Formation R Perfectionnement

15-16 janvier 2018

Martin Chevalier (Insee)

Objectifs et organisation

Apprendre à perfectionner son utilisation de R : acquérir des points de repères, des réflexes, des méthodes de travail.

Effectuer un panorama structuré et hiérarchisé de méthodes et outils largement utilisés.

Prendre du recul sur le logiciel, comprendre certains modes de fonctionnement complexes.

Objectifs et organisation

1. Travailler sur des données dans R :
   - travailler efficacement (+++) ;
   - améliorer les performances (+++) ;
   - programmer avec R (++) ;
   - interroger des bases de données (+).
2. Présenter des résultats avec R :
   - faire des graphiques avec base R et `ggplot2` (++) ;
   - faire du reporting (+).

Pédagogie : équilibre entre présentations et cas pratiques.

Horaires (proposition !) : 9h30-12h20, 13h40-16h30

Objectifs et organisation

Lundi 15 janvier

- Introduction + Travailler sur des données 1 (2h)
- Cas pratiques (2h)
- Faire des graphiques avec `ggplot2` (2h)

Mardi 16 janvier

- Travailler sur des données 2 + R Markdown (2h)
- Cas pratiques (au choix, 4h)

Introduction :
Se perfectionner avec R

Connaître plus ou connaître mieux ?

Comme tout langage statistique ou de programmation, R repose sur un ensemble d'instructions plus ou moins complexes.

Se perfectionner dans la maîtrise de R peut donc signifier deux choses :

- étendre son « vocabulaire » d'instructions connues ;
- mieux comprendre les instructions déjà connues.

En pratique, les deux **vont de pair** : en découvrant de nouvelles fonctions, on est souvent amené à mieux comprendre le fonctionnement de celles que l'on croyait maîtriser.

Plan de la partie

Chercher (et trouver !) de l'aide

Découvrir de nouvelles fonctionnalités

Utiliser de nouveaux *packages*

Savoir utiliser l'aide du logiciel

À tout moment, taper `help(nomFonction)` ou `? nomFonction` affiche l'aide de la fonction `nomFonction`.

```
# Aide de la fonction read.csv
? read.csv
```

Remarque Pour afficher l'aide sur une fonction d'un *package*, il faut que celui-ci soit au préalable chargé (avec `library()` ou `require()`).

La fonction `help.search()` ou la commande `??` permettent d'effectuer une recherche approximative :

```
# Recherche à partir du mot-clé csv
?? csv
```

Chercher de l'aide en ligne

Bien souvent, le problème que l'on rencontre a **déjà été rencontré par d'autres**.

Pour progresser dans la maîtrise de R, il ne faut donc surtout pas hésiter à s'appuyer sur les forums de discussion, comme par exemple stackoverflow.

On gagne ainsi souvent beaucoup de temps en formulant le problème que l'on rencontre dans un **moteur de recherche** pour consulter certaines réponses.

Quand une question semble ne pas avoir été déjà posée, ne pas hésiter à la poser soi-même, en joignant alors un **exemple reproductible** (*minimal working example* ou MWE).

Afficher le code d'une fonction

Quand l'utilisation d'une fonction pose problème (message d'erreur inattendu), il est souvent utile d'**afficher son code** pour comprendre d'où vient le problème.

Pour ce faire, il suffit de saisir son nom sans parenthèse.

```
# Code de la fonction read.csv
read.csv
  ## function (file, header = TRUE, sep = ",", quote = "\"", dec
  ##     fill = TRUE, comment.char = "", ...)
  ## read.table(file = file, header = header, sep = sep, quote =
  ##     dec = dec, fill = fill, comment.char = comment.char, ..
  ## <bytecode: 0x873b900>
  ## <environment: namespace:utils>
```

Afficher le code d'une fonction est dans certains cas plus difficile, *cf.* stackoverflow.

Se repérer dans les CRAN *Task Views*

Les CRAN *Task Views* recensent les fonctions et *packages* de façon thématique. Elles sont mises à jour régulièrement et portent sur des thèmes variés :

Bayesian, ChemPhys, ClinicalTrials, Cluster, DifferentialEquations, Distributions, Econometrics, Environmetrics, ExperimentalDesign, ExtremeValue, Finance, FunctionalData, Genetics, Graphics, HighPerformanceComputing, MachineLearning, MedicalImaging, MetaAnalysis, Multivariate, NaturalLanguageProcessing, NumericalMathematics, OfficialStatistics, Optimization, Pharmacokinetics, Phylogenetics, Psychometrics, ReproducibleResearch, Robust, SocialSciences, Spatial, SpatioTemporal, Survival, TimeSeries, WebTechnologies, gR

La liste de toutes les *Task Views* est accessible à la page : https://cran.r-project.org/web/views.

Consulter des sites, tutoriels, livres

De plus en plus de supports sont consacrés à la présentation et à l'enseignement des fonctionnalités de R, comme par exemple :

- le site R-bloggers : articles en général courts sur des exemples d'applications (de qualité inégale) ;
- le site bookdown.org : dépôt de livres numériques consacrés à R élaborés avec R Markdown (très riches et très complets) ;
- le site de RStudio : nombreux aides-mémoires ou articles présentant les fonctionnalités de l'écosystème RStudio ;
- les ouvrages de Hadley Wickham : ggplot2: elegant graphics for data analysis (à compiler soi-même), Advanced R.

Accéder à la documentation d'un *package*

Une des principales forces de R est d'être un langage hautement modulaire comptant **plusieurs milliers de *packages*** (12 094 au 26/01/2018).

Toutes les informations sur un *package* sont accessibles sur sa page du *Comprehensive R Archive Network* (CRAN).

Exemple https://CRAN.R-project.org/package=haven

On trouve en particulier sur cette page :

- les **dépendances** du *package* (*Depends* et *Imports*) ;
- un lien vers sa **page de développement** (*URL*) ;
- une **version .pdf de son aide** (*Reference manual*)
- éventuellement un ou plusieurs **documents de démonstration** (*Vignettes*).

Installer un *package* automatiquement

La fonction `install.packages("nomPackage")` permet d'installer automatiquement le *package* `nomPackage`.

Les données nécessaires sont téléchargées depuis un des dépôts du CRAN (*repositories* ou en abrégé `repos`).

