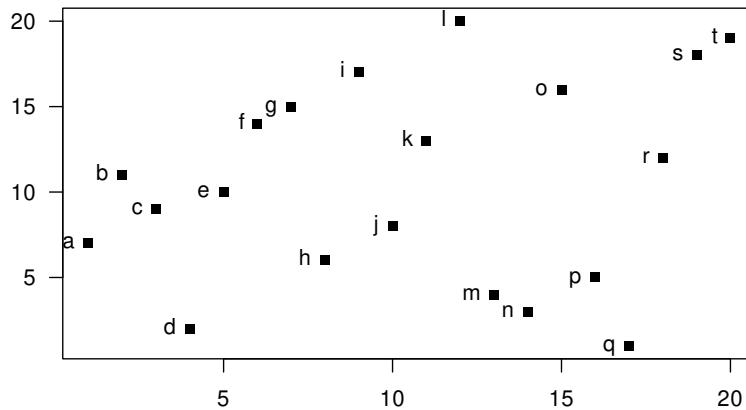
```
(nom <- letters[1 : length(var1)])
#> [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
#> [20] "t"
par(mfrow = c(1, 2))
plot(var1, var2, pch = 15)
plot(var1, var2, type = "n")
text(x = var1, y = var2, labels = nom, font = 2)
```



L'argument `font` contrôle la graisse du texte, c.-à-d. que le texte pourra être écrit normalement (1), en gras (2), en italique (3), etc. Ici, le texte se place exactement aux coordonnées fournies. Mais, on imagine facilement que si on superpose à la fois les points et les étiquettes, le graphique sera illisible. Heureusement, cette fonction présente l'argument `pos` qui contrôle le positionnement du texte par rapport aux coordonnées. Le tableau suivant fournit les valeurs possibles pour cet argument.

Valeur de pos & Signification\ - 0 & Aux coordonnées - 1 & En-dessous des coordonnées - 2 & À gauche des coordonnées - 3 & Au-dessus des coordonnées - 4 & À droite des coordonnées

```
plot(var1, var2, pch = 15, ann = FALSE, las = 1)
text(x = var1, y = var2, labels = nom, pos = 2)
```

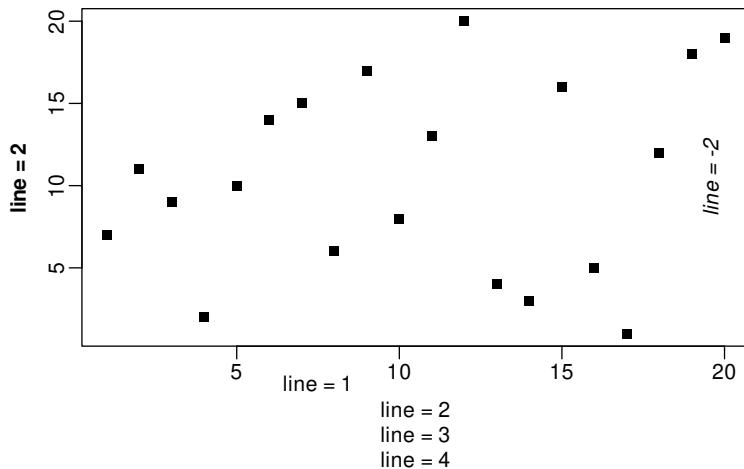


Remarque : la fonction `locator()` pourra s'appliquer ici. Pour rajouter du texte dans les marges, nous utiliserons la fonction `mtext()`. Cependant, cette fonction s'écrit différemment : l'argument `side` indique dans quelle marge doit être affiché le texte (1 en bas, 2 à gauche, 3 en haut, 4 à droite). L'argument `line` permet quant à lui de positionner le texte par rapport aux limites de la région du plot. Enfin, l'argument `at`

permet d'indiquer la coordonnée de placement sur l'axe en question. Vu qu'on ne fournit pas de coordonnées dans cette fonction, la fonction `locator()` ne fonctionnera pas.

Regardons cela avec un exemple simple.

```
par(mgp = c(2, .5, 0))
plot(var1, var2, pch = 15, ann = FALSE)
mtext(side = 1, line = 1, text = "line = 1", at = 7.5)
mtext(side = 1, line = 2, text = "line = 2")
mtext(side = 1, line = 3, text = "line = 3")
mtext(side = 1, line = 4, text = "line = 4")
mtext(side = 2, line = 2, text = "line = 2", font = 2)
mtext(side = 4, line = -2, text = "line = -2", font = 3)
```



5.8 Ajout d'une légende

Que serait un graphe sans légende ? La fonction `legend()` permet d'insérer une légende assez élaborée dans la région du plot et offre de nombreuses possibilités.

```
example(legend)
```

Tous les éléments de la légende sont modifiables, à l'exception de la police de caractères, ce qui peut être problématique si les autres éléments textuels du graphe (titre, nom des axes, etc.) n'utilisent pas la police par défaut. Nous verrons comment contourner cette difficulté plus loin. Le positionnement de la légende peut s'effectuer de deux manières différentes : - soit en indiquant les coordonnées x et y du coin supérieur gauche; - soit en spécifiant un mot-clé prédéfini (`top`, `bottom`, `topleft`, `center`, etc.). Le tableau ci-après présente les principaux arguments de cette fonction `legend()`. {ll}

Argument & Signification

- `legend` & Nom des items
- `bty` & Type de boîte (défaut ‘o’)
- `bg` & Couleur du fond de la boîte
- `box.lwd` & Épaisseur de la bordure de la boîte
- `box.col` & Couleur de la bordure de la boîte
- `title` & Titre de la légende
- `title.col` & Couleur du titre

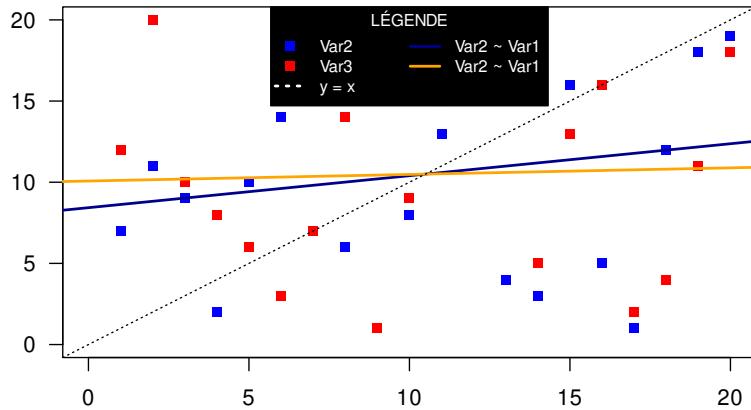
- `text.col` & Couleurs du nom des items
- `text.font` & Graisse du nom des items
- `col` & Couleurs des items (lignes ou symboles)
- `cex` & Taille des symboles
- `pch` & Symbole des items
- `pt.cex` & Taille des symboles
- `lty` & Type de ligne (si les items sont des lignes)
- `lwd` & Épaisseur des lignes (si les items sont des lignes)
- `ncol` & Nombre de colonnes pour représenter les items
- `horiz` & Items répartis en lignes ou en colonnes (défaut)
- `plot` & **TRUE** ou **FALSE**

Cette fonction, si attribuée à un objet, retourne des informations intéressantes sur le positionnement et les dimensions de la légende. Regardons plutôt.

```
plot(var1, var2, xlim = c(0, 20), ylim = c(0, 20), pch = 15)
(leg <- legend("center", legend = c("Item 1", "Item 2"), pch = 15))
#> $rect
#> $rect$w
#> [1] 3.422053
#>
#> $rect$h
#> [1] 4.87218
#>
#> $rect$left
#> [1] 8.288973
#>
#> $rect$top
#> [1] 12.43609
#>
#>
#> $text
#> $text$x
#> [1] 9.520913 9.520913
#>
#> $text$y
#> [1] 10.81203 9.18797
```

Les informations renvoyées correspondent à la boîte `$rect` (largeur, hauteur, coordonnées du coin supérieur gauche) et au positionnement des différents noms des items (`$text`). Regardons le comportement de quelques arguments de la fonction `legend()`.

```
plot(0, xlim = c(0, 20), ylim = c(0, 20), type = "n", ann = FALSE, las = 1)
points(var1, var2, pch = 15, col = "blue")
points(var1, var3, pch = 15, col = "red")
abline(reg = lm(var2 ~ var1), lwd = 2, col = "darkblue")
abline(reg = lm(var3 ~ var1), lwd = 2, col = "orange")
abline(a = 0, b = 1, lty = 3)
legend("top", c("Var2", "Var3", "y = x", "Var2 ~ Var1", "Var2 ~ Var1"), bg = "black", col = c("blue", "red", "black", "darkblue", "orange"))
```



5.9 Ajout d'un axe

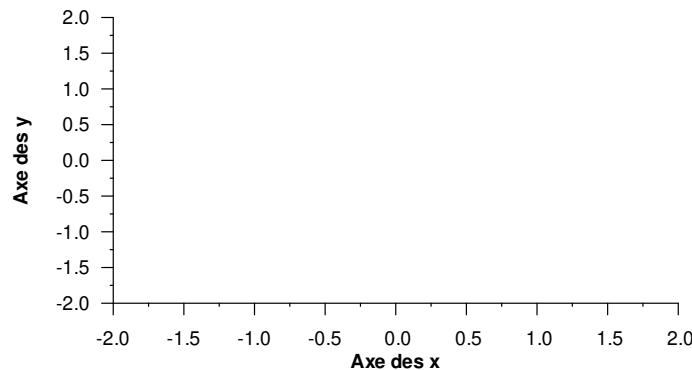
Regardons comment ajouter des axes à un graphe. Dans un premier temps, nous allons faire un plot vide et créer nous-même les axes avec la fonction `axis()`. Cette fonction accepte plusieurs arguments détaillés dans le tableau suivant.

Argument & Signification

- `side` & **1** (bas), **2** (gauche), **3** (haut), **4** (gauche)
- `at` & Coordonnées où placer la graduation
- `labels` & Étiquettes des graduations (même longueur que `at`)
- `pos` & Coordonnée de position sur l'axe perpendiculaire
- `tick` & **TRUE** ou **FALSE** (l'axe et la graduation ne sont pas tracés)
- `lty` & Type de ligne de l'axe
- `lwd` & Épaisseur de ligne de l'axe
- `lwd.ticks` & Épaisseur de ligne de la graduation
- `col` & Couleur de l'axe
- `col.ticks` & Couleur de la graduation
- `col.axis` & Couleur des étiquettes

```
plot0(xlim = c(-2, 2), ylim = c(-2, 2))
title(main = "Plot retravaillé")
grad <- seq(-2, 2, by = 0.5)
axis(side = 1, at = grad, labels = format(grad), pos = -2)
axis(side = 2, at = grad, labels = format(grad), pos = -2, las = 2)
axis(side = 1, at = seq(-1.75, 1.75, by = 0.5), pos = -2, tck = -0.01, labels = FALSE, lwd = -2, lwd.ticks = 1)
axis(side = 2, at = seq(-1.75, 1.75, by = 0.5), pos = -2, tck = -0.01, labels = FALSE, lwd = -2, lwd.ticks = 1)
mtext(side = 1, line = 1.5, text = "Axe des x", font = 2)
mtext(side = 2, line = 2.5, text = "Axe des y", font = 2, las = 0)
```

Plot retravaillé



Ici, nous avons défini une graduation secondaire dépourvue d'étiquette sur les axes avec une graduation plus fine. Nous avons également rajouté un nom aux axes avec la fonction `mtext()` puisque cette option n'est pas disponible dans la fonction `axis()`.

5.10 Ajout d'une image

Pour terminer ce chapitre, nous allons nous intéresser à l'inclusion d'une image dans un graphe. En effet, il peut être fort intéressant de pouvoir mettre une image quelconque sur un graphique. Par ex., sur une carte, il pourra s'agir de symboles divers permettant de figurer certains éléments caractéristiques (e.g. parc de stationnement, hôtel, station météo, etc.) ou des repères (e.g. nord géographique, etc.). Il pourra également s'agir du logo d'une institution, et bien d'autres. La fonction `rasterImage()` du package `graphics` permet d'ajouter à un graphe existant une image sous forme matricielle. Il peut s'agir d'une image sous format **JPEG**, **PNG**, **GIF**, etc. Dans la suite, nous n'importerons que des images au format **PNG**. Pour ce faire, nous avons besoin de charger un package complémentaire, spécialement dédié à cette tâche : le package `png`.

```
## install.packages("png")
library(png)
```

Nous avons développé une fonction, que nous avons appelée `plotimage()`, basée sur la fonction `rasterImage()`, qui permet d'importer dans un graphe n'importe quelle image au format **PNG**, **JPEG** ou **TIFF**. Cette fonction permet, soit d'ajouter l'image à un graphe existant, soit de créer un nouveau graphe avec cette image. De plus, elle permet de redimensionner l'image en conservant le rapport hauteur/largeur d'origine. Le tout s'adaptant aux dimensions du graphe. Enfin, cette fonction permet de positionner l'image soit en fournissant les coordonnées du centre, soit en utilisant des positions prédéfinies (e.g. "center", "topleft", etc.). Notons que tous les paramètres graphiques de la fonction `plot()` peuvent être modifiés (dans le cas où l'image est insérée dans une nouvelle fenêtre).

Commençons par définir cette fonction dans R.

```
plotimage <- function(file, x = NULL, y = NULL, size = 1, add = FALSE,
angle = 0, pos = 0, bg = "lightgray", ...){
if (length(grep(".png", file)) > 0){
require("png")
img <- readPNG(file, native = TRUE)
}
if (length(grep(".tif", file)) > 0){
require("tiff")
img <- readTIFF(file, native = TRUE)
}
```

```

}

if (length(grep(".jp", file)) > 0){
require("jpeg")
img <- readJPEG(file, native = TRUE)
}
res <- dim(img)[2:1]
if (add){
xres <- par()$usr[2] - par()$usr[1]
yres <- par()$usr[4] - par()$usr[3]
res <- c(xres, yres)
}else{
par(mar = c(1, 1, 1, 1), bg = bg, xaxs = "i", yaxs = "i")
dims <- c(0, max(res))
plot(0, type = "n", axes = FALSE, xlim = dims, ann = FALSE,
ylim = dims, ...)
}
if (is.null(x) && is.null(y)){
if (pos == "center" || pos == 0){
x <- par()$usr[1]+(par()$usr[2]-par()$usr[1])/2
y <- par()$usr[3]+(par()$usr[4]-par()$usr[3])/2
}
if (pos == "bottom" || pos == 1){
x <- par()$usr[1]+(par()$usr[2]-par()$usr[1])/2
y <- par()$usr[3]+res[2]*size/2
}
if (pos == "left" || pos == 2){
x <- par()$usr[1]+res[1]*size/2
y <- par()$usr[3]+(par()$usr[4]-par()$usr[3])/2
}
if (pos == "top" || pos == 3){
x <- par()$usr[1]+(par()$usr[2]-par()$usr[1])/2
y <- par()$usr[4]-res[2]*size/2
}
if (pos == "right" || pos == 4){
x <- par()$usr[2]-res[1]*size/2
y <- par()$usr[3]+(par()$usr[4]-par()$usr[3])/2
}
if (pos == "bottomleft" || pos == 5){
x <- par()$usr[1]+res[1]*size/2
y <- par()$usr[3]+res[2]*size/2
}
if (pos == "topleft" || pos == 6){
x <- par()$usr[1]+res[1]*size/2
y <- par()$usr[4]-res[2]*size/2
}
if (pos == "topright" || pos == 7){
x <- par()$usr[2]-res[1]*size/2
y <- par()$usr[4]-res[2]*size/2
}
if (pos == "bottomright" || pos == 8){
x <- par()$usr[2]-res[1]*size/2
y <- par()$usr[3]+res[2]*size/2
}
}

```

```

}
xx <- res[1]*size/2
yy <- res[2]*size/2
rasterImage(img, x-xx, y-yy, x+xx, y+yy, angle = angle)
}

```

Voici les différents arguments possibles pour cette fonction.

Argument & Signification - `file` & Nom de l'image à ouvrir (avec extension png) - `x` & Coordonnée en x du centre de l'image - `y` & Coordonnée en y du centre de l'image - `pos` & Position pré définie. Alternative à x et y. (voir la fonction `legend()`) - `size` & Coefficient réducteur de l'image (entre 0 et 1) - `angle` & Degré de rotation de l'image (entre 0 et 360) - `bg` & Couleur du fond de la figure - `add` & `TRUE` ou `FALSE` - ... & Autres paramètres graphiques de la fonction `plot()`

Le site Web <http://www.flaticon.com> permet de télécharger plus de 60 000 icônes gratuitement à différentes résolutions et sous différents formats. Les images suivantes sont issues de ce site. Merci aux auteurs !

```
# plotimage(file = "img/chap5/icon4.png", add = FALSE)
```

Regardons les différents placements par défaut.

```

par(mfrow = c(3, 3))
for (i in 0 : 8){
  plotimage("./icon8.png", size = 0.25, pos = i)
  box("figure")
}

plot(0, type = "n", axes = FALSE, ann = FALSE)
for (i in 0 : 8)
  plotimage("./icon6.png", size = 0.25, pos = i, add = TRUE)
  box("figure")

```

Pour terminer, regardons l'impact de l'angle en superposant la même image tous les 45 degrés. Les formes résultantes n'étaient pas du tout prévu par les auteurs. Et, le résultat est très surprenant et esthétique. De la pure sérendipité !!!

```

plot0(xlim = c(-1, 1), ylim = c(-1, 1), xaxs = "i", yaxs = "i")
for (i in seq(0, 360, 45)){
  plotimage("./icon2.png", size = 0.25, add = TRUE, angle = i, x = .25, y = .25)
}
plotimage("./icon2.png", size = 0.2, add = TRUE, pos = 7)
box("figure")

```

Un petit dernier, parce qu'on aime ça.

6 Graphiques classiques

Regardons tout d'abord quelques fonctions permettant de réaliser des graphiques parmi les plus communs dans la recherche scientifique. Sous R, de tels graphes sont réalisés avec des *High-level plotting functions*, c.-à-d. que l'appel à ces fonctions effacera le précédent contenu du périphérique graphique actif. Mais, nous verrons dans le dernier chapitre qu'il est possible de contourner cet obstacle. On opposera ces fonctions aux *Low-level plotting functions* qui elles, permettront d'ajouter des éléments à un graphique pré-existent. C'est l'objet du chapitre suivant.

6.1 Diagramme de dispersion

Il s'agit d'un graphe classique permettant de représenter deux variables continues l'une en fonction de l'autre dans un nuage de points. Nous allons réutiliser pour cela la fonction `plot()`. Créons une variable contenant une série de valeurs allant de 1 à 20.

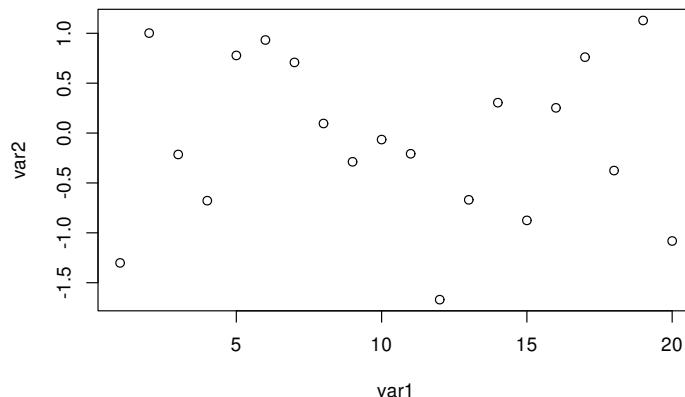
```
(var1 <- seq(from = 1, to = 20, by = 1))
#> [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Remarque : les parenthèses permettent d'afficher dans la console le résultat de l'assignation. Générons une seconde variable avec 20 valeurs tirées aléatoirement selon une distribution normale de moyenne 0 et d'écart-type 1.

```
(var2 <- rnorm(n = 20, mean = 0, sd = 1))
#> [1] -1.30158592 1.00199023 -0.21469905 -0.67772829 0.77796330 0.93286144
#> [7] 0.70814503 0.09586562 -0.28800979 -0.06531592 -0.20758215 -1.66985489
#> [13] -0.66917716 0.30468606 -0.87530490 0.25275209 0.76092100 -0.37592154
#> [19] 1.12856708 -1.08185323
```

Représentons maintenant le nuage de points (*scatterplot*) formés des valeurs de `var1` et `var2`.

```
plot(x = var1, y = var2)
```



Anticipons légèrement sur les chapitres suivants et intéressons-nous à l'argument `type` de la fonction `plot()`. Celui-ci permet de représenter les données de différentes manières : nuage de points, barres verticales, lignes, etc. {ll}

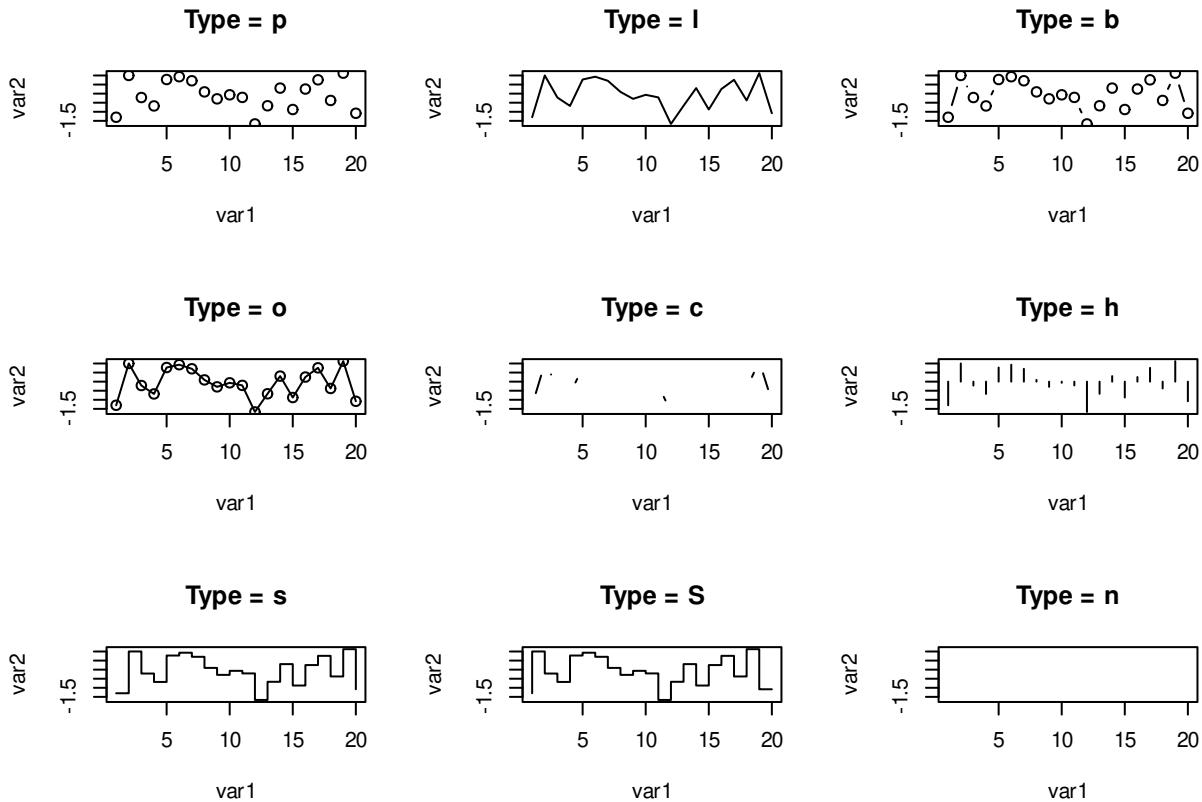
Valeur & Représentation\

- `type = 'p'` & Points
- `type = 'l'` & Lignes reliées
- `type = 'c'` & Lignes non reliées
- `type = 'b'` & Points et lignes non reliées
- `type = 'o'` & Points et lignes reliées
- `type = 'h'` & Barres verticales
- `type = 's'` & Plateau puis pente
- `type = 'S'` & Pente puis creux
- `type = 'n'` & Aucun symbole

Afin de bien comprendre les différences, partitionnons la fenêtre graphique en neuf régions distinctes (trois lignes et trois colonnes), chacune destinée à recevoir un plot spécifique avec une valeur différente pour l'argument `type`. Nous allons donc modifier le paramètre graphique `mfrow` de l'objet `par()`. Avec cet argument, les régions graphiques seront remplies en lignes.

Nous allons également rajouter un titre à chaque graphique qui contient la valeur de l'argument `type`. L'argument `main` permet de rajouter un titre principal à un graphe qui se positionnera en haut du graphique.

