Université du Québec à Rimouski
Département de Biologie, Chimie et Géographie

L'ART GRAPHIQUE SOUS R

Nicolas Casajus
Kévin Cazelles

- Novembre 2014 -

Avant-propos

Le logiciel R est un environnement de statistiques *open-source* librement distribué sous les termes de la licence publique générale GNU. 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 *The 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 concédons 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 clareté 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,
Kévin Cazelles,
le 28 novembre 2014

Table des matières

Liste des figures

Liste des tableaux

Introduction

Bienvenu dans le monde fascinant des graphiques sous R. Tout au long de cet enseignement, vous apprendrez à maîtriser les rudiments du langage R dédié aux graphes. Ici, il ne sera pas question de `ggplot2`, la nouvelle tendance en matière de graphiques sous R qui en a séduit plus d'un. Faites une recherche sur Internet avec les mots clés *graph*, *R* et *tutorial*, et vous verrez de quoi nous parlons.

Non, ici vous allez apprendre la production de graphiques à l'ancienne, façon maison. Pourquoi un tel choix ? Tout simplement parce qu'une telle approche va vous permettre de mieux appréhender l'ensemble des options que proposent R en matière de graphique. Même si cet enseignement est loin d'être exhaustif, vous verrez qu'il dresse un portrait assez complet des possibilités offertes par ce langage de programmation.

Selon nous, il est important de commencer par le commencement, et de maîtriser les différents éléments qui composent un graphique. Ainsi, nous allons voir comment modifier les marges, les axes, la fonte de caractères, la couleur, comment ajouter des éléments à un graphique (édition d'un graphe), comment ajouter un graphe dans un autre graphe, etc. Une fois cette maitrise acquise, vous pourrez laisser libre cours à votre imagination et à votre créativité pour améliorer le rendu visuel de vos graphiques.

Le package `graphics`

Cet enseignement repose sur un seul package : le package `graphics`. Celui-ci fait partie de la bibliothèque de base de R, c.-à-d. qu'il n'a besoin d'être ni téléchargé en complément de R, ni chargé à l'ouverture d'une nouvelle session. Notons que tous les autres packages dédiés aux graphiques sous R (`ggplot2`, `lattice`, `grid`, etc.) reposent sur ce package. En fait, le package `graphics` est bien plus qu'un package, c'est un système graphique à part entière.

Les graphiques avec R sont le sujet du livre (et de la thèse de doctorat) de Paul Murrel, membre actif du `R Development Core Team` ayant participé à l'élaboration du package `graphics`. Nous vous conseillons vivement la lecture de cet ouvrage (*R graphics*, ISBN 1-58488-486-X).

Voici un aperçu des possibilités offertes par ce package.

```
> demo(graphics)
```

La commande suivante liste toutes les fonctions disponibles dans ce package. Nous passerons au travers de plusieurs d'entre elles tout au long de ce document.

```
> ls(envir = as.environment("package:graphics"))

##  [1] "abline"          "arrows"          "assocplot"
##  [4] "axis"            "Axis"            "axis.Date"
##  [7] "axis.POSIXct"    "axTicks"         "barplot"
## [10] "barplot.default" "box"             "boxplot"
## [13] "boxplot.default" "boxplot.matrix"  "bxp"
## [16] "cdplot"          "clip"            "close.screen"
## [19] "co.intervals"    "contour"         "contour.default"
## [22] "coplot"          "curve"           "dotchart"
## [25] "erase.screen"    "filled.contour"  "fourfoldplot"
## [28] "frame"           "grconvertX"      "grconvertY"
## [31] "grid"            "hist"            "hist.default"
## [34] "identify"        "image"           "image.default"
## [37] "layout"          "layout.show"     "lcm"
## [40] "legend"          "lines"           "lines.default"
## [43] "locator"         "matlines"        "matplot"
## [46] "matpoints"       "mosaicplot"      "mtext"
## [49] "pairs"           "pairs.default"   "panel.smooth"
## [52] "par"             "persp"           "pie"
## [55] "plot"            "plot.default"    "plot.design"
## [58] "plot.function"   "plot.new"        "plot.window"
## [61] "plot.xy"         "points"          "points.default"
## [64] "polygon"         "polypath"        "rasterImage"
## [67] "rect"            "rug"             "screen"
## [70] "segments"        "smoothScatter"   "spineplot"
## [73] "split.screen"    "stars"           "stem"
## [76] "strheight"       "stripchart"      "strwidth"
## [79] "sunflowerplot"   "symbols"         "text"
## [82] "text.default"    "title"           "xinch"
## [85] "xspline"         "xyinch"          "yinch"
```

Mon « hello world » plot

Mais, trêve de palabres. Réalisons notre premier graphe à l'aide de la fonction `plot()`. Vous avez déjà probablement utilisés cette fonction au moins une fois dans votre vie. Le graphique affiché après l'appel à cette fonction dépend de ce qu'on lui demande d'afficher. Ici, nous allons représenter la valeur **1** dans un diagramme de dispersion. Étant donné que la seconde variable n'est pas précisé, la valeur **1** sera représentée en fonction d'elle-même (en fonction de son indice pour être précis).

```
> plot(1)
```

Vous êtes-vous déjà demandé ce qu'il se passait exactement après l'appel à cette fonction ? Une fenêtre graphique, aussi appelée périphérique graphique (ou *device* en anglais) s'est d'abord ouverte. Puis, elle a été récupérer les valeurs par défaut des paramètres graphiques (propriétés des axes, des marges, couleurs, etc.) de R. Enfin, dans la région du plot, elle a affiché l'objet que nous voulions représenter (ici, la valeur **1**).

Nous développerons plus loin la notion de périphérique graphique. Mais, glissons rapidement un mot sur les paramètres graphiques. Leurs valeurs par défaut sont stockées dans l'objet `par()`, qui est à la fois une fonction et une liste. On accède à ces paramètres de la manière suivante :

```
> par()
```

Comme cette liste contient 72 paramètres graphiques, nous allons simplement afficher les dix premiers :

```
> par()[1:10]

## $xlog
## [1] FALSE
##
## $ylog
## [1] FALSE
##
## $adj
## [1] 0.5
##
## $ann
## [1] TRUE
##
## $ask
## [1] FALSE
##
## $bg
## [1] "transparent"
##
## $bty
## [1] "o"
##
## $cex
## [1] 1
##
## $cex.axis
## [1] 1
##
## $cex.lab
## [1] 1
```

Par exemple, l'argument `bg` spécifie la couleur du fond (par défaut, le fond est transparent). Ainsi, en récupérant ces valeurs, R a automatiquement déterminé les axes, la police, les couleurs, les marges, le type de symboles, etc. Toutes ces informations sont modifiables, et heureusement, car le rendu par défaut de R laisse un peu à désirer... Nous verrons plus loin comment modifier certains de ces paramètres.

Chapitre 1

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-existant. C'est l'objet du chapitre suivant.

1.1 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] -0.18574 -0.51555  0.63983  1.74350 -0.08721  2.46926  0.39635
##  [8]  0.56607 -0.56347  0.56846  0.98468  0.72502 -0.07076  1.58130
## [15]  0.91076 -0.73832  0.02637 -0.13624 -0.97297 -1.70856
```

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

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

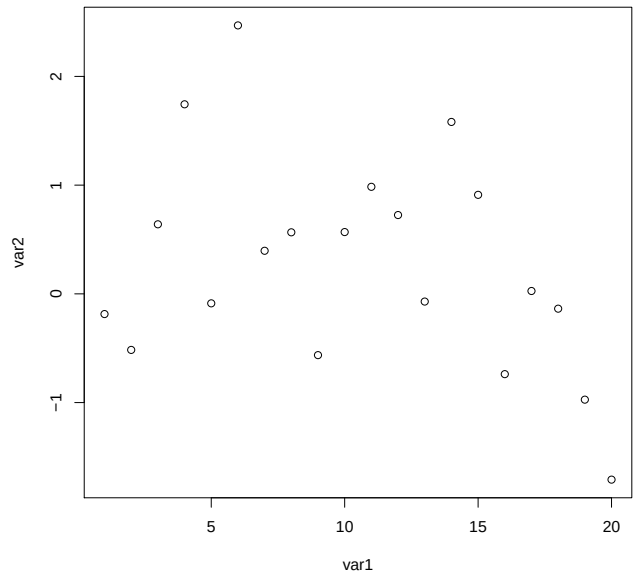

FIGURE 1.1 – Scatterplot de deux variables

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.

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

TABLE 1.1 – Catégories de scatterplot

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")
```

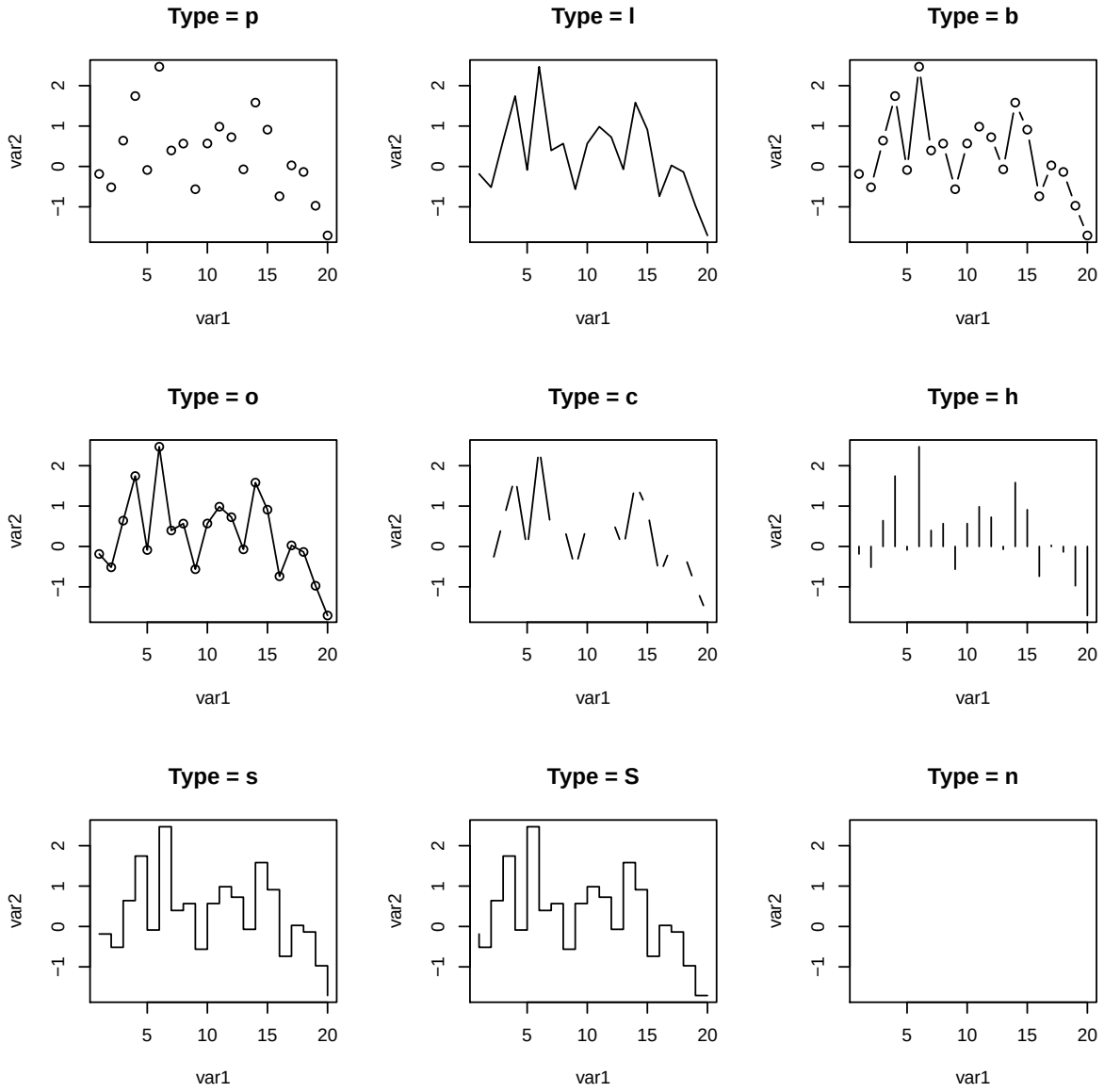

FIGURE 1.2 – Catégories de scatterplot

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.

1.2 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, 1er et 3ème quartiles, minimum et maximum) d'une variable quantitative. Sous R, la fonction utilisée sera `boxplot()`.

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

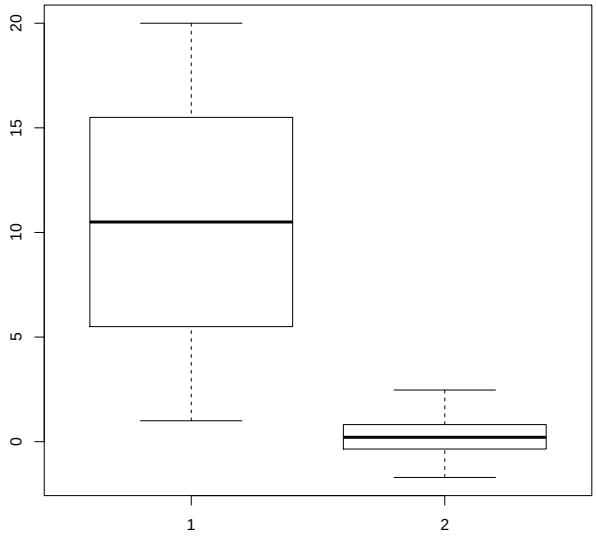

FIGURE 1.3 – Boîte à moustaches de deux variables continues

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.

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

TABLE 1.2 – Options de la fonction `boxplot()`

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,] -1.7086
## [2,] -0.3506
## [3,]  0.2114
## [4,]  0.8179
## [5,]  2.4693
##
## $n
## [1] 20
##
## $conf
##         [,1]
## [1,] -0.2015
## [2,]  0.6242
##
## $out
## numeric(0)
##
## $group
## numeric(0)
##
## $names
## [1] "1"
```

1.3 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 = T)
> echn[1:20]
```

```
##  [1] "Bleu"  "Bleu"  "Bleu"  "Vert"  "Blanc" "Blanc" "Blanc" "Rouge"
##  [9] "Jaune" "Blanc" "Bleu"  "Bleu"  "Jaune" "Vert"  "Vert"  "Vert"
## [17] "Rouge" "Vert"  "Jaune" "Blanc"
```

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

```
> (var4 <- table(echn))

## echn
## Blanc  Bleu Jaune  Noir Rouge  Vert
##   165   184   136   149   196   170
```

Visualisons cette nouvelle variable catégorique.

```
> barplot(var4)
```

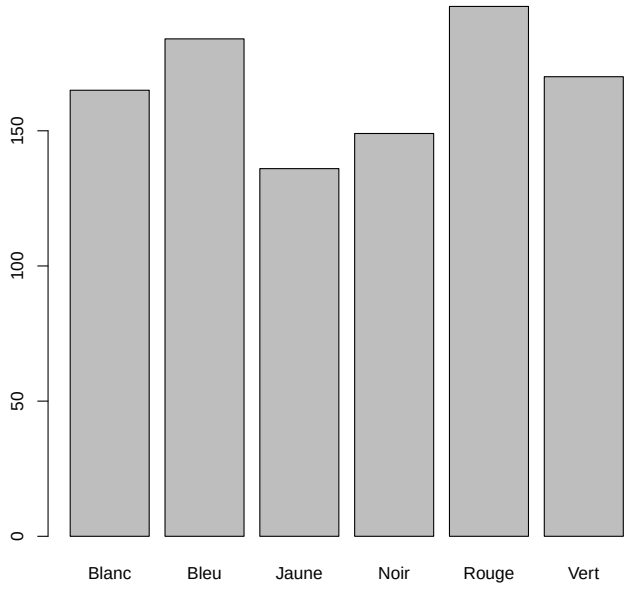

FIGURE 1.4 – Diagramme en bâtons (effectifs)

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
## 0.165 0.184 0.136 0.149 0.196 0.170
```

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))
```

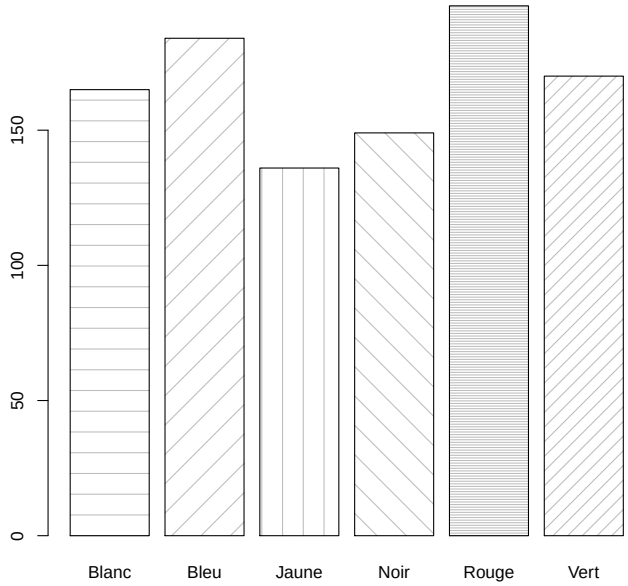

FIGURE 1.5 – Diagramme en bâtons (hachures)