C'est la **méthode à privilégier** : les dépendances nécessaires au bon fonctionnement du *package* sont détectées et automatiquement installées.

Remarque Cette méthode fonctionne à l'Insee :

- pour les installations locales de R sur les postes de travail ;
- sur AUS, *via* un dépôt local spécifique ;
- mais PAS sur les sessions des postes de formation.

Installer un *package* manuellement

La page d'information d'un *package* comporte également des liens vers les fichiers qui le composent.

Quand l'installation directe depuis un dépôt du CRAN est indisponible, il suffit de **télécharger ces fichiers** et d'**installer manuellement le *package***.

Pour une installation sous Windows, il faut privilégier les **fichiers compilés** (*Windows binaries*).

```
# Note : Le fichier haven._1.1.0.zip est situé
# dans le répertoire de travail
install.packages(
  "haven_1.1.0.zip", repos = NULL, type = "binaries"
)
```

Installer des *packages* depuis github

En règle générale, le développement de *packages* s'appuie sur des plate-formes de **développement collaboratif** comme Github.

La **page de développement** d'un *package* comporte plusieurs informations préciseuses :

- la dernière version du *package* et de sa documentation ;
- des informations sur son développement ;
- une zone pour rapporter d'éventuels *bugs* (*bug reports*).

Exemple https://github.com/tidyverse/haven

La fonction `install_github()` du *package* `devtools` permet d'installer un *package* directement depuis GitHub.

```
library(devtools)
install_github("tidyverse/haven")
```

Utiliser les données d'exemples d'un *package*

La plupart des **packages** contiennent des **données d'exemples** utilisées notamment dans son aide ou ses vignettes.

Une fois le *package* installé, il suffit d'utiliser la fonction `data(package = "nomPackage")` pour afficher les données qu'il contient.

```
library(ggplot2)
data(package = "ggplot2")
```

Pour « rapatrier » dans l'environnement global les données d'un *package*, c'est de nouveau la fonction `data()` qu'il faut utiliser.

```
data(mpg)
```

Travailler efficacement sur des données avec R

Qu'est-ce que travailler efficacement avec R ?

Appliqué au travail sur des données, l'efficacité peut avoir au moins deux significations distinctes :

- efficacité **algorithmique** : minimisation du temps passé par la machine pour réaliser une série d'opérations ;
- **productivité** du programmeur : minimisation du temps passé à coder une série d'opération.

En règle générale, on peut avoir l'idée que plus on souhaite être efficace algorithmiquement, plus la programmation risque d'être longue et difficile.

Ce n'est pas toujours vrai : on perd souvent beaucoup de temps à (ré)inventer une méthode peu efficace quand une beaucoup plus simple et rapide existe déjà.

Référence Gillepsie C., Lovelace R., *Efficient R programming* (disponible sur bookdown.org)

Mesure l'efficacité algorithmique

La fonction `system.time()` permet de mesurer la durée d'un traitement.

```
system.time(rnorm(1e6))
  ## utilisateur     système      écoulé
  ##        0.07        0.00        0.07
```

Néanmoins, elle est inadaptée aux traitements de très courte durée. Dans ces situations, privilégier la fonction `microbenchmark()` du package `microbenchmark`.

```
library(microbenchmark)
microbenchmark(times = 10, rnorm(1e6))
  ## Unit: milliseconds
  ##          expr      min       lq     mean   median
  ##  rnorm(1e+06) 70.50072 75.59383 83.88297 85.28535
  ##        uq      max neval
  ##  90.59321 95.62014    10
```

Mesurer la taille d'un objet en mémoire

R stocke l'ensemble des fichiers sur lesquels il travaille dans la mémoire vive.

Afin de loger les objets les plus gros mais aussi d'optimiser les performances, il est souvent utile de **limiter la taille des objets** sur lesquels portent les traitements.

Pour mesurer la taille des objets, utiliser la fonction `object_size()` du *package* `pryr`.

```
library(pryr)
object_size(rnorm(1e6))
  ## 8 MB
```

Construire un exemple reproductible (MWE)

Lorsque l'on cherche à améliorer les performances d'un programme, il est important de pouvoir le tester sur des données **autonomes et reproductibles**.

Pour ce faire, les **fonctions de générations de nombres aléatoires** de R sont particulièrement utiles.

```
# Graine pour pouvoir reproduire l'aléa
set.seed(2018)

# Vecteur de nombres de taille 1 000
a <- rnorm(1000)

# Vecteur de lettres de taille 1 000
b <- letters[sample(1:26, 1000, replace = TRUE)]

# Matrice logique 1 000 x 100 avec 1 % de TRUE
c <- matrix(runif(100000) > 0.99, ncol = 100)
```

Plan de la partie

De l'importance des fonctions dans R

Vectoriser : `*apply()`, `Reduce()` et `do.call()`

Coder efficacement en base R

dplyr : une grammaire du traitement des données

data.table : un data.frame optimisé

Aller plus loin avec R

« Tout ce qui agit est un appel de fonction »

> *To understand computations in R, two slogans are helpful :*
>
> - *Everything that exists is an object.*
> - *Everything that happens is a function call.*
>
> *John Chambers*

```
# ... même assigner une valeur
is.function(`<-`)
  ## [1] TRUE
`<-`(a, 10)

# ... même afficher la valeur d'un objet
a
  ## [1] 10
print(a)
  ## [1] 10
```

Définir une fonction dans R

Utilisé avec <-, `function()` définit une nouvelle fonction :

```
# Définition de la fonction monCalcul()
monCalcul <- function(a, b){
  resultat <- 10 * a + b
  return(resultat)
}

# Code de monCalcul()
monCalcul
  ## function(a, b){
  ##   resultat <- 10 * a + b
  ##   return(resultat)
  ## }

# Appel de la fonction monCalcul()
monCalcul(2, 3)
  ## [1] 23
```

Valeurs par défaut des paramètres

Des valeurs par défaut peuvent être renseignées pour les paramètres.

```
monCalcul <- function(a, b = 3) 10 * a + b
monCalcul(8)
  ## [1] 83
```

Les valeurs par défaut peuvent dépendre des autres paramètres.

```
monCalcul <- function(a, b = a * 2) 10 * a + b
monCalcul(2)
  ## [1] 24
```

Remarque Ceci est la conséquence de la *lazy evaluation* des arguments dans R (*cf.* *Advanced R*).