```
par(mfrow = c(3, 3))
plot(var1, var2, type = "p", main = "Type = p")
plot(var1, var2, type = "l", main = "Type = l")
plot(var1, var2, type = "b", main = "Type = b")
plot(var1, var2, type = "o", main = "Type = o")
plot(var1, var2, type = "c", main = "Type = c")
plot(var1, var2, type = "h", main = "Type = h")
plot(var1, var2, type = "s", main = "Type = s")
plot(var1, var2, type = "S", main = "Type = S")
plot(var1, var2, type = "n", main = "Type = n")
```

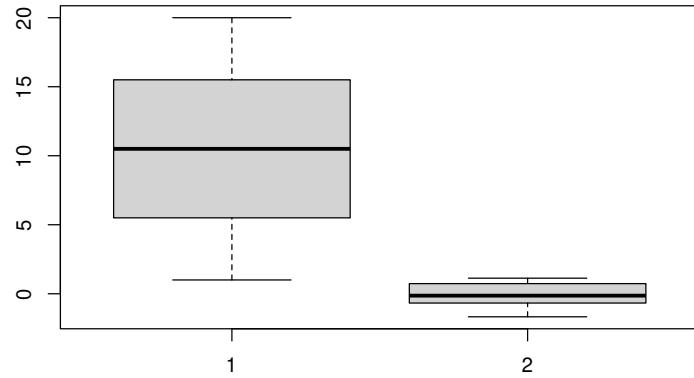


La fonction `plot()` offre de nombreux autres arguments qu'il est possible de modifier. C'est en partie ce que nous verrons tout au long de ce document, puisque cet enseignement met essentiellement l'accent sur cette fonction.

6.2 Boîte à moustaches

La boîte à moustaches est une représentation graphique très utile en statistiques, puisqu'elle permet de résumer les caractéristiques de position (médiane, 1^{er} et 3^{ème} quartiles, minimum et maximum) d'une variable quantitative. Sous R, la fonction utilisée sera `boxplot()`.

```
boxplot(var1, var2)
```



Cette fonction s'applique sur des vecteurs, mais aussi sur des data frames. Elle possède de nombreux arguments. Par exemple, le tableau suivant liste les paramètres les plus courants. {ll}

Argument & Signification:

- **width** & Largeur des boîtes (valeurs à fournir)
- **varwidth** & Largeur des boîtes (proportionnelle au n)
- **outline** & Suppression des outliers
- **horizontal** & Vertical ou horizontal
- **add** & Rajout d'une boîte
- **at** & Coordonnée en x de la nouvelle boîte

L'argument **plot** mis à la valeur **FALSE** n'affiche pas de boîte à moustaches, mais retourne les différentes statistiques associées sous la console. Par exemple:

```
boxplot(var2, plot = FALSE)
#> $stats
#>      [,1]
#> [1,] -1.6698549
#> [2,] -0.6734527
#> [3,] -0.1364490
#> [4,]  0.7345330
#> [5,]  1.1285671
#>
#> $n
#> [1] 20
#>
#> $conf
#>      [,1]
#> [1,] -0.6338886
#> [2,]  0.3609906
#>
#> $out
#> numeric(0)
#>
#> $group
#> numeric(0)
#>
```

```
#> $names
#> [1] "1"
```

6.3 Diagramme en bâtons

Ce type de représentation est utile pour visualiser des données discrètes ou catégoriques. Chaque modalité de la variable catégorique (ou discrète) sera représentée par une barre verticale (ou horizontale) dont la longueur sera proportionnelle à son effectif (relatif ou absolu) parmi l'ensemble des modalités. Sous R, on réalise un tel graphique avec la fonction `barplot()`. Créons tout d'abord un vecteur contenant six modalités (chaînes de caractères).

```
(nom <- c("Vert", "Jaune", "Rouge", "Blanc", "Bleu", "Noir"))
#> [1] "Vert"  "Jaune" "Rouge" "Blanc" "Bleu"   "Noir"
```

Maintenant, nous allons tirer aléatoirement 1000 valeurs (avec remise donc) dans ce vecteur afin que chaque couleur soit présente plusieurs fois.

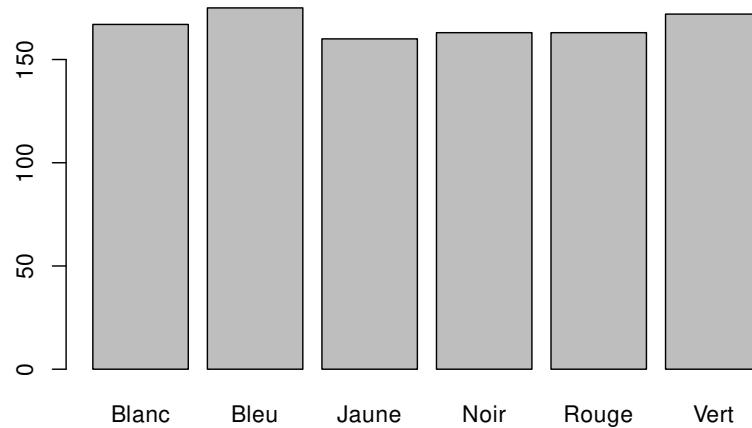
```
echn <- sample(x = nom, size = 1000, replace = TRUE)
echn[1:20]
#> [1] "Jaune" "Bleu"   "Bleu"   "Blanc" "Blanc" "Jaune" "Noir"   "Bleu"   "Noir"
#> [10] "Rouge"  "Noir"   "Jaune"  "Noir"   "Rouge"  "Rouge"  "Rouge"  "Jaune"  "Vert"
#> [19] "Jaune"  "Rouge"
```

Comptons combien de fois se retrouve chaque modalité dans cette variable.

```
(var4 <- table(echn))
#> echn
#> Blanc  Bleu  Jaune  Noir  Rouge  Vert
#> 167   175   160   163   163   172
```

Visualisons cette nouvelle variable catégorique.

```
barplot(var4)
```



Nous pouvons également représenter cette même variable sous forme relative. Calculons la fréquence de chaque modalité et produisons un nouveau graphe.

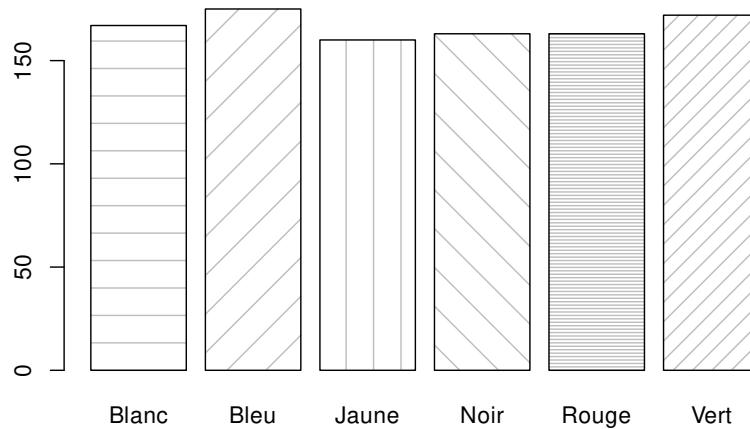
```
(var5 <- var4/sum(var4))
#> echn
#> Blanc Bleu Jaune Noir Rouge Vert
#> 0.167 0.175 0.160 0.163 0.163 0.172
```

Visuellement, rien ne changera, mis à part les valeurs portées sur l'axe des ordonnées. La fonction `barplot()` possède aussi de nombreux arguments. Nous vous invitons à consulter l'aide associée à cette fonction.

```
help(barplot)
```

Il est possible de hachurer les rectangles plutôt que de leur associer une couleur. Pour ce faire, deux arguments doivent être spécifiés : - `density` : nombre de hachures par pouce; - `angle` : orientation des hachures dans le sens trigonométrique. Par exemple,

```
barplot(var4, density = c(rep(5, 4), 40, 10), angle = c(0, 45, 90, 135))
```



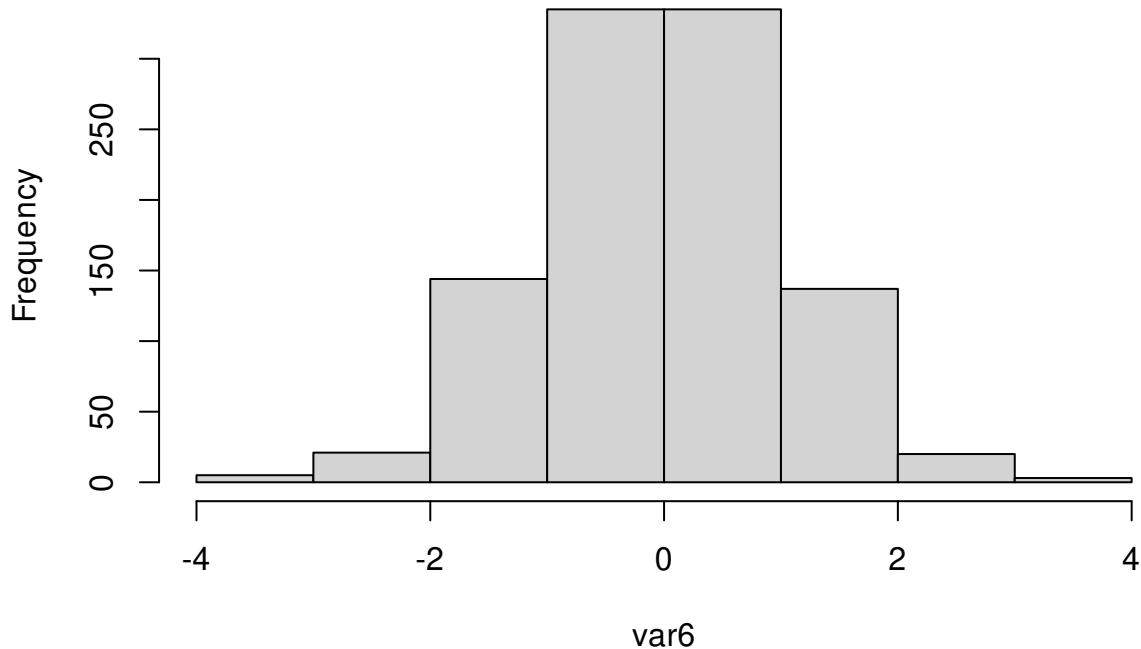
6.4 Histogramme

L'histogramme permet de visualiser la répartition des valeurs d'une variable continue en créant des classes de valeurs. Il est très utile pour connaître la loi de distribution que suivent les valeurs (loi normale, loi de Poisson, etc.). Sous R, ce graphe se fera à l'aide de la fonction `hist()`. Générons 1000 valeurs aléatoires selon une loi gaussienne.

```
var6 <- rnorm(n = 1000)
var6[1:20]
#> [1] -0.84662648  0.88428710 -0.61045109  0.40266609  1.10209985  0.09268090
#> [7]  0.31801968 -0.06017932 -0.97767294 -1.07471188 -0.71395050 -1.01911101
#> [13]  1.69670482 -0.53455813 -0.32133670 -1.43342050 -0.09313099  0.52829319
#> [19] -1.03240366 -0.47266098
```

```
hist(var6)
```

Histogram of var6

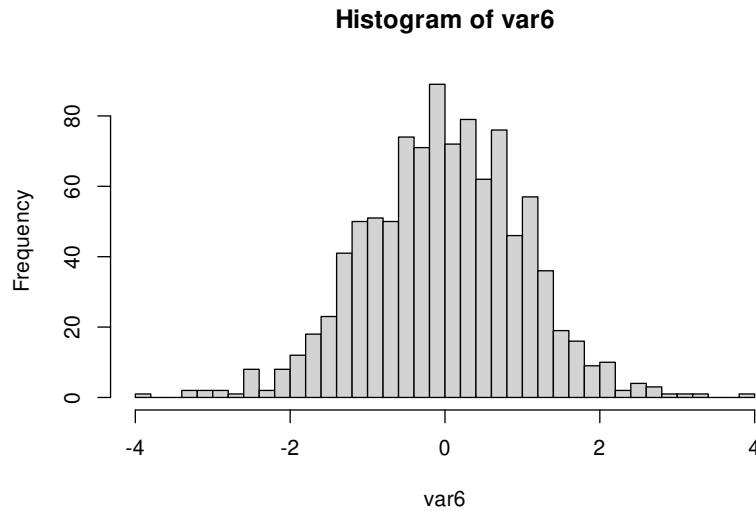


De la même manière que pour les boîtes à moustaches, l'argument `plot = FALSE` ne trace pas l'histogramme, mais affiche dans la console les statistiques associées.

```
hist(var6, plot = FALSE)
#> $breaks
#> [1] -4 -3 -2 -1  0  1  2  3  4
#>
#> $counts
#> [1]  5 21 144 335 335 137  20   3
#>
#> $density
#> [1] 0.005 0.021 0.144 0.335 0.335 0.137 0.020 0.003
#>
#> $mids
#> [1] -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5
#>
#> $xname
#> [1] "var6"
#>
#> $equidist
#> [1] TRUE
#>
#> attr(,"class")
#> [1] "histogram"
```

L'argument `breaks` permet de modifier les classes de l'histogramme. Une façon simple de procéder consiste à donner le nombre de classes que l'on souhaite représenter.

```
hist(var6, breaks = 30)
```



Nous aurions également pu indiquer les bornes de chaque classe désirée (par ex. des classes tous les 0.25). De même, plusieurs algorithmes ont été implémentés afin de déterminer un nombre de classes optimum . Consultez l'aide de cette fonction pour en savoir plus.

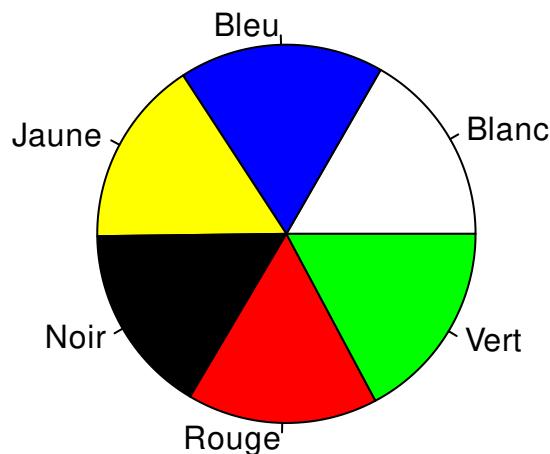
```
help(hist)
```

Finalement, mentionnons que les arguments **density** et **angle** sont aussi disponibles pour la fonction **hist()**. Adéquatement définis, ils permettront d'hachurer les rectangles.

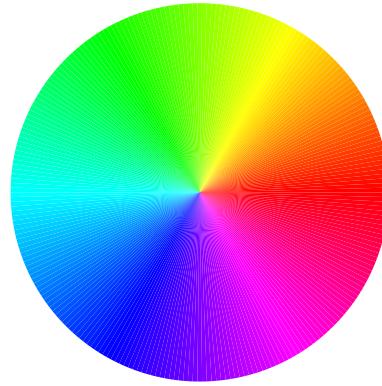
6.5 Diagramme sectoriel

Une alternative au diagramme en bâtons est le diagramme sectoriel (camembert). Regardons ce que ça donne avec les données précédentes (couleurs).

```
pie(var4, col = c("white", "blue", "yellow", "black", "red", "green"))
```



```
pie(rep(1, 250), col = rainbow(250), border = NA, labels = "")
```



Dans les deux exemples précédents, nous avons défini des couleurs à l'aide de l'argument `col`. Dans le premier cas, nous avons indiqué le nom des couleurs : R implémente une palette de couleurs prédéfinies assez conséquente. Dans le second cas, nous avons utilisé la fonction `rainbow()` qui sélectionne un nombre donné de couleurs parmi les couleurs de l'arc-en-ciel. Nous y reviendrons dans le troisième chapitre.

Pour terminer, exécutez le code suivant sous R.

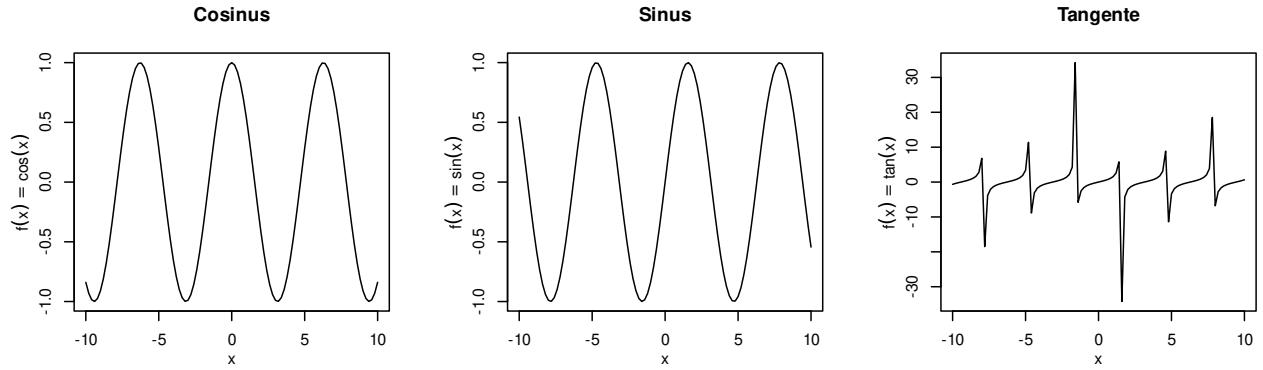
```
for (i in 1 : 250){
  if (i > 1){
    cols <- rainbow(250)[c(i:250, 1:(i-1))]
    pie(rep(1, 250), col = cols, border = NA, labels = "")
  }else{
    pie(rep(1, 250), col = rainbow(250), border = NA, labels = "")
  }
}
```

Plutôt sympa, non ?

6.6 Fonctions mathématiques

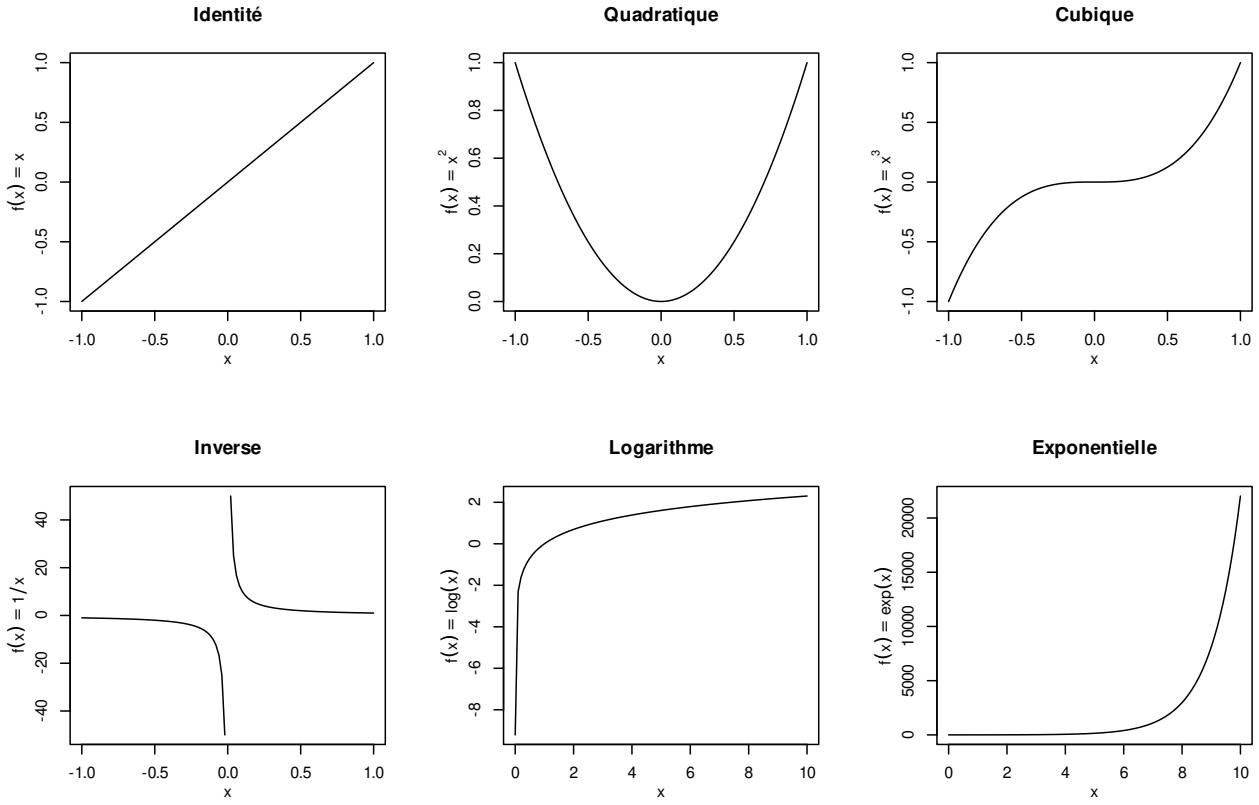
Pour terminer ce chapitre, introduisons une fonction qui pourrait vous être utile. Il s'agit de la fonction `curve()` qui permet de tracer le comportement d'une fonction mathématique bornée. Regardons un exemple avec des fonctions trigonométriques.