1.4 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]  1.74621 -1.14138 -1.64806 -1.72683  0.40916  0.08049 -0.70761
##  [8]  1.11852 -0.25218  1.21423 -0.18305  0.51924  0.19751 -0.99848
## [15] -0.44532  2.70914  0.35190 -2.12571  0.44178  0.42279
```

```
> hist(var6)
```

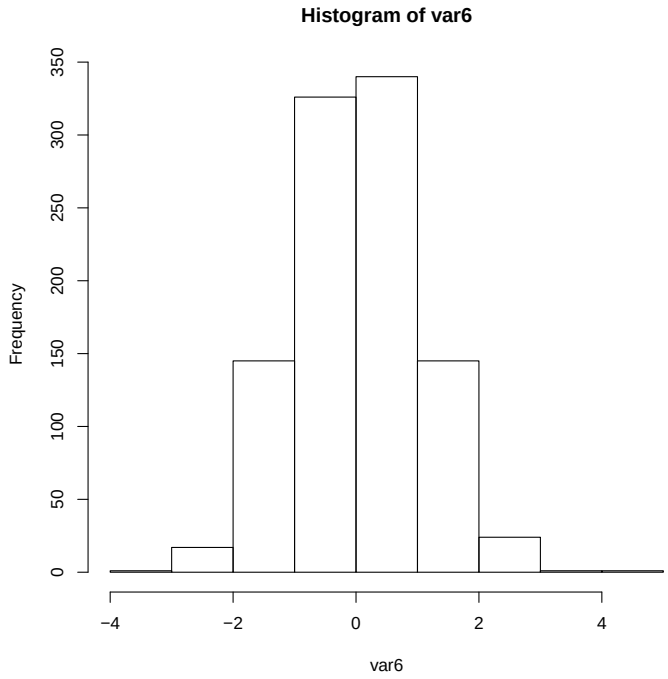

FIGURE 1.6 – Histogramme d'une varible normale

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  5
##
## $counts
## [1]   1  17 145 326 340 145  24   1   1
##
## $density
## [1] 0.001 0.017 0.145 0.326 0.340 0.145 0.024 0.001 0.001
##
## $mids
## [1] -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5
##
## $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)
```

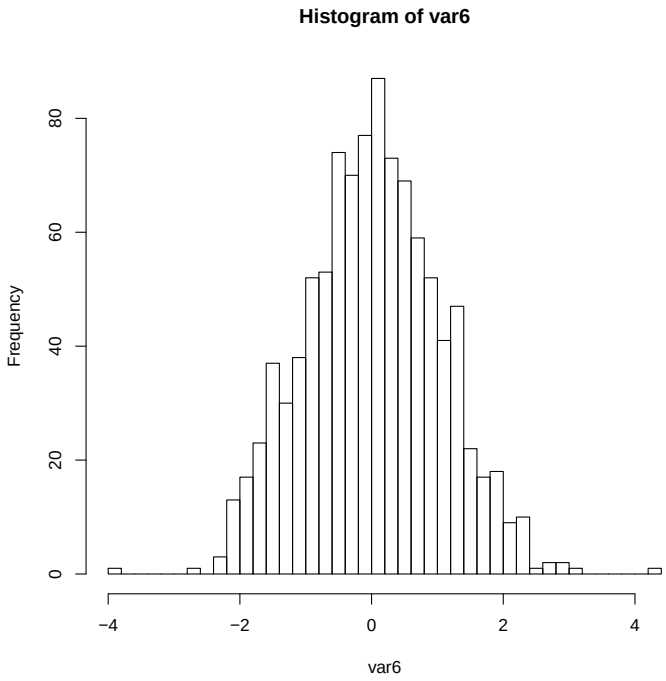

FIGURE 1.7 – Histogramme à 30 classes

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.

1.5 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"))
```

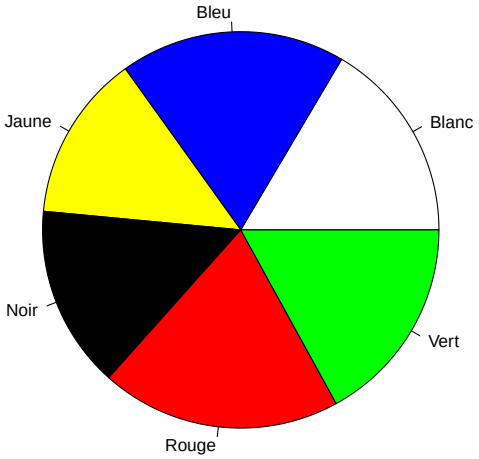

FIGURE 1.8 – Diagramme sectoriel

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

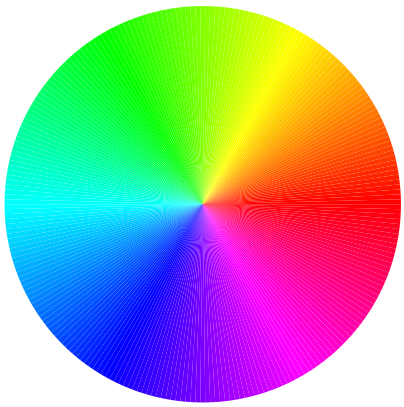

FIGURE 1.9 – Diagramme sectoriel arc-en-ciel

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 ?

1.6 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)
```

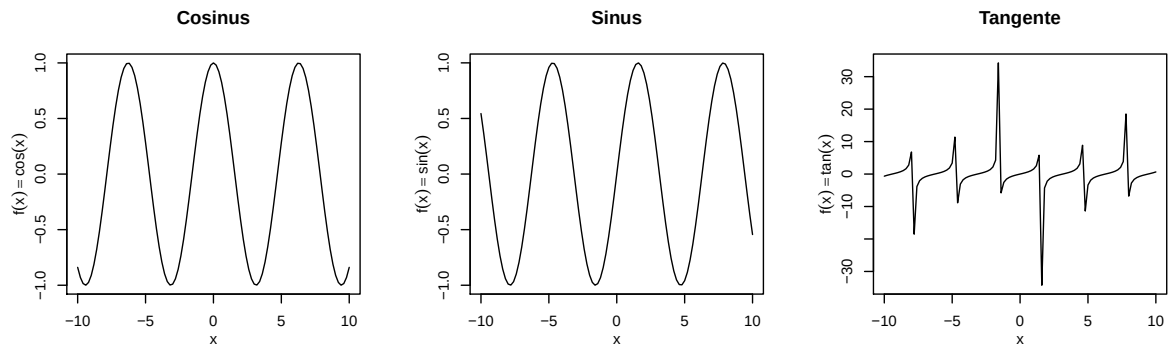

FIGURE 1.10 – Fonctions trigonométriques

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 = 1e-04, to = 10, main = "Logarithme", ylab = txt)
> txt <- expression(f(x) == exp(x))
> curve(exp(x), from = 1e-04, to = 10, main = "Exponentielle", ylab = txt)
```

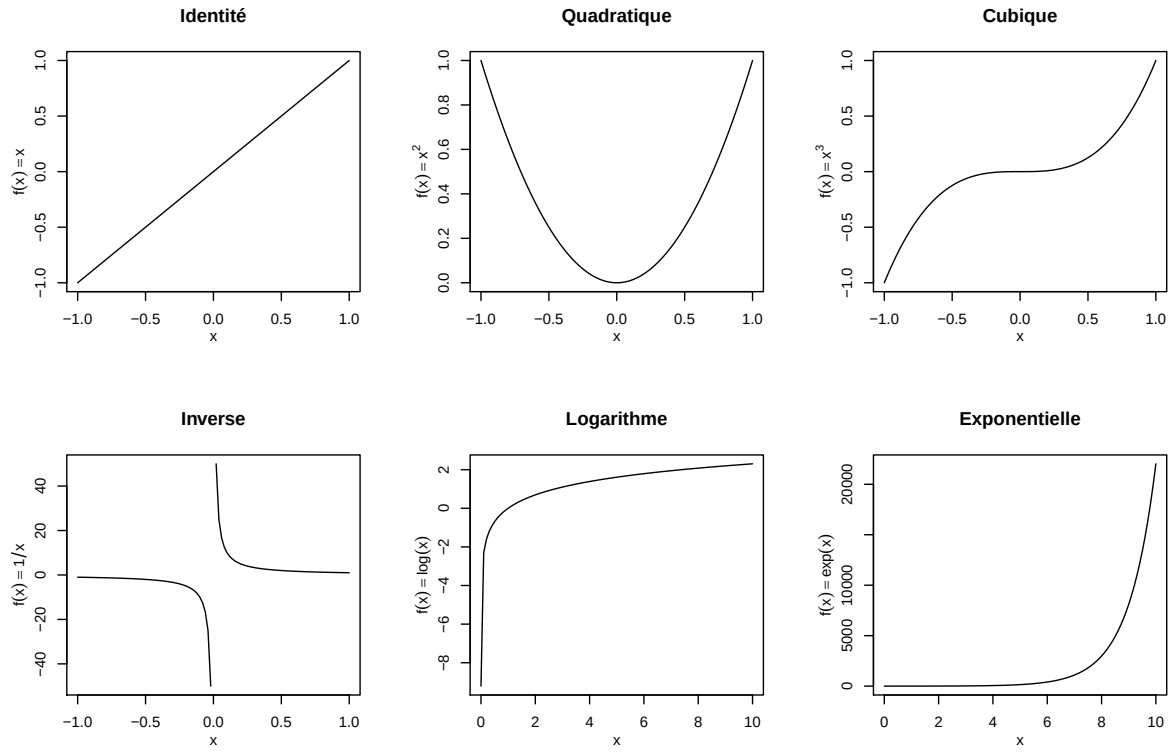

FIGURE 1.11 – Fonctions mathématiques

Chapitre 2

Édition d'un graphe

Dans le chapitre précédent, nous avons rapidement fait le tour des grands types de graphes existants sous R. Bien sûr, il en existe d'autres. Maintenant nous allons voir comment éditer un graphe. En d'autres termes, nous allons ajouter des éléments à un graphique, éléments qui pourront être des points, des lignes, des symboles, du texte ou encore une légende. L'appel aux fonctions précédentes avait pour conséquence d'ouvrir un nouveau périphérique graphique. Ce qui entraînait inévitablement l'effacement du contenu du périphérique actif. Ici, nous allons utiliser des *Low-level plotting functions*, qui n'ouvriront pas de nouveau périphérique graphique mais utiliseront celui déja ouvert (et actif) pour l'éditer. Une condition est alors requise pour que ces fonctions ne génèrent pas d'erreur : un périphérique graphique doit être préalablement ouvert avant leur utilisation.

2.1 Graphe 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 = T))

##  [1]  1  7 10  4  4  9  1  6  6  2  3  6  9  3  3  4 10 10  2  2

> (x2 <- sample(x = 10:20, size = 20, replace = T))

##  [1] 17 19 16 10 13 16 11 16 20 12 16 17 13 11 17 13 11 20 10 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))
```

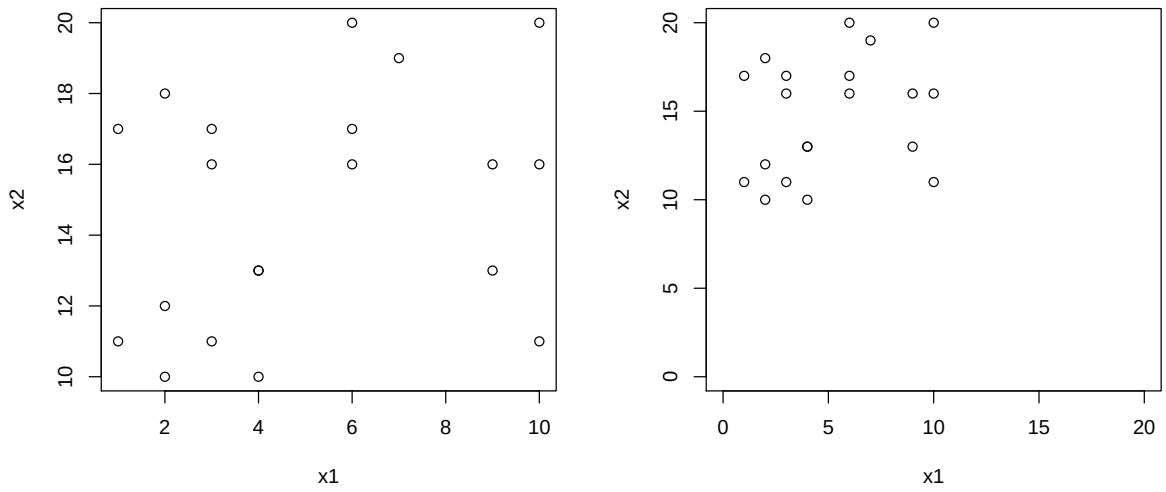

FIGURE 2.1 – Graphe avec axes prédéfinis

Par défaut, la fonction `plot()` affiche une boîte autour de la région graphique. C'est l'argument `bty` qui défini 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")
```

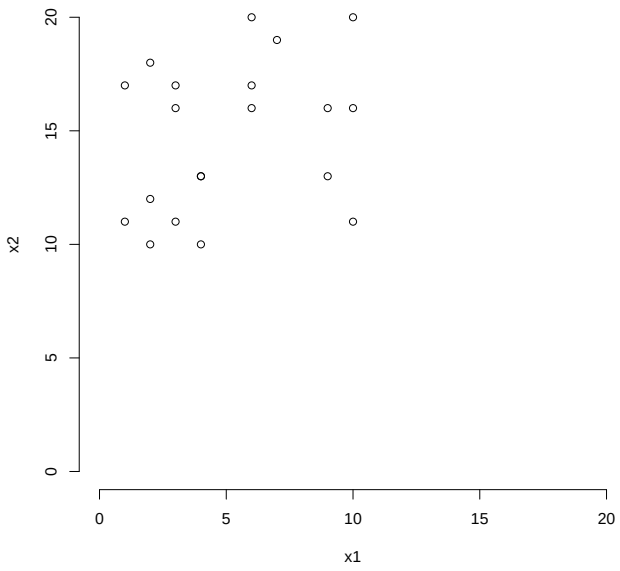

FIGURE 2.2 – Suppression du cadre

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 = F)
```

FIGURE 2.3 – Suppression des axes

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 = F, ann = F)
```

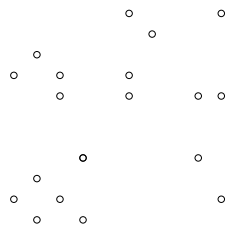

FIGURE 2.4 – Suppression du nom des axes

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 = F, type = "n")
```

FIGURE 2.5 – Graphique vierge

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 = F, ann = F, 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 = F, ann = F, ...) {
+     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")
```

FIGURE 2.6 – Graphique vierge bis

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"`.

2.2 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 = F)
> var3 <- sample(var1, 20, replace = F)
```

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)
```

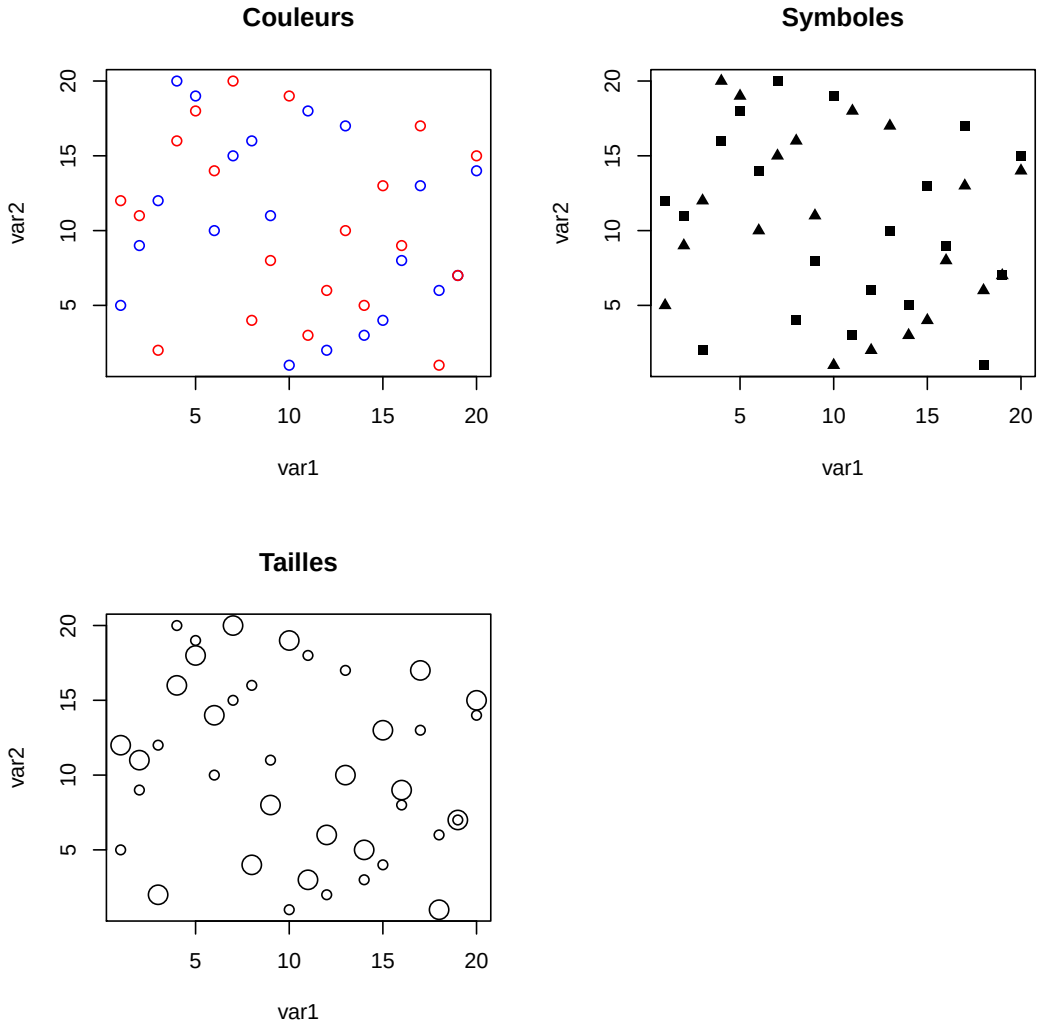

FIGURE 2.7 – Rajout de points

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).

2.3 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))
```