Contrôle de la valeur des paramètres

Des structures conditionnelles `if()` permettent de contrôler la valeur des arguments.

```
monCalcul <- function(a = NULL, b = NULL){
  if(is.null(a)) stop("a n'est pas renseigné.")
  if(is.null(b)){
    b <- a * 2
    warning("b n'est pas renseigné.")
  }
  return(10 * a + b)
}

monCalcul(b = 3)
  ## Error in monCalcul(b = 3): a n'est pas renseigné.
monCalcul(a = 1)
  ## Warning in monCalcul(a = 1): b n'est pas renseigné.
  ## [1] 12
```

Portée des variables et environnements (1)

Dans R **chaque objet est repéré par son nom et son environnement** : cela permet d'éviter les conflits de noms.

```
# Création d'une fonction sum() un peu absurde
sum <- function(...) "Ma super somme !"
sum(2, 3)
  ## [1] "Ma super somme !"

# Cette fonction est rattachée à l'environnement global
ls()
  ## [1] "a"          "b"          "c"          "monCalcul"
  ## [5] "sum"

# Mais on peut toujours accéder à la fonction
# de base en utilisant ::
base::sum(2, 3)
  ## [1] 5
```

Portée des variables et environnements (2)

À chaque appel d'une fonction, un **environnement d'exécution** est créé.

```
maFun <- function() environment()
maFun()
  ## <environment: 0xae9f8a8>
maFun()
  ## <environment: 0xae552b0>
```

En conséquence, les instructions exécutées à l'intérieur d'une fonction **ne modifient pas l'environnement global**.

```
a <- 10
maFonction3 <- function(){
  a <- 5
}
maFonction3()
a
  ## [1] 10
```

Portée des variables et environnements (3)

En revanche, les objets définis dans l'environnement global sont accessibles au sein d'une fonction.

```
a <- 10
maFonction4 <- function(){
  a + 5
}
maFonction4()
  ## [1] 15
```

Ceci est dû au fait que les environnements dans lequel R recherche des objets sont **emboîtés les uns dans les autres** (*cf.* la fonction `search()`).

Pour en savoir plus *Advanced R*, obeautifulcode.com

Valeur de retour d'une fonction

La fonction `return()` spécifie la valeur à renvoyer. Pour renvoyer plusieurs valeurs, utiliser une liste.

```
maFonction1 <- function(){
  a <- 1:5; b <- 6:10; return(a)
}
maFonction1()
  ## [1] 1 2 3 4 5

maFonction2 <- function(){
  a <- 1:5; b <- 6:10; return(list(a = a, b = b))
}
maFonction2()
  ## $a
  ## [1] 1 2 3 4 5
  ##
  ## $b
  ## [1]  6  7  8  9 10
```

Effets de bord et programmation fonctionnelle

Par défaut, les fonctions dans R :

- ne modifient pas l'environnement d'origine (il n'y a **pas d'effets de bord**) ;
- peuvent être utilisées en lieu et place des valeurs qu'elles retournent.

```
monCalcul <-  function(a, b) 10 * a + b
monCalcul(2, 3) + 5
  ## [1] 28
```

Ces éléments font de R un **langage particulièrement adapté à la programmation fonctionnelle**.

Quelques principes de la programmation fonctionnelle

1. **Ne jamais créer d'effets de bord** Toute modification apportée à l'environnement par une fonction passe par sa valeur de sortie.

2. **Vectoriser *i.e.* appliquer des fonctions systématiquement à un ensemble d'éléments** Fonctions `*apply()`, `Reduce()`, `do.call()`.

3. **Structurer les traitements à l'aide de fonctions courtes et explicites** Faciliter la relecture, la maintenance et la modularisation.

Pour en savoir plus Wikipedia, maryrosecook.com.

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur chaque indépendamment : apply()

La fonction apply(X, MARGIN, FUN) applique la fonction FUN à la **matrice** X selon la dimension MARGIN.

```
# Définition et affichage de la matrice m
m <- matrix(1:6, ncol = 3)
m
  ##      [,1] [,2] [,3]
  ## [1,]    1    3    5
  ## [2,]    2    4    6

# Application de la fonction sum() selon les lignes
apply(m, 1, sum)
  ## [1]  9 12

# Application de la fonction sum() selon les colonnes
apply(m, 2, sum)
  ## [1]  3  7 11
```

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur chaque indépendamment : lapply()

La fonction `lapply(X, FUN)` applique la fonction `FUN` au **vecteur** ou à la **liste** `X`.

```
l <- list(1:5, c(6:9, NA))
l
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1]  6  7  8  9 NA
lapply(l, sum)
  ## [[1]]
  ## [1] 15
  ##
  ## [[2]]
  ## [1] NA
```

Exemple d'utilisation Appliquer une fonction à toutes les variables d'une table.

Appliquer sur chaque indépendamment : sapply()

La fonction sapply() est analogue à la fonction lapply(), mais simplifie le résultat produit quand c'est possible.

```
sapply(l, sum)
  ## [1] 15 NA
```

Les arguments optionnels de la fonction utilisée peuvent être ajoutés à la suite dans toutes les fonctions *apply().

```
sapply(l, sum, na.rm = TRUE)
  ## [1] 15 30
```

Exemple d'utilisation Calcul de statistiques sur toutes les variables d'une table.

Définir une fonction à la volée dans *apply()

Il est fréquent que l'opération que l'on souhaite appliquer ne corresponde pas exactement à une fonction pré-existante.

Dans ce cas, on peut définir une **fonction à la volée** dans la fonction *apply().

```
# On souhaite sélectionner le second élément de
# de chaque vecteur de la liste l
l
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1]  6  7  8  9 NA

# On définit une fonction dans sapply()
sapply(l, function(x) x[2])
  ## [1] 2 7
```

Appliquer sur chaque par groupe : tapply()

La fonction `tapply(X, INDEX, FUN)` applique la fonction `FUN`, à l'objet `X` ventilé selon les modalités de `INDEX`.

```
# Variables d'âge et de sexe
age <- c(45, 50, 35, 20)
sexe <- c("H", "F", "F", "H")

# Âge moyen par sexe
tapply(age, sexe, mean)
  ##    F    H
  ## 42.5 32.5

# Même résultat avec une combinaison de sapply() et de split()
sapply(split(age, sexe), mean)
  ##    F    H
  ## 42.5 32.5
```

Exemple d'utilisation Calcul de statistiques agrégées par catégories.