```
par(mfrow = c(1, 3), mgp = c(2, 1, 0))
txt <- expression(f(x)==cos(x))
curve(cos(x), from = -10, to = 10, main = "Cosinus", ylab = txt)
txt <- expression(f(x)==sin(x))
curve(sin(x), from = -10, to = 10, main = "Sinus", ylab = txt)
txt <- expression(f(x)==tan(x))
curve(tan(x), from = -10, to = 10, main = "Tangente", ylab = txt)
```



Ici, des éclaircissements s'imposent. Premièrement, nous avons modifié un autre paramètre graphique : `mgp`. Celui permet de contrôler le positionnement du nom des axes, des étiquettes des axes et des axes eux-mêmes. Ces positionnements sont relatifs à la région du plot. Nous y reviendrons plus loin. La fonction `expression()` permet quant à elle d'utiliser l'écriture mathématique dans des graphiques. Consultez l'aide de cette fonction ainsi que celle de la fonction `plotmath()` pour en savoir plus.

```
par(mfrow = c(2, 3), mgp = c(2, 1, 0))
txt <- expression(f(x)==x)
curve(x^1, from = -1, to = 1, main = "Identité", ylab = txt)
txt <- expression(f(x)==x^2)
curve(x^2, from = -1, to = 1, main = "Quadratique", ylab = txt)
txt <- expression(f(x)==x^3)
curve(x^3, from = -1, to = 1, main = "Cubique", ylab = txt)
txt <- expression(f(x)==1/x)
curve(1/x, from = -1, to = 1, main = "Inverse", ylab = txt)
txt <- expression(f(x)==log(x))
curve(log(x), from = 0.0001, to = 10, main = "Logarithme", ylab = txt)
txt <- expression(f(x)==exp(x))
curve(exp(x), from = 0.0001, to = 10, main = "Exponentielle", ylab = txt)
```



7 Paramètres graphiques

Au cours des deux derniers chapitres, nous avons vu comment créer et éditer un graphe à l'aide des principales fonctions contenues dans le package `graphics`. Dans ce troisième chapitre, nous allons approfondir les notions introduites précédemment, notamment en ce qui a trait aux paramètres graphiques principaux, tels que les fontes de caractères, les types de symboles et de lignes, les axes, les marges, etc. Nous terminerons ce chapitre en nous attardant sur les couleurs sous R, et plus généralement, dans le monde informatique. En effet, ce que nous dirons sur les fontes de caractères et les couleurs sera aussi valable dans le développement Web (e.g. CSS pour *Cascading Style Sheets*, ou feuilles de style en français).

7.1 La fonction `par()`

Précédemment, nous avons mentionné que lorsque R trace ou édite un graphe, il va récupérer les valeurs des paramètres graphiques pour adapter les axes, le background, les couleurs, les tailles de caractères, etc. Celles-ci sont stockées dans l'objet `par()`, qui est également une fonction. En effet, l'affichage d'un paramètre se fait en appelant l'objet (le paramètre est un élément de la liste `par()`), mais le changement de valeur d'un paramètre se fait en appelant la fonction (le paramètre devient un argument de la fonction `par()`).

```
par()$bg
#> [1] "white"
par()$col
#> [1] "black"
par()$bty
#> [1] "o"
```

Ces paramètres possèdent des valeurs par défaut afin d'éviter à l'utilisateur de devoir les définir à chaque nouveau graphe. Bien entendu, ces valeurs peuvent être modifiées : heureusement, car même si les valeurs par défaut conviennent à n'importe quelle représentation graphique, le rendu visuel laisse vraiment à désirer. Nous avons déjà modifié les valeurs par défaut de certains paramètres (axes, couleurs, etc.), soit directement dans les fonctions appelées (e.g. `plot()`, `axis()`, `legend()`, `polygon()`, etc.), soit dans la fonction `par()`. Et, c'est là une notion très importante : les paramètres graphiques peuvent être modifiés soit dans le `par()`, soit à la volée, dans les fonctions graphiques. Mentionnons tout de même que certains paramètres ne peuvent être modifiés que via la fonction `par()`. C'est le cas notamment de `mar`, `oma`, `new`, `mfcol` et `mfrow`.

Mais, la modification dans le `par()` n'aura pas le même effet qu'une modification à la volée. En effet, modifier la couleur du texte dans la fonction `plot()` n'aura pas pour conséquence de mettre à jour la valeur de cet argument dans le `par()`. Ainsi, si derrière nous rajoutons, par ex. un titre avec la fonction `title()` sans spécifier de couleur, celui-ci s'affichera avec la valeur par défaut contenue dans le `par()`. Au contraire, si on modifie la couleur du texte dans le `par()`, et qu'aucune précision n'est apportée à la volée concernant ce paramètre, toutes les fonctions graphiques afficheront la couleur du texte selon la nouvelle valeur définie dans le `par()`. Un dernier point important : toute modification dans la fonction `par()` sera effective tant qu'un périphérique graphique restera ouvert. La fermeture des périphériques graphiques entraînera la remise à zéro des valeurs des paramètres graphiques. Cependant, il est de coutume de sauvegarder les paramètres graphiques avec leurs valeurs par défaut dans un objet, et de redéfinir le `par()` avec cet objet une fois le graphique réalisé. Ceci permet de s'assurer que le `par()` est bien réinitialisé.

```
## Sauvegarde du par() d'origine
opar <- par()
## Modification du par()
par(bg = "steelblue", mar = c(1, 1, 0, 0), col = "white")
## Commandes graphiques...
## Restauration du par()
par(opar)
```

La fonction `par()` comporte 72 paramètres graphiques dont la plupart sont modifiables (66 pour être précis). Au cours de ce chapitre, nous allons en détailler une bonne trentaine, ceux que nous avons jugés les plus pertinents.

7.2 Fonte de caractères

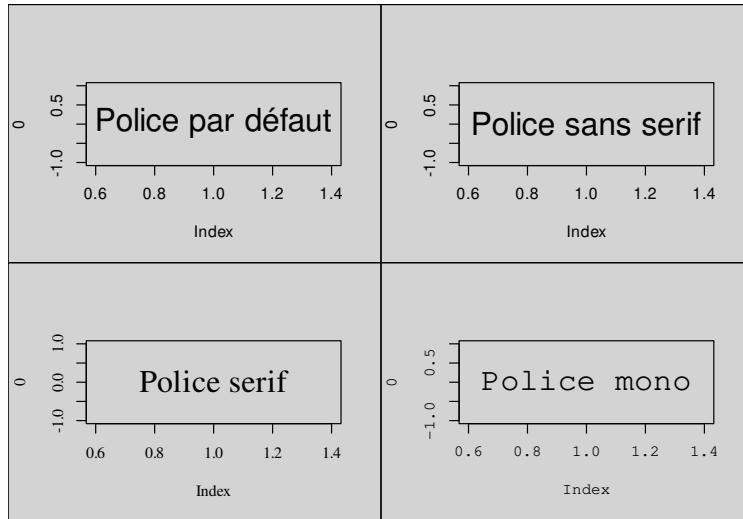
Abordons tout d'abord la notion de fonte de caractères. En typographie, une fonte de caractères est un ensemble de règles qui va déterminer le rendu visuel d'une expression textuelle. Il est très fréquent que police et fonte soient confondues. Une fonte de caractère est caractérisée par : - une police de caractères (*font family* ou *typeface* en anglais); - un style (normal, italique ou oblique); - une graisse (normal ou gras); - un corps (taille de police).

Ainsi, le Helvetica normal 12 points est une fonte de caractères, mais le Helvetica est une police de caractères. De nombreuses classifications existent pour les polices de caractères. Celle que nous présentons ici à l'avantage de se rapprocher des polices disponibles dans R (et dans le monde du Web). Sous R, trois polices principales de caractères sont disponibles :

- sans-serif (noté `sans`) : regroupe les polices sans empattement (c.-à-d. sans les extensions qui viennent terminer les caractères) et à chasse proportionnelle (la largeur des caractères varie en fonction du caractère). Citons le Helvetica, Arial et Verdana comme police sans-serif.
- serif (noté `serif`) : regroupe les polices à empattement et à chasse proportionnelle. C'est le cas du Times (New Roman) et du Garamond.
- monospace (noté `mono`) : possède la caractéristique d'avoir une chasse fixe. Ses polices sont préférées pour l'écriture de code informatique car elles permettent un alignement vertical des caractères. R, sous Windows, utilise la police Courier New et sous Mac, le Monaco. Par défaut, la police `sans` est utilisée

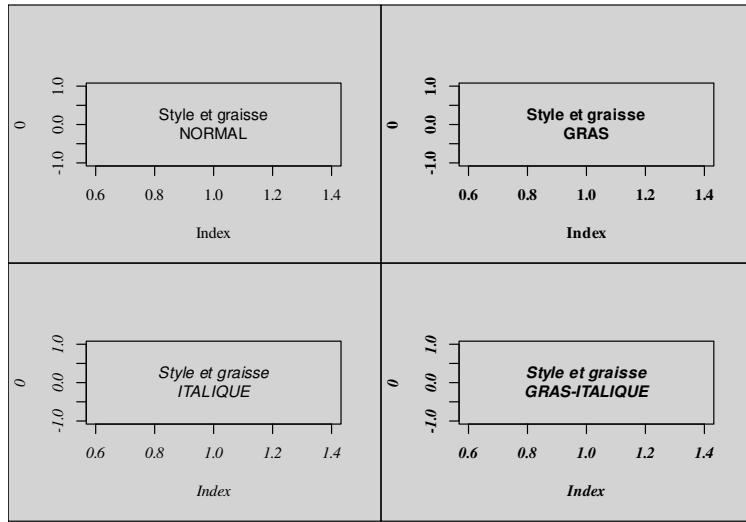
sous R pour afficher l'information textuelle sur les graphes. Cette valeur est stockée dans l'argument `family` du `par()`. Regardons les différences entre ces trois polices de caractères.

```
par(mfrow = c(2, 2), bg = "lightgray")
plot(0, type = "n")
text(1, 0, "Police par défaut", cex = 2)
plot(0, type = "n", family = "sans")
text(1, 0, family = "sans", "Police sans serif", cex = 2)
plot(0, type = "n", family = "serif")
text(1, 0, family = "serif", "Police serif", cex = 2)
plot(0, type = "n", family = "mono")
text(1, 0, family = "mono", "Police mono", cex = 2)
```



Sous R, le style et la graisse sont regroupés sous le même argument : `font`. Mais, celui-ci est plus précis que `family` (qui s'applique sur tous les éléments textuels) dans le sens où il se décline en `font.lab`, `font.main` et `font.sub`. Regardons les différents styles disponibles.

```
par(mfrow = c(2, 2), bg = "lightgray")
plot(0, type = "n", family = "serif", font.lab = 1, font.axis = 1)
text(1, 0, "Style et graisse\nNORMAL", font = 1)
plot(0, type = "n", family = "serif", font.lab = 2, font.axis = 2)
text(1, 0, "Style et graisse\nGRAS", font = 2)
plot(0, type = "n", family = "serif", font.lab = 3, font.axis = 3)
text(1, 0, "Style et graisse\nITALIQUE", font = 3)
plot(0, type = "n", family = "serif", font.lab = 4, font.axis = 4)
text(1, 0, "Style et graisse\nGRAS-ITALIQUE", font = 4)
```

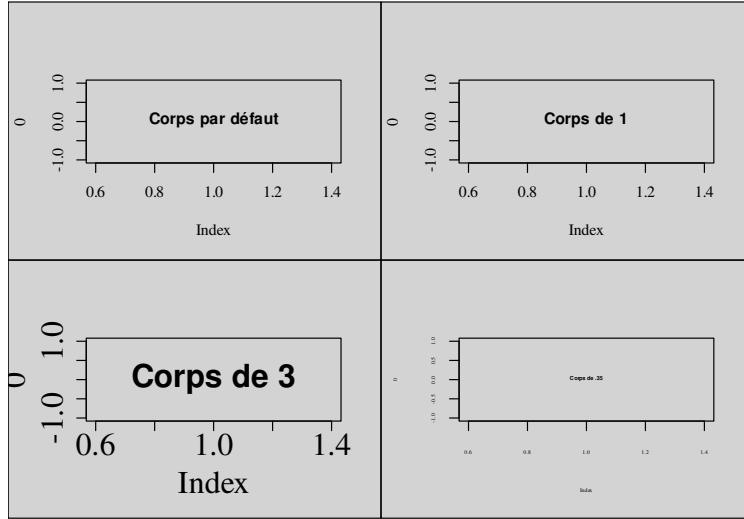


Dans les fonctions `text()` et `mtext()`, seul l'argument `font` est disponible. De plus, tous ces paramètres auraient pu être modifiés dans le `par()` avant de réaliser le graphique. Le corps de police se modifiera avec les arguments `cex.axis`, `cex.lab`, `cex.main`, `cex.sub`. Attention : l'argument `cex` modifie la taille des symboles ponctuels (sauf dans les fonctions `text()` et `mtext()`). Regardons dans le `par()` les valeurs par défaut de chacun de ces paramètres.

```
par() [grep("cex", names(par()))]
#> $cex
#> [1] 1
#>
#> $cex.axis
#> [1] 1
#>
#> $cex.lab
#> [1] 1
#>
#> $cex.main
#> [1] 1.2
#>
#> $cex.sub
#> [1] 1
```

Modifions ces paramètres de corps de police.

```
par(mfrow = c(2, 2), bg = "lightgray")
plot(0, family = "serif", type = "n")
text(1, 0, "Corps par défaut", font = 2)
plot(0, family = "serif", type = "n", cex.lab = 1, cex.axis = 1)
text(1, 0, "Corps de 1", font = 2, cex = 1)
plot(0, family = "serif", type = "n", cex.lab = 2, cex.axis = 2)
text(1, 0, "Corps de 3", font = 2, cex = 2)
plot(0, family = "serif", type = "n", cex.lab = .35, cex.axis = .35)
text(1, 0, "Corps de .35", font = 2, cex = .35)
```

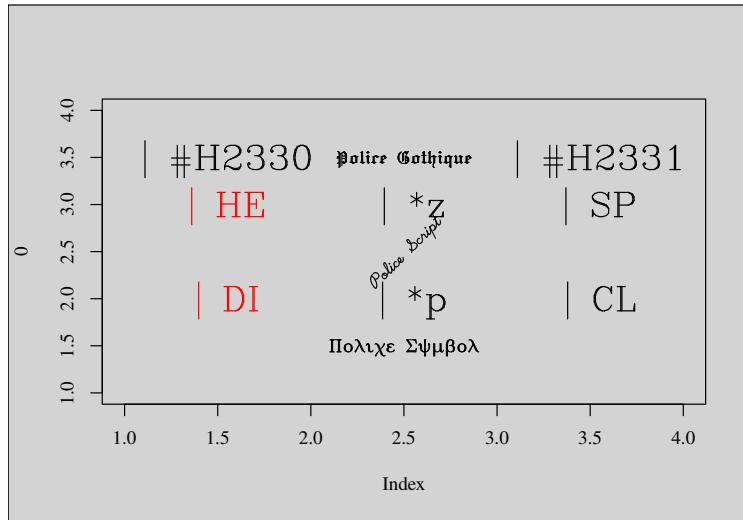


Revenons maintenant aux polices de caractères. Il en existe deux autres sous R et celles-ci sont regroupées dans les fontes **Hershey**. Il s'agit des polices **script** (également appelée *cursive*) qui imite l'écriture manuscrite et **gothic** (ou *fantaisie*) avant tout décorative. Cet ensemble de fontes regroupe des polices permettant d'afficher toute sorte de symboles (grecs, musicaux, japonais, pictogrammes, etc.). Le meilleur moyen d'en faire le tour reste encore d'utiliser la commande suivante.

```
demo(Hershey)
```

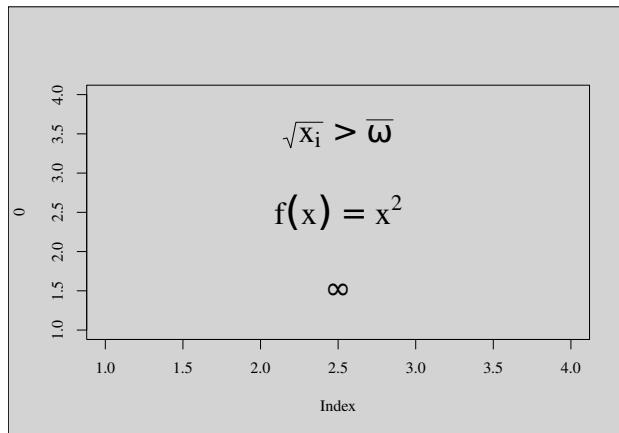
Regardons rapidement comment utiliser ces polices et caractères spéciaux.

```
plot(0, type = "n", xlim = c(1, 4), ylim = c(1, 4), family = "serif")
text(3.5, 2, " | CL", vfont = c("serif", "plain"), cex = 2)
text(1.5, 2, " | DI", vfont = c("serif", "plain"), cex = 2, col = "red")
text(1.5, 3, " | HE", vfont = c("serif", "plain"), cex = 2, col = "red")
text(3.5, 3, " | SP", vfont = c("serif", "plain"), cex = 2)
text(2.5, 3.5, "Police Gothique",
vfont = c("gothic english", "plain"))
text(1.5, 3.5, " | #H2330", vfont = c("serif", "plain"), cex = 2)
text(3.5, 3.5, " | #H2331", vfont = c("serif", "plain"), cex = 2)
text(2.5, 3, " | *z", vfont = c("serif", "plain"), cex = 2)
text(2.5, 2, " | *p", vfont = c("serif", "plain"), cex = 2)
text(2.5, 2.5, "Police Script",
vfont = c("script", "italic"), srt = 45)
text(2.5, 1.5, "Police Symbol",
vfont = c("serif symbol", "bold"))
```



Finalement, regardons comment ajouter des expressions mathématiques sur un graphe avec la fonction `expression`. N'hésitez pas à vous reporter aux rubriques d'aide de cette fonction et de la fonction `plotmath()` pour en savoir plus.

```
plot(0, type = "n", xlim = c(1, 4), ylim = c(1, 4))
text(2.5, 2.5, expression(f(x) == x^2), cex = 2)
text(2.5, 1.5, expression(infinity), cex = 2)
text(2.5, 3.5, expression(sqrt(x[i]) > bar(omega)), cex = 2)
```

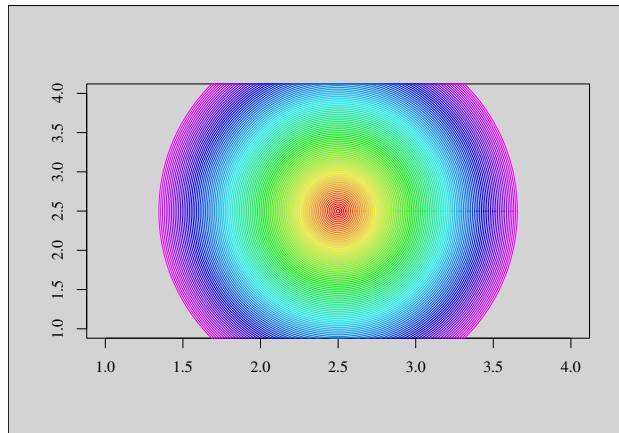


7.3 Symboles ponctuels

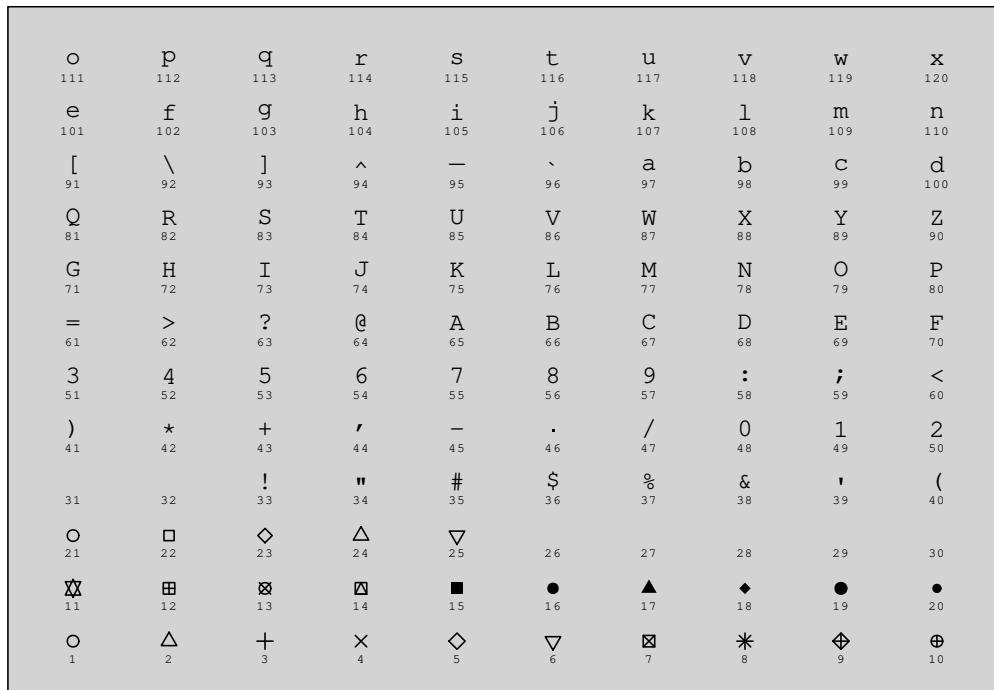
Les points possèdent trois caractéristiques : un type de symbole, une taille et une couleur. Le type de symbole est défini par l'argument `pch`, sa taille par `cex`, et sa couleur par `col`. Modifions les deux derniers. Nous allons incrémenter progressivement le symbole par défaut de R, et à chaque augmentation de taille, on va attribuer une couleur différente, en respectant l'ordre des couleurs dans l'arc-en-ciel.

```
k <- 1
plot(0, xlim = c(1, 4), ylim = c(1, 4), type = "n", ann = FALSE)
for (i in seq(.5, 50, by = .5)){
  points(2.5, 2.5, cex = i, col = rainbow(120)[k])
```

```
k <- k + 1
}
```

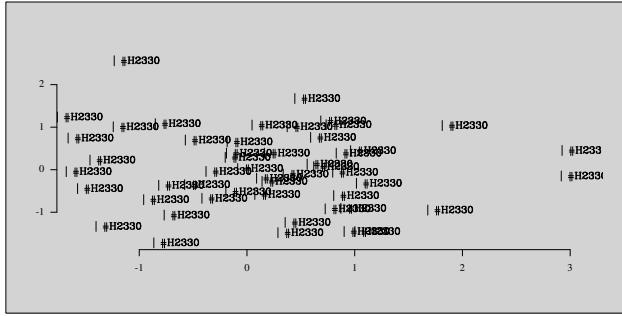


Attention, certains symboles sont caractérisés par deux couleurs : le contour et le fond. L'argument `col` contrôlera la couleur de contour alors que l'argument `bg` définira la couleur de fond. C'est le cas notamment des symboles 21 à 25. La figure suivante illustre différentes valeurs possibles pour l'argument `pch` qui contrôle le type de symbole. Notons que la valeur **1** sera différente de la valeur '**1**', la première affichant le premier symbole alors que le second affichera la valeur 1.



Finalement, mentionnons qu'il est possible d'insérer des symboles `Hershey` via la fonction `text()`.