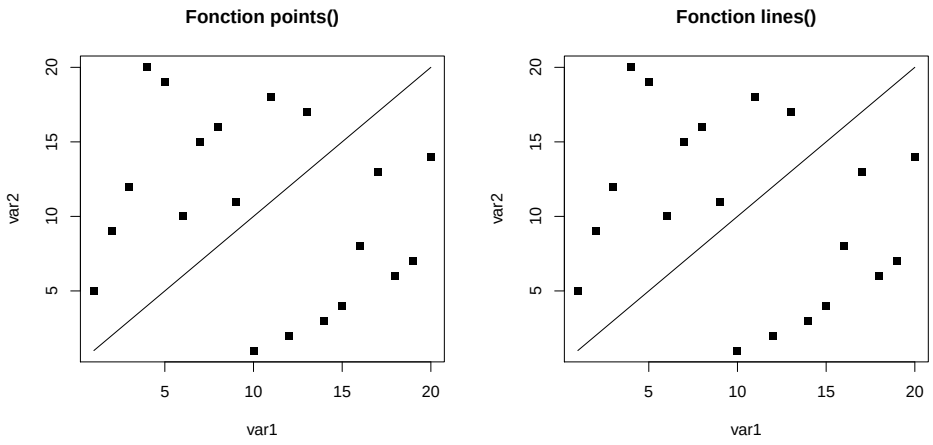

FIGURE 2.8 – Rajout d'une ligne

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])
```

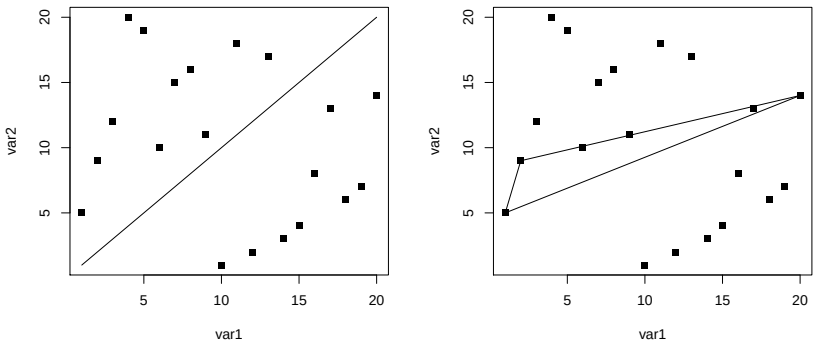

FIGURE 2.9 – Rajout de segments

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'étendront 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)
```

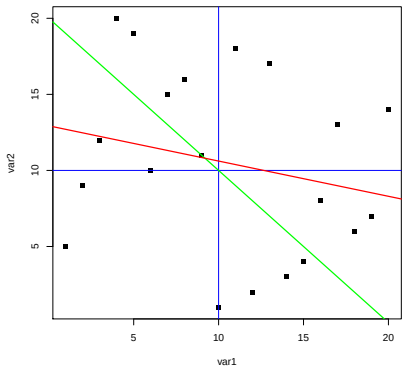

FIGURE 2.10 – Rajout de droites caractéristiques

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")
```

2.4 Ajout de polygones

Insérer une forme polygonale sur un graphe 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)
```

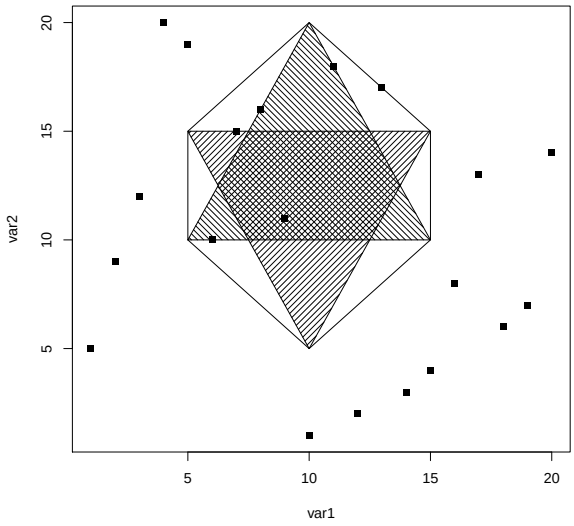

FIGURE 2.11 – Ajout de polygones

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")
```

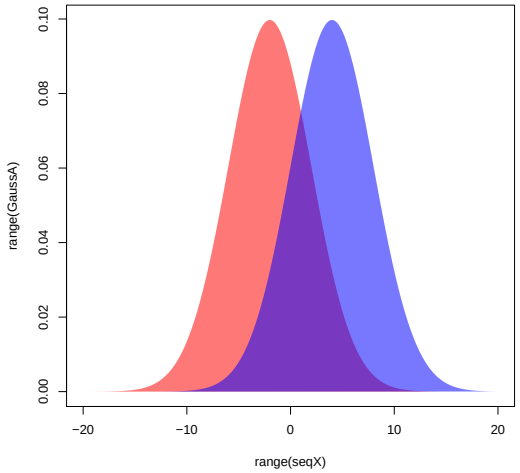

FIGURE 2.12 – Densité de probabilités de fonctions normales

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")
```

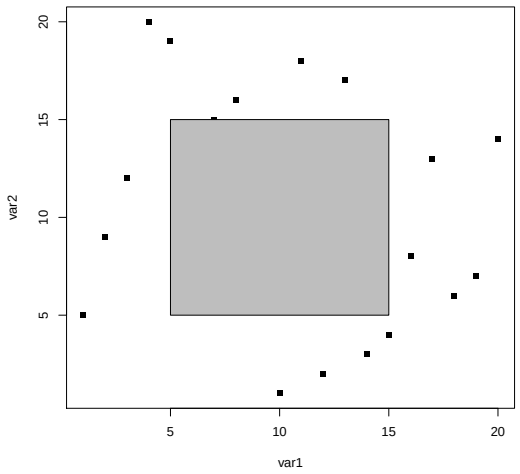

FIGURE 2.13 – Ajout d'un rectangle

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 = F, 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 = "lightgray", border = 0)
> 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, border = "white")
```

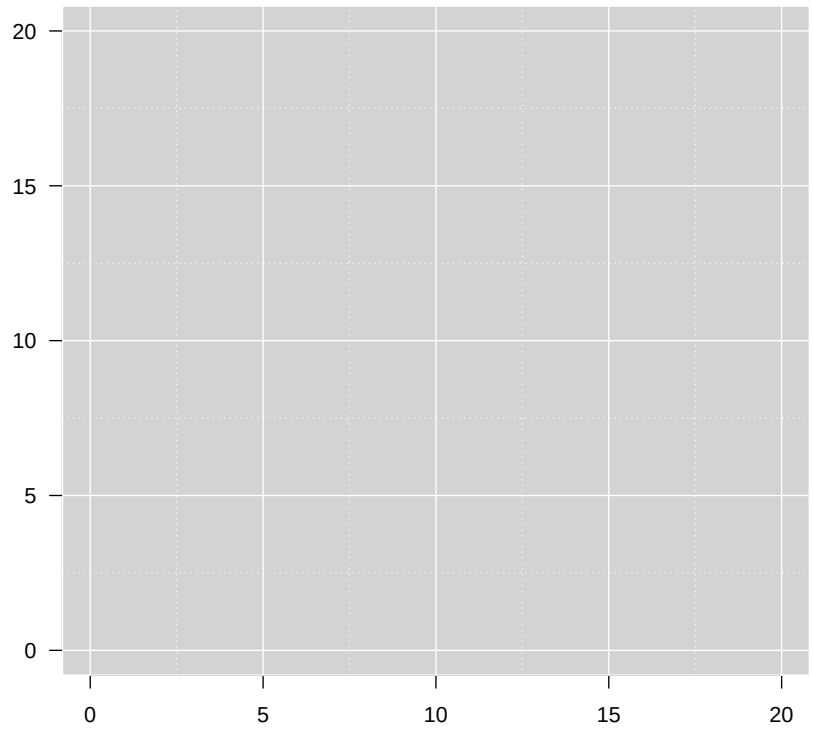

FIGURE 2.14 – Background à la `ggplot2`

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
```

2.5 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 = F)
> 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 = 0.1, angle = 45, code = 3)
```

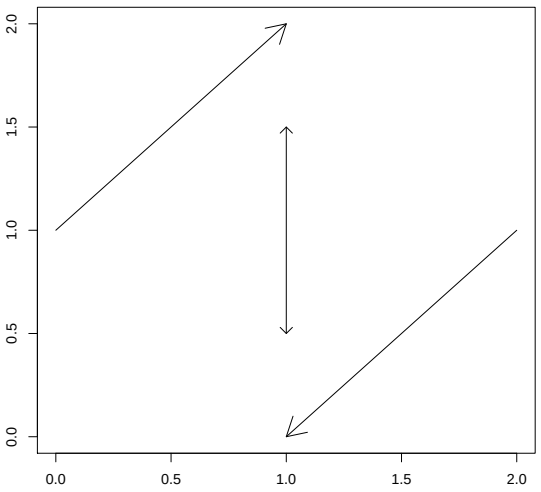

FIGURE 2.15 – Ajout d'une flèche

Nous n'en dirons pas plus sur les flèches.

2.6 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.).

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

TABLE 2.1 – Arguments de la fonction `title()`

Utilisons ces arguments.

```
> par(mgp = c(2, 1, 0))
> plot(var1, var2, pch = 15, ann = F, 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)
```

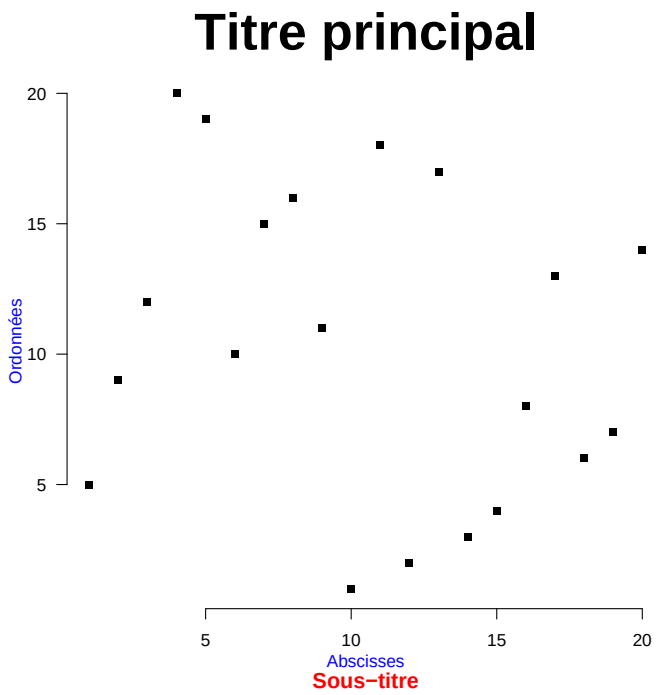

FIGURE 2.16 – Ajout de titres

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()`.

2.7 Ajout de texte

Intéressons-nous maintenant aux annotations textuelles. Nous distinguerons les expressions textuelles insérées dans la zone de plot de celles destinées à apparaitre 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.

Dans la zone de plot

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"
## [17] "q" "r" "s" "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)
```

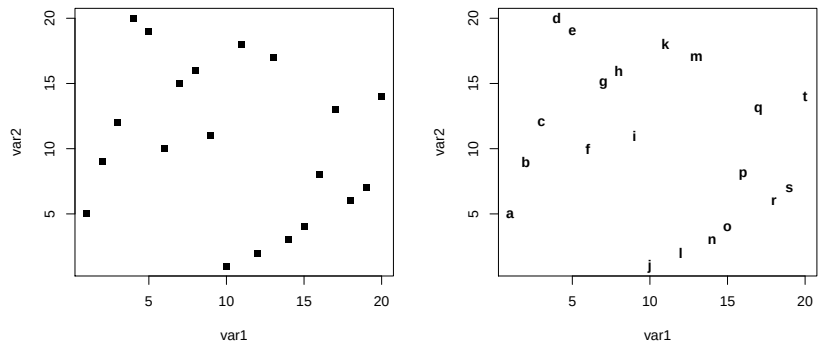

FIGURE 2.17 – Ajout d'un texte

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

TABLE 2.2 – Positionnement d'une expression textuelle

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

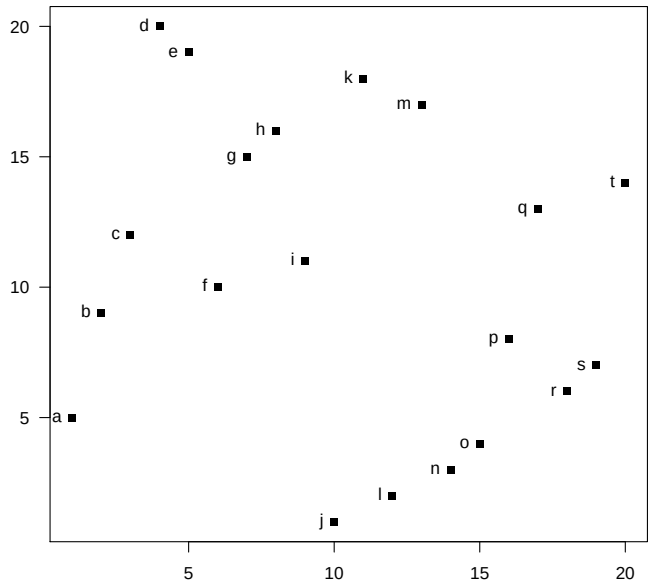

FIGURE 2.18 – Positionnement d'un texte

Remarque : la fonction `locator()` pourra s'appliquer ici.

Dans les marges

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, 0.5, 0))
> plot(var1, var2, pch = 15, ann = F)
> 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)
```

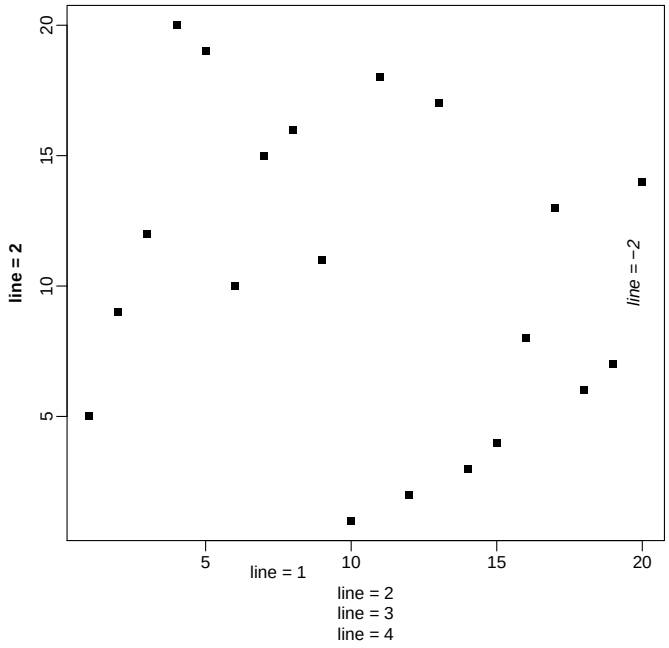

FIGURE 2.19 – Ajout d'un texte dans les marges

2.8 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()`.

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**

TABLE 2.3 – Principaux arguments de la fonction `legend()`

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] 3.143
##
## $rect$h
## [1] 2.512
##
## $rect$left
## [1] 8.428
##
## $rect$top
## [1] 11.26
##
##
## $text
## $text$x
## [1] 9.553 9.553
##
## $text$y
## [1] 10.419  9.581
```

Les informations retournées 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 = F, 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", "white", "darkblue", "orange"),
+     pch = 15, ncol = 2, pt.cex = c(1, 1, 0, 0, 0), text.col = "white",
+     lwd = c(0, 0, 2, 2, 2), lty = c(0, 0, 3, 1, 1), title = "LÉGENDE",
+     cex = 0.75)
```

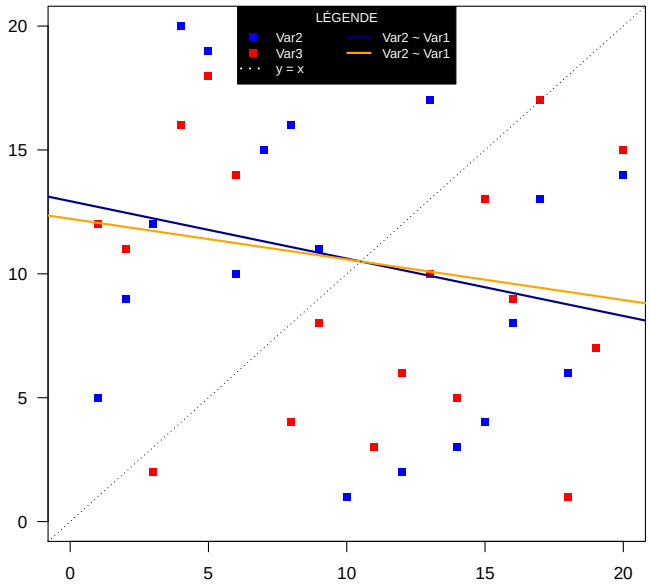

FIGURE 2.20 – Ajout d'une légende

2.9 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

TABLE 2.4 – Principaux arguments de la fonction `axis()`