Appliquer sur tous : do.call()

La fonction `do.call(what, args)` permet d'appliquer la fonction `what()` à un **ensemble** d'arguments `args` spécifié comme une liste (alors que les fonctions `*apply()` appliqueraient `what()` à **chaque** élément de `args`).

```
# Concaténation des vecteurs de l
do.call(base::c, l)
  ##  [1]  1  2  3  4  5  6  7  8  9 NA

# Equivalent à
base::c(l[[1]], l[[2]])
  ##  [1]  1  2  3  4  5  6  7  8  9 NA
```

Exemple d'utilisation Concaténer de nombreuses tables avec `rbind()` ou `cbind()`.

Vectoriser : *apply(), Reduce() et do.call()

Appliquer sur tous successivement : Reduce()

La fonction `Reduce(f, x)` permet d'appliquer la fonction `f()` **successivement** à l'ensemble des éléments de `x` (alors que `do.call()` applique `f` **simultanément**).

```
# Application successive de la division au vecteur 1:4
Reduce(`/`, 1:4)
  ## [1] 0.04166667

# Equivalent à
((1/2)/3)/4
  ## [1] 0.04166667
```

Exemple d'utilisation Fusionner de nombreuses tables avec `merge()` (sur les mêmes identifiants).

L'idée : En faire faire le moins possible à R

R est un langage dit « de haut niveau » : les objets qui le composent sont relativement faciles d'utilisation, au prix de performances limitées.

À l'inverse, des langages dits de « bas niveau » (par exemple C++) sont plus difficiles à utiliser mais aussi plus efficaces.

La plupart des fonctions fondamentales de R font appel à des fonctions compilées à partir d'un langage de plus bas niveau.

D'où le principe : **limiter au maximum la surcharge liée à R** pour retomber au plus vite sur des fonctions pré-compilées.

Remarque Il est très facile en pratique d'utiliser R comme une interface vers des langages de plus bas niveau, *cf. infra* à propos de `Rcpp`.

Utiliser les boucles avec parcimonie (1)

Comme la plupart des langages de programmation, R dispose de **structures de contrôles** permettant de réaliser des boucles.

```
boucle <- function(x){
  cumul <- rep(NA, length(x))
  for(i in seq_along(x))
    cumul[i] <- if(i == 1) x[i] else cumul[i - 1] + x[i]
  return(cumul)
}
boucle(1:5)
  ## [1]  1  3  6 10 15
```

Ces opérations présentent plusieurs inconvénients :

1. Elles sont longues à écrire et assez peu claires ;
2. Elles reposent sur des effets de bord ;
3. Elles sont en général très peu **efficaces algorithmiquement**.

Utiliser les boucles avec parcimonie (2)

Les méthodes de vectorisation sont en général beaucoup plus efficaces que les boucles en R :

- vectorisation de haut niveau (*cf. supra*) ;
- vectorisation de bas niveau : la vectorisation est opérée par le langage de bas niveau auquel fait appel R.

```
summary(microbenchmark(times = 10L
  , boucle = boucle(1:1e4)
  , Reduce = Reduce(`+`, 1:1e4, accumulate = TRUE)
  , cumsum = cumsum(1:1e4)
))[, 1:4]
##      expr        min         lq       mean
## 1 boucle 18686.838 20569.758 21284.5156
## 2 Reduce  6402.722  6963.981  8102.1192
## 3 cumsum    37.218    38.982    46.4275
```

Tirer le meilleur parti de la compilation (1)

On distingue souvent deux familles de langages informatiques :

- les langages **compilés** (C, C++) : l'ensemble du code est transformé en langage machine par un *compilateur* puis soumis par le système d'exploitation ;
- les langages **interprétés** (R, Python) : les instructions du code sont soumises les unes après les autres par un *interpréteur*, ce qui est moins efficace (*cf.* boucles en R).

La fonction `compiler::cmpfun()` permet néanmoins de **compiler** des fonctions R avant utilisation.

```
# Compilation de la fonction boucle()
boucle_compil <- compiler::cmpfun(boucle)
microbenchmark(boucle(1:1e4), boucle_compil(1:1e4))
  ## Unit: milliseconds
  ##                      expr       min        lq      mean
  ##          boucle(1:10000) 18.256964 22.946919 28.623216
  ##  boucle_compil(1:10000)  1.673866  1.907869  2.841323
```

Tirer le meilleur parti de la compilation (2)

Une autre fonctionnalité du *package* `compiler` est la compilation « juste-à-temps » (ou *just-in-time*, JIT) : le code n'est plus interprété mais **compilé au fur et à mesure**.

Dans R, on active le mode JIT pour une session grâce à la fonction `compiler::enableJIT()` en spécifiant le niveau de compilation JIT (de 0 à 3).

```
# Passage au niveau maximal de compilation JIT
compiler::enableJIT(3)
  ## [1] 0
summary(microbenchmark(boucle(1:1e4), boucle_compil(1:1e4)))[, 1
  ##                       expr      min       lq     mean
  ## 1        boucle(1:10000) 1.683376 1.806746 2.656830
  ## 2 boucle_compil(1:10000) 1.678782 1.907039 2.755148
```

Remarque Depuis R 3.4.0, `enableJIT()` vaut 3 par défaut.

Utiliser l'opérateur [au lieu de ifelse()

Lorsqu'on crée une variable en faisant intervenir une condition, il est fréquent d'utiliser la fonction `ifelse()` :

```
notes <- runif(n = 100000, min = 0, max = 20)
mavar <- ifelse(notes >= 10, "Reçu", "Recalé")
```

Il est néanmoins beaucoup plus efficace d'utiliser l'opérateur `[`.

```
microbenchmark(times = 10L
  , ifelse = ifelse(notes >= 10, "Reçu", "Recalé")
  , "[" = {
    mavar <- rep("Recalé", length(notes))
    mavar[notes >= 10] <- "Reçu"
  }
)
  ## Unit: milliseconds
  ##    expr       min        lq      mean    median
  ##  ifelse 36.374660 46.54510 70.446984 51.274090
  ##       [  1.636993  1.95378  2.749823  2.353574
```

Simplifier les données : le type `factor`

On utilise souvent des chaînes de caractère pour coder une variable de nature catégorielle.