```
x <- rnorm(50)
y <- rnorm(50)
plot(x, y, type = "n", ann = FALSE, bty = "n", las = 1)
text(x, y, " | #H2330", vfont = c("serif", "plain"))
```



7.4 Types de lignes

Les lignes, tout comme les bordures de polygones et les axes, possèdent trois caractéristiques sur lesquelles on peut jouer : le type de ligne (`lty`), son épaisseur (`lwd`) et sa couleur (`col`). La figure suivante illustre ces différentes caractéristiques (inutile de détailler la couleur).



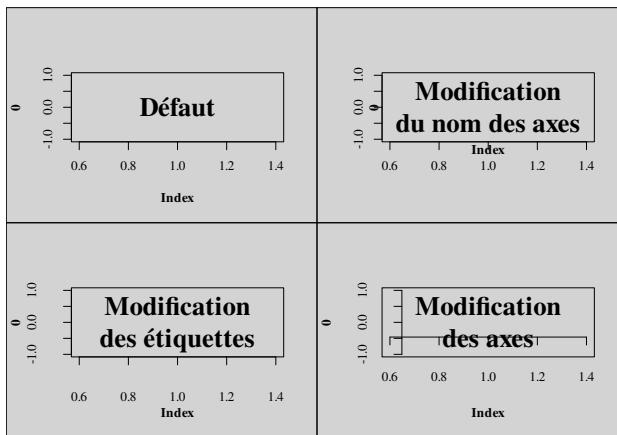
7.5 Modification des axes

Les axes sont un des éléments graphiques possédant probablement le plus grand nombre de paramètres modifiables. Et pour cause, c'est un élément clé pour la compréhension de l'information illustrée sur le graphique. Les arguments `cex.axis`, `col.axis`, `font.axis` ont déjà été abordés dans les sections précédentes. Nous n'y reviendrons pas. Les arguments `xaxt` et `yaxt` vont contrôler l'affichage des axes : s'ils ont la valeur ‘n’, les axes ne seront pas affichés après l'appel à des fonctions de haut niveau graphique (e.g. `plot()`). Cela aura le même effet qu'utiliser l'argument `axes` de la fonction `plot()` et de lui attribuer la valeur `FALSE`. Mais, dans ce dernier cas, la boîte délimitant la région du plot ne sera pas affichée non plus. Un autre argument peut être intéressant. Il s'agit de `mgp`. Celui-ci possède trois valeurs numériques qui vont contrôler le positionnement du nom des axes, des étiquettes des axes et des axes eux-mêmes. Ces positions sont relatives à la délimitation de la région graphique.

```

plot(0, pch = 15, type = "n")
text(1, 0, "Défaut", font = 2, cex = 2)
par(mgp = c(0, 1, 0))
plot(0, pch = 15, type = "n")
text(1, 0, "Modification\ndu nom des axes", font = 2, cex = 2)
par(mgp = c(3, 2, 0))
plot(0, pch = 15, type = "n")
text(1, 0, "Modification\ndes étiquettes", font = 2, cex = 2)
par(mgp = c(3, 1, -1.25))
plot(0, pch = 15, type = "n")
text(1, 0, "Modification\ndes axes", font = 2, cex = 2)

```



Le tableau suivant liste les paramètres graphiques se rapportant aux axes qu'il est intéressant de connaître.

Table 2: Arguments de `par()`

Argument	Signification	<code>par()</code>	À la volée
<code>ann</code>	Contrôle la présence du nom des axes	x	x
<code>axes</code>	TRUE ou FALSE (pas d'axes ni de boîte)	& x	
<code>cex.axis</code>	Taille de caractères des étiquettes des axes	x	x
<code>cex.lab</code>	Taille de caractères du nom des axes	x	x
<code>col.axis</code>	Couleur des axes et de leurs étiquettes	x	x
<code>col.lab</code>	Couleur du nom des axes	x	x
<code>col.ticks</code>	Couleur de la graduation	& x	
<code>font.axis</code>	Style et graisse des étiquettes	x	x
<code>font.lab</code>	Style et graisse du nom des axes	x	x
<code>las</code>	Orientation des étiquettes des axes (0 , 1 , 2 , 3)	x	x
<code>mgp</code>	Voir page précédente	x	
<code>tck</code>	Longueur de la graduation	x	x
<code>tick</code>	TRUE ou FALSE (pas de graduation)	& x	
<code>lty</code>	Type de tracé des axes	x	x
<code>lty.ticks</code>	Type de tracé de la graduation	& x	
<code>lwd</code>	Épaisseur des axes	x	x
<code>lwd.ticks</code>	Épaisseur de la graduation	& x	
<code>xaxp</code>	Nombre de graduation en abscisse	x	x
<code>xaxs</code>	' r ' (ajout de 4% aux limites de l'axe) ou ' i '	x	x
<code>xaxt</code>	TRUE ou FALSE (pas d'axe des x)	x	x
<code>xlab</code>	Nom de l'axe des x	& x	

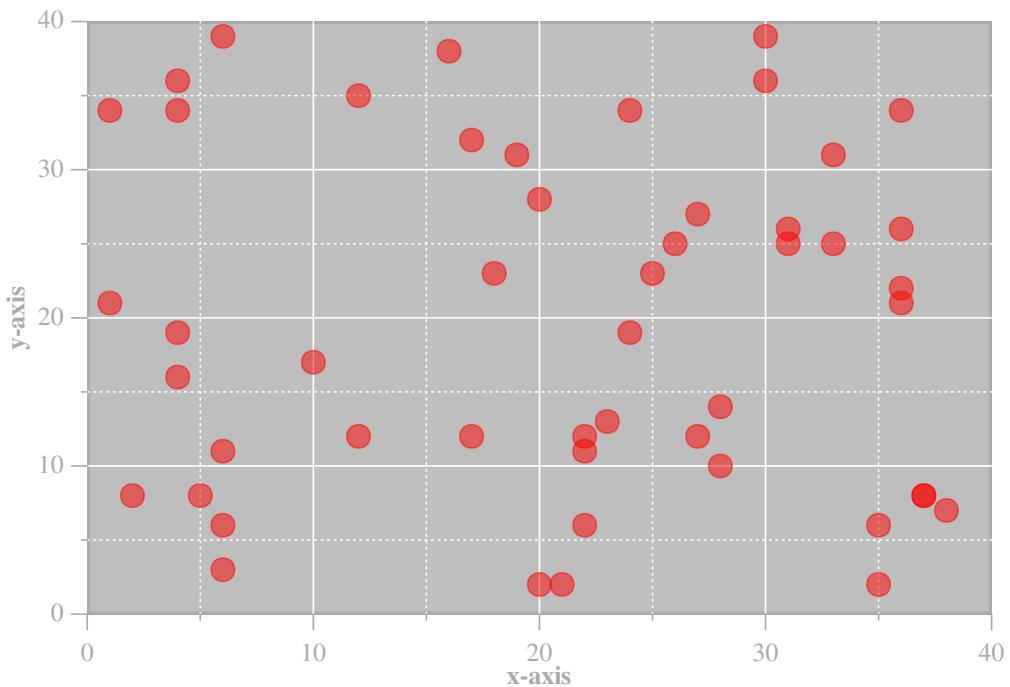
Argument	Signification	par()	À la volée
xlim	Étendue de l'axe des x	& x	
yaxp	Nombre de graduation en ordonnée	x	x
yaxs	Voir <code>xaxs</code>	x	x
yaxt	Voir <code>xaxt</code>	x	x
ylab	Nom de l'axe des y	& x	
ylim	Étendue de l'axe des y	& x	

Une modification à la volée signifie que le paramètre en question verra sa valeur par défaut changée dans les fonctions graphiques telles que `plot()`, `axis()`, etc. Le paramètre `las` pourra prendre les valeurs :

- `las = 0` : étiquettes parallèles aux axes;
- `las = 1` : étiquettes horizontales;
- `las = 2` : étiquettes perpendiculaires aux axes;
- `las = 3` : étiquettes verticales.

Pour terminer, regardons un exemple faisant appel à certains de ces paramètres graphiques.

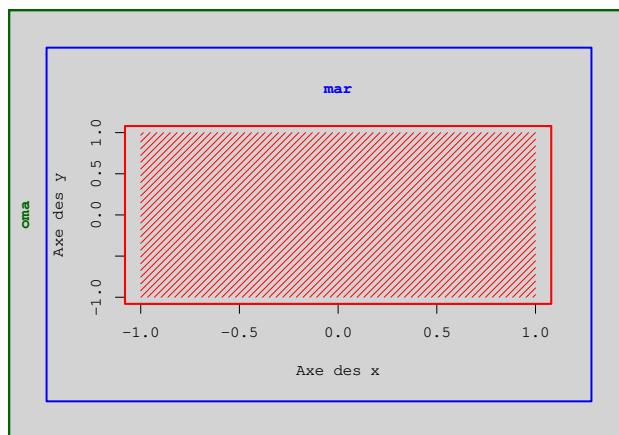
```
## Empty plot
par(mgp = c(1.75, 0.75, 0))
plot(0, type = "n", xlim = c(0, 40), ylim = c(0, 40), axes = FALSE, ann = FALSE, xaxs = "i", yaxs = "i")
## Background
rect(0, 0, 40, 40, col = "gray", border = "darkgray", lwd = 3)
for (i in seq(10, 30, 10)){
  points(c(0, 40), c(i, i), col = "white", type = "l")
  points(c(i, i), c(0, 40), col = "white", type = "l")
}
for (i in seq(5, 35, 10)){
  points(c(0, 40), c(i, i), col = "white", type = "l", lty = 3)
  points(c(i, i), c(0, 40), col = "white", type = "l", lty = 3)
}
## Axes principaux
axis(side = 1, at = seq(0, 40, by = 10), labels = seq(0, 40, by = 10), lwd = 0, pos = 0, lwd.ticks = 1,
col = "darkgray", family = "serif", col.axis = "darkgray")
axis(side = 2, at = seq(0, 40, by = 10), labels = seq(0, 40, by = 10), lwd = 0, pos = 0, lwd.ticks = 1,
col = "darkgray", family = "serif", las = 2, col.axis = "darkgray")
## Axes secondaires
axis(side = 1, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, pos = 0, tck = -0.01, lwd.ticks = 1,
col.ticks = "darkgray")
axis(side = 2, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, pos = 0, tck = -0.01, lwd.ticks = 1,
col.ticks = "darkgray")
## Nom des axes
mtext(text = "x-axis", side = 1, line = 1.5, family = "serif", font = 2, col = "darkgray")
mtext(text = "y-axis", side = 2, line = 1.75, family = "serif", las = 0, font = 2, col = "darkgray")
## Informations
x <- sample(1:39, 50, replace = TRUE)
y <- sample(1:39, 50, replace = TRUE)
points(x, y, col = "#FF000080", pch = 19, cex = 2)
```



7.6 Ajustement des marges

Les marges sont également une notion importante d'un graphique. Plusieurs paramètres graphiques permettent de les contrôler. Nous n'en verrons que deux : `oma` (pour *outer margin*) et `mar` (pour *figure margin*). Regardons à quoi elles correspondent au travers de deux exemples.

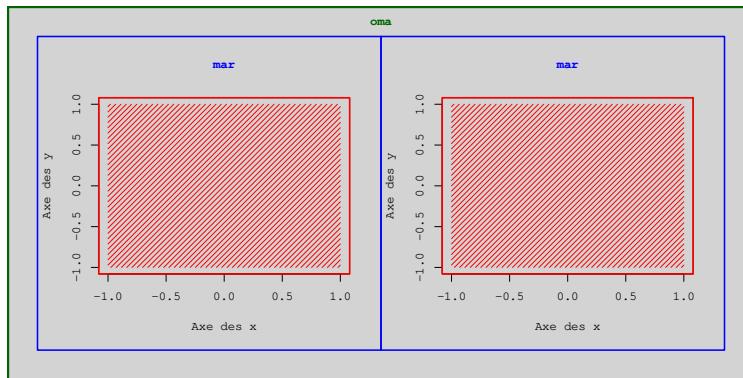
```
par(oma = c(2, 2, 2, 2), bg = "lightgray", family = "mono")
plot(0, 0, type = "n", xlab = "Axe des x", ylab = "Axe des y")
box("plot", col = "red", lwd = 2)
rect(-1, -1, 1, 1, border = NA, col = "red", density = 20, angle = 45)
box("figure", col = "blue", lwd = 2)
box("outer", col = "darkgreen", lwd = 4)
mtext(side = 2, line = 4.75, text = "oma", col = "darkgreen", font = 2)
mtext(side = 3, line = 1.5, text = "mar", col = "blue", font = 2)
```



```

par(oma = c(2, 2, 2, 2), bg = "lightgray", mfrow = c(1, 2))
par(family = "mono")
plot(0, 0, type = "n", xlab = "Axe des x", ylab = "Axe des y")
box("plot", col = "red", lwd = 2)
rect(-1, -1, 1, 1, border = NA, col = "red", density = 20, angle = 45)
box("figure", col = "blue", lwd = 2)
mtext(side = 3, line = 1.75, text = "mar", col = "blue", font = 2)
plot(0, 0, type = "n", xlab = "Axe des x", ylab = "Axe des y")
box("plot", col = "red", lwd = 2)
rect(-1, -1, 1, 1, border = NA, col = "red", density = 20, angle = 45)
box("figure", col = "blue", lwd = 2)
mtext(side = 3, line = 1.75, text = "mar", col = "blue", font = 2)
box("outer", col = "darkgreen", lwd = 4)
mtext(side = 3, line = 0.5, text = "oma", col = "darkgreen", font = 2, outer = TRUE)

```



L'espace compris entre les bordures verte et bleue correspond à la marge extérieure à la région graphique (*outer margin*). Elle est contrôlée par le paramètre `oma`. Par défaut, sa valeur est :

```

par()$oma
#> [1] 0 0 0 0

```

Ces quatre chiffres correspondent respectivement aux marges en bas, à gauche, en haut et à droite. Comme on peut le voir sur la figure 3.13, ce paramètre peut être intéressant à ajuster dans le cas d'une figure composite. En effet, il permettra de définir des marges communes à tous les graphes de la figure. L'argument `mar`, quant à lui, contrôle la taille de la région du plot (excluant les axes). Son ajustement est donc très important. Par défaut, il vaut :

```

par()$mar
#> [1] 5.1 4.1 4.1 2.1

```

La marge du bas est destinée à accueillir à la fois le nom de l'axe des x et un sous-titre : pour cela, sa valeur est plus importante que les autres marges. Notamment, la marge de droite, qui par défaut, ne contiendra aucun élément. Il n'existe pas de règle absolue concernant ce paramètre : la conception graphique de la figure guidera sa définition.

```

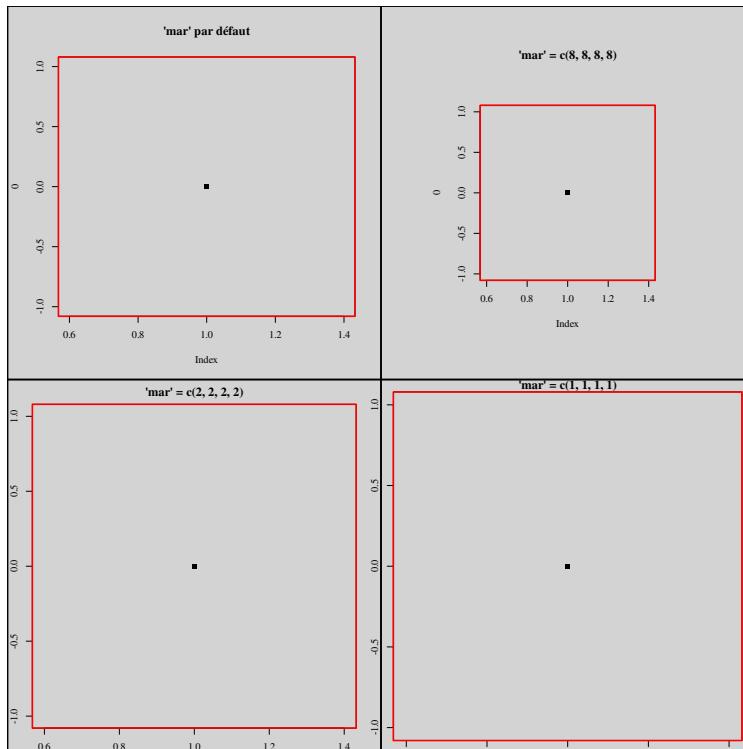
par(mfrow = c(2, 2), bg = "lightgray", family = "serif")
plot(0, pch = 15, main = "'mar' par défaut")
box("figure", lwd = 2)
box("plot", col = "red", lwd = 2)

```

```

par(mar = c(8, 8, 8, 8))
plot(0, pch = 15, main = "'mar' = c(8, 8, 8, 8)")
box("figure", lwd = 2)
box("plot", col = "red", lwd = 2)
par(mar = c(2, 2, 2, 2))
plot(0, pch = 15, main = "'mar' = c(2, 2, 2, 2)")
box("figure", lwd = 2)
box("plot", col = "red", lwd = 2)
par(mar = c(1, 1, 1, 1))
plot(0, pch = 15, main = "'mar' = c(1, 1, 1, 1)")
box("figure", lwd = 2)
box("plot", col = "red", lwd = 2)

```



7.7 Les couleurs sous R

Terminons ce chapitre par les couleurs. C'est un domaine très vaste. En informatique, il existe plusieurs systèmes de représentation des couleurs. On peut utiliser des palettes de couleurs prédéfinies, le système Rouge-Vert-Bleu, RVB (ou *RGB* en anglais) ou encore le système hexadécimal. D'autres systèmes existent, mais nous ne les verrons pas aujourd'hui. Commençons par le plus simple : les palettes. R dispose d'une palette de base dans laquelle figurent huit couleurs prédéfinies.

```

palette()
#> [1] "black"    "#DF536B"  "#61D04F"  "#2297E6"  "#28E2E5"  "#CDOBBC" "#F5C710"
#> [8] "gray62"

```

Cette palette, bien que peu garnie, est intéressante, car elle permet de choisir une couleur par son nom ou par sa position dans le vecteur `palette()`. Outre cette palette, R met à notre disposition la palette `colors()` qui comporte 657 couleurs, chacune avec un nom.

```
colors() [1:8]
#> [1] "white"          "aliceblue"        "antiquewhite"    "antiquewhite1"
#> [5] "antiquewhite2"  "antiquewhite3"  "antiquewhite4"  "aquamarine"
```

D'autres palettes existent sous R, mais nous en parlerons plus loin, car nous devons voir avant certaines notions importantes de colorimétrie. Qu'est-ce-qu'une couleur ? En informatique, on utilise souvent (mais pas tout le temps) la synthèse additive des trois couleurs primaires : Rouge-Vert-Bleu. Dans ce système, 100% de rouge, 100% de vert et 100% de bleu donnera du blanc. En quantités égales, on obtiendra du gris dont la teinte dépendra de la quantité de couleur. Les valeurs de chaque couleur primaire s'étalonneront de 0 à 1 (100%) ou de 0 à 255.

Nom	Rouge	Vert	Bleu
Rouge	100%	0%	0%
Bleu	0%	0%	100%
Vert	0%	100%	0%
Noir	0%	0%	0%
Blanc	100%	100%	100%
Gris clair	80%	80%	80%
Gris foncé	20%	20%	20%
Cyan	0%	100%	100%
Magenta	100%	0%	100%
Jaune	100%	100%	0%

La fonction `rgb()` permet de construire des couleurs en fournissant la quantité de chaque couleur primaire. Par défaut, ces quantités doivent être indiquées dans l'intervalle [0, 1]. Mais, l'argument `maxColorValue` permet de modifier cet intervalle. Ainsi, les trois graphiques générés avec les commandes suivantes seront identiques :

```
k <- rgb(red = 1, green = 0, blue = 1)
plot(0, pch = 15, cex = 10, col = k)
k <- rgb(100, 0, 100, maxColorValue = 100)
plot(0, pch = 15, cex = 10, col = k)
k <- rgb(255, 0, 255, maxColorValue = 255)
plot(0, pch = 15, cex = 10, col = k)
```

Une autre fonction intéressante est la fonction `col2rgb()`. Celle-ci convertie le nom d'une couleur (prédéfinie dans les palettes de R) en code RGB dans l'intervalle [0, 255]. Elle permet aussi de convertir un code hexadécimal.

```
col2rgb("red")
#>      [,1]
#> red     255
#> green    0
#> blue     0
col2rgb("skyblue")
#>      [,1]
#> red     135
#> green   206
#> blue    235
col2rgb(c("red", "cyan", "lightgray", "salmon"))
#>      [,1] [,2] [,3] [,4]
```

```
#> red    255    0  211  250
#> green   0  255  211 128
#> blue    0  255  211 114
```

Et le codage en hexadécimal dans tout ça. C'est quoi ? Wikipédia nous dit que c'est un *système de numération positionnel en base 16*. Plutôt brutal... En version courte, c'est une manière de représenter les nombres différemment du système classique : le système décimal que l'on connaît tous. A la différence du système binaire (système reposant sur une base 2 [0 et 1]), le système hexadécimal utilise 16 symboles : les dix chiffres arabes et les six premières lettres de l'alphabet (de A à F). Le tableau suivant donne la correspondance avec le système décimal.

Hexadécimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Décimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pour retranscrire des couleurs, on utilisera six caractères hexadécimaux : les deux premiers pour le rouge, les deux suivants pour le vert et les deux derniers pour le bleu. Le tout précédé du symbole dièse. Voici le code hexadécimal de quelques couleurs.

Nom	Code hexadécimal
Rouge	\#FF0000
Bleu	\#0000FF
Vert	\#00FF00
Noir	\#000000
Blanc	\#FFFFFF
Gris clair	\#CCCCCC
Gris foncé	\#333333
Cyan	\#00FFFF
Magenta	\#FF00FF
Jaune	\#FFFF00

Comment convertir une valeur RGB en écriture hexadécimale ? Là encore, c'est très simple : on prend la quantité de rouge (exprimée dans un intervalle [0, 255]), et on la divise par 16. Ensuite, on prend le modulo (partie entière de la division), et on le convertit en hexadécimal d'après la correspondance donnée dans le tableau 3.3. On obtient ainsi le premier caractère hexadécimal de la couleur rouge. Puis, on fait de même avec le reste de la division et on obtient finalement le second symbole hexadécimal de la couleur rouge. On procède de même pour les deux autres couleurs RGB et voilà, une couleur exprimée en code hexadécimal. Voyons un exemple avec le gris clair pour lequel la quantité de chaque couleur primaire est 80% (soit 204 dans un intervalle [0, 255]).