```
> 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 = F, lwd = -2, lwd.ticks = 1)
> axis(side = 2, at = seq(-1.75, 1.75, by = 0.5), pos = -2, tck = -0.01,
+     labels = F, 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)
```

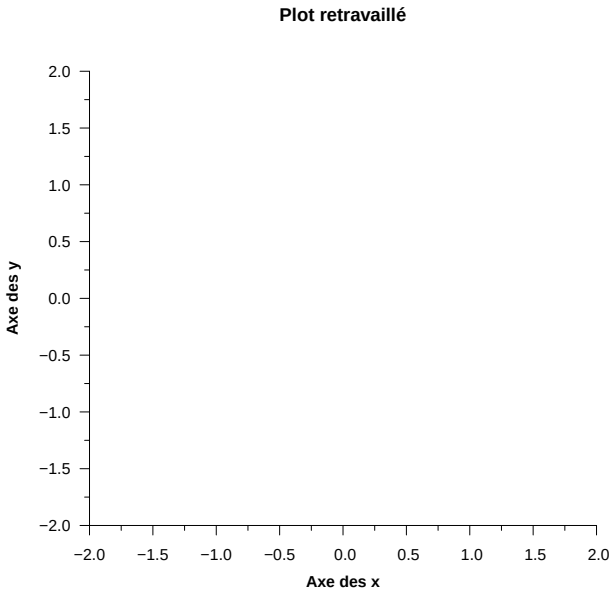

FIGURE 2.21 – Ajout d'un axe

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()`.

2.10 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 = F, xlim = dims, ann = F, 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 prédéfinie. 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()`

TABLE 2.5 – Description de la fonction `plotimage()`

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 = "./icon4.png", add = F)
```

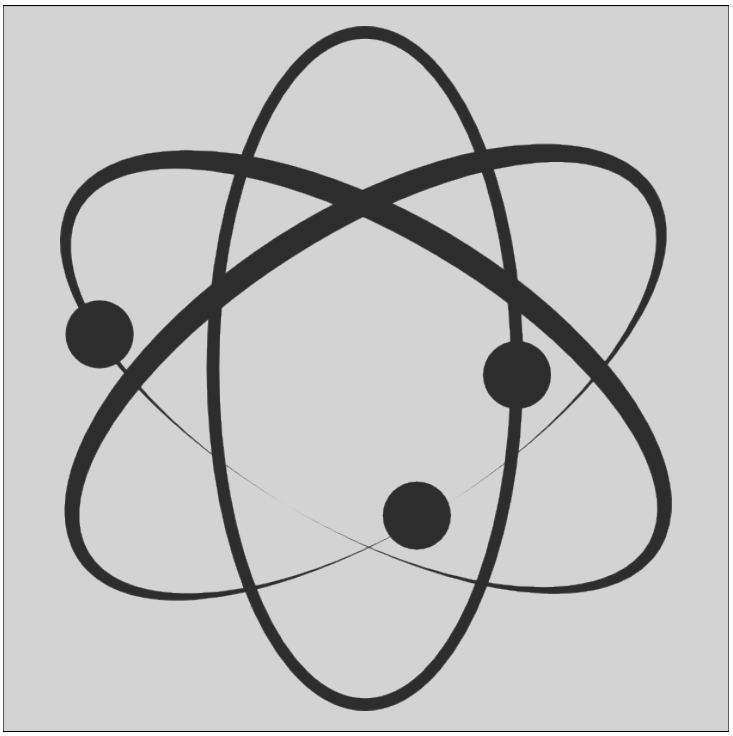

FIGURE 2.22 – Insertion d'une image

Regardons les différents placements par défaut.

```
> par(mfrow = c(3, 3))
> for (i in 0:8) {
+     par(mar = c(1, 1, 1, 1))
+     plotimage("./icon8.png", size = 0.25, pos = i)
+     box("figure")
+ }
```

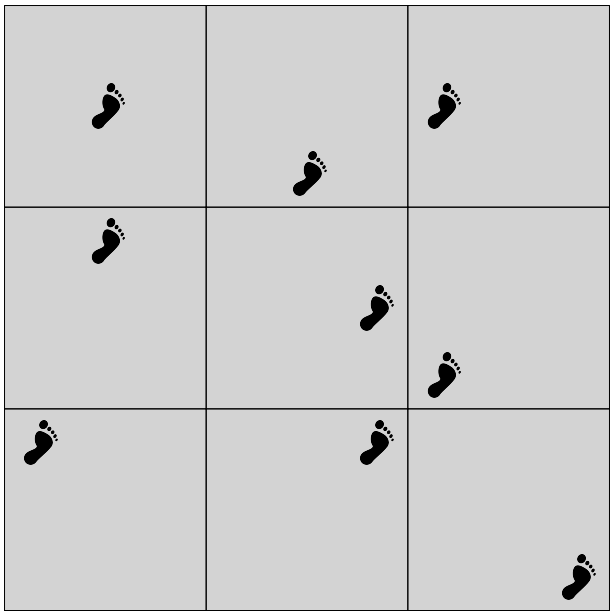

FIGURE 2.23 – Positionnement d'une image

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

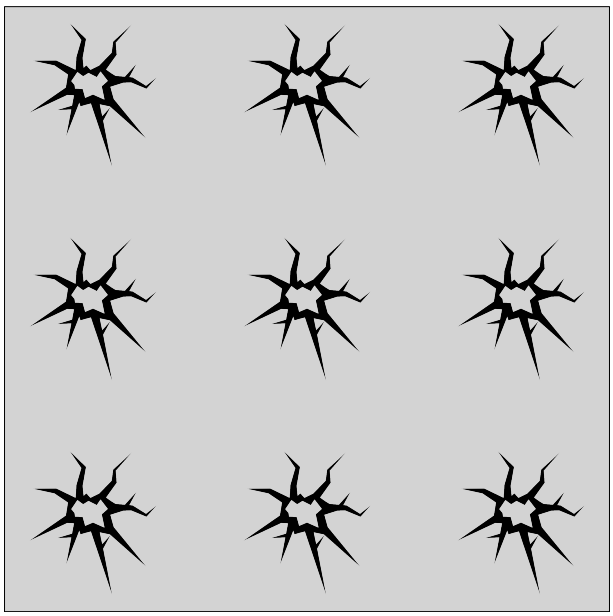

FIGURE 2.24 – Superposition d'images

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 = T, angle = i, x = 0.25,
+         y = 0.25)
+ }
> plotimage("./icon2.png", size = 0.2, add = T, pos = 7)
> box("figure")
```

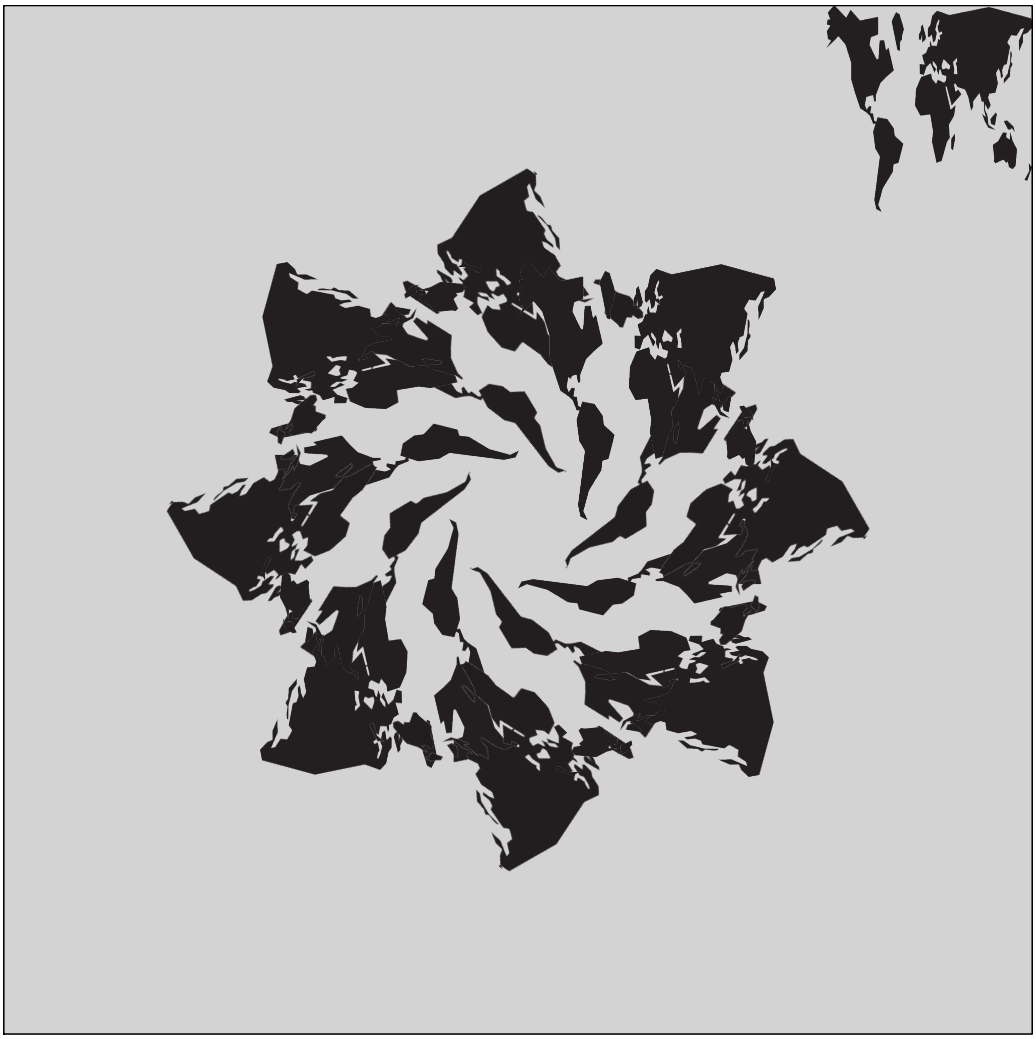

FIGURE 2.25 – Sérendipité artistique

Un petit dernier, parce qu'on aime ça.

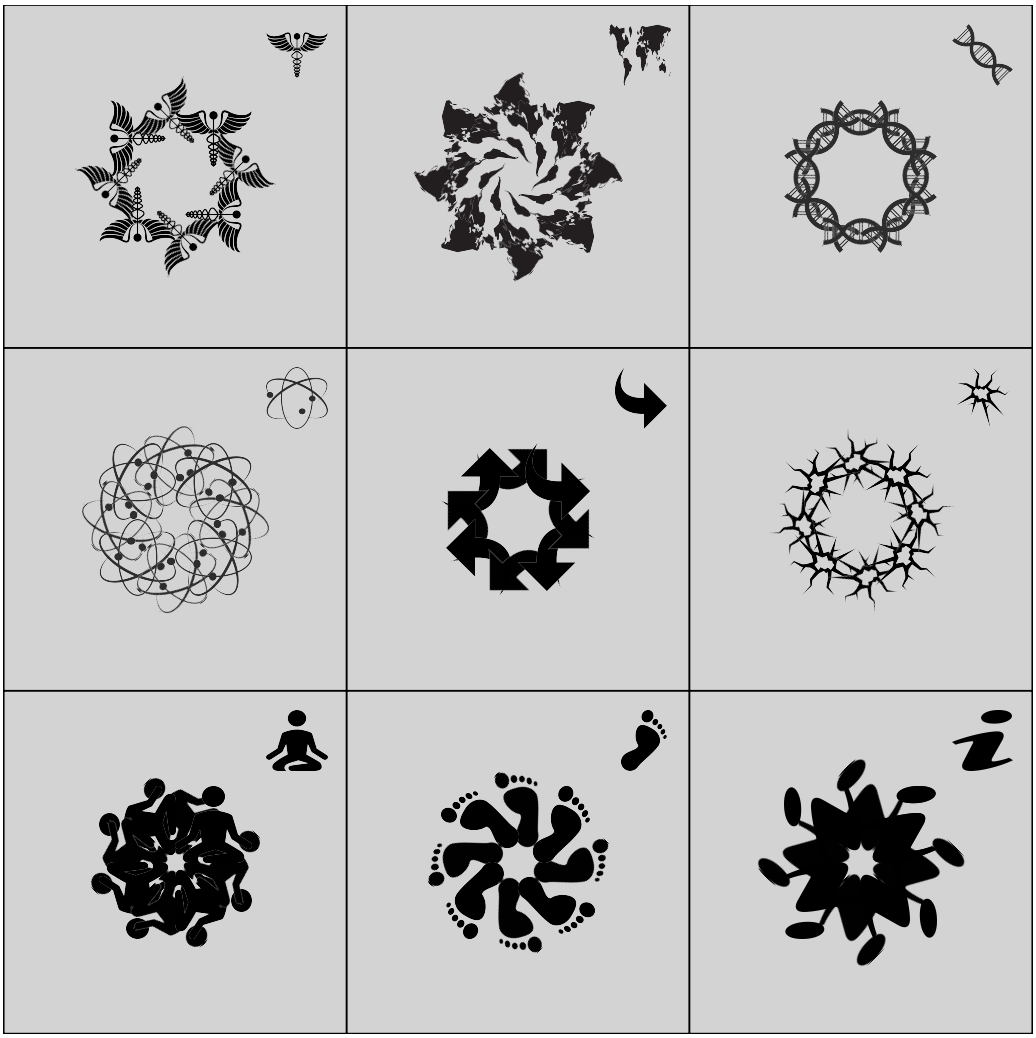

FIGURE 2.26 – Sérendipité artistique bis

Chapitre 3

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).

3.1 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] "transparent"

> 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`, `fig`, `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.

3.2 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 caracterisée 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)
```

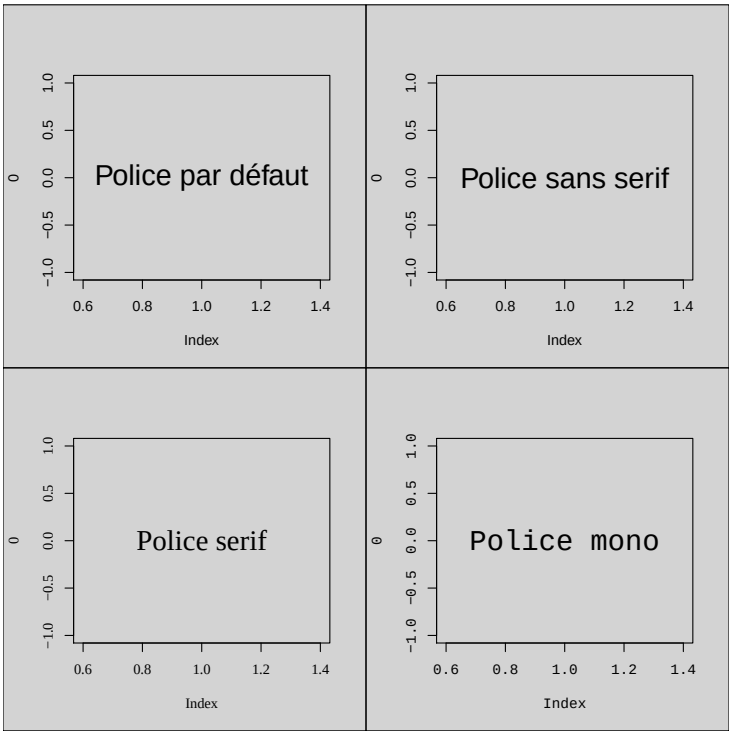

FIGURE 3.1 – Polices de caractères

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.axis`, `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)
```

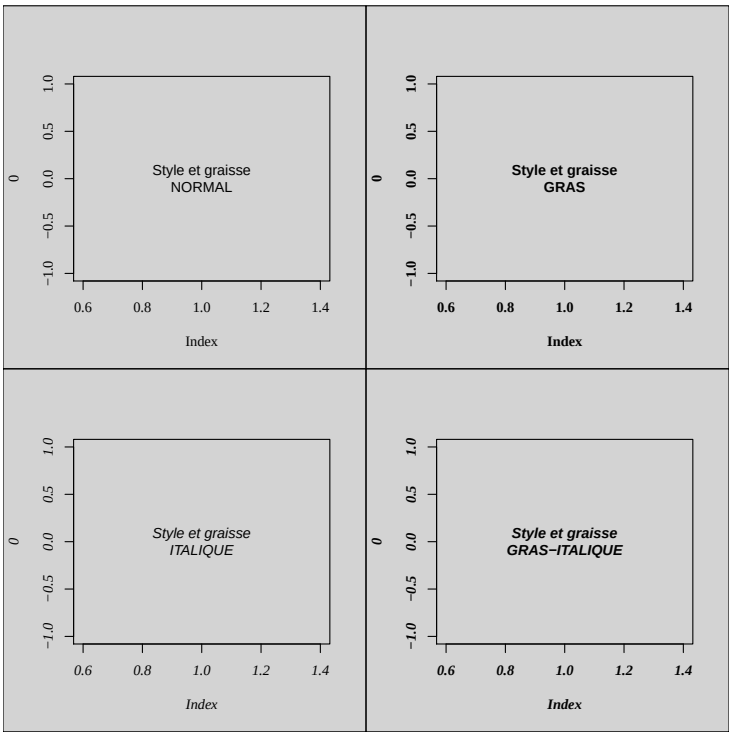

FIGURE 3.2 – Style et graisse de police

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 = 0.35, cex.axis = 0.35)
> text(1, 0, "Corps de .35", font = 2, cex = 0.35)
```