Le type `factor` permet de remplacer chaque valeur distincte par un entier en sauvegardant la table de correspondance. Il est **beaucoup plus léger**.

```
# Variable à deux modalités codées en caractères
sexe <- sample(c("H", "F"), 120000, replace = TRUE)
object_size(sexe)
  ## 960 kB

# Conversion en facteur
f.sexe <- factor(sexe)
str(f.sexe)
  ##  Factor w/ 2 levels "F","H": 1 2 1 2 1 2 2 2 1 1 ...
object_size(f.sexe)
  ## 481 kB
```

Utiliser les noms à bon escient (1)

La plupart des objets manipulés couramment dans R peuvent être **nommés** : vecteurs, matrices, listes, `data.frame`.

Utiliser des noms est une méthode souvent **très rapide** pour **accéder aux éléments** qui composent ces objets.

Exemple On cherche à extraire les observations d'une table *via* leur identifiant `id`. On compare l'utilisation des noms à une fusion réalisée avec `merge()`.

```
# Création de la table df
id <- as.character(sample(1e5))
sexe <- sample(1:2, 1e5, replace = TRUE)
df <- data.frame(id, sexe)
```

Utiliser les noms à bon escient (2)

```r
# Affectation de noms à df
row.names(df) <- id

# Liste des identifiants à extraire
extract <- c("234", "12", "7890")

# Comparaison
microbenchmark(times = 10L
  , merge = merge(data.frame(id = extract), df, sort = FALSE)
  , names = df[extract, ]
)
  ## Unit: milliseconds
  ##   expr       min        lq      mean    median
  ##  merge 16.127307 17.827670 20.804243 20.165223
  ##  names  3.395627  3.473018  3.678273  3.575197
  ##         uq       max neval
  ##  24.251721 26.346721    10
  ##   3.658939  4.807671    10
```

À propos des matrices (1)

Quand c'est possible, **travailler sur des matrices** (plutôt que des `data.frame`) est souvent source d'efficacité :

- de nombreuses opérations sont **vectorisées** pour les matrices : sommes en lignes et en colonnes (`rowSums()` et `colSums()`), etc. ;
- l'**algèbre matricielle** (le produit matriciel notamment) est très bien optimisée ;
- selon la nature du problème, l'utilisation de **matrices lacunaires** (*sparse*) peut faire gagner et en empreinte mémoire et en temps de calcul (*cf.* le *package* `Matrix`).

À propos des matrices (2)

```
# Création d'une matrice m avec 99 % de 0
v <- rep(0, 1e6); v[sample(1e6, 1e4)] <- rnorm(1e4)
m <- matrix(v, ncol = 100)

# Transformation en matrice lacunaire
library(Matrix)
M <- Matrix(m)

# Gain en espace (en ko)
c(object_size(m), object_size(M))
  ## [1] 8000200  121824

# Gain de performances pour la fonction colSums()
microbenchmark(dense = colSums(m), sparse = colSums(M))
  ## Unit: microseconds
  ##    expr      min        lq       mean   median
  ##   dense 1255.196 1282.6595 1360.35790 1335.667
  ##  sparse   57.559   74.1685   89.06236   89.084
  ##              uq      max neval
```

Philosophie de dplyr

`dplyr` est un *package* développé par RStudio et en particulier par Hadley Wickham. Il constitue un véritable **écosystème** visant à faciliter le travail sur des tables statistiques :

- il fournit un ensemble de **fonctions élémentaires** (les « verbes ») pour effectuer les manipulations de données ;
- plusieurs verbes peuvent facilement être **combinés en utilisant l'opérateur %>%** (*pipe*) ;
- toutes les opérations sont optimisées par du **code de bas niveau**.

```
library(dplyr)
```

Pour en savoir plus De nombreuses vignettes très pédagogiques sont disponibles sur la page du *package*. Un aide-mémoire est également disponible sur le site de RStudio.

Données d'exemple : table `flights` de `nycflights13`

Les exemples relatifs aux *packages* `dplyr` et `data.table` s'appuient sur les données du *package* `nycflights13`.

```
library(nycflights13)
```

Ce *package* contient des données sur tous les vols au départ de la ville de New-York en 2013.

```
data(package = "nycflights13")
dim(flights)
  ## [1] 336776     19
names(flights)[1:9]
  ## [1] "year"           "month"          "day"
  ## [4] "dep_time"       "sched_dep_time" "dep_delay"
  ## [7] "arr_time"       "sched_arr_time" "arr_delay"
```

Simplifier des opérations de base R

`dplyr` propose plusieurs verbes pour simplifier certaines opérations parfois fastidieuses en base R :

- `filter()` sélectionne des observations selon une ou plusieurs conditions ;

```
filter(flights, month == 7, day == 4)
```

- `arrange()` trie le fichier selon une ou plusieurs variables ;

```
arrange(flights, month, desc(distance))
```

- `select()` sélectionne des variables par leur noms ;

```
select(flights, year:arr_delay)
```

- `rename()` renomme des variables.

```
rename(flights, annee = year)
```

Calculer des statistiques avec `summarise()`

La fonction `summarise()` permet de facilement calculer des statistiques sur des données.

```
summarise(flights
  , distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
)
```

```
##   distance_moyenne retard_max
## 1         1039.913       1272
```

Remarque Comme toutes les fonctions de `dplyr`, `summarise()` prend un `data.frame` en entrée et produit un `data.frame` en sortie.

Ventiler des traitements avec group_by()

Appliqué au préalable à un data.frame, group_by() ventile tous les traitements ultérieurs selon les modalités d'une ou plusieurs variables.

```
flights_bymonth <- group_by(flights, month)
summarise(flights_bymonth
  , distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
)[1:3, ]
```

```
##   month distance_moyenne retard_max
## 1     1         1006.844        1272
## 2     2         1000.982         834
## 3     3         1011.987         915
```

Enchaîner des opérations avec %>%

L'utilisation des verbes de `dplyr` ne prend tout son intérêt que quand ils sont enchaînés en utilisant l'opérateur *pipe* %>%.