```
## Premier caractère hexa du gris clair
204%/%16
#> [1] 12
## Soit C en hexadécimal
## Second caractère hexa du gris clair
204%/%16
#> [1] 12
## Soit C en hexadécimal
## Code complet
hexa <- "#CCCCCC"
## Vérification
col2rgb(hexa)
```

```
#>      [,1]
#> red    204
#> green  204
#> blue   204
```

Malheureusement, il n'existe pas de fonction sous R pour convertir une couleur en hexadécimal. Nous allons donc en créer une qui permettra de convertir en écriture hexadécimale soit un nom de couleur (présent dans les palettes de R), soit un code RGB. Cette fonction marchera pour plusieurs couleurs simultanément. Dans le cas des noms, ils devront être dans un vecteur, alors que pour les codes RGB, ils devront être dans une matrice telle que celle obtenue après l'appel à la fonction `col2rgb()`. Nous allons appeler cette fonction `col2hex()`. Notons que le degré de transparence sera pris en compte. Commençons par définir cette fonction.

```
col2hex <- function(cols, maxColorValue = 1){
  if (missing(cols))
    stop("Color(s) argument is missing.")
  if (is.matrix(cols)){
    if (nrow(cols) > 4)
      stop("Color matrix has to be in the col2rgb format.")
    ncols <- ncol(cols)
  }
  if (is.character(cols)) ncols <- length(cols)
  if (!is.character(cols) && !is.matrix(cols))
    stop("Colors have to be a vector of names or a RGB matrix.")
  mat <- data.frame(Hex = c(0:9, LETTERS[1:6]), Dec = 0:15)
  for (i in 1 : 2) mat[, i] <- as.character(mat[, i])
  hexa <- NULL
  for (i in 1 : ncols){
    loc <- "#"
    if (is.character(cols)){
      col <- tolower(cols[i])
      pos <- which(colors() == col)
      if (length(pos) == 0)
        stop(paste("Color", i, "not found."))
      col <- col2rgb(col)
    }else{
      col <- cols[, i]
      if (min(col) < 0)
        stop("RGB colors are not valid.")
      if (maxColorValue == 1 && max(col) > 1)
        stop("Inappropriate maxColorValue argument.")
      if (maxColorValue != 255)
        col <- col * (255/maxColorValue)
      col <- as.matrix(col)
    }
    for (k in 1 : nrow(col)){
      first <- as.character(col[k, 1] %% 16)
      c1 <- mat[which(mat[, "Dec"] == first), "Hex"]
      secon <- as.character(col[k, 1] %% 16)
      c2 <- mat[which(mat[, "Dec"] == secon), "Hex"]
      loc <- paste(loc, c1, c2, sep = "")
    }
    hexa <- c(hexa, loc)
  }
}
```

```

return(hexa)
}

```

Essayons cette fonction.

```

## Avec des noms de couleurs
col2hex("red")
#> [1] "#FF0000"
col2hex(c("red", "cyan", "skyblue"))
#> [1] "#FF0000" "#00FFFF" "#87CEEB"
## Avec des codes RGB
(color <- col2rgb("red")/255)
#>      [,1]
#> red     1
#> green   0
#> blue    0
col2hex(color)
#> [1] "#FF0000"
(color <- col2rgb(c("red", "cyan", "skyblue")))
#>      [,1] [,2] [,3]
#> red    255    0  135
#> green   0  255  206
#> blue    0  255  235
col2hex(color, maxValue = 255)
#> [1] "#FF0000" "#00FFFF" "#87CEEB"
## Verifications
(color <- col2rgb(c("red", "cyan", "skyblue")))
#>      [,1] [,2] [,3]
#> red    255    0  135
#> green   0  255  206
#> blue    0  255  235
col2rgb(col2hex(color, maxValue = 255))
#>      [,1] [,2] [,3]
#> red    255    0  135
#> green   0  255  206
#> blue    0  255  235
## Avec de la transparence
(color <- col2rgb("#FF000088", alpha = TRUE))
#>      [,1]
#> red    255
#> green   0
#> blue    0
#> alpha   136
col2hex(color, maxValue = 255)
#> [1] "#FF000088"

```

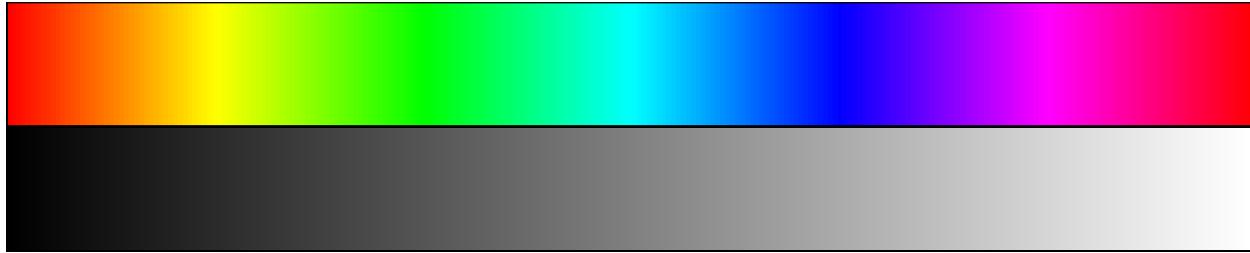
Quelques mots sur la transparence. Le logiciel R gère très bien la transparence des couleurs. Pour une couleur au format RGB, la transparence sera renseignée avec l'argument `alpha`. La valeur 0 signifiera une transparence totale, alors qu'une valeur de 100% (ou 1 ou 255) une opacité complète (valeur par défaut). En hexadécimal, il suffira de rajouter à la fin du code deux autres caractères hexadécimaux indiquant le pourcentage d'opacité. La traduction de décimal à hexadécimal suit la même règle de conversion que pour les couleurs. Ainsi, une totale opacité est équivalente à FF. Mais, pourquoi compliquer les choses en parlant d'hexadécimal ? Il se trouve qu'il existe sous R d'autres palettes de couleurs, mais contrairement aux autres

palettes vu précédemment (`colors()` et `palette()`), elles ne retournent pas des noms de couleurs, mais du code hexadécimal. Voici les deux principales.

```
## Arc-en-ciel
rainbow(24)
#> [1] "#FF0000" "#FF4000" "#FF8000" "#FFBF00" "#FFFF00" "#BFFF00" "#80FF00"
#> [8] "#40FF00" "#00FF00" "#00FF40" "#00FF80" "#00FFBF" "#00FFFF" "#00BFFF"
#> [15] "#0080FF" "#0040FF" "#0000FF" "#4000FF" "#8000FF" "#BF00FF" "#FF00FF"
#> [22] "#FF00BF" "#FF0080" "#FF0040"
## Dégradé de gris
gray(seq(0, 1, length.out = 10))
#> [1] "#000000" "#1C1C1C" "#393939" "#555555" "#717171" "#8E8E8E" "#AAAAAA"
#> [8] "#C6C6C6" "#E3E3E3" "#FFFFFF"
```

Et en image.

```
par(mar = c(0, 0, 0, 0), mfrow = c(2, 1))
image(matrix(1:255, ncol = 1), col = rainbow(255), axes = FALSE)
box("figure", lwd = 2)
image(matrix(c(0:255), ncol = 1), col = gray(c(0:255)/255), axes = FALSE)
box("figure", lwd = 2)
```



Pour terminer ce chapitre, nous vous présentons une fonction que nous avons implémentée et qui pourrait vous être très utile. Celle-ci va vous permettre de retourner le code hexadécimal de couleurs que vous aurez sélectionnées en cliquant sur une palette. Vous aurez le choix des palettes, et vous pourrez récupérer les codes hexadécimaux directement dans la console R. Voici le cœur de la fonction `pickcolor()`.

```
pickcolor <- function(ramp, n){
  ncols <- length(ramp)
  switch(.Platform$OS.type,
  unix = {quartz(width = 7, height = .5)},
  windows = {x11(width = 7, height = .5)})
  par(mar = c(0, 0, 0, 0))
  image(matrix(1 : ncols, ncol = 1), col = ramp, axes = FALSE)
  mat <- as.data.frame(matrix(ncol = 3, nrow = ncols))
  for (i in 1 : ncols){
    xx <- (par()$usr[2]-par()$usr[1])/ncols
    mat[i, 1] <- par()$usr[1]+(i-1)*xx
    xx <- (par()$usr[2]-par()$usr[1])/ncols
    mat[i, 2] <- par()$usr[1]+(i)*xx
    mat[i, 3] <- ramp[i]
  }
  i <- 0
```

```

while (n > i^2)
i <- i + 1
dims <- c(i, i)
xy <- locator(n, type = "p", pch = 4)
switch(.Platform$OS.type,
unix = {quartz(width = 6, height = 6)},
windows = {x11(width = 6, height = 6)})
par(mfrow = dims)
cols <- NULL
for (i in 1 : n){
par(mar = c(0, 0, 0, 0), family = "serif")
pos <- which(mat[, 1] <= xy$x[i] | mat[, 2] >= xy$x[i])
image(matrix(1), col = mat[pos, 3], axes = FALSE)
rvb <- col2rgb(mat[pos, 3])
if (rvb[1, 1] == rvb[2, 1] && rvb[1, 1] == rvb[3, 1] && rvb[1, 1] < 50){
text(0, 0, mat[pos, 3], cex = 2, col = "white")
} else{
text(0, 0, mat[pos, 3], cex = 2)
}
box("figure", col = "lightgray")
cols <- c(cols, mat[pos, 3])
}
return(cols)
}

```

Cette fonction possède deux arguments :

- **ramp** : une palette de couleurs (vecteur de couleurs hexadécimales);
- **n** : nombre de couleurs à cliquer. Voyons un premier exemple avec la fonction **gray()**.

```
pickcolor(ramp = gray(c(0:255)/255), n = 9)
```



Après avoir cliquer neuf fois sur cette palette, les neuf codes hexadécimaux sont retournés dans la console et la figure suivante s'affiche.

La fonction **colorRampPalette()** du package **graphics** permet de créer ses propres palettes de couleurs. Il suffit pour cela d'indiquer un certain nombre de couleurs (minimum deux), et cette fonction retournera une fonction d'interpolation entre ces couleurs. Il suffira d'utiliser cette nouvelle fonction pour créer sa rampe de couleur. Dans l'exemple ci-dessous, on génère une palette de 255 couleurs partant du blanc et arrivant au rouge, en passant par le jaune.

```

rampcols <- colorRampPalette(c("white", "yellow", "red"))
rampcols(255)[1:12]
#> [1] "#FFFFFF" "#FFFFFC" "#FFFFFA" "#FFFFF8" "#FFFFF6" "#FFFFF4" "#FFFFF2"
#> [8] "#FFFFF0" "#FFFFEE" "#FFFFEC" "#FFFFEA" "#FFFFE8"

```

Utilisons cette palette de couleurs avec notre fonction interactive.

```
pickcolor(ramp = rampcols(255), n = 16)
```



Ceci clôture notre chapitre sur les paramètres graphiques. Les deux chapitres suivants sont un peu plus avancés mais ils vont vous permettre d'automatiser vos productions graphiques et de créer des compositions aussi esthétiques que sous *Adobe Illustrator*. Ou presque...

8 Périphériques et exportation

Dans ce chapitre, nous allons apprendre à exporter un graphique en lignes de commande. Ceci est vital pour toute procédure d'automatisation. Imaginez que vous deviez produire des centaines de graphiques. Il ne vous viendrait pas à l'esprit (du moins nous l'espérons) de devoir cliquer pour sauvegarder un à un chacun de vos graphes. Mais, avant de parler de cette étape d'exportation, nous devons développer la notion de périphérique graphique.

8.1 Types de périphériques

Qu'est-ce-qu'un périphérique graphique ? Nous avons déjà mentionné dans l'introduction, que lorsqu'on appelle une *High-level plotting function* (e.g. la fonction `plot()`), ceci a pour conséquence d'ouvrir une fenêtre graphique dans laquelle sera affichée l'information visuelle souhaitée. Et bien, cette fenêtre est un périphérique graphique. Cependant, c'est un périphérique un peu particulier sous R : on parle de **périphérique graphique interactif**, dans le sens où l'on voit le résultat de la commande à l'écran (le graphe). Mais, sachez que la plupart des périphériques de sortie disponibles dans R sont ce qu'on va appeler des **périphériques d'exportation**. Et lors de leur sollicitation, l'utilisateur ne verra aucun graphique s'afficher à l'écran. Nous y reviendrons à la fin de ce chapitre. Le type de périphérique graphique interactif que vous allez utiliser dépend de votre système d'exploitation. Sous Windows, le moteur graphique de base est X11, alors que sous les machines Unix, c'est le système QUARTZ qui prévaut (bien que le moteur X11 puisse être installé et qu'il soit le moteur par défaut sélectionné lorsque R est utilisé dans le SHELL). Sous Mac OSX, QUARTZ est appelé AQUA.

```
## Système d'exploitation
.Platform$OS.type
#> [1] "unix"
## Moteur graphique
.Platform$GUI
#> [1] "X11"
```

Pour ouvrir un nouveau périphérique graphique, il faudra utiliser la commande `x11()` (sous Windows) ou `quartz()` (sous Unix). Voici quelques caractéristiques de ce périphérique graphique.

```
## x11.options()
quartz.options()
#> $title
#> [1] "Quartz %d"
#>
#> $width
#> [1] 7
```

```

#>
#> $height
#> [1] 7
#>
#> $pointsize
#> [1] 12
#>
#> $family
#> [1] "Helvetica"
#>
#> $antialias
#> [1] TRUE
#>
#> $type
#> [1] "native"
#>
#> $bg
#> [1] "transparent"
#>
#> $canvas
#> [1] "white"
#>
#> $dpi
#> [1] NA

```

Ces options sont modifiables. Ainsi, on peut redimensionner une fenêtre graphique à l'ouverture (c'est ce que fait la fonction `pickcolor()` lorsqu'elle affiche la palette de couleurs). Voici comment faire.

```

x11(width = 12, height = 7)
quartz(width = 12, height = 7)

```

Maintenant, une astuce pour ceux qui développent leurs propres fonctions graphiques sous R. Si vous développez des fonctions ou des packages qui seront distribués et donc potentiellement utilisés sur n'importe quel système d'exploitation, vous pouvez utiliser la commande suivante pour ouvrir un nouveau périphérique graphique. Celle-ci s'adapte à n'importe quel OS : Windows et Unix (Mac OSX et Linux).

```

switch(.Platform$OS.type, unix = {quartz()}, windows = {x11()})

```

Nous avons vu que l'appel aux *High-level plotting functions* avait pour conséquence d'ouvrir un nouveau périphérique graphique. C'est vrai si aucun périphérique n'est ouvert. Par contre, si un périphérique graphique est déjà ouvert et actif, son contenu sera remplacé par le nouveau plot, mais les paramètres graphiques spécifiés pour ce périphérique seront eux conservés (sauf s'ils sont modifiés à la volée). En revanche, dès l'ouverture d'un nouveau périphérique graphique, les valeurs des paramètres graphiques sont réinitialisés. En d'autres termes, toute modification directe dans le `par()` est propre à une fenêtre graphique. La commande suivante permet de fermer tous les périphériques graphiques ouverts (même les périphériques d'exportation, cachés à l'utilisateur). Alors attention !!!

```
graphics.off()
```

8.2 Les fonctions `dev.x()`

Cette famille de fonctions permet de manipuler les périphériques graphiques ouverts. Bien que peu utilisées (à l'exception de `dev.off()`), il nous a semblé important de les mentionner ici. Le tableau ci-dessous liste

les fonctions principales.

Fonction	Action
<code>dev.list()</code>	Affiche la liste des périphériques ouverts
<code>dev.cur()</code>	Affiche le périphérique actif
<code>dev.prev()</code>	Affiche le périphérique précédent
<code>dev.next()</code>	Affiche le périphérique suivant
<code>dev.set(n)</code>	Sélectionne le périphérique n
<code>dev.off()</code>	Ferme le périphérique actif
<code>dev.copy()</code>	Copie le contenu d'un périphérique dans un autre

La commande `graphics.off()` sera préférée à `dev.off()` dans le cas où de nombreux périphériques graphiques sont ouverts.

8.3 Exportation d'un graphe

Pour exporter un graphe, c.-à-d. l'enregistrer sur le disque dur, trois possibilités existent. La première, c'est en cliquant. Vous savez sûrement déjà comment faire. La seconde consiste à copier le contenu d'un périphérique graphique AQUA ou X11 dans un périphérique de sortie (e.g. **PDF**, **PNG**, **TIFF**, **Postscript**, etc.). Avant de voir comment procéder, regardez les périphériques de sortie disponibles sur votre système d'exploitation.

```
capabilities()
#>      jpeg      png      tiff      tcltk      X11      aqua
#>      TRUE      TRUE      TRUE      TRUE      FALSE     FALSE
#> http/ftp    sockets libxml      fifo      cledit    iconv
#>      TRUE      TRUE      TRUE      TRUE      FALSE     TRUE
#>      NLS   profmem    cairo    ICU long.double libcurl
#>     FALSE      TRUE      TRUE      TRUE      TRUE     TRUE
```

Les formats **PDF**, **SVG** et **Postscript** ne sont pas présents dans cette liste. En fait, ils sont regroupés sous le type **CAIRO**. Regardons comment exporter un graphe en **PDF** avec la fonction `dev.copy()`.

```
## Production du graphe en mode interactif
x <- rnorm(50)
y <- rnorm(50)
plot(x, y, pch = 15, main = "My plot")
abline(reg = lm(y ~ x), col = "red")
## Exportation
dev.copy(device = pdf)
dev.off()
```

Quelques remarques sur ce qu'on vient de faire. Premièrement, pour que l'exportation s'effectue, il faut fermer la connexion au périphérique de sortie (ici le périphérique **PDF**) avec la commande `dev.off()`. On vient de créer un fichier **PDF**. Les dimensions de ce fichier sont les mêmes que celles du périphérique graphique sous R. De plus, le fichier est exporté dans le répertoire courant et R choisit un nom par défaut. Bien évidemment, nous pouvons spécifier un nom au fichier exporté.

```
plot(x, y, pch = 15, main = "My plot")
abline(reg = lm(y ~ x), col = "red")
dev.copy(device = pdf, "MyPlot.pdf")
dev.off()
```

Remarque : les fonctions `dev.print()` et `dev.copy2pdf()` permettent de faire la même chose. Cependant, dans le cas de la première, si aucun nom n'est spécifié, le périphérique sera envoyé à l'imprimante par défaut à laquelle votre ordinateur est connecté. Donc, attention si vous l'utilisez. Enfin, la troisième façon d'exporter un graphe est d'avoir recours aux *File-based devices*. Derrière ce nom se cachent en fait des périphériques connus de tous : **PDF**, **PNG**, **TIFF**, **Postscript**, etc. Contrairement à la fonction `dev.copy()`, avec ce genre de fonctions, on ouvre le périphérique graphique de sortie avant de faire le graphe. Ceci présente un certain inconvénient : c'est qu'on ne voit pas le résultat s'afficher dans le GUI de R après l'exécution de chaque ligne de code. Ce n'est qu'une fois la connexion au périphérique de sortie coupée qu'on pourra voir le résultat en **PDF** par ex. Voici quelques exemples.

```
## Exportation en PNG.
png("MyPlot2.png")
plot(x, y, pch = 15, main = "My plot 2")
abline(lm(y ~ x))
dev.off()
```

```
# Exportation en PDF.
pdf("MyPlot2.pdf")
plot(x, y, pch = 15, main = "My plot 2")
abline(lm(y ~ x))
dev.off()
```

Bien évidemment, ces périphériques de sortie possèdent des options qu'on peut modifier selon nos propres besoins. Voici les options pour le périphérique **PDF**.

```
pdf.options()
#> $width
#> [1] 7
#>
#> $height
#> [1] 7
#>
#> $onefile
#> [1] TRUE
#>
#> $family
#> [1] "Helvetica"
#>
#> $title
#> [1] "R Graphics Output"
#>
#> $fonts
#> NULL
#>
#> $version
#> [1] "1.4"
#>
#> $paper
#> [1] "special"
#>
#> $encoding
#> [1] "default"
#>
#> $bg
```

```

#> [1] "transparent"
#>
#> $fg
#> [1] "black"
#>
#> $pointsize
#> [1] 12
#>
#> $pagecentre
#> [1] TRUE
#>
#> $colormodel
#> [1] "srgb"
#>
#> $useDingbats
#> [1] FALSE
#>
#> $useKerning
#> [1] TRUE
#>
#> $fillOddEven
#> [1] FALSE
#>
#> $compress
#> [1] TRUE

```

Ainsi, on peut redimensionner le fichier exporté et en modifier la résolution. Cependant, vous verrez qu'il peut être parfois difficile d'ajuster la résolution. Et bien souvent, vous devrez aussi jouer dans le `par()`, notamment au niveau des marges.

```

pdf("MyPlot2ter.pdf", width = 12, height = 6, pointsize = 16)
plot(x, y, pch = 15, main = "My plot 2")
abline(lm(y ~ x))
dev.off()

```

Consultez les rubriques d'aide des fonctions `pdf()` et `png()` pour en savoir plus.

9 Partitionnement et composition

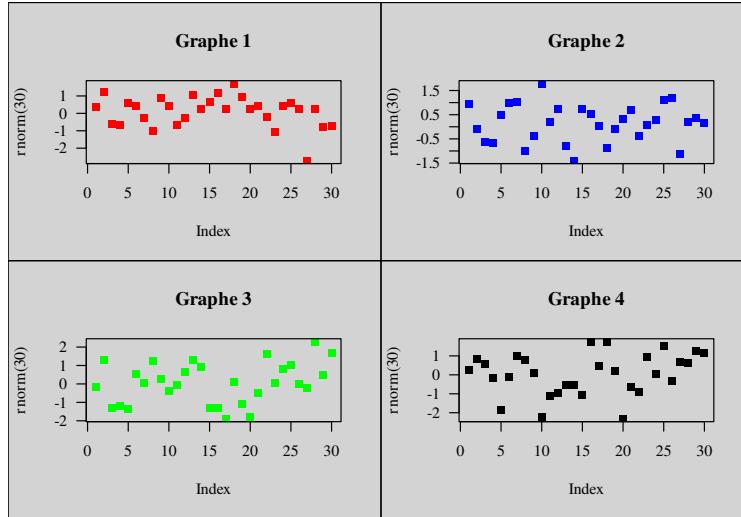
Dans ce dernier chapitre, nous allons voir comment créer des compositions graphiques avancées. Nous allons voir comment partitionner un périphérique graphique afin d'y inclure plusieurs graphes. Nous avons déjà vu l'argument `mfrw` de la fonction `par()`. Nous en rappellerons rapidement les principes, mais nous verrons surtout deux autres fonctions (`layout()` et `split.screen()`) offrant beaucoup plus de souplesse dans l'arrangement des figures au sein du périphérique. Finalement, nous verrons comment inclure un graphique dans un autre graphique, par ex. une inclusion en médaillon. Pour ce faire, nous discuterons de deux derniers paramètres graphiques contenus dans le `par()` : `new` et `fig`.

9.1 Partitionnement basique

Nous avons déjà utilisé l'argument `mfrw` de la fonction `par()` à de multiples reprises. Cet argument permet de partitionner la fenêtre graphique en différentes régions, chacune destinée à accueillir un graphe

différent. Avec `mfrow`, les régions seront remplies en lignes. Il existe un autre paramètre, `mfcol`, avec lequel l'ordre de remplissage des régions partitionnées se fera en colonnes. Mais, son principe d'utilisation est le même que `mfrow` : la première valeur indique le nombre de lignes et la seconde, le nombre de colonnes. Le partitionnement créé avec ces deux arguments possède la caractéristique suivante : toutes les régions graphiques possèdent les mêmes dimensions. Ce qui peut présenter un certain avantage, mais aussi constituer une limite dans la composition de la figure. Notons qu'en ajustant les arguments contrôlant les marges (`mar` et `oma`), il est tout de même possible de faire varier les dimensions des sous-figures. Voici un exemple illustrant l'utilisation de l'argument `mfrow`.