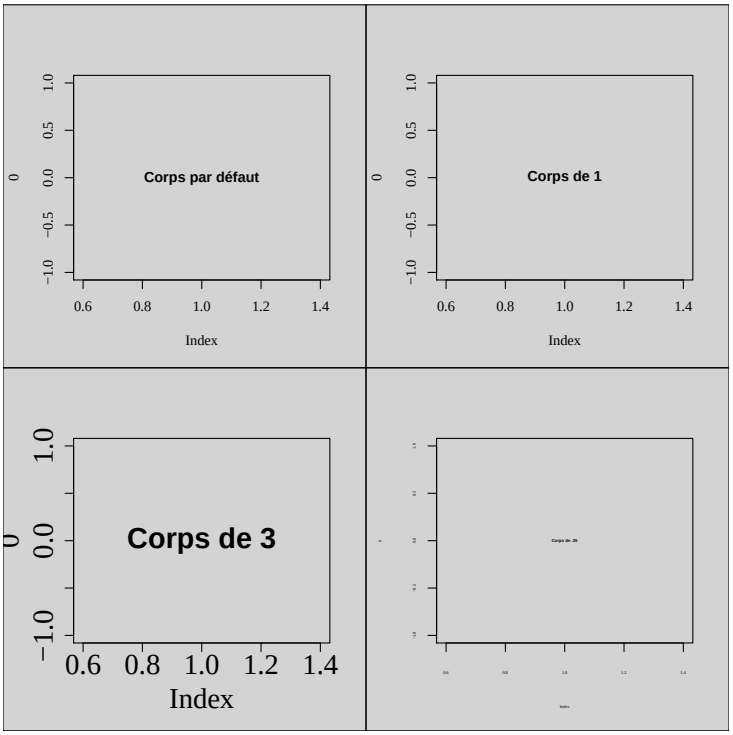

FIGURE 3.3 – Corps de police

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"))
```

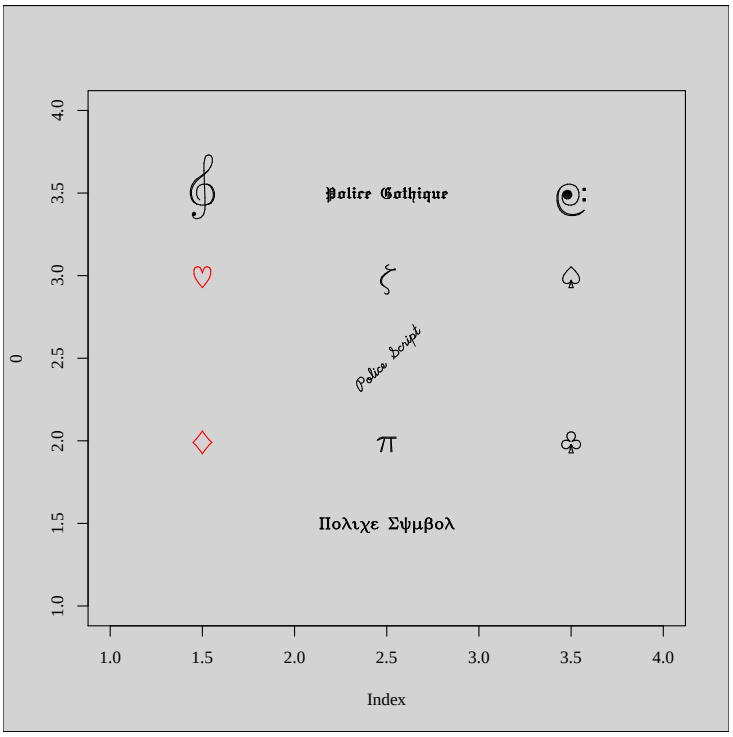

FIGURE 3.4 – Fontes `Hershey`

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)
```

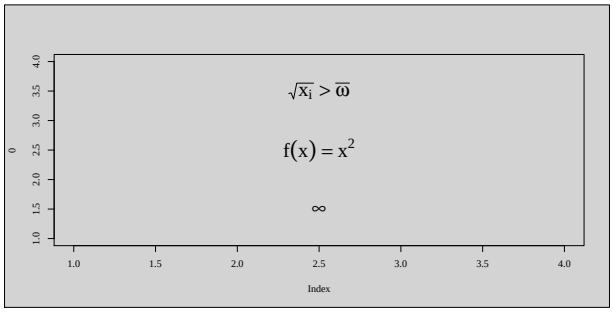

FIGURE 3.5 – Expressions mathématiques

3.3 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 = F)
> for (i in seq(0.5, 50, by = 0.5)) {
+     points(2.5, 2.5, cex = i, col = rainbow(120)[k])
+     k <- k + 1
+ }
```

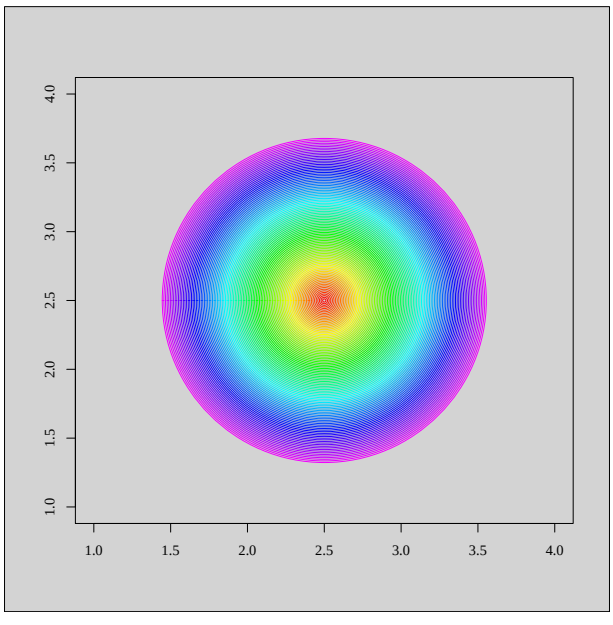

FIGURE 3.6 – Taille et couleur d'un symbole ponctuel

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.

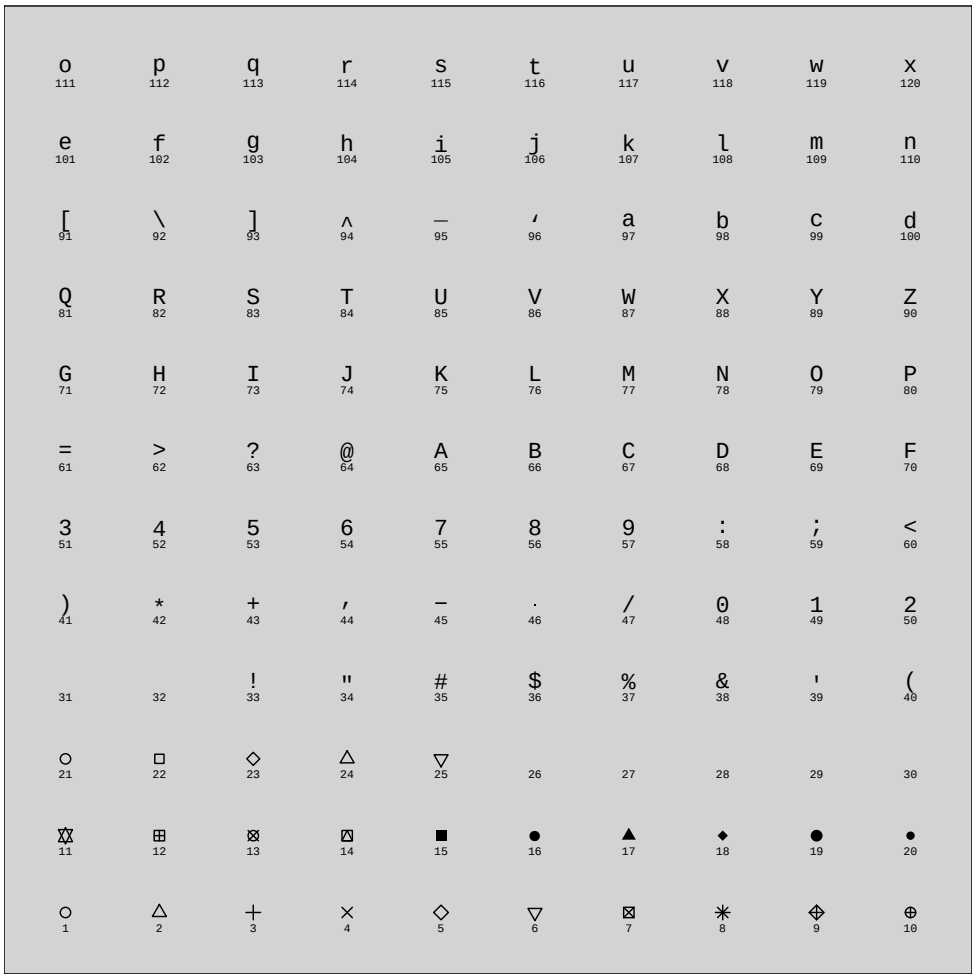

FIGURE 3.7 – Types de symbole ponctuel

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 = F, bty = "n", las = 1)
> text(x, y, "\\#H2330", vfont = c("serif", "plain"))
```

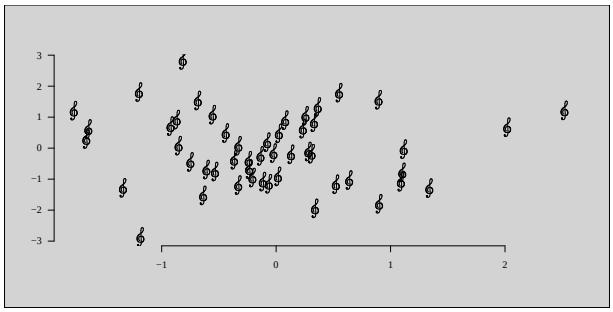

FIGURE 3.8 – Symbole `Hershey`

3.4 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).

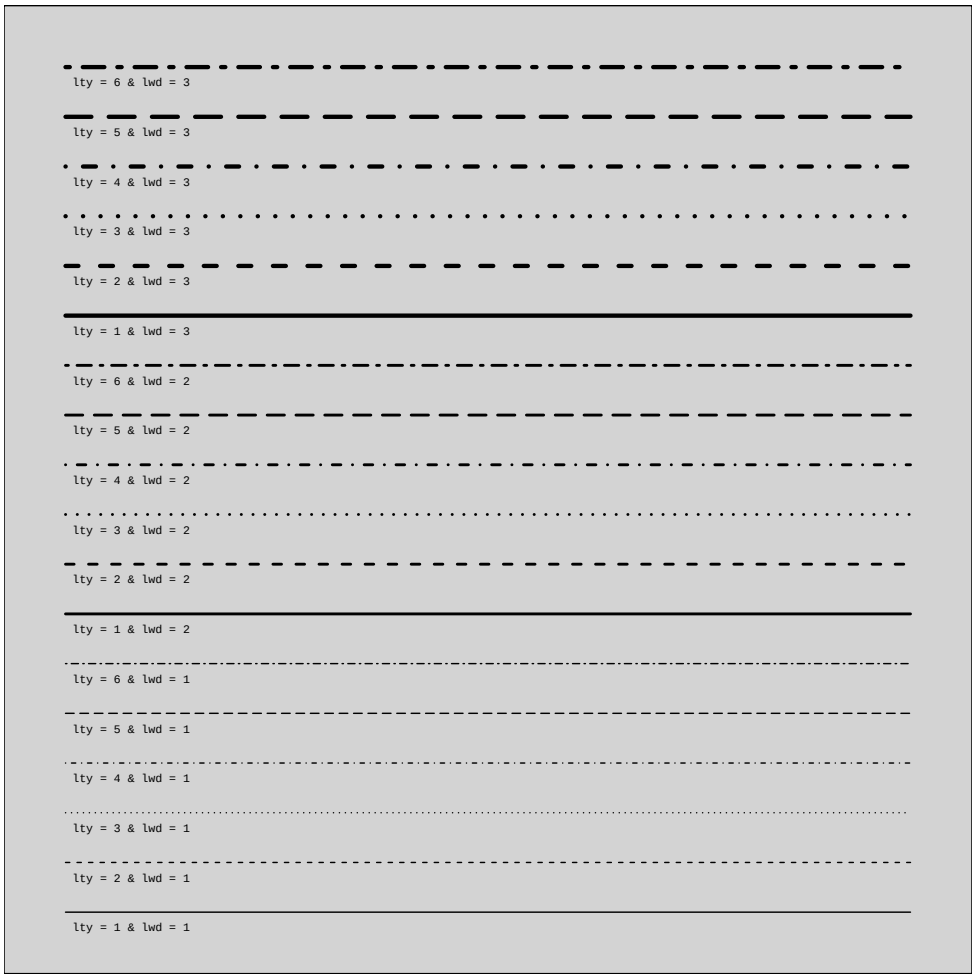

FIGURE 3.9 – Types de lignes

3.5 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-même. 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)
```

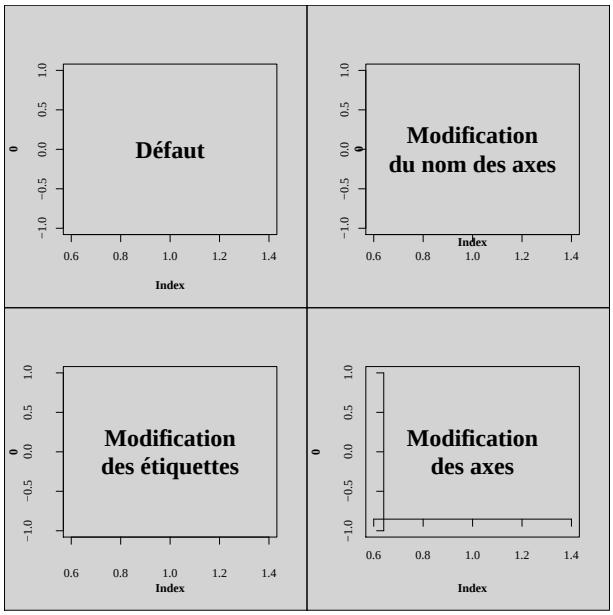

FIGURE 3.10 – Paramètre `mgp`

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

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

TABLE 3.1 – Paramètres graphiques relatifs aux axes

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 = F, ann = F,
+     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 = F, lwd = 0, pos = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = "darkgray")
> axis(side = 2, at = seq(5, 35, by = 10), labels = F, 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 = T)
> y <- sample(1:39, 50, replace = T)
> points(x, y, col = "#FF000080", pch = 19, cex = 2)
```

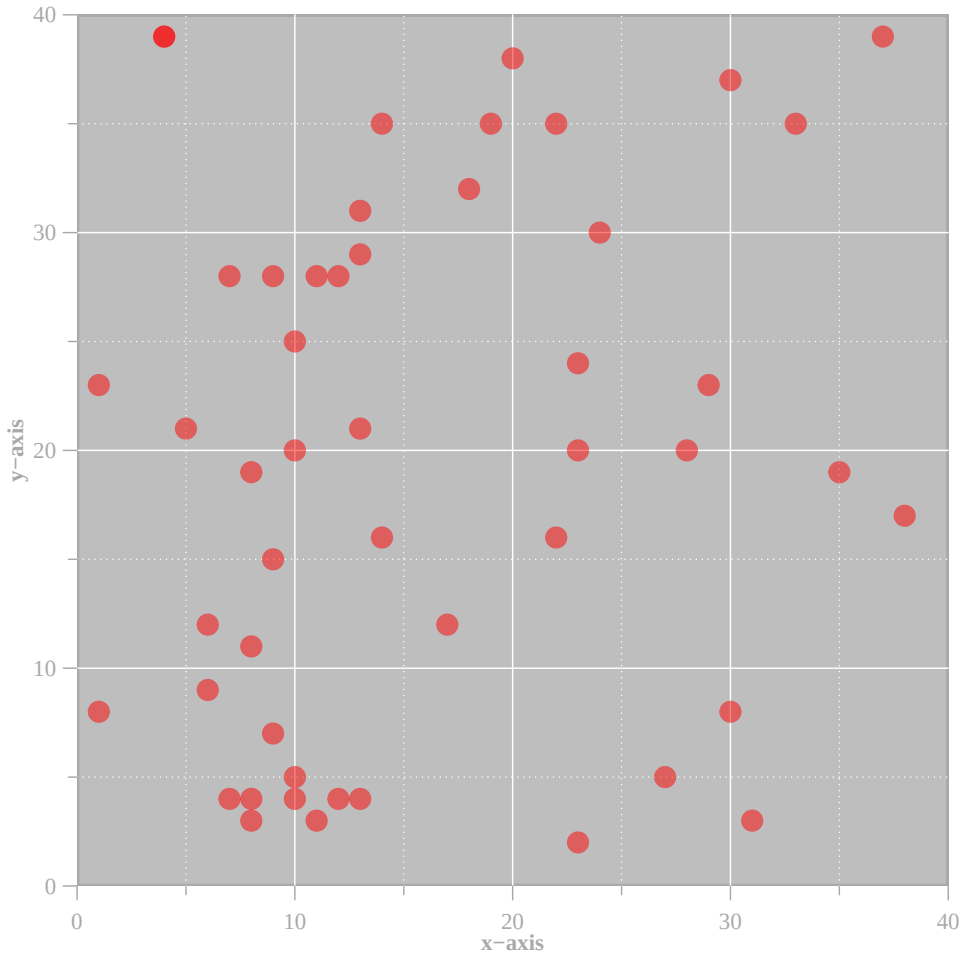

FIGURE 3.11 – Graphique retravaillé

3.6 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)
```

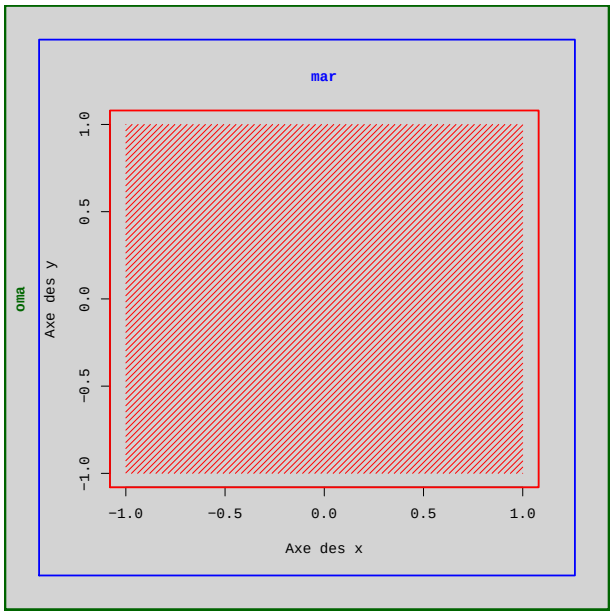

FIGURE 3.12 – Marges d'une figure