`maTable %>% maFonction(param1, param2)` est équivalent à `maFonction(maTable, param1, param2)`.

Ainsi, l'**enchaînement de nombreuses opérations** devient beaucoup plus facile à mettre en œuvre et à comprendre.

```
flights %>%
  group_by(year, month, day) %>%
  summarise(
    retard_arrivee = mean(arr_delay, na.rm = TRUE),
    retard_depart = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(retard_arrivee > 30 | retard_depart > 30)
```

Fusionner des tables avec `*_join()`

`dplyr` dispose de nombreuses fonctions très utiles pour fusionner une ou plusieurs tables ensemble, qui **s'inspirent très fortement de SQL** :

- `a %>% left_join(b, by = "id")` : fusionne `a` et `b` en conservant toutes les observations de `a` ;
- `a %>% right_join(b, by = "id")` : fusionne `a` et `b` en conservant toutes les observations de `b` ;
- `a %>% inner_join(b, by = "id")` : fusionne `a` et `b` en ne conservant que les observations dans `a` et `b` ;
- `a %>% full_join(b, by = "id")` : fusionne `a` et `b` en conservant toutes les observations.

Pour en savoir plus Une vignette est consacrée à la présentation des fonctions de `dplyr` portant sur deux tables.

Comparaison de base R et de dplyr

dplyr est particulièrement intéressant pour travailler sur des données par groupe. On compare donc l'utilisation de tapply() de base R avec group_by() de dplyr.

```
df <- data.frame(
  x = rnorm(1e6)
  , by = sample(1e3, 1e6, replace = TRUE)
)

microbenchmark(times = 10L
  , base = tapply(df$x, df$by, sum)
  , dplyr = df %>% group_by(by) %>% summarise(sum(x))
)
  ## Unit: milliseconds
  ##   expr      min       lq     mean   median       uq
  ##   base 37.92114 46.84232 56.65085 50.55141 70.85616
  ##  dplyr 48.34361 50.91462 64.66221 55.71249 73.45178
  ##        max neval
  ##   85.66136    10
```

Philosophie de data.table

Contrairement à dplyr, data.table ne cherche pas à se substituer à base R mais à le compléter.

Il introduit un nouveau type d'objet, le data.table, qui **hérite** du data.frame (tout data.table est un data.frame).

Appliqué à un data.table, l'opérateur [est **enrichi et optimisé**.

```
library(data.table)
flights_DT <- data.table(flights)
```

Pour en savoir plus Là encore des vignettes très pédagogiques sont disponibles sur la page du *package*.

L'opérateur [du data.table : i, j et by

La syntaxe de l'opérateur [appliqué à un data.table est la suivante (DT représente le data.table) :

```
DT[i, j, by]
```

- i : sélectionner des observations selon une condition ;
- j : sélectionner ou **créer** une ou plusieurs variables ;
- by : ventiler les traitements selon les modalités d'une ou plusieurs variables.

Exemple Retard quotidien maximal au mois de janvier.

```
flights_DT[
  month == 1, max(arr_delay, na.rm = TRUE), by = day
]
```

Sélectionner des observations avec `i`

Il est beaucoup plus simple et efficace de sélectionner des observations dans un `data.table` que dans un `data.frame` :

- il n'y a pas à répéter le nom du `data.frame` dans `[` ;
- il est possible d'indexer un `data.table` par une ou plusieurs « clés » permettant une recherche souvent plus rapide.

```
setkey(flights_DT, origin)
microbenchmark(times = 100L
  , base = flights[flights$origin == "JFK",]
  , dt1 = flights_DT[origin == "JFK"]
  , dt2 = flights_DT[list("JFK")]
)
  ## Unit: milliseconds
  ##  expr      min       lq     mean   median       uq
  ##  base 41.98297 47.92040 55.90588 50.78441 58.67894
  ##   dt1 11.15860 11.49826 17.72419 12.97871 19.45241
  ##   dt2 10.57523 11.04820 15.11194 12.74932 17.22756
```

Calculer des statistiques avec j

L'argument j permet de calculer des statistiques agrégées.

```
flights_DT[, j = list(
  distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
)]
  ##    distance_moyenne retard_max
  ## 1:          1039.913       1272
```

Utilisé avec := il permet de les refusionner automatiquement avec les données d'origine.

```
flights_DT <- flights_DT[, j := list(
  distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
)]
```

Ventiler des traitements avec by et keyby

L'argument by de [ventile tous les traitements renseignés dans j selon les modalités d'une ou plusieurs variables.

```
flights_DT[, j = list(
  distance_moyenne = mean(distance)
  , retard_max = max(arr_delay, na.rm = TRUE)
), by = month][1:3,]
  ##    month distance_moyenne retard_max
  ## 1:     1          1006.844       1272
  ## 2:    10          1038.876        688
  ## 3:    11          1050.305        796
```

Remarque Par défaut, by ordonne les résultats dans l'ordre des groupes dans le data.table. keyby trie les données selon la variable d'agrégation (comme group_by de dplyr).

Chaîner les opérations dans un data.table

Il est très facile de chaîner les opérations sur un data.table en enchaînant les [.

```
flights_DT[
   , j = list(
     retard_arrivee = mean(arr_delay, na.rm = TRUE)
     , retard_depart = mean(dep_delay, na.rm = TRUE)
   )
   , keyby = list(year, month, day)
][retard_arrivee > 30 | retard_depart > 30]
```

Remarque Ces chaînages sont possibles avec un data.table mais pas avec un data.frame.

Comparaison de base R, dplyr et data.table

```
# Conversion de la table de test en data.table
dt <- data.table(df)

microbenchmark(times = 10L
   , base = tapply(df$x, df$by, sum)
   , dplyr = df %>% group_by(by) %>% summarise(sum(x))
   , data.table = dt[, sum(x), keyby = by]
)
```

```
##          expr       lq     mean       uq
## 1        base 38.32042 47.52636 48.74153
## 2       dplyr 49.53867 62.07801 74.74132
## 3 data.table 21.10773 38.63131 25.69521
```

Pour en savoir plus Cette discussion sur stackoverflow.com (notamment entre les auteurs des *packages*) aborde les avantages et les inconvénients de `dplyr` et `data.table`.

Les limites du logiciel

Les outils présentés jusqu'à présent correspondent à une utilisation « classique » de R : production d'une enquête, redressements, études.