```
par(mfrow = c(2, 2), bg = "lightgray", las = 1, family = "serif")
plot(rnorm(30), pch = 15, col = "red", main = "Graphe 1")
box("figure")
plot(rnorm(30), pch = 15, col = "blue", main = "Graphe 2")
box("figure")
plot(rnorm(30), pch = 15, col = "green", main = "Graphe 3")
box("figure")
plot(rnorm(30), pch = 15, col = "black", main = "Graphe 4")
box("figure")
```



Avec ces arguments, il est possible d'ignorer une région graphique et de passer à la suivante en utilisant la fonction `plot.new()`.

```
par(mfrow = c(1, 3), bg = "lightgray", las = 1, family = "serif")
plot(rnorm(30), pch = 15, col = "red", main = "Graphe 1")
box("figure")
plot.new()
box("figure")
plot(rnorm(30), pch = 15, col = "blue", main = "Graphe 2")
box("figure")
```

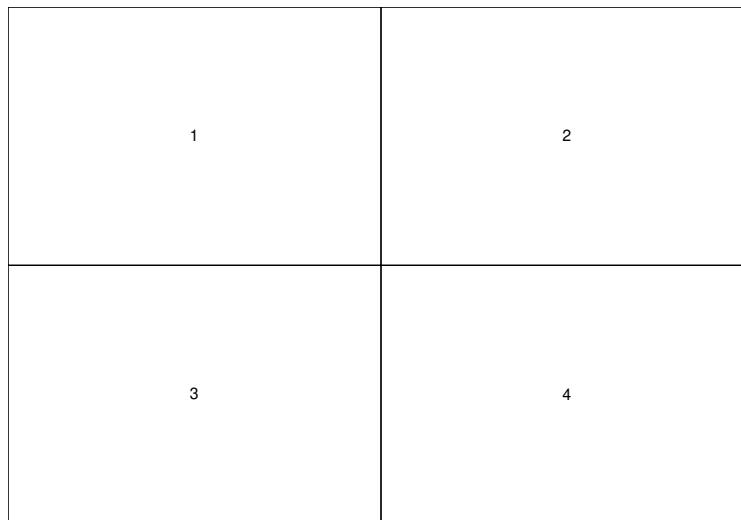


Nous avons fait le tour des possibilités offertes par ces arguments de la fonction `par()`. Regardons maintenant des fonctions plus élaborées.

9.2 Partitionnement avancé

Dans la plupart des situations courantes, le partitionnement basique tel que vu dans la section précédente suffira. Mais, si vous avez besoin de créer des compositions graphiques encore plus poussées, vous allez devoir utiliser les fonctions que nous allons voir maintenant. La première fonction dont nous allons parler est la fonction `layout()` contenue dans le package `graphics`. Celle-ci va diviser la fenêtre graphique d'après le contenu d'une matrice. Regardons un premier exemple afin de faire connaissance avec cette fonction.

```
(mat <- matrix(1:4, ncol = 2, byrow = TRUE))
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
layout(mat)
layout.show(n = 4)
```

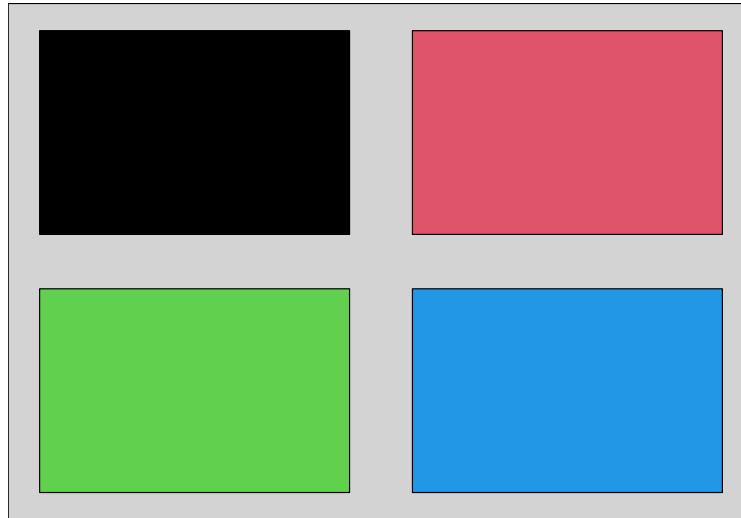


La fonction `layout.show()` permet de visualiser le partitionnement réalisé. L'argument `n` mentionné correspond au nombre de régions graphiques issues du partitionnement que l'on souhaite afficher. Par ex., si `n` avait pris la valeur **2**, seules les deux premières régions graphiques auraient été affichées. L'ordre de remplissage est dicté par les numéros des régions graphiques. Dans notre cas, le remplissage se fera par ligne. Vérifions.

```

layout(mat)
par(bg = "lightgray")
for (i in 1 : 4){
  par(mar = c(1, 1, 1, 1))
  plot(c(-1, 1), c(-1, 1), type = "n", ann = FALSE, axes = FALSE)
  rect(-1, -1, 1, 1, col = palette()[i])
}
box("outer")

```



Par contre, si on transposait la matrice, le remplissage se ferait en colonnes. Ici, le résultat est très similaire à ce que nous aurions obtenu avec les arguments `mffrow` ou `mfcoll`. Cependant, la fonction `layout()` permet de partitionner la fenêtre graphique en un nombre impair de régions : c'est ce qui fait sa force, car cela implique un redimensionnement des régions graphiques. Pour ce faire, nous devons modifier la matrice de base de manière à ce que certaines cellules de la matrice possède la même valeur.

```

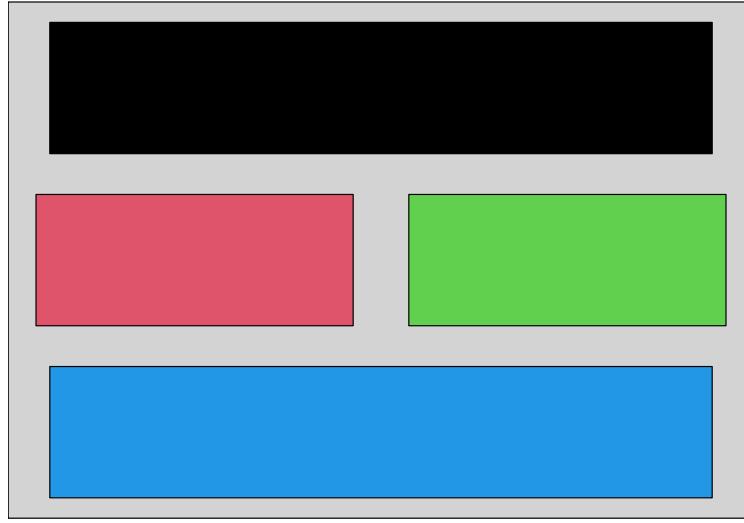
(mat <- matrix(c(1, 1, 2, 3, 4, 4), ncol = 2, byrow = TRUE))
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    3
#> [3,]    4    4

```

```

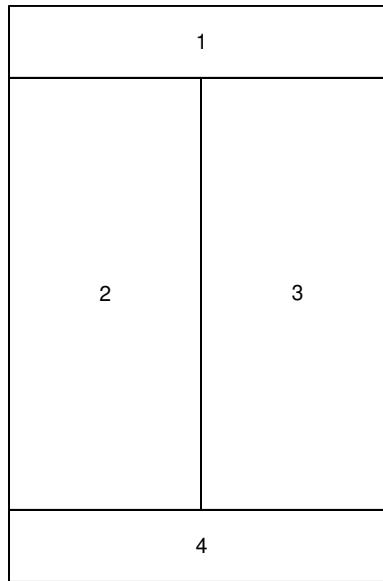
layout(mat)
par(bg = "lightgray")
for (i in 1 : 4){
  par(mar = c(1, 1, 1, 1))
  plot(c(-1, 1), c(-1, 1), type = "n", ann = FALSE, axes = FALSE)
  rect(-1, -1, 1, 1, col = palette()[i])
}
box("outer")

```



Cependant, nous remarquons que les marges définies ont été modifiées pour les régions fusionnées. Ce qui peut être problématique dans la recherche d'un alignement vertical des graphes (ce qui est sûrement le cas vu qu'on touche à une composition très avancée). Il va donc falloir réajuster les marges en fonction de la région graphique. La fonction `layout()` possèdent deux arguments qui vont nous permettre de contrôler les dimensions des régions graphiques : `widths`, `heights`. Le premier va contrôler la largeur des colonnes des régions graphiques, alors que le second s'occupera de la hauteur des lignes. Voyons cela.

```
layout(mat, widths = c(4, 4), heights = c(1, 6, 1))
layout.show(4)
```



Le premier exercice du chapitre 6 présente une utilisation avancée de la fonction `layout()` et de tous ses arguments.

Introduisons maintenant la fonction `split.screen()`. Celle-ci offre encore plus d'interactivité que la fonction `layout()`. Le partitionnement de la fenêtre graphique est dit récursif : chaque région peut-être redivisée autant de fois que souhaité. Mais, la puissance de cette fonction réside dans le fait de pouvoir choisir la région à éditer : il n'y a pas d'ordre de remplissage. De plus, il est facile (même si ce n'est pas recommandé)

de revenir à une région précédemment éditée afin d'y rajouter des éléments (ou d'en effacer son contenu). La sélection d'une région donnée se fera avec la fonction `screen()`.

```
## Division en 3 lignes et 1 colonne
split.screen(figs = c(3, 1))
#> [1] 1 2 3
## Division de la region 2 en 2 colonnes
split.screen(figs = c(1, 2), screen = 2)
#> [1] 4 5
## Region active
screen()
#> [1] 4
## Division de la region 5 en 2 lignes
split.screen(figs = c(2, 1), screen = 5)
#> [1] 6 7
## Region active
screen()
#> [1] 6
## Noms des regions
close.screen()
#> [1] 1 2 3 4 5 6 7
```

Attention, car les régions subdivisées existent toujours. Ainsi, si vous éditez la région **2**, vous éditez également ses sous-régions **4** et **5**, et donc **6** et **7**, les sous-régions de **5**. Nous n'en dirons pas plus sur cette fonction `split.screen()`. Mais, nous vous invitons à consulter la rubrique d'aide de cette fonction si vous êtes intéressés par son potentiel.

9.3 Graphe dans un graphe

Pour terminer, regardons un cas de figure auquel vous serez peut-être confronté un jour. Il s'agit de superposer plusieurs graphes dans une même fenêtre graphique sans avoir recours au partitionnement. La difficulté, c'est que chacun de ces graphes doit être créé avec une *High-level plotting function*, qui par définition, écrasera le contenu du périphérique graphique actif, et donc le graphe précédent. Heureusement, la fonction `par()` met à notre disposition un argument fort utile : l'argument `new`. Celui-ci, s'il prend la valeur **TRUE**, permettra de réinitialiser le système de coordonnées du périphérique ouvert et défini par le graphe précédent. Notons que les paramètres graphiques du périphérique seront eux conservés.

La première situation que nous pouvons rencontrer est la suivante : nous souhaitons superposer deux graphiques qui partagent un même axe (par ex. l'axe des x), mais qui diffèrent en y. Dit autrement, il s'agit de rajouter un second axe y (et les valeurs associées) qui n'a rien à voir avec le premier.

Par exemple, nous pourrions vouloir représenter à la fois la température et les précipitations en fonction de l'altitude sur un même graphique. Cependant, ces variables présentent des valeurs qui ne s'étendent pas sur le même range (environ -40 à +30 degrés Celsius pour la température, et 0 à 2000 millimètres pour les précipitations annuelles cumulées). C'est là qu'intervient l'argument `new` du `par()`. L'idée est donc de faire le graphe de la température en fonction de l'altitude, puis, d'utiliser ce paramètre graphique pour réinitialiser le système de coordonnées de ce graphe, et finalement de redéfinir un nouveau système de coordonnées dans cette même fenêtre en ajoutant le graphe des précipitations en fonction de l'altitude.

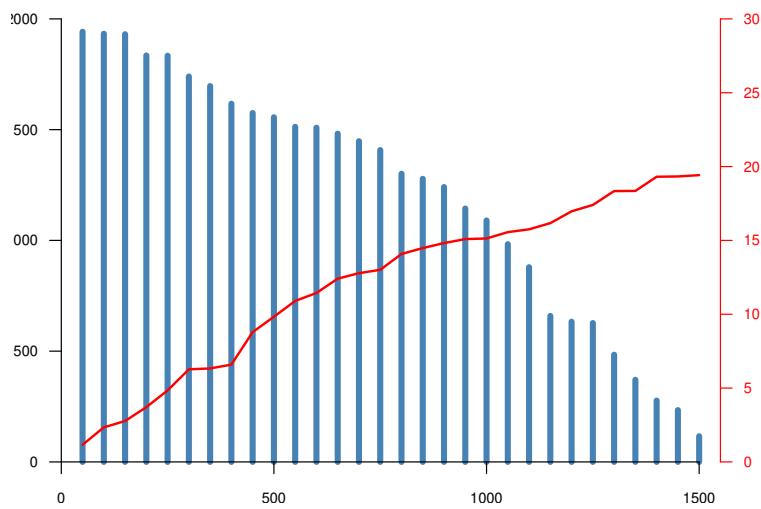
Voyons un exemple avec des données fictives.

```
## Creation des variables
x <- seq(50, 1500, by = 50)
y1 <- sample(100:2000, size = length(x), replace = TRUE)
```

```

y2 <- sort(y1)/100
y1 <- sort(y1, decreasing = TRUE)
## Premier graphe
par(cex.axis = .75, ann = FALSE)
plot(x, y1, col = "steelblue", type = "h", lwd = 5, ylim = c(0, 2000), axes = FALSE, xlim = c(0, 1550))
axis(1, pos = 0, seq(0, 1500, 500), seq(0, 1500, 500))
axis(2, pos = 0, seq(0, 2000, 500), seq(0, 2000, 500), las = 1)
## Second graphe
par(new = TRUE)
plot(x, y2, type = "l", col = "red", lwd = 2, ann = FALSE, ylim = c(0, 30), axes = FALSE, xlim = c(0, 1550))
axis(4, pos = 1550, seq(0, 30, 5), seq(0, 30, 5), las = 1, col = "red", col.axis = "red")

```



Mise à part l'inclusion de la commande `par(new = TRUE)`, tout se passe normalement. En enlevant cette ligne de code, les graphiques s'afficheraient bien (sauf que le second aurait écrasé le premier). Dans le second graphe, nous n'affichons pas l'axe des x puisque celui-ci est déjà tracé dans le premier graphe. L'exercice 2 du chapitre 6 montre un exemple de graphique plus élaboré. Le second cas que vous pourriez rencontrer concerne l'inclusion d'un graphique dans une région restreinte d'un autre graphique. On appelle cela l'inclusion en médaillon. C'est très fréquent en cartographie, le médaillon représente une carte générale et le graphe principal une portion agrandie de ce médaillon. Pour réaliser ce genre de graphique, nous allons encore utiliser le paramètre graphique `new`. Mais, cette fois-ci nous aurons besoin de le combiner avec un autre argument : `fig`. Regardons ses valeurs par défaut.

```

par()$fig
#> [1] 0 1 0 1

```

Ce paramètre définit, dans un format standardisé, le format NDC (*Normalized Device Coordinates*), les coordonnées de la figure dans le périphérique graphique. Les deux premières valeurs correspondent aux minimum et maximum en x, et les deux suivantes les minimum et maximum en y. Par défaut donc, la figure occupera tout l'espace disponible dans le périphérique graphique (marges comprises). Pour inclure une graphe en médaillon, il faudra donc modifier ces valeurs de `fig` après avoir avoir tracer le premier graphe, mais avant d'appeler la fonction qui affichera le second. En fait, on redéfinira ces valeurs dans le `par()` en même temps qu'on modifiera la valeur du paramètre `new`.

Par ex., si on souhaite inclure un médaillon dans le quart supérieur-droit du périphérique, nous utiliserons la commande suivante.

```
par(fig = c(.5, 1, .5, 1), new = TRUE)
```

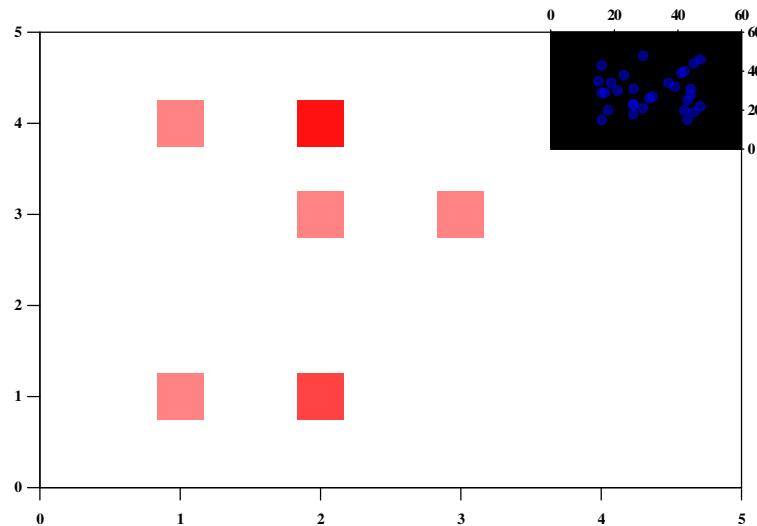
Et pour un médaillon placé au centre,

```
par(fig = c(.25, .75, .25, .75), new = TRUE)
```

Une remarque maintenant : si vous modifiez les marges des graphes, et que vous souhaitez un alignement parfait des graphes, il faudra que les marges des côtés sur lesquels les graphes doivent s'aligner (par ex. marge en haut et à droite, pour un médaillon placé en haut et à droite) soient identiques.

Regardons maintenant un exemple visuel.

```
par(mgp = c(0, .75, 0), xaxs = "i", yaxs = "i")
par(family = "serif", font.axis = 2)
par(mar = c(2, 2, 2, 2), cex.axis = .75)
plot(c(0, 5), c(0, 5), xaxt = "n", yaxt = "n", ann = F, type = "n")
x <- sample(1:3, 10, replace = TRUE)
y <- sample(1:4, 10, replace = TRUE)
points(x, y, cex = 5, col = "#FF00007D", pch = 15)
axis(1, at = 0:5, 0:5, col = par()$col.axis)
axis(2, at = 0:5, 0:5, col = par()$col.axis, las = 1)
par(new = TRUE)
par(fig = c(.7, 1, .7, 1), mar = c(0, 0, 2, 2), cex.axis = .75)
par(xaxs = "i", yaxs = "i", mgp = c(0, .25, 0), tck = -.02)
plot(c(0, 60), c(0, 60), xaxt = "n", yaxt = "n", ann = F, type = "n")
rect(0, 0, 60, 60, col = "black")
x <- sample(15:50, 30, replace = TRUE)
y <- sample(15:50, 30, replace = TRUE)
points(x, y, cex = 1, col = "#0000FF7D", pch = 19)
axis(3, seq(0, 60, 20), seq(0, 60, 20), col = par()$col.axis)
axis(4, seq(0, 60, 20), seq(0, 60, 20), col = par()$col.axis, las = 1)
box("outer", col = "white")
```

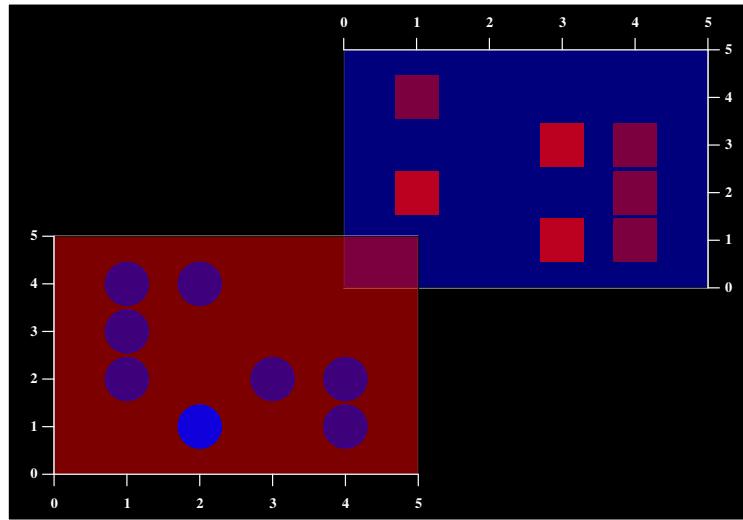


Voilà, ce n'est pas plus compliqué. D'ailleurs, si on y réfléchit bien, si on ne modifie pas le paramètre `fig`, on se retrouverait dans le cas de figure vu dans la section précédente. Maintenant, si vous vous rappelez la

fonction `plotimage()` que nous avons développée au chapitre 2, nous pourrions aussi exporter le graphique qui doit prendre la place du médaillon (en PNG par ex.), puis, le rajouter avec cette fonction `plotimage()` dans le graphe occupant toute la fenêtre graphique. Et nous aurions le choix de la disposition grâce aux arguments implémentés (précision des coordonnées ou positions prédéfinie). Vous pouvez essayer pour voir.

Pour terminer, regardons un dernier exemple pour bien comprendre le rôle de l'argument `fig` de la fonction `par()`.