```
> 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 = T)
```

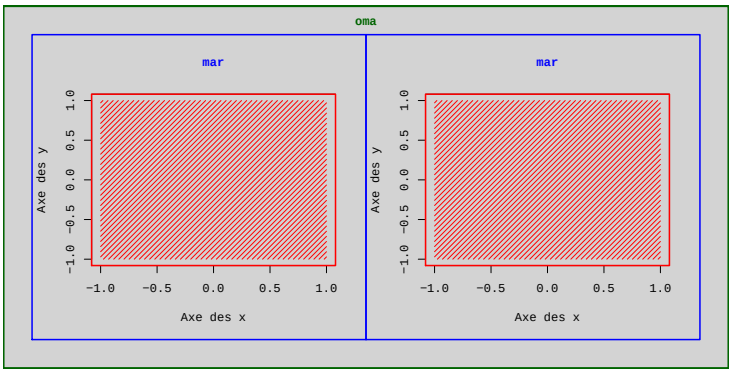

FIGURE 3.13 – Marges d'une figure composite

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)
```

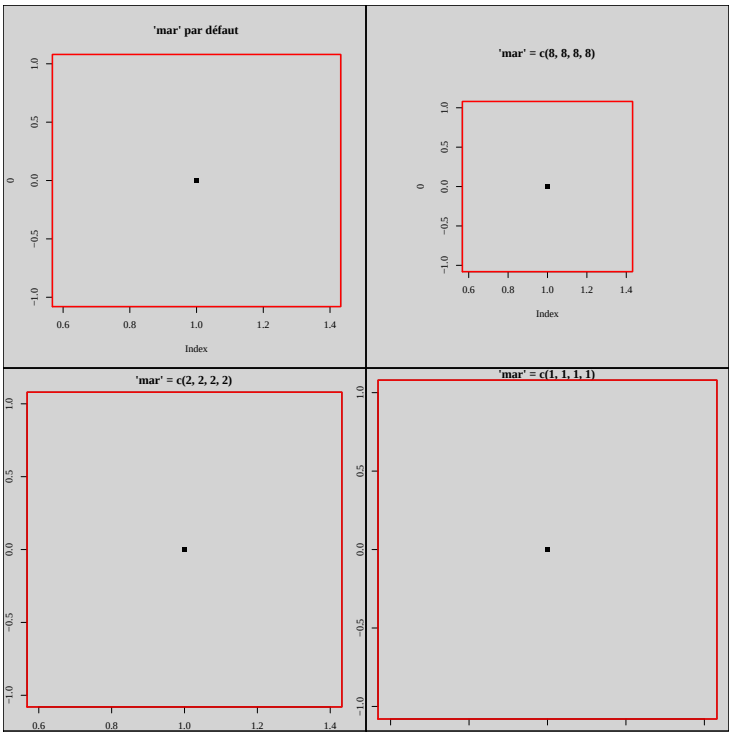

FIGURE 3.14 – Effet du paramètre `mar`

3.7 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.

Palettes prédéfinies

Commençons par le plus simple : les palettes. R dispose d'une palette de base dans laquelle figurent huit couleurs prédéfinies.

```
> palette()

## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta"
## [7] "yellow"  "gray"
```

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.

Le système RGB

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 systeme, 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'étalonnent 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%

TABLE 3.2 – Code RGB de quelques couleurs

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
```

Nomenclature hexadécimale

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 a F). Le tableau suivant donne la correspondance avec le systeme 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

TABLE 3.3 – Correspondance hexadécimal-décimal

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`

TABLE 3.4 – Code hexadécimal de quelques couleurs

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, maxColorValue = 255)

## [1] "#FF0000" "#00FFFF" "#87CEEB"

>
> ## Verifions
> (color <- col2rgb(c("red", "cyan", "skyblue")))

##       [,1] [,2] [,3]
## red    255    0  135
## green    0  255  206
## blue     0  255  235

> col2rgb(col2hex(color, maxColorValue = 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, maxColorValue = 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] "#FF0000FF" "#FF4000FF" "#FF8000FF" "#FFBF00FF" "#FFFF00FF"
##  [6] "#BFFF00FF" "#80FF00FF" "#40FF00FF" "#00FF00FF" "#00FF40FF"
## [11] "#00FF80FF" "#00FFBFFF" "#00FFFFFF" "#00BFFFFF" "#0080FFFF"
## [16] "#0040FFFF" "#0000FFFF" "#4000FFFF" "#8000FFFF" "#BF00FFFF"
## [21] "#FF00FFFF" "#FF00BFFF" "#FF0080FF" "#FF0040FF"

>
> ## Dégradé de gris
> gray(seq(0, 1, length.out = 10))

##  [1] "#000000" "#1C1C1C" "#393939" "#555555" "#717171" "#8E8E8E"
##  [7] "#AAAAAA" "#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 = F)
> box("figure", lwd = 2)
> image(matrix(c(0:255), ncol = 1), col = gray(c(0:255)/255), axes = F)
> box("figure", lwd = 2)
```

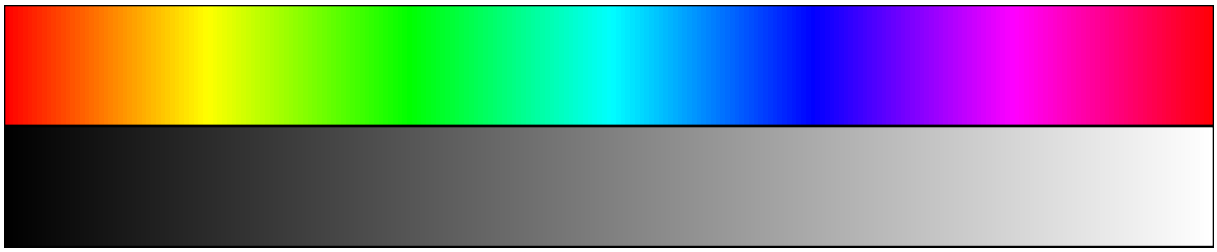

FIGURE 3.15 – Palettes de couleurs hexadécimales

Sélection interactive

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 = 0.5)
+     }, windows = {
+         x11(width = 7, height = 0.5)
+     })
+     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)
```

FIGURE 3.16 – Dégradés de gris

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

FIGURE 3.17 – Couleurs sélectionnées

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"
##  [7] "#FFFFF2" "#FFFFF0" "#FFFFEE" "#FFFFEC" "#FFFFEA" "#FFFFE8"
```

Utilisons cette palette de couleurs avec notre fonction interactive.

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

FIGURE 3.18 – Palette personnalisée

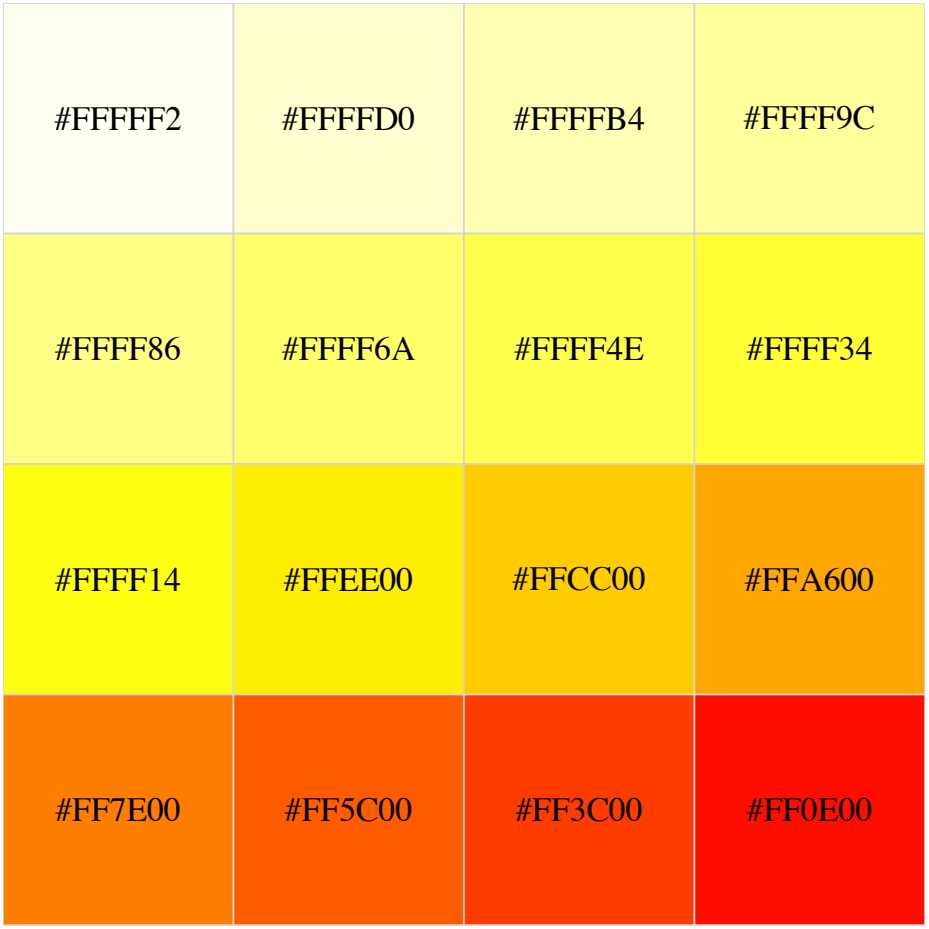

FIGURE 3.19 – Couleurs sélectionnées bis

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...

Chapitre 4

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.

4.1 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 systeme 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 developpez 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 peripherique 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()
```

4.2 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
`dev.list()`	Affiche la liste des périphériques ouverts
`dev.cur()`	Affiche le périphérique actif
`dev.prev()`	Affiche le périphérique précédent
`dev.next()`	Affiche le périphérique suivant
`dev.set(n)`	Sélectionne le périphérique n
`dev.off()`	Ferme le périphérique actif
`dev.copy()`	Copie le contenu d'un périphérique dans un autre

TABLE 4.1 – Les fonctions de la famille `dev.x()`

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

4.3 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 disponible sur votre système d'exploitation.

```
> capabilities()

##     jpeg      png     tiff    tcltk      X11     aqua http/ftp
##     TRUE     TRUE     TRUE     TRUE    FALSE     TRUE     TRUE
##  sockets   libxml     fifo   cledit    iconv      NLS  profmem
##     TRUE     TRUE     TRUE     TRUE     TRUE     TRUE     TRUE
##    cairo
##     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] TRUE
##
## $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.

Chapitre 5

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 `mfrow` 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`.

5.1 Partitionnement basique

Nous avons déjà utilisé l'argument `mfrow` 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")
```

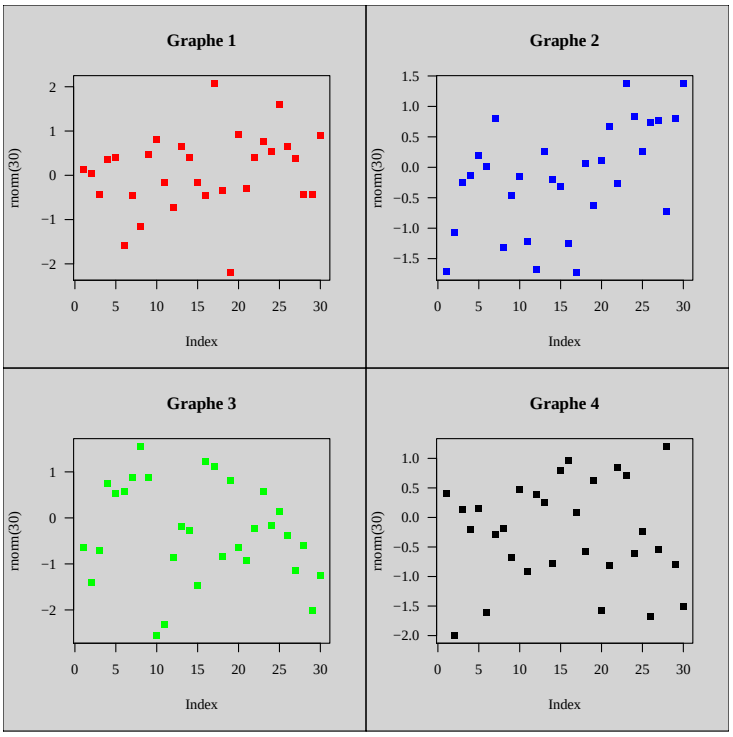

FIGURE 5.1 – Partitionnement basique

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")
```

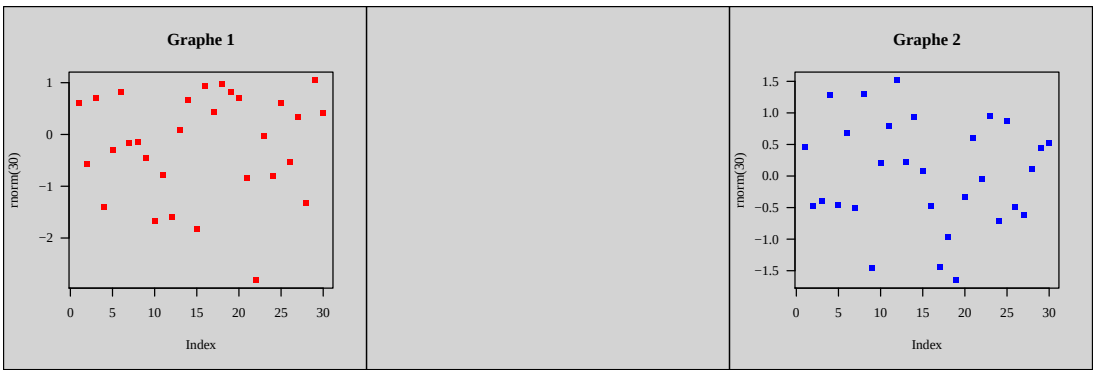

FIGURE 5.2 – Partitionnement basique avec sélection de régions

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

5.2 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 = T))

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4

> layout(mat)
> layout.show(n = 4)
```

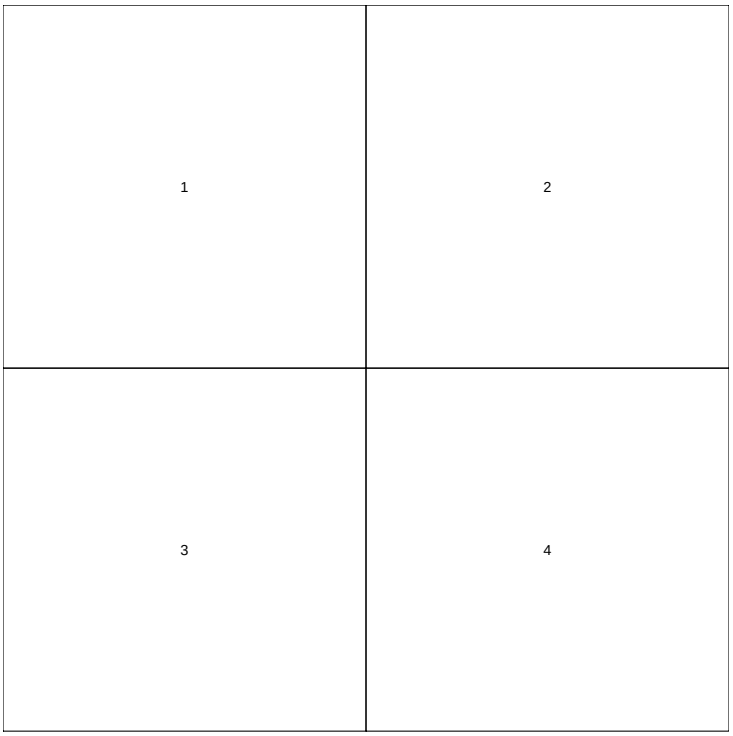

FIGURE 5.3 – Partitionnement avec la fonction `layout()`

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 = F, axes = F)
+     rect(-1, -1, 1, 1, col = palette()[i])
+ }
> box("outer")
```