Il arrive néanmoins que certains traitements soient rendus **difficiles par les caractéristiques du logiciel** :

- travail sur des volumes de données impossibles à loger en mémoire ;
- temps de calcul trop longs et impossibles à réduire.

Dans ce genre de situations, la solution consiste en général à utiliser R comme une **interface** vers des techniques ou langages susceptibles de répondre au problème posé.

Se connecter à des bases de données

Une autre solution pour exploiter de grands volumes de données dans R est de l'utiliser pour **interroger des bases de données**, *via* par exemple le *package* `RPostgreSQL`.

```
library(RPostgreSQL)

# Connexion à la base de données maBdd
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "maBdd"
  , host = "localhost", port = 5432
  , user = "utilisateur", password = "motDePasse"
)

# Requête SQL sur la table maTable
dbGetQuery(con, "SELECT COUNT(*) FROM maTable")
```

Remarque Différents *packages* permettent de se connecter à différents types de base de données : `RMySQl` pour MySQL, etc.

Se connecter à des bases de données avec dplyr

dplyr a la particularité de pouvoir fonctionner de façon totalement transparente sur des bases de données de différents types.

```
library(dplyr)

# Connexion à la base de données maBdd
con <- src_postgres(
  dbname = "maBdd", host = "localhost", port = 5432
  , user = "utilisateur", password = "motDePasse"
)

# Requête SQL sur la table maTable...
tbl(con, "SELECT COUNT(*) FROM maTable")

# ... ou utilisation des verbes de dplyr
tbl(con) %>% summarise(n())
```

Paralléliser des traitements avec `parallel` (1)

La plupart des ordinateurs possèdent aujourd'hui plusieurs cœurs (*core*) susceptibles de mener des traitements **en parallèle** (8 sur chaque serveur d'AUS par exemple).

Par défaut, R n'expoite qu'un seul cœur : le *package* `parallel` (mais aussi les *packages* `snow` ou `foreach` par exemple) permettent de **paralléliser des structures du type `*apply`**.

Ce type d'opérations est composé de plusieurs étapes :

1. Création et paramétrage du « *cluster* » de cœurs à utiliser (chargement des fonctions et *packages* nécessaires sur chaque cœur) ;
2. Lancement du traitement parallélisé avec `parLapply()` ;
3. Arrêt des processus du *cluster* avec `stopCluster()`.

Paralléliser des traitements avec `parallel` (2)

Dans cet exemple, on cherche à appliquer la fonction `f` à chaque matrice de la liste `l`.

```
library(MASS)
f <- function(x) rowSums(ginv(x))
l <- lapply(1:100, function(x) matrix(runif(1e4), ncol = 1e2))

# Création et paramétrage du cluster
library(parallel)
cl <- makeCluster(4)
clusterEvalQ(cl, library(MASS))
clusterExport(cl, "f")

# Lancement du calcul parallélisé
parLapply(cl, l, f)

# Arrêt des processus du cluster
stopCluster(cl)
```

Paralléliser des traitements avec parallel (3)

```
microbenchmark(times = 10
  , lapply(l, f)
  , parLapply(cl, l, f)
)
  ## Unit: milliseconds
  ##                   expr      min       lq     mean
  ##          lapply(l, f) 636.0039 641.6147 663.0252
  ##  parLapply(cl, l, f) 341.5059 383.9122 412.6761
  ##    median       uq      max neval
  ##  650.1038 658.3229 782.8332    10
  ##  405.4428 437.5670 480.6239    10
```

Rcpp : un package R pour utiliser C++ (1)

Le *package* `Rcpp` permet d'intégrer facilement des fonctions codées en C++ dans un programme R.

```
library(Rcpp)
cppFunction('int add(int x, int y) {
  int result = x + y;
  return result;
}')

add(1, 2)
  ## [1] 3
```

Remarque Il est également possible de soumettre un fichier contenant des fonctions C++ écrit par ailleurs à l'aide de la fonction `sourceCpp()`.

Pour en savoir plus *Advanced R*

Rcpp : un package R pour utiliser C++ (2)

Contrairement à R, C++ est un langage de bas niveau : les boucles y sont en particulier extrêmement rapides.

Exemple Somme cumulée par colonne

```
# Fonction C++
cppFunction('NumericMatrix cumColSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericMatrix out(nrow, ncol);
  for (int j = 0; j < ncol; j++) {
    double acc = 0;
    for(int i = 0; i < nrow; i++){
      acc += x(i, j);
      out(i, j) = acc;
    }
  }
  return out;
}')
```

Rcpp : un package R pour utiliser C++ (3)

```
# Fonction R
cumColSumsR <- function(x){
  apply(x, 2, cumsum)
}

# Les deux fonctions produisent les mêmes résultats...
x <- matrix(rnorm(1e6), ncol = 1e2)
all.equal(cumColSumsR(x), cumColSumsC(x))
  ## [1] TRUE

# ... mais cumColSumsC() est beaucoup plus rapide !
summary(microbenchmark(times = 10
  , cumColSumsR(x)
  , cumColSumsC(x)
))[, c("expr", "lq", "mean", "uq")]
  ##             expr        lq      mean        uq
  ## 1 cumColSumsR(x) 17.589140 21.515209 25.846904
  ## 2 cumColSumsC(x)  4.759599  6.959464  9.219795
```

Réaliser des graphiques avec R

R et la réalisation de graphiques

La réalisation de graphiques dans un logiciel statistique est une opération souvent longue et complexe.

Dans la plupart des cas, l'ajustement fin des paramètres par le biais de lignes de code relève de la gageure.

R dispose néanmoins de plusieurs caractéristiques qui facilitent la réalisation de graphiques :

- **souplesse** : la très grande variété des types d'objets simplifie les paramétrages ;
- **rigueur** : la dimension fonctionnelle du langage aide à systématiser l'utilisation des paramètres graphiques ;
- **adaptabilité** : la liberté de développement de modules complémentaires rend possible de profondes innovations dans la conception des graphiques.

Base R ou ggplot2 ?

Il existe aujourd'hui troix principaux paradigmes pour produire des graphiques avec R :

- les fonctionnalités de base du logiciel du *package* `graphics` ;
- les fonctionnalités plus élaborées des *packages* `grid` et `lattice` (non-abordées dans cette formation) ;
- la « grammaire des graphiques » du *package* `ggplot2`.