```
par(bg = "black", mgp = c(0, .75, 0), xaxs = "i", yaxs = "i")
par(family = "serif", font.axis = 2, col.axis = "white", col = "white")
par(fig = c(.45, 1, .45, 1), mar = c(0, 0, 2, 2), cex.axis = .75)
plot(c(0, 5), c(0, 5), xaxt = "n", yaxt = "n", ann = F, type = "n")
rect(0, 0, 5, 5, col = "#0000FF7D")
x <- sample(1:4, 10, replace = TRUE)
y <- sample(1:4, 10, replace = TRUE)
points(x, y, cex = 5, col = "#FF00007D", pch = 15)
axis(3, at = 0:5, 0:5, col = par()$col.axis)
axis(4, at = 0:5, 0:5, col = par()$col.axis, las = 1)
par(new = TRUE)
par(fig = c(0, .55, 0, .55), mar = c(2, 2, 0, 0), cex.axis = .75)
par(xaxs = "i", yaxs = "i", mgp = c(0, .75, 0))
plot(c(0, 5), c(0, 5), xaxt = "n", yaxt = "n", ann = F, type = "n")
rect(0, 0, 5, 5, col = "#FF00007D")
x <- sample(1:4, 10, replace = TRUE)
y <- sample(1:4, 10, replace = TRUE)
points(x, y, cex = 5, col = "#0000FF7D", pch = 19)
axis(1, at = 0:5, 0:5, col = par()$col.axis)
axis(2, at = 0:5, 0:5, col = par()$col.axis, las = 1)
box("outer", col = "white")
```



10 Exercices

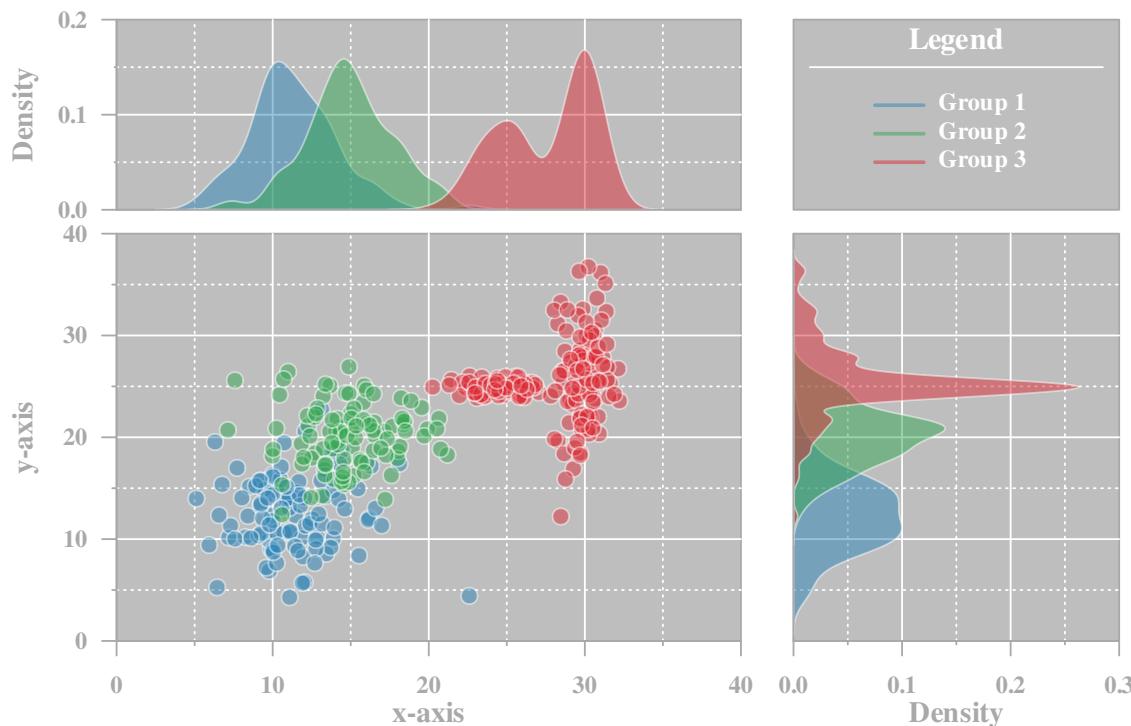
Nous allons maintenant mettre en pratique tout ce qui a été vu au cours de cet enseignement au travers de trois exemples pratiques. Dans le premier, nous allons avoir recours à la fonction `layout()` afin de créer une figure composée de trois graphiques ayant un axe en commun. Dans le second, nous allons voir comment

composer un graphique dans lequel seront représentées deux séries de données ayant en commun un axe mais dont le second axe ne présente pas les mêmes dimensions. Enfin, nous verrons comment insérer un graphe en médaillon dans un autre. Les lignes de commandes permettant de réaliser chaque figure seront disponibles dans le chapitre 7. Essayez de reproduire ces figures sans avoir recours au code, sauf si vous bloquez bien évidemment.

10.1 Partitionnement avancé

Importez-les données dans R (attention à vous placer dans le répertoire contenant les données téléchargées). Remarque : ces données n'ont aucune signification particulière.

```
## Importation des données
load(file = "extdata/datademo1.Rdata")
head(dat)
#>      x      y      z
#> 1  9.781518 6.872631 Group1
#> 2  7.184751 10.158595 Group1
#> 3  8.259131 10.193993 Group1
#> 4 13.876945 10.059613 Group1
#> 5  9.997379 16.339577 Group1
#> 6 10.249272 14.765262 Group1
```

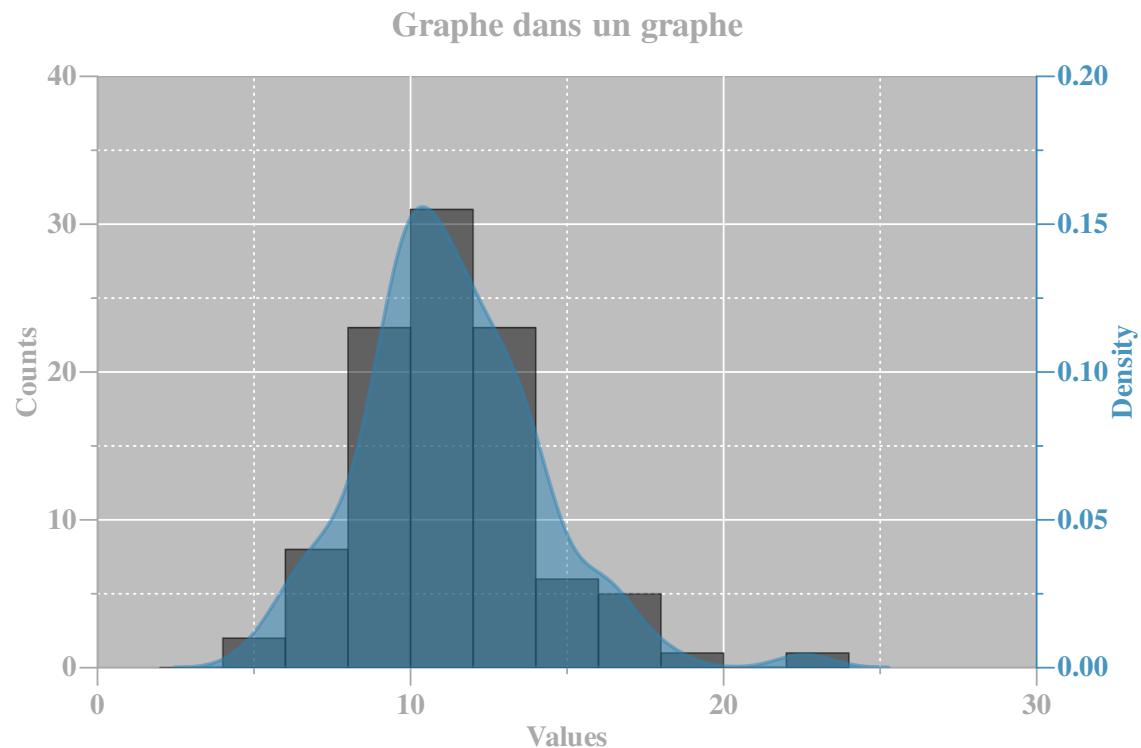


Bien, maintenant vous avez tout en main pour commencer. Amusez-vous bien !

10.2 Superposition de graphes

Passons au second exercice. La figure ?? est le résultat auquel vous devriez arriver.

```
## Importation des données
load(file = "extdata/datadem2.Rdata")
head(dat)
#> [1] 9.781518 7.184751 8.259131 13.876945 9.997379 10.249272
```

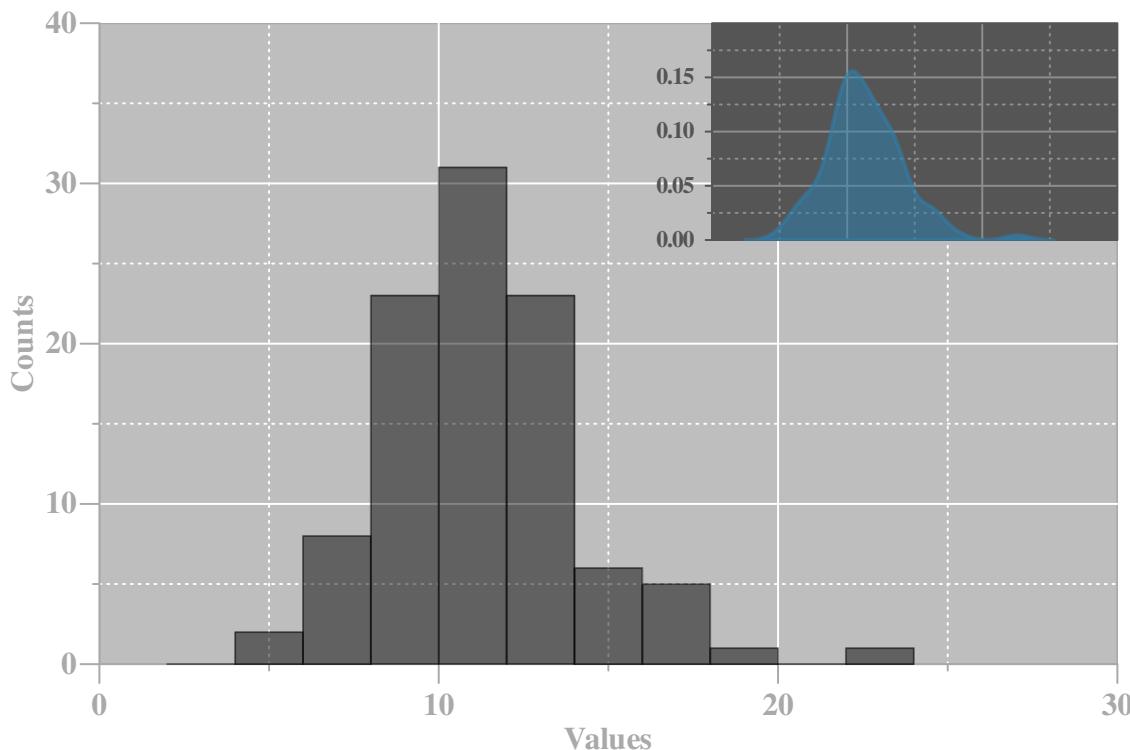


10.3 Inclusion en médaillon

Ce dernier exercice va vous amener à créer une figure dans laquelle un graphe sera inclus en médaillon (c.-à-d. en plus petit et disposé dans un coin) dans un autre (Figure ??). Importez les données

```
## Importation des données
load(file = "extdata/datadem3.Rdata")
head(dat)
#> [1] 9.781518 7.184751 8.259131 13.876945 9.997379 10.249272
```

Graphe dans un graphe



11 Solutions des exercices

Voici une solution pour réaliser les figures du précédent chapitre.

11.1 Partitionnement avancé

```
###  
### CONFIGURATION DU PERIPHERIQUE  
###  
## Partitionnement de la fenetre graphique  
mat <- matrix(c(2, 4, 1, 3), byrow = TRUE, ncol = 2)  
layout(mat, widths = c(6, 3), heights = c(3, 6))  
###  
### CONCEPTION DU SCATTERPLOT  
##  
## Empty plot  
par(mar = c(3, 3, 0, 0), family = "serif", col.axis = "darkgray")  
plot(0, type = "n", xlim = c(0, 40), ylim = c(0, 40), axes = FALSE, ann = FALSE)  
## Background  
rect(0, 0, 40, 40, col = "gray", border = par()$col.axis)  
for (i in c(10, 20, 30)){  
  points(x = c(0, 40), y = c(i, i), col = "white", type = "l")  
  points(x = c(i, i), y = c(0, 40), col = "white", type = "l")  
}
```

```

for (i in c(5, 15, 25, 35)){
  points(c(0, 40), c(i, i), col = "white", type = "l", lty = 3)
  points(c(i, i), c(0, 40), col = "white", type = "l", lty = 3)
}
## Axes principaux
axis(1, pos = 0, at = seq(0, 40, 10), labels = seq(0, 40, 10), font = 2, col = par()$col.axis)
axis(2, pos = 0, at = seq(0, 40, 10), labels = seq(0, 40, 10), font = 2, col = par()$col.axis, las = 2)
## Axes secondaires
axis(side = 1, pos = 0, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = par()$col.axis)
axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = par()$col.axis, las = 0)
## Noms des axes
mtext("x-axis", side = 1, line = 1.50, font = 2, col = par()$col.axis)
mtext("y-axis", side = 2, line = 1.75, font = 2, col = par()$col.axis, las = 0)
## Ajout des points
subtab <- dat[dat[, "z"] == "Group1", ]
points(x = subtab[, "x"], y = subtab[, "y"], pch = 21, col = "#FFFFFF7D", bg = "#2B84B67D", cex = 1.5)
subtab <- dat[dat[, "z"] == "Group2", ]
points(x = subtab[, "x"], y = subtab[, "y"], pch = 21, col = "#FFFFFF7D", bg = "#32A74F7D", cex = 1.5)
subtab <- dat[dat[, "z"] == "Group3", ]
points(x = subtab[, "x"], y = subtab[, "y"], pch = 21, col = "#FFFFFF7D", bg = "#DC29337D", cex = 1.5)
#####
#### CONCEPTION DU GRAPHE DU HAUT
#####
## Empty plot
par(mar = c(0, 3, 2, 0), family = "serif", col.axis = "darkgray")
plot(c(0, 40), c(0, .2), type = "n", axes = FALSE, ann = FALSE)
## Background
rect(0, 0, 40, .2, col = "gray", border = par()$col.axis)
points(x = c(0, 40), y = c(.1, .1), col = "white", type = "l")
for (i in c(10, 20, 30))
  points(x = c(i, i), y = c(0, .2), type = "l", col = "white")
  points(c(0, 40), c(.05, .05), type = "l", col = "white", lty = 3)
  points(c(0, 40), c(.15, .15), type = "l", col = "white", lty = 3)
for (i in c(5, 15, 25, 35)){
  points(c(i, i), c(0, .2), type = "l", col = "white", lty = 3)
}
## Axe principal
axis(2, pos = 0, at = seq(0, .2, .1), col = par()$col.axis, las = 2, labels = format(seq(0, .2, .1)), font = 2)
## Axe secondaire
axis(side = 2, pos = 0, at = seq(.05, .15, .1), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = par()$col.axis)
## Nom de l'axe y
mtext("Density", side = 2, line = 1.75, font = 2, col = par()$col.axis, las = 0)
## Density functions
dens <- density(dat[dat[, "z"] == "Group1", "x"])
polygon(x = dens$x, y = dens$y, col = "#2B84B67D", border = "#FFFFFF7D")
dens <- density(dat[dat[, "z"] == "Group2", "x"])
polygon(x = dens$x, y = dens$y, col = "#32A74F7D", border = "#FFFFFF7D")
dens <- density(dat[dat[, "z"] == "Group3", "x"])
polygon(x = dens$x, y = dens$y, col = "#DC29337D", border = "#FFFFFF7D")
## Correction
lines(x = c(0, 40), y = c(0, 0), col = par()$col.axis)
#####
#### CONCEPTION DU GRAPHE DE DROITE

```

```

####

## Empty plot
par(mar = c(3, .5, 0, .5), family = "serif", col.axis = "darkgray")
plot(c(0, .3), c(0, 40), type = "n", axes = FALSE, ann = FALSE)
## Background
rect(0, 0, .3, 40, col = "gray", border = par()$col.axis)
points(x = c(.1, .1), y = c(0, 40), col = "white", type = "l")
points(x = c(.2, .2), y = c(0, 40), col = "white", type = "l")
for (i in c(10, 20, 30))
points(x = c(0, .3), y = c(i, i), type = "l", col = "white")
for (i in seq(.05, .25, by = .1))
points(y = c(0, 40), x = c(i, i), type = "l", col = "white", lty = 3)
for (i in seq(5, 35, by = 10)){
points(c(0, .3), c(i, i), type = "l", col = "white", lty = 3)
}

## Axe principal
axis(1, pos = 0, at = seq(0, .3, .1), col = par()$col.axis, las = 1, labels = format(seq(0, .3, .1)), f
## Axe secondaire
axis(side = 1, pos = 0, at = seq(.05, .25, .1), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, co
## Nom de l'axe
mtext("Density", side = 1, line = 1.5, font = 2, col = par()$col.axis)
## Density functions
dens <- density(dat[dat[, "z"] == "Group1", "y"])
polygon(x = dens$y, y = dens$x, col = "#2B84B67D", border = "#FFFFFF7D")
dens <- density(dat[dat[, "z"] == "Group2", "y"])
polygon(x = dens$y, y = dens$x, col = "#32A74F7D", border = "#FFFFFF7D")
dens <- density(dat[dat[, "z"] == "Group3", "y"])
polygon(x = dens$y, y = dens$x, col = "#DC29337D", border = "#FFFFFF7D")
## Correction
lines(x = c(0, 0), y = c(0, 40), col = par()$col.axis)

####

#### LEGENDE
####

## Empty plot
par(mar = c(0, 0.5, 2, 0.5), family = "serif")
plot(0, type = "n", ylim = c(0, 4), xlim = c(0, 4), axes = FALSE, ann = FALSE)
## Background
rect(0, 0, 4, 4, col = "gray", border = par()$col.axis)
## Titre de la legende
text(2, 3.5, labels = "Legend", col = "white", font = 2, cex = 1.25)
lines(x = c(0.2, 3.8), y = c(3, 3), col = "white")
## Texte de la legende
text(1.6, 2.2, labels = "Group 1", pos = 4, col = "white", font = 2)
text(1.6, 1.6, labels = "Group 2", pos = 4, col = "white", font = 2)
text(1.6, 1.0, labels = "Group 3", pos = 4, col = "white", font = 2)
## Ajout des symboles
lines(x = c(1, 1.6), y = c(2.2, 2.2), col = "#2B84B67D", lwd = 2)
lines(x = c(1, 1.6), y = c(1.6, 1.6), col = "#32A74F7D", lwd = 2)
lines(x = c(1, 1.6), y = c(1.0, 1.0), col = "#DC29337D", lwd = 2)

```

11.2 Superposition de graphes

```
### CONCEPTION DE L'HISTOGRAMME
## Empty plot
par(mar = c(2.5, 2.5, 3, 3), family = "serif", xaxs = "i", yaxs = "i", col.axis = "darkgray", mgp = c(0, 1, 1))
plot(c(0, 30), c(0, 40), type = "n", axes = FALSE, ann = FALSE)
## Background
rect(0, 0, 30, 40, col = "gray", border = par()$col.axis)
for (i in c(10, 20)){
  lines(x = c(0, 30), y = c(i, i), col = "white")
  lines(x = c(i, i), y = c(0, 40), col = "white")
}
lines(x = c(0, 30), y = c(30, 30), col = "white")
for (i in c(5, 15, 25)){
  lines(x = c(0, 30), y = c(i, i), col = "white", lty = 3)
  lines(x = c(i, i), y = c(0, 40), col = "white", lty = 3)
}
points(x = c(0, 30), y = c(35, 35), col = "white", type = "l", lty = 3)
## Axes principaux
axis(1, pos = 0, at = seq(0, 30, 10), labels = seq(0, 30, 10), font = 2, col = par()$col.axis)
axis(2, pos = 0, at = seq(0, 40, 10), labels = seq(0, 40, 10), font = 2, col = par()$col.axis, las = 2)
## Axes secondaires
axis(side = 1, pos = 0, at = seq(5, 25, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = "black")
axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = "black")
## Noms des axes
mtext("Values", 1, line = 1.50, font = 2, col = par()$col.axis)
mtext("Counts", 2, line = 1.50, font = 2, col = par()$col.axis, las = 0)
## Ajout de l'histogramme
x <- seq(2, 24, by = 2)
hist(dat, add = TRUE, border = "#0000007D", col = "#0000007D", breaks = x)
### CONCEPTION DE LA FONCTION DE DENSITE
## Empty plot
par(mar = c(2.5, 2.5, 3, 3), family = "serif", col.axis = "#2B84B6DD", new = TRUE)
plot(c(0, 30), c(0, .2), type = "n", axes = FALSE, ann = FALSE)
## Axes principaux
axis(side = 4, pos = 30, at = seq(0, .2, .05), labels = format(seq(0, .2, .05)), col = par()$col.axis, las = 0)
axis(4, pos = 30, at = seq(0.025, .175, .05), labels = FALSE, tck = -0.01, lwd = 0, lwd.ticks = 1, col = "black")
## Noms des axes
mtext("Density", 4, line = 2, font = 2, col = par()$col.axis, las = 0)
## Ajout de la courbe de densite
den <- density(dat)
polygon(den$x, den$y, col = "#2B84B67D", border = "#2B84B67D", lwd = 2)
## Rajout d'un titre
title("Graphe dans un graphe", col.main = "darkgray")
```

11.3 Inclusion en médaillon

```
### CONCEPTION DE L'HISTOGRAMME
## Empty plot
par(mar = c(2.5, 2.5, 3, 3), family = "serif", xaxs = "i", yaxs = "i", col.axis = "darkgray", mgp = c(0, 1, 1))
plot(c(0, 30), c(0, 40), type = "n", axes = FALSE, ann = FALSE)
```