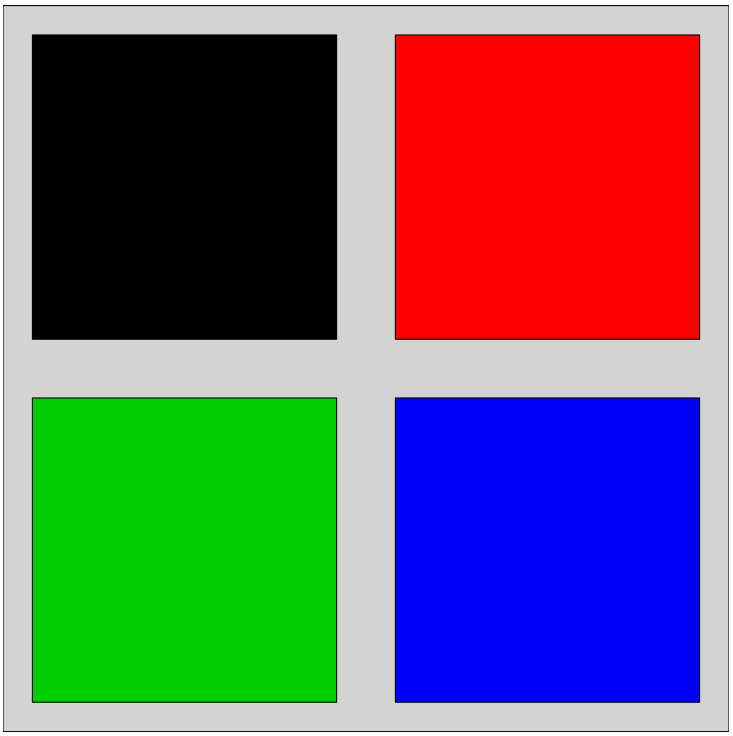

FIGURE 5.4 – Ordre de remplissage avec `layout()`

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 `mfrow` ou `mfcol`. 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 = T))

##      [,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 = F, axes = F)
+     rect(-1, -1, 1, 1, col = palette()[i])
+ }
> box("outer")
```

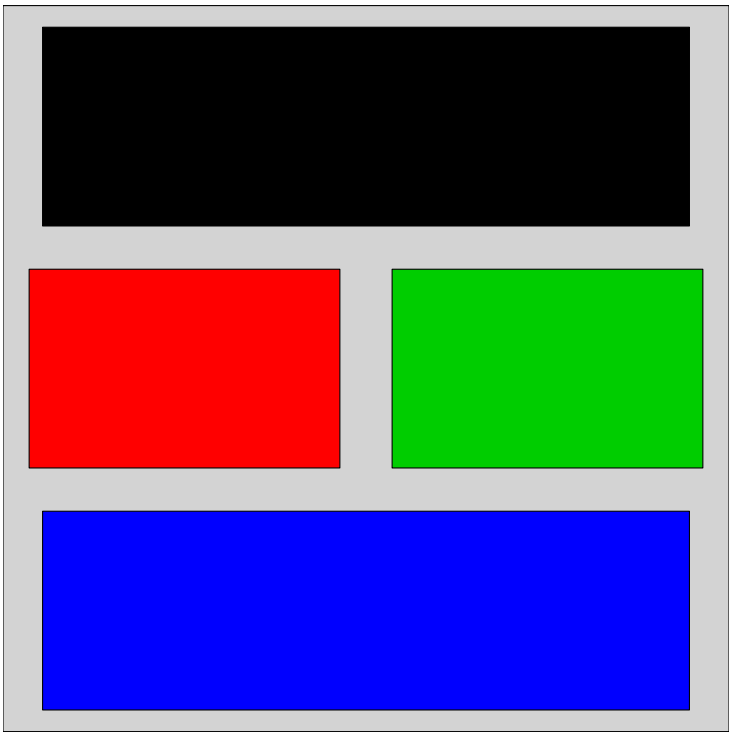

FIGURE 5.5 – Fusion de régions

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)
```

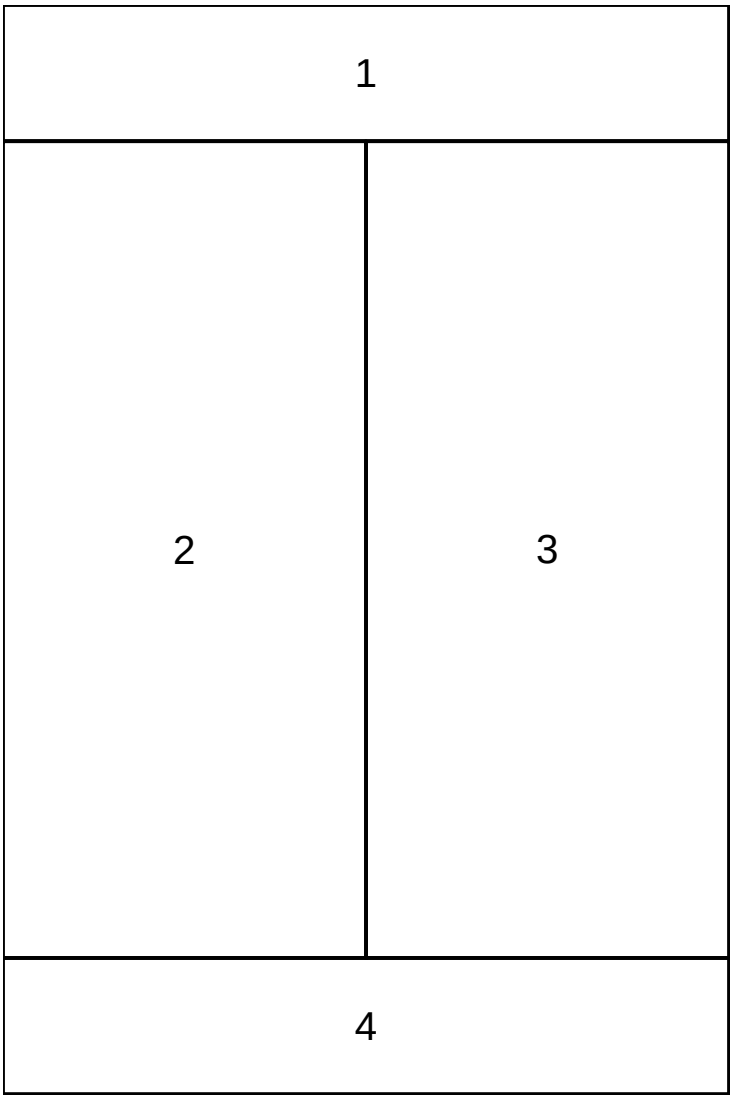

FIGURE 5.6 – Redimensionnement des régions d'un `layout()`

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 éditerez 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.

5.3 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.

Superposition de graphes

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 = T)
> y2 <- sort(y1)/100
> y1 <- sort(y1, decreasing = T)
>
> ## Premier graphe
> par(cex.axis = 0.75, ann = F)
> plot(x, y1, col = "steelblue", type = "h", lwd = 5, ylim = c(0, 2000),
+     axes = F, 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 = T)
> plot(x, y2, type = "l", col = "red", lwd = 2, ann = F, ylim = c(0, 30),
+     axes = F, xlim = c(0, 1550))
> axis(4, pos = 1550, seq(0, 30, 5), seq(0, 30, 5), las = 1, col = "red",
+     col.axis = "red")
```

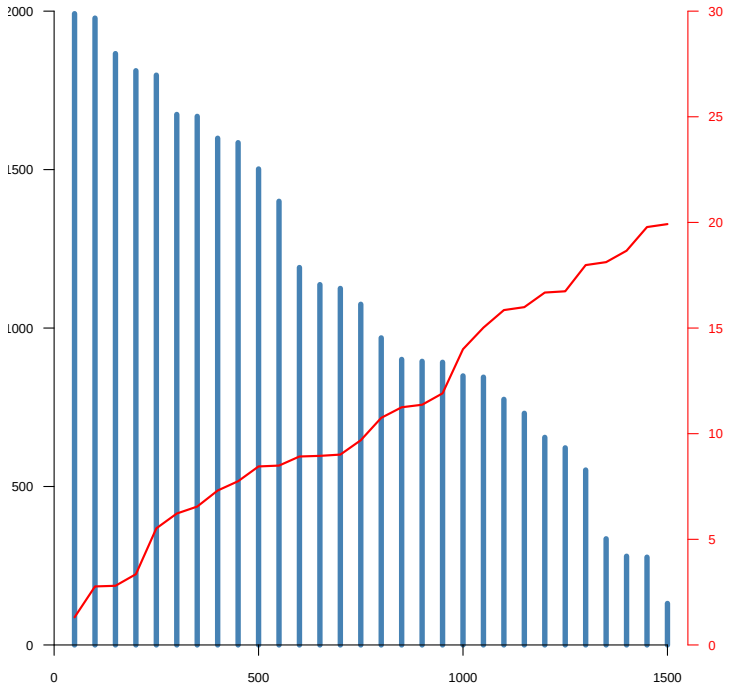

FIGURE 5.7 – Superposition de graphes

Mise à part l'inclusion de la commande `par(new = T)`, 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.

Inclusion en médaillon

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(0.5, 1, 0.5, 1), new = T)
```

Et pour un médaillon placé au centre,

```
> par(fig = c(0.25, 0.75, 0.25, 0.75), new = T)
```

Une remarque maintenant : si vous modifiez les marges des graphes, et que vous souhaitiez 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, 0.75, 0), xaxs = "i", yaxs = "i")
> par(family = "serif", font.axis = 2)
> par(mar = c(2, 2, 2, 2), cex.axis = 0.75)
> plot(c(0, 5), c(0, 5), xaxt = "n", yaxt = "n", ann = F, type = "n")
> x <- sample(1:3, 10, replace = T)
> y <- sample(1:4, 10, replace = T)
> 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 = T)
> par(fig = c(0.7, 1, 0.7, 1), mar = c(0, 0, 2, 2), cex.axis = 0.75)
> par(xaxs = "i", yaxs = "i", mgp = c(0, 0.25, 0), tck = -0.02)
> 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 = T)
> y <- sample(15:50, 30, replace = T)
> 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")
```

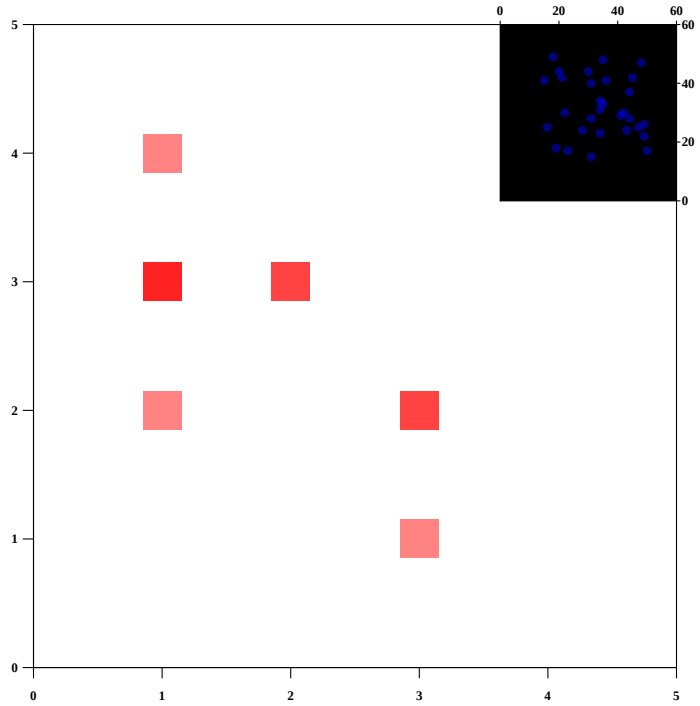

FIGURE 5.8 – Inclusion en médaillon

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.

Chevauchement de graphes

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, 0.75, 0), xaxs = "i", yaxs = "i")
> par(family = "serif", font.axis = 2, col.axis = "white", col = "white")
> par(fig = c(0.45, 1, 0.45, 1), mar = c(0, 0, 2, 2), cex.axis = 0.75)
> 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 = T)
> y <- sample(1:4, 10, replace = T)
> 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 = T)
> par(fig = c(0, 0.55, 0, 0.55), mar = c(2, 2, 0, 0), cex.axis = 0.75)
```

```
> par(xaxs = "i", yaxs = "i", mgp = c(0, 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 = T)
> y <- sample(1:4, 10, replace = T)
> 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")
```

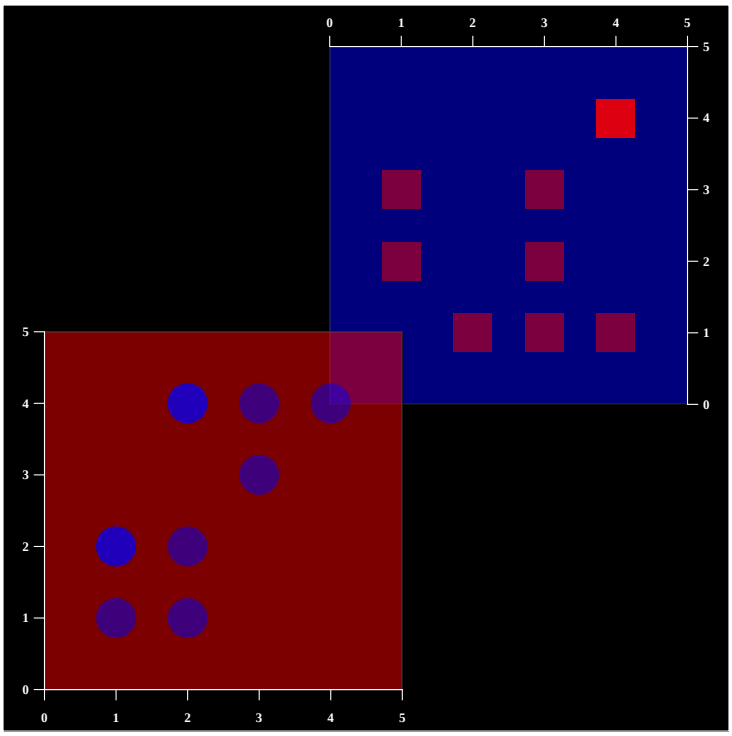

FIGURE 5.9 – Chevauchement de graphes

Chapitre 6

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 même 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.

6.1 Partitionnement avancé

Le but de cet exercice est de réaliser la figure 6.1. Pour ce faire, vous aurez besoin des données (téléchargez-les sur cette page) :

```
> ## URL fixe
> url <- "https://drive.google.com/open"
> ## Requete PHP
> php <- "id=0B28sTqSLvpcsUXVpSlU5YW5lM2M&authuser=0"
> ## Ouverture du lien
> browseURL(url = paste(url, php, sep = "?"))
```

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

```
> ## Importation des donnees
> load(file = "datadem1.Rdata")
> head(dat)

##        x      y      z
## 1  9.782  6.873 Group1
## 2  7.185 10.159 Group1
## 3  8.259 10.194 Group1
## 4 13.877 10.060 Group1
## 5  9.997 16.340 Group1
## 6 10.249 14.765 Group1
```