Plan de la partie

Réaliser des graphiques avec `graphics`

Réaliser des graphiques avec `ggplot2`

Données d'exemple : table mpg de ggplot2

La plupart des exemples de cette partie sont produits à partir de la table mpg du *package* ggplot2.

```
library(ggplot2)
dim(mpg)
  ## [1] 234  11
names(mpg)
  ##  [1] "manufacturer" "model"        "displ"
  ##  [4] "year"         "cyl"          "trans"
  ##  [7] "drv"          "cty"          "hwy"
  ## [10] "fl"           "class"
```

- displ : cylindrée ;
- drv : transmission (f traction, r propulsion, 4 quatre roues motrices) ;
- cty et hwy : nombre de *miles* parcourus par *gallon* d'essence en ville et sur autoroute respectivement.

Réaliser des graphiques avec `graphics`

Beaucoup de fonctions, des paramètres communs

La création de graphiques avec le *package* de base `graphics` s'appuie sur la **fonction `plot()`** ainsi que sur des **fonctions spécifiques** :

- `plot(hist(x))`, `plot(density(x))` : histogrammes et densités ;
- `plot(ts)` : représentation de séries chronologiques ;
- `plot(x, y)` : nuages de points ;
- `barplot(table(x))` et `pie(table(x))` : diagrammes en bâtons et circulaires.

Si ce n'est quelques **arguments spécifiques**, ces fonctions partagent un ensemble de **paramètres graphiques communs**.

Pour en savoir plus Le site statmethods.net recense et illustre la plupart des fonctions du *package* `graphics`.

Histogrammes et densités

Les fonctions `histogram()` et `density()` calculent les statistiques ensuite utilisées par la fonction `plot()` pour construire les graphiques.

Arguments spécifiques à `hist()` :

- `breaks` : méthode pour déterminer les limites des classes ;
- `labels = TRUE` : ajoute l'effectif de chaque classe.

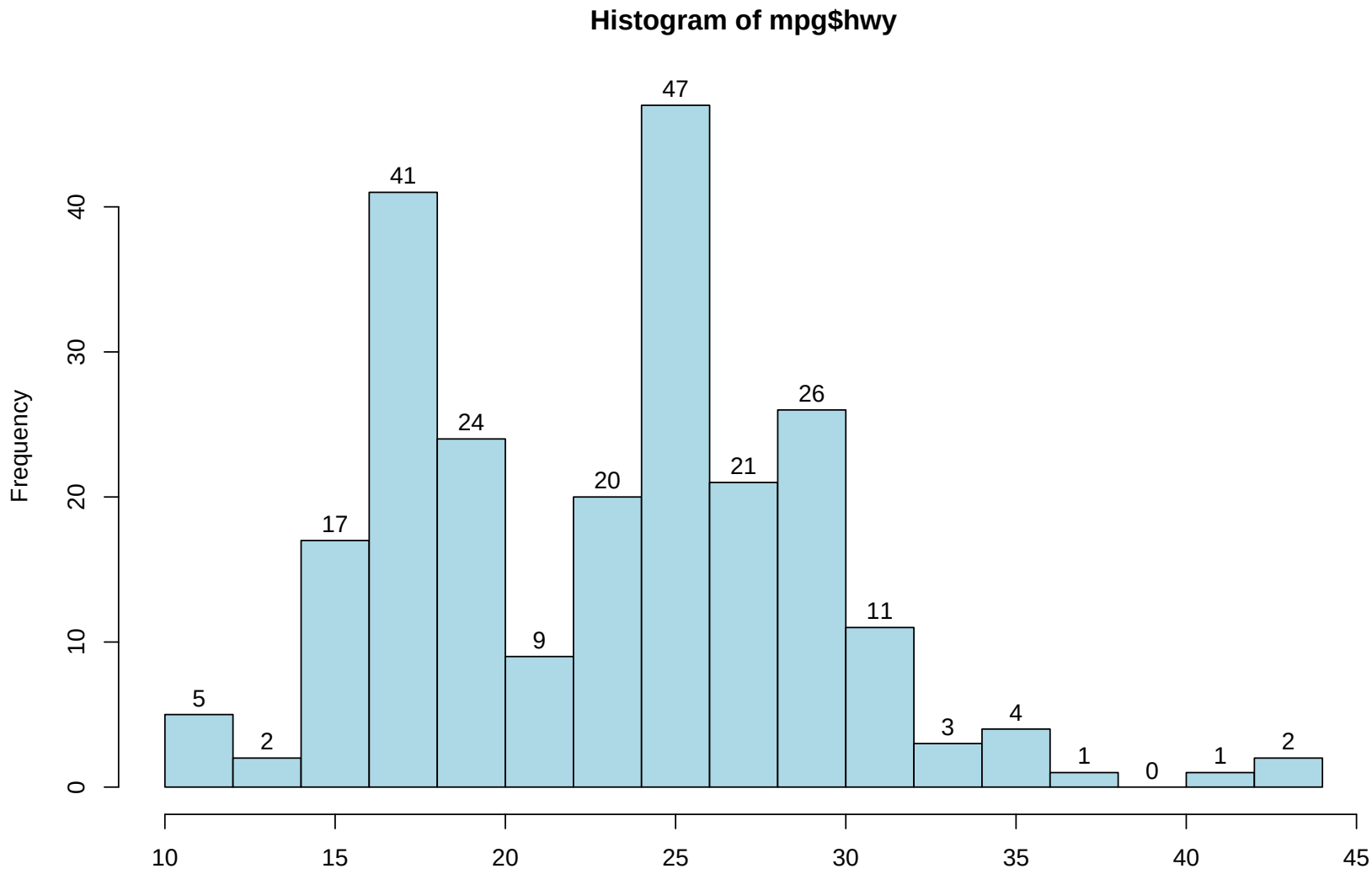
Arguments spécifiques à `density()` :

- `bw` : largeur de la fenêtre utilisée par la fonction de lissage ;
- `kernel` : fonction de lissage utilisée.

Remarque L'argument `plot` de la fonction `hist()` (`TRUE` par défaut) affiche automatiquement un graphique, sans avoir à appeler explicitement la fonction `plot()`.

Histogrammes et densités