```

## Background
rect(0, 0, 30, 40, col = "gray", border = par()$col.axis)
for (i in c(10, 20)){
  lines(x = c(0, 30), y = c(i, i), col = "white")
  lines(x = c(i, i), y = c(0, 40), col = "white")
}
lines(x = c(0, 30), y = c(30, 30), col = "white")
for (i in c(5, 15, 25)){
  lines(x = c(0, 30), y = c(i, i), col = "white", lty = 3)
  lines(x = c(i, i), y = c(0, 40), col = "white", lty = 3)
}
points(x = c(0, 30), y = c(35, 35), col = "white", type = "l", lty = 3)
## Axes principaux
axis(1, pos = 0, at = seq(0, 30, 10), labels = seq(0, 30, 10), font = 2, col = par()$col.axis)
axis(2, pos = 0, at = seq(0, 40, 10), labels = seq(0, 40, 10), font = 2, col = par()$col.axis, las = 2)
## Axes secondaires
axis(side = 1, pos = 0, at = seq(5, 25, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = par()$col.axis)
axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = FALSE, lwd = 0, tck = -0.01, lwd.ticks = 1, col = par()$col.axis)
## Noms des axes
mtext("Values", 1, line = 1.50, font = 2, col = par()$col.axis)
mtext("Counts", 2, line = 1.50, font = 2, col = par()$col.axis, las = 0)
## Ajout de l'histogramme
x <- seq(2, 24, by = 2)
hist(dat, add = TRUE, border = "#0000007D", col = "#0000007D", breaks = x)
## Rajout d'un titre
title("Graphe dans un graphe", col.main = "darkgray")
### CONCEPTION DU MEDAILLON
## Empty plot
par(mar = c(2.5, 2.5, 3, 3), family = "serif", cex.axis = .75, new = TRUE, col.axis = "#555555", fig = 1)
plot(c(0, 30), c(0, .2), type = "n", axes = FALSE, ann = FALSE)
## Background
rect(0, 0, 30, .2, col = par()$col.axis, border = par()$col.axis)
for (i in seq(.05, .15, .05)){
  lines(x = c(0, 30), y = c(i, i), col = "#8E8E8E")
}
for (i in seq(10, 20, 10)){
  lines(x = c(i, i), y = c(0, .2), col = "#8E8E8E")
}
for (i in c(5, 15, 25)){
  lines(x = c(i, i), y = c(0, 40), col = "#8E8E8E", lty = 3)
}
for (i in seq(0.025, .175, .05)){
  lines(x = c(0, 30), y = c(i, i), col = "#8E8E8E", lty = 3)
}
box(col = par()$col.axis)
## Axe principal
axis(side = 2, pos = 0, at = seq(0, .15, .05), labels = format(seq(0, .15, .05)), col = par()$col.axis, col.ticks = "#555555")
## Axe secondaire
axis(2, pos = 0, at = seq(0.025, .175, .05), labels = FALSE, tck = -0.02, lwd = 0, lwd.ticks = 1, col.ticks = "#555555")
## Ajout de la courbe de densite
den <- density(dat)
polygon(den$x, den$y, col = "#2B84B67D", border = "#2B84B67D", lwd = 2)

```

Part III

Utiliser ggplot2

12 Introduction au package grid

Expliquer grid.

12.1 Principes

12.2 Examples

12.3 Application

12.3.1 Lattice

12.3.2 ggplot2

13 Introduction au package ggplot2

Expliquer ggplot2.

14 ggplot2 in action

14.1 Principes

14.2 Reprendre ce qui a été fait dans la partie 1

15 Les extensions de ggplot2

Voir <https://www.ggplot2-exts.org/>

16 Graphiques interactives

Dans cette partie nous aborderons les graphiques interactives via des bibliothèques web.

Annexe 1 - courte introduction à R

16.0.1 Les grands principes de R

Le dernier livre de John Chambers (Chambers, 2016) insistent beaucoup sur les trois principes fondamentaux de R pour aller plus loin dans la compréhension des mécanismes d'extension de R:

1. **objet:** tout ce qui existe dans R est un objet;

2. **function:** tout ce qui se passe dans R est un appelle à une fonction;
3. **interface:** les interfaces avec les autres software sont une partie de R.

Le langage R est performant pour manipuler, analyser et visualiser des données. Utiliser R, c'est utiliser une syntaxe précise pour écrire des requêtes. Ces requêtes induisent, au niveau des unités de calculs de l'ordinateur, la série d'opérations nécessaire pour réaliser les tâches associées aux requêtes soumises. Pour mener une analyse de données avec R, il faut être capable de formuler correctement les requêtes et donc maîtriser la syntaxe de ce langage. Aussi, pour améliorer ces compétences il faut enrichir son vocabulaire (en augmentant ses connaissances des fonctions et packages disponibles) et creuser la question du fonctionnement de R (et se confronter à des questions plus techniques).

L'objectif de cette annexe est de familiariser le lecteur néophyte avec la syntaxe de R en utilisant des exemples de lignes de commande que nous commentons. Nous discutons brièvement de quelques principes fondamentaux du langage et nous couvrons un vocabulaire très limité (nous mentionnons un nombre restreint de fonctions et d'opérateurs). Cette annexe a été conçue pour rendre l'ouvrage autonome. Cependant, cette annexe ne peut constituer à elle seule une introduction au langage R. C'est pourquoi, nous vous recommandons vivement de consulter les trois premiers chapitres de *R et espace* (REF+ouvrage disponible sur le site de Framabook <https://framabook.org/r-et-espace/>), et le manuel *R pour les débutants* d'Emmanuel Paradis (REF+disponible sur le CRAN https://cran.r-project.org/doc/contrib/Paradis-rdebut斯_fr.pdf).

Première commande

```
var1 <- 1 + 1
```

Avant de lister les opérations les plus courantes, arrêtons-nous sur cette première ligne de commande pour mieux la comprendre. La première ligne est une assignation, nous demandons à R de réaliser l'opération $1+1$ et de stocker le résultat dans une variable, dont le nom est `var1`. Lorsque la commande est soumise (ce qui correspond à utiliser la touche entrée dans une console R), un certain nombre d'opérations ont été réalisées par l'ordinateur, afin que les `1` soient correctement interprétés comme des nombres devant être additionnés (opérateur `+` qui va engendrer une addition au niveau des unités de calculs) et finalement le résultat de cette opération va être associé à la variable `var1` (opérateur `<-`)⁵⁴. Un espace dans la mémoire vive de l'ordinateur a été alloué pour y inclure la variable (incluant son nom et sa valeur) et la variable est disponible dans la session de R.

Il est important de noter dès maintenant que les noms donnés aux variables sont soumis à deux contraintes. Premièrement, un nom de variable doit systématiquement commencer par une lettre latine (minuscule ou majuscule) ou un point (“.”). Deuxièmement, le reste du nom peut-être formé par des lettres latines (minuscules ou majuscules), des chiffres, des points (“.”) et des tirets-bas (“_”). Pour ne pas se perdre dans les noms de vos variables, il est préférable de rester concis et aussi explicite que possible. Notez que si un nom de variable est utilisé dans la console alors que la variable n'a pas été définie, un message d'erreur est retourné.

```
var0
#> Error in eval(expr, envir, enclos): object 'var0' not found
```

Si nous rappelons maintenant l'objet `var1`, il est correctement afficher :

```
var1
#> [1] 2
```

⁵⁴R propose trois opérateurs d'assignation: `<-`, `=` (`var1 = 1 + 1`) et `->` (`1 + 1 -> var1`) nous utilisons toujours le premier dans cette ouvrage.

Notons au passage que cette ligne est en réalité équivalent à `print(var1)`, `print()` étant une fonction (voir le paraphe dédié REF) qui permet d'afficher les objets de R. Parfois, l'utilisateur peut désirer créer une variable et l'afficher immédiatement (ce que nous faisons à plusieurs reprises dans l'ouvrage). Pour ce faire, la ligne de code peut être passer `print()` :

```
print(var2 <- 35+48)
#> [1] 83
```

Pour une économie de cinq lettres, l'utilisateur peut simplement utiliser les parenthèses:

```
(var2 <- 35+48)
#> [1] 83
```

Pour voir l'ensemble des variables utilisé dans la session, utiliser la commande `ls()` :

```
ls()
#> [1] "var1" "var2"
```

Jusqu'ici, nous avons utilisé deux variables: `var1` et `var2` qui sont donc affichées.

Pour R, tout est objet

La portion structurée de mémoire vers lequel renvoie un nom de variable est un objet. Pour R, tout est objet, depuis les nombres jusqu'aux fonctions. Pour faire simple, nous pouvons diviser les objets en deux grandes catégories. D'un côté, nous avons les objets de base (p. ex. les vecteurs), de l'autre côté, il y a les objets composés (de type S3 ou S4). Pour se donner une image, les premiers sont les briques élémentaires de R avec lesquelles les seconds (S3 ou S4) sont bâties. Comme nous l'avons énoncé plus haut, les objets S4 étant les plus formels que les S3. Cette distinction technique permet de bien comprendre comment R fonctionne et de mieux comprendre certains messages d'erreur. Nous assignons maintenant différents objets de base à différentes variables tout en excluant les fonctions dont nous parlons plus bas :

```
var3 <- "cool"
var4 <- TRUE
var5 <- NA
var6 <- 1L
var7 <- var1
```

votre session de R, la variable

R a été capable de comprendre et d'indexer en mémoire ces différents objets. Ce dernier point enlève une épine du pied de l'utilisateur de R⁵⁵. Nous pouvons savoir comment R a interprété ces objets avec la fonction `typeof()`⁵⁶.

```
typeof(var3)
typeof(var4)
typeof(var5)
typeof(var6)
typeof(var7)
#> [1] "character"
#> [1] "logical"
```

⁵⁵C'est un moment difficile à passer quand on utilise des langages de plus bas niveau comme le langage C, sur lequel R repose.

⁵⁶Il est possible d'utiliser la fonction `mode()` qui est simplement un alias de cette fonction.

```
#> [1] "logical"
#> [1] "integer"
#> [1] "double"
```

Ces objets sont en fait des vecteurs de différents types et de taille 1. Un vecteur est constitué d'un ou plusieurs éléments de même nature. Il sont construits avec la fonction `c()`. La taille des vecteurs peut être retournée avec `length()`. Noter que dans la suite, on ajoute des parenthèses à une ligne donnée pour pouvoir afficher le résultat de l'assignation :

```
(var7 <- c(2,4,9))
typeof(var7)
length(var7)
(var8 <- c("aa","ab","ac","ad"))
typeof(var8)
length(var8)
#> [1] 2 4 9
#> [1] "double"
#> [1] 3
#> [1] "aa" "ab" "ac" "ad"
#> [1] "character"
#> [1] 4
```

Un ensemble d'objet de même nature peut être contenu dans un tableau doublement indexé, c'est ce qu'on appelle une matrice. Les matrices peuvent être créées à l'aide de la fonction `matrix()`. Les tableaux peuvent même être indexés dans un nombre quelconque de dimensions grâce aux objets “array” créés avec la fonction `array()`. Pour appeler une des valeurs de ces tableaux, on utilise les crochets, comme nous le montrons ci-dessous :

```
var9 <- matrix(data=c(3,4,5,6,8,9), ncol=2, nrow=3)
var9[1,2]
var9[1,]
var10 <- array(2,c(2,2,3))
var10[1,2,2]
#> [1] 6
#> [1] 3 6
#> [1] 2
```

Un dernier objet permet de rassembler des objets de nature différentes, il s'agit des listes, pour accéder à l'un de ces objets, on utilise le double crochet et si on souhaite accéder à un objet en particulier dans la liste, on ajoute un crochet.

```
(var11 <- list(var7,var2,var8))
typeof(var11)
var11[[2]]
var11[[3]][2]
#> [[1]]
#> [1] 2 4 9
#>
#> [[2]]
#> [1] 83
#>
#> [[3]]
```

```
#> [1] "aa" "ab" "ac" "ad"
#>
#> [1] "list"
#> [1] 83
#> [1] "ab"
```

Nous avons vu les objets de bases les plus importants. Parmi les objets les plus utilisés de R, il y a les “dataframe”. Il s’agit d’objets de type S3, construits avec l’utilisation de la fonction `data.frame()`.

```
(var12 <- data.frame(var7, Format=c("A1", "A2",
  "A3")))
#> var7 Format
#> 1   2     A1
#> 2   4     A2
#> 3   9     A3
```

Ces objets sont finalement des tableaux dont les colonnes sont des vecteurs. C’est un format très pratique pour travailler sur les données. L’utilisation du `$` vous permet d’accéder à l’un de ces vecteurs, en utilisant le nom de la colonne :

```
var12$var7
#> [1] 2 4 9
```

Vous pouvez également utiliser les crochets pour accéder aux différents éléments, comme dans une matrice. Un dernier détail pur savoir s’il s’agit d’un objet S3 ou S4 on utilise la fonction `is.object()`, si elle retourne “TRUE”, alors il s’agit d’un objet S3 ou S4. Nous l’utilisons sur notre “matrix” et notre “data.frame”.

```
is.object(var9)
is.object(var12)
#> [1] FALSE
#> [1] TRUE
```

Pour ces objets composites, une notion importante est la classe de l’objet c’est-à-dire l’espèce à laquelle cet objet appartient. Pour le savoir, il suffit d’utiliser la fonction `class()`.

```
class(var12)
#> [1] "data.frame"
```

C’est grâce à la classe que R reconnaît les objets composés. En fait, il est très facile de créer un objet S3 personnalisé dans R grâce à la fonction `structure()` :

```
(
  var13 <- structure(
    list(
      donnees=var12 , auteur="KCNC"
    ),
    class="monobjet")
)
#> $donnees
#> var7 Format
#> 1   2     A1
```

```
#> 2     4      A2
#> 3     9      A3
#>
#> $auteur
#> [1] "KCNC"
#>
#> attr(,"class")
#> [1] "monobjet"
```

Conditions

Lorsqu'on utilise un langage de programmation, on a souvent besoin de comparer des valeurs. Pour cela, on utilise des tests logiques. Il s'agit de comparaisons entre deux objets dont le résultat est soit “TRUE” (vrai) soit “FALSE” (faux).

```
var1==85
var1<50
var1>50
var1!=45
```

Avec R, les tests sont élargis à différents type d'objet. Par exemple, les comparaisons peuvent aussi se faire sur les lettres, c'est alors une question d'ordre alphabétique :

```
"a">>"b"
#> [1] FALSE
"bestiaire"<"bestiole"
#> [1] TRUE
```

Pour créer des tests logiques un peu plus élaborés et utilisant, on utilise **&** (le “et” logique) et **|** (le “ou” logique).

```
var1>50 & var1%%5==0
var1>50 | var1%%3
```

Avec ces tests logiques, on peut construire des conditions, c'est-à-dire des structures logiques qui permettent de réaliser une opération différente selon le résultat des test.

```
if (var1 %% 9 == 0) {
  print("var1 est un multiple de 9.")
} else if (var1 %% 3 == 0) {
  print("var1 est un multiple de 3.")
} else {
  print("var1 n'est pas multiple de 9 et 3.")
}
#> [1] "var1 n'est pas multiple de 9 et 3."
```

Selon la valeur de “var1”, l'un des trois messages est affiché. Si c'est la première ou la seconde option qui est choisie, les suivantes ne seront pas explorées. La dernière condition (**else** seul) signifie que si les options précédentes n'ont pas été retenues, on utilise celle-ci. La fonction **print()** permet d'afficher un objet, ici ce sont des chaînes de caractères que l'on affiche dans la console.

Boucles

Lorsqu'on fait des analyses, il se peut qu'on répète plusieurs fois les mêmes opérations mais pour des données différentes, par exemple pour les différentes colonnes d'un tableau. Dans ce cas, nous utilisons des boucles. Il existe les boucles **for** et les boucles **while**. Les boucles **for** consiste en une itération de commande sur un ensemble de valeurs pré-déterminée.

```
for (i in 1:3) print(i)
#> [1] 1
#> [1] 2
#> [1] 3
```

L'opérateur : est très utile, **a:b** crée un ensemble de valeur de 1 en 1 depuis “a” jusqu’à “b”. La variable “i” va prendre les valeurs 1,2 et 3 que l'on affiche avec la fonction **print()**. Il est possible de répéter un ensemble d'opérations qui doivent alors être placées entre accolades. Comme nous l'avons mentionné plus tôt, R dispose d'une structure très flexible de boucle **for** : une itération sur les valeurs d'un vecteur donnée.

```
vec <- c(2,8,10)
for (j in vec) {
  s <- j^2
  print(s)
}
#> [1] 4
#> [1] 64
#> [1] 100
```

Au contraire des boucles **for**, les boucles **while** répètent des instructions pour un ensemble de valeur qui n'est pas pré-déterminé. Les instructions sont répétées indéfiniment (donc méfiance avec ces boucles) tant qu'une condition n'est pas respectée.

```
d <- 3
i <- 0
while (i<d) {
  s <- i^2
  print(s)
  i <- i+1
}
#> [1] 0
#> [1] 1
#> [1] 4
```

La manipulation et visualisation des données nécessite rarement l'utilisation des boucles **while**, les boucles **for** sont quant à elles fréquemment utilisées.

Les fonctions

De manière générale, en informatique, une fonction est un ensemble de lignes de code dédié à la réalisation d'une tâche spécifique concrétisée par le renvoie d'une valeur (un objet dans R). Nous avons jusqu'ici déjà utiliser différentes fonctions. Le lecteur attentif a certainement remarqué que les fonctions de R présentent toutes une structure commune: elles commencent par une suite de caractères (son nom) suivie de parenthèses à l'intérieur desquelles se trouvent éventuellement un ou plusieurs arguments. Les arguments sont des objets transmis à la fonction qui seront soient ceux manipuler soit précise comment la tâche doit être effectuées. Par exemple, quand nous avons utilisé la ligne :

```
var9 <- matrix(
  data=c(3,4,5,6,8,9),
  ncol=2,
  nrow=3
)
```

l'argument “data” sera l'ensemble des valeurs est à utiliser pour remplir la matrice, “ncol” est le nombre de colonne et “nrow”, le nombre de ligne de la matrice. De manière générale, la forme caractéristique de l'appel d'une fonction est :

```
nomfomction(arg1=obj1, arg2=obj2, arg3=obj3)
```

Une fonction retourne un objet, et cet objet peut être donnée à une variable. Dans R, c'est un objet qui est retourné que nous pouvons passer à une variable :

```
var14 <- nomfomction(arg1=obj1, arg2=obj2, arg3=obj3)
```

Parmi les arguments il existe des argument “par défaut”. Pour ces arguments, il existe un objet prédéfinie qui défini comment la tâche sera réalisée si l'utilisateur ne précise rien. Ainsi, nous n'avons pas défini l'argument “byrow” dont la valeur par défaut est “FALSE”, c'est pourquoi la matrice est remplie colonne par colonne. Pour changer ce comportement nous précisons changeons cela :

```
(  
  var15 <- matrix(  
    data=c(3,4,5,6,8,9),  
    ncol=2,  
    nrow=3,  
    byrow=TRUE)  
)  
#>      [,1] [,2]  
#> [1,]    3    4  
#> [2,]    5    6  
#> [3,]    8    9
```

Dans R, les fonctions sont aussi des objets de type particulier, il existe deux catégories de fonctions :

```
typeof(typeof)  
typeof(sum)  
#> [1] "closure"  
#> [1] "builtin"
```

Pour R, une fonction est un bel et bien un objet dont le contenu est finalement du code. Quand une fonction est appelée sans parenthèse, c'est sont codes qui est renvoyé. De plus, pour une fonction dans un package R, des indications supplémentaires sont retournées entre <>, pour savoir où cette fonction est localisée.

```
typeof  
#> function (x)  
#> .Internal(typeof(x))  
#> <bytecode: 0x55f9145e0bc8>  
#> <environment: namespace:base>
```

Cette remarque vous facilitera la compréhension des messages d'erreur : lorsque R parle d'un objet de type "closure" il parle d'une fonction. R est un langage où votre efficacité repose très fortement sur votre vocabulaire. Pour engranger rapidement un grand nombre de fonctions, la compilation de fonctions proposées par le site du CRAN est très utile. Toutes fonctions de R sont assorties d'une documentation qui vous explique, en anglais, comment l'utiliser. Pour y accéder, l'utilisateur a deux possibilités, précédé par un ? le nom de la fonction ou utiliser la fonction `help()`, cette dernière permet une recherche plus détaillée.

```
?matrix  
help(matrix)
```

Les fonctions de R sont très flexibles. Avec les arguments par défaut qui permettent de ne pas utiliser l'ensemble des arguments, il y a aussi l'argument sous forme de "..." qui permet de passer des arguments supplémentaires. Ces arguments sont utilisés comme argument de fonctions utilisée à l'intérieur d'une fonction donnée. C'est une utilisation très fréquent dans R.

En utilisant R, on est facilement amené à créer nos propres fonctions, c'est-à-dire des objets de type "closure". Dans l'exemple ci-dessous, nous créons la fonction `affine()` qui a trois arguments. Le premier n'a pas de valeur par défaut, l'utilisateur est obligé de le spécifier, pour les deux autres, nous avons ajouté des valeurs par défaut, l'utilisateur à le choix de les changer

```
affine <- function(x,a=1,b=0){  
  y <- a*x+b  
  return(y)  
}
```

Dans une fonction, il n'est pas nécessaire d'écrire tous les noms d'arguments utilisés, nous pouvons simplement indiquer les valeurs. Dans ce cas, l'ordre des valeurs doit être l'ordre des arguments.

```
affine(4,2,3)
```

L'exemple suivant montre comment profiter des arguments supplémentaires, "...". Nous créons la fonction `affinen()` qui retourne la valeur de la fonction affine à la puissance n. Nous pourrions ré-écrire l'ensemble des paramètres de la fonction affine mais nous pouvons aussi utiliser les "...", de la sorte :

```
affinen <- function(..., pow=2){  
  y <- affine(...)  
  return(y^pow)  
}  
affine(x=4,a=2,b=3,pow=3)
```

Les oubli de l'orthographe exact d'une fonction sont fréquents. Pour chercher efficacement les noms de fonctions par motifs, il existe la fonction `apropos()` :

```
apropos("plo")
```

Annexe 2 - une revue des packages existant

Voir inSileco.

17 Annexe 3 - quel package pour tel graphique ?

Voir <https://www.r-graph-gallery.com/> aussi <https://cran.r-project.org/web/views/Graphics.html> (pas trop à jour...)

References

- Becker, R. A. (1994). A brief history of S. *cahier de recherche, AT&T Bell La.*
- Becker, R. A. and Chambers, J. M. (1984). *S: An Interactive Environment for Data Analysis and Graphics.* The Wadsworth statistics/probability series. Wadsworth Advanced Book Program, Belmont, Calif.
- Becker, R. A. and Chambers, J. M. (1985). *Extending the S System.* Wadsworth Advanced Books and Software, Monterey, Calif.
- Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics.* Wadsworth & Brooks/Cole computer science series. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, Calif.
- Chambers, J. M. (1977). *Computational Methods for Data Analysis.* Wiley series in probability and mathematical statistics. Wiley, New York.
- Chambers, J. M. (1998). *Programming with Data: A Guide to the S Language.* Springer, New York.
- Chambers, J. M. (2016). *Extending R.* The R series. CRC Press, Boca Raton London New York. OCLC: 965932522.
- Chambers, J. M. and Hastie, T. J., editors (1993). *Statistical Models in S.* Chapman & Hall computer science series. Chapman & Hall, New York. OCLC: 832177818.
- ElementR, G. (2014). *R et espace: traitement de l'information géographique.* Frambook, Lyon. OCLC: 910872278.
- Fox, J. (2009). Aspects of the Social Organization and Trajectory of the R Project. *The R Journal*, 1:9.
- Ihaka, R. and Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299.
- Millot, G. (2018). *Comprendre et réaliser les tests statistiques à l'aide de R: manuel de biostatistique.* OCLC: 1023590131.
- Utrilla, P., Mazo, C., Sopena, M., Martínez-Bea, M., and Domingo, R. (2009). A palaeolithic map from 13,660 calbp: engraved stone blocks from the late magdalenian in abauntz cave (navarra, spain). *Journal of Human Evolution*, 57(2):99 – 111.