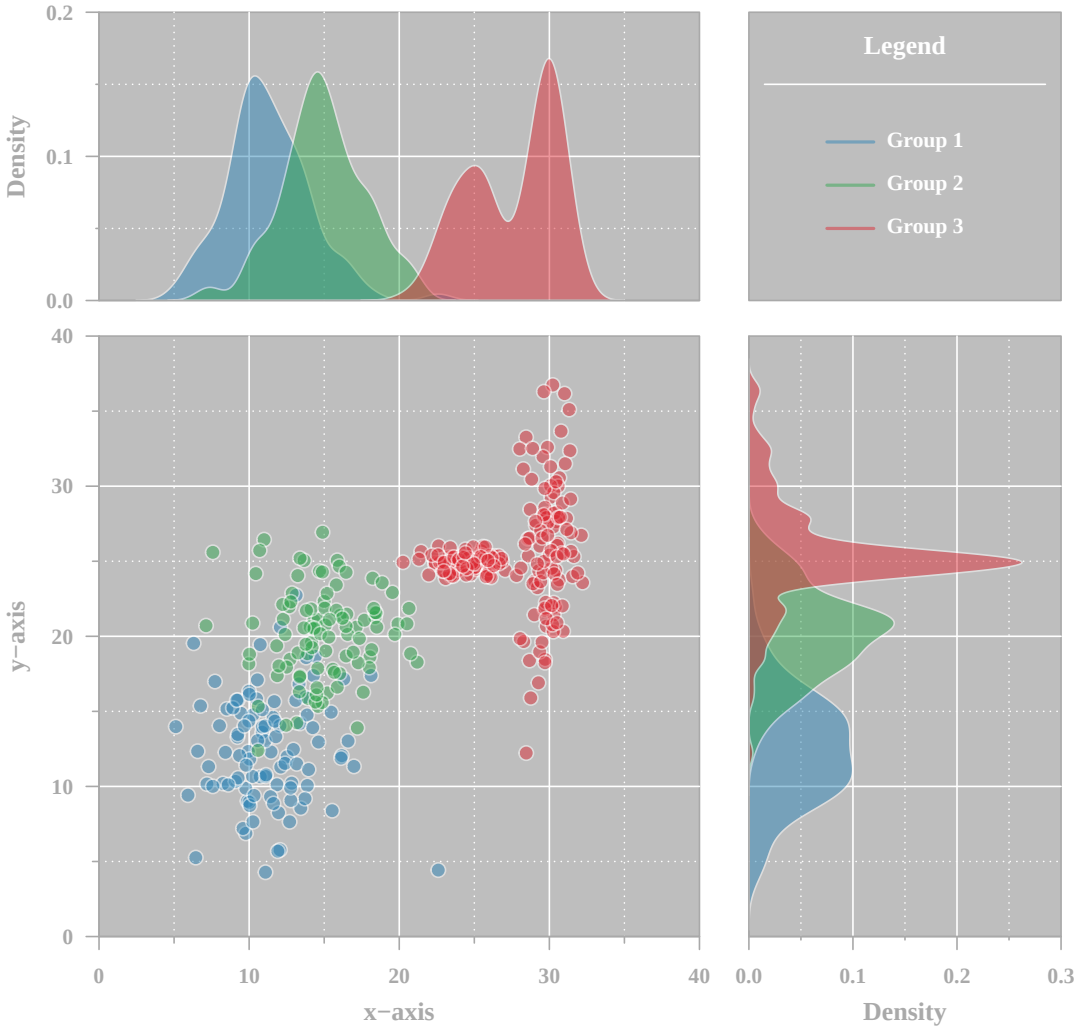

FIGURE 6.1 – Exercice - Partitionnement avancé

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

6.2 Superposition de graphes

Passons au second exercice. La figure 6.2 est le résultat auquel vous devriez arriver. Comme précédemment, téléchargez les données nécessaires avec les lignes de commande ci-dessous, et importez-les.

```
> ## URL fixe
> url <- "https://drive.google.com/open"
> ## Requete PHP
> php <- "id=0B28sTqSLvpcsemNsamRYTE93bUk&authuser=0"
> ## Ouverture du lien
> browseURL(url = paste(url, php, sep = "?"))
```

```
> ## Importation des donnees
> load(file = "datadem2.Rdata")
> head(dat)

## [1]  9.782  7.185  8.259 13.877  9.997 10.249
```

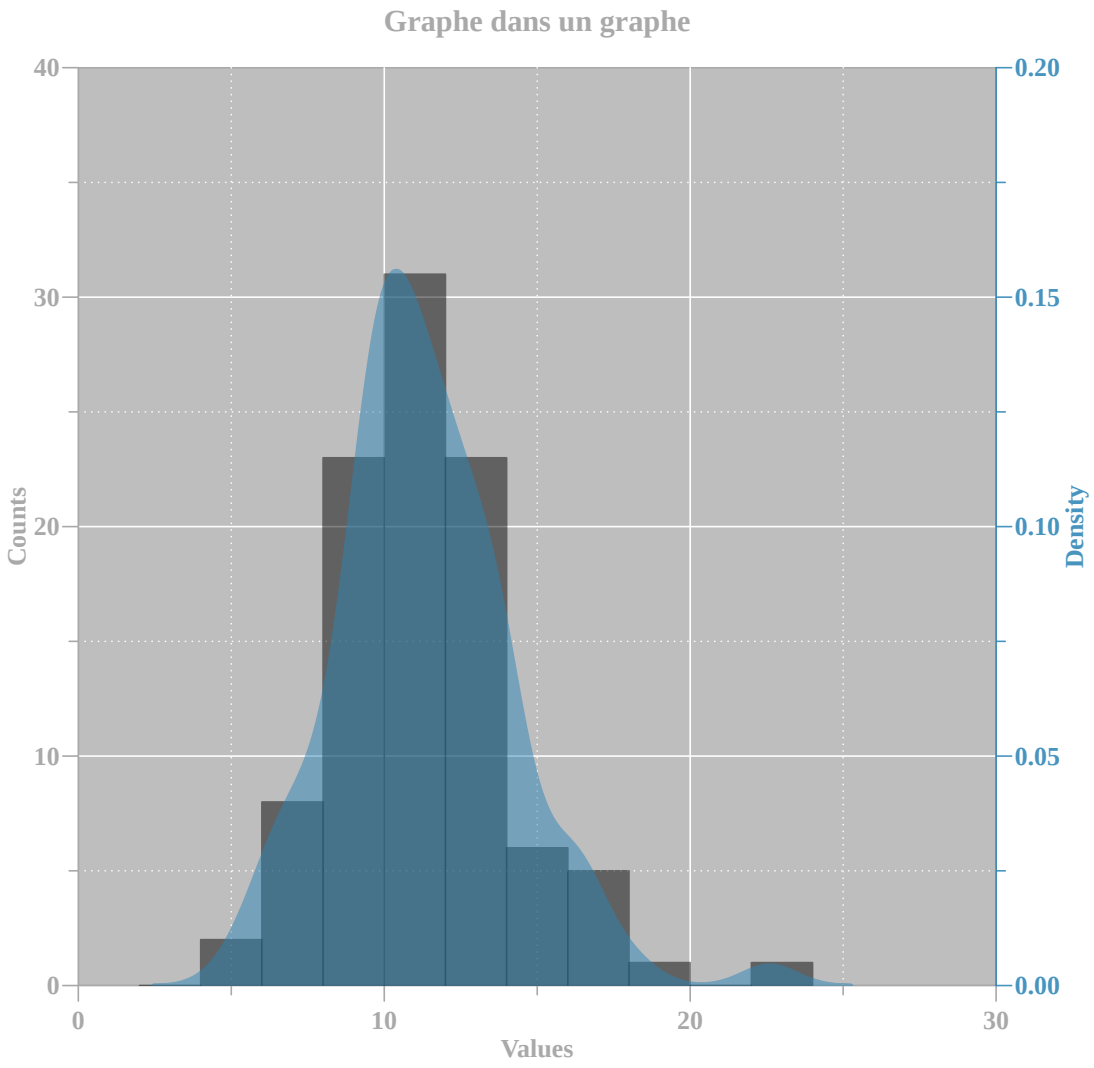

FIGURE 6.2 – Exercice - Superposition de graphes

6.3 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 6.3). Vous pouvez télécharger les données sur cette page :

```
> ## URL fixe
> url <- "https://drive.google.com/open"
> ## Requete PHP
```

```
> php <- "id=0B28sTqSLvpcsX0RBT3lYeEdXMm8&authuser=0"
> ## Ouverture du lien
> browseURL(url = paste(url, php, sep = "?"))
```

```
> ## Importation des donnees
> load(file = "datadem3.Rdata")
> head(dat)

## [1]  9.782  7.185  8.259 13.877  9.997 10.249
```

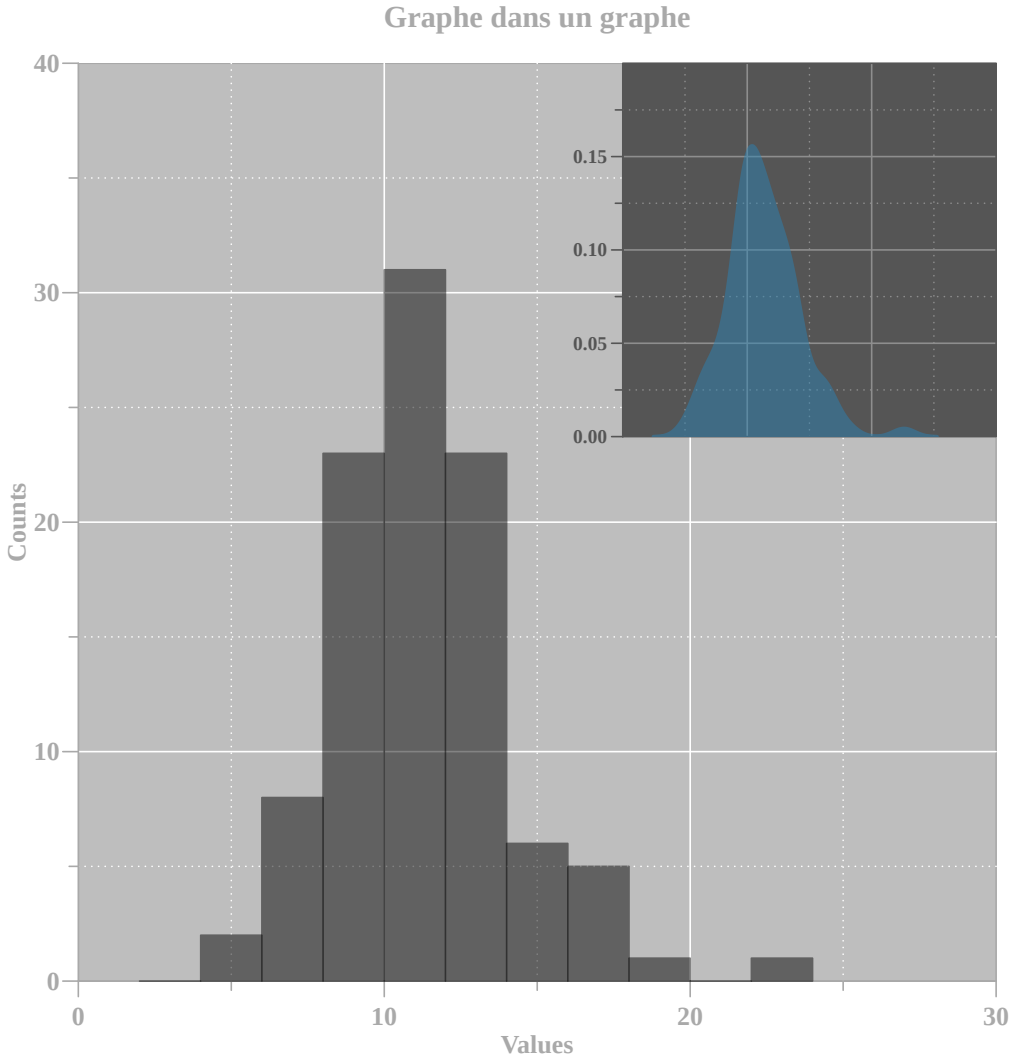

FIGURE 6.3 – Exercice - Inclusion en médaillon

Chapitre 7

Solutions des exercices

Voici la (une) solution ayant permis de réaliser les figures du précédent chapitre.

7.1 Partitionnement avancé

```
> ### CONFIGURATION DU PERIPHERIQUE
>
> ## Partitionnement de la fenetre graphique
> mat <- matrix(c(2, 4, 1, 3), byrow = T, 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 = F, ann = F)
>
> ## 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 = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
> axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## Noms des axes
> mtext("x-axis", side = 1, line = 1.5, 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, 0.2), type = "n", axes = F, ann = F)
>
> ## Background
> rect(0, 0, 40, 0.2, col = "gray", border = par()$col.axis)
> points(x = c(0, 40), y = c(0.1, 0.1), col = "white", type = "l")
> for (i in c(10, 20, 30)) points(x = c(i, i), y = c(0, 0.2), type = "l",
+     col = "white")
> points(c(0, 40), c(0.05, 0.05), type = "l", col = "white", lty = 3)
> points(c(0, 40), c(0.15, 0.15), type = "l", col = "white", lty = 3)
> for (i in c(5, 15, 25, 35)) {
+     points(c(i, i), c(0, 0.2), type = "l", col = "white", lty = 3)
+ }
>
> ## Axe principal
> axis(2, pos = 0, at = seq(0, 0.2, 0.1), col = par()$col.axis, las = 2,
+     labels = format(seq(0, 0.2, 0.1)), font = 2)
>
> ## Axe secondaire
> axis(side = 2, pos = 0, at = seq(0.05, 0.15, 0.1), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = 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, 0.5, 0, 0.5), family = "serif", col.axis = "darkgray")
> plot(c(0, 0.3), c(0, 40), type = "n", axes = F, ann = F)
>
> ## Background
> rect(0, 0, 0.3, 40, col = "gray", border = par()$col.axis)
> points(x = c(0.1, 0.1), y = c(0, 40), col = "white", type = "l")
> points(x = c(0.2, 0.2), y = c(0, 40), col = "white", type = "l")
> for (i in c(10, 20, 30)) points(x = c(0, 0.3), y = c(i, i), type = "l",
+     col = "white")
> for (i in seq(0.05, 0.25, by = 0.1)) points(y = c(0, 40), x = c(i, i),
+     type = "l", col = "white", lty = 3)
> for (i in seq(5, 35, by = 10)) {
+     points(c(0, 0.3), c(i, i), type = "l", col = "white", lty = 3)
+ }
>
> ## Axe principal
> axis(1, pos = 0, at = seq(0, 0.3, 0.1), col = par()$col.axis, las = 1,
+     labels = format(seq(0, 0.3, 0.1)), font = 2)
>
> ## Axe secondaire
> axis(side = 1, pos = 0, at = seq(0.05, 0.25, 0.1), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## Nom de l'axe
> mtext("Density", side = 1, line = 1.5, font = 2, col = par()$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()$col.axis)
>
>
> ### LEGENDE
>
> ## Empty plot
> par(mar = c(0, 0.5, 2, 0.5), family = "serif")
> plot(0, type = "n", ylim = c(0, 4), xlim = c(0, 4), axes = F, ann = F)
>
> ## Background
> rect(0, 0, 4, 4, col = "gray", border = par()$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, 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, 1), col = "#DC29337D", lwd = 2)
```

7.2 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, 0.6, 0))
> plot(c(0, 30), c(0, 40), type = "n", axes = F, ann = F)
>
```

```
> ## 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()$col.axis, las = 2)
>
> ## Axes secondaires
> axis(side = 1, pos = 0, at = seq(5, 25, by = 10), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
> axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## Noms des axes
> mtext("Values", 1, line = 1.5, font = 2, col = par()$col.axis)
> mtext("Counts", 2, line = 1.5, font = 2, col = par()$col.axis, las = 0)
>
> ## Ajout de l'histogramme
> x <- seq(2, 24, by = 2)
> hist(dat, add = T, border = "#0000007D", col = "#0000007D", breaks = x)
>
>
> ### CONCEPTION DE LA FONCTION DE DENSITE
>
> ## Empty plot
> par(mar = c(2.5, 2.5, 3, 3), family = "serif", col.axis = "#2B84B6DD",
+     new = T)
> plot(c(0, 30), c(0, 0.2), type = "n", axes = F, ann = F)
>
> ## Axes principaux
> axis(side = 4, pos = 30, at = seq(0, 0.2, 0.05), labels = format(seq(0,
+     0.2, 0.05)), col = par()$col.axis, font = 2, las = 2)
> axis(4, pos = 30, at = seq(0.025, 0.175, 0.05), labels = F, tck = -0.01,
+     lwd = 0, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## 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")
```

7.3 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, 0.6, 0))
> plot(c(0, 30), c(0, 40), type = "n", axes = F, ann = F)
>
> ## 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()$col.axis, las = 2)
>
> ## Axes secondaires
> axis(side = 1, pos = 0, at = seq(5, 25, by = 10), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
> axis(side = 2, pos = 0, at = seq(5, 35, by = 10), labels = F, lwd = 0,
+     tck = -0.01, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## Noms des axes
> mtext("Values", 1, line = 1.5, font = 2, col = par()$col.axis)
```

```
> mtext("Counts", 2, line = 1.5, font = 2, col = par()$col.axis, las = 0)
>
> ## Ajout de l'histogramme
> x <- seq(2, 24, by = 2)
> hist(dat, add = T, border = "#0000007D", col = "#0000007D", breaks = x)
>
> ## Rajout d'un titre
> title("Graphe dans un graphe", col.main = "darkgray")
>
>
> ### CONCEPTION DU MEDAILLON
>
> ## Empty plot
> par(mar = c(2.5, 2.5, 3, 3), family = "serif", cex.axis = 0.75, new = T,
+     col.axis = "#555555", fig = c(0.5, 1, 0.5, 1), mgp = c(0, 0.5, 0))
> plot(c(0, 30), c(0, 0.2), type = "n", axes = F, ann = F)
>
> ## Background
> rect(0, 0, 30, 0.2, col = par()$col.axis, border = par()$col.axis)
> for (i in seq(0.05, 0.15, 0.05)) {
+     lines(x = c(0, 30), y = c(i, i), col = "#8E8E8E")
+ }
> for (i in seq(10, 20, 10)) {
+     lines(x = c(i, i), y = c(0, 0.2), col = "#8E8E8E")
+ }
> for (i in c(5, 15, 25)) {
+     lines(x = c(i, i), y = c(0, 40), col = "#8E8E8E", lty = 3)
+ }
> for (i in seq(0.025, 0.175, 0.05)) {
+     lines(x = c(0, 30), y = c(i, i), col = "#8E8E8E", lty = 3)
+ }
> box(col = par()$col.axis)
>
> ## Axe principal
> axis(side = 2, pos = 0, at = seq(0, 0.15, 0.05), labels = format(seq(0,
+     0.15, 0.05)), col = par()$col.axis, font = 2, las = 2, tck = -0.03)
>
> ## Axe secondaire
> axis(2, pos = 0, at = seq(0.025, 0.175, 0.05), labels = F, tck = -0.02,
+     lwd = 0, lwd.ticks = 1, col.ticks = par()$col.axis)
>
> ## Ajout de la courbe de densite
> den <- density(dat)
> polygon(den$x, den$y, col = "#2B84B67D", border = "#2B84B67D", lwd = 2)
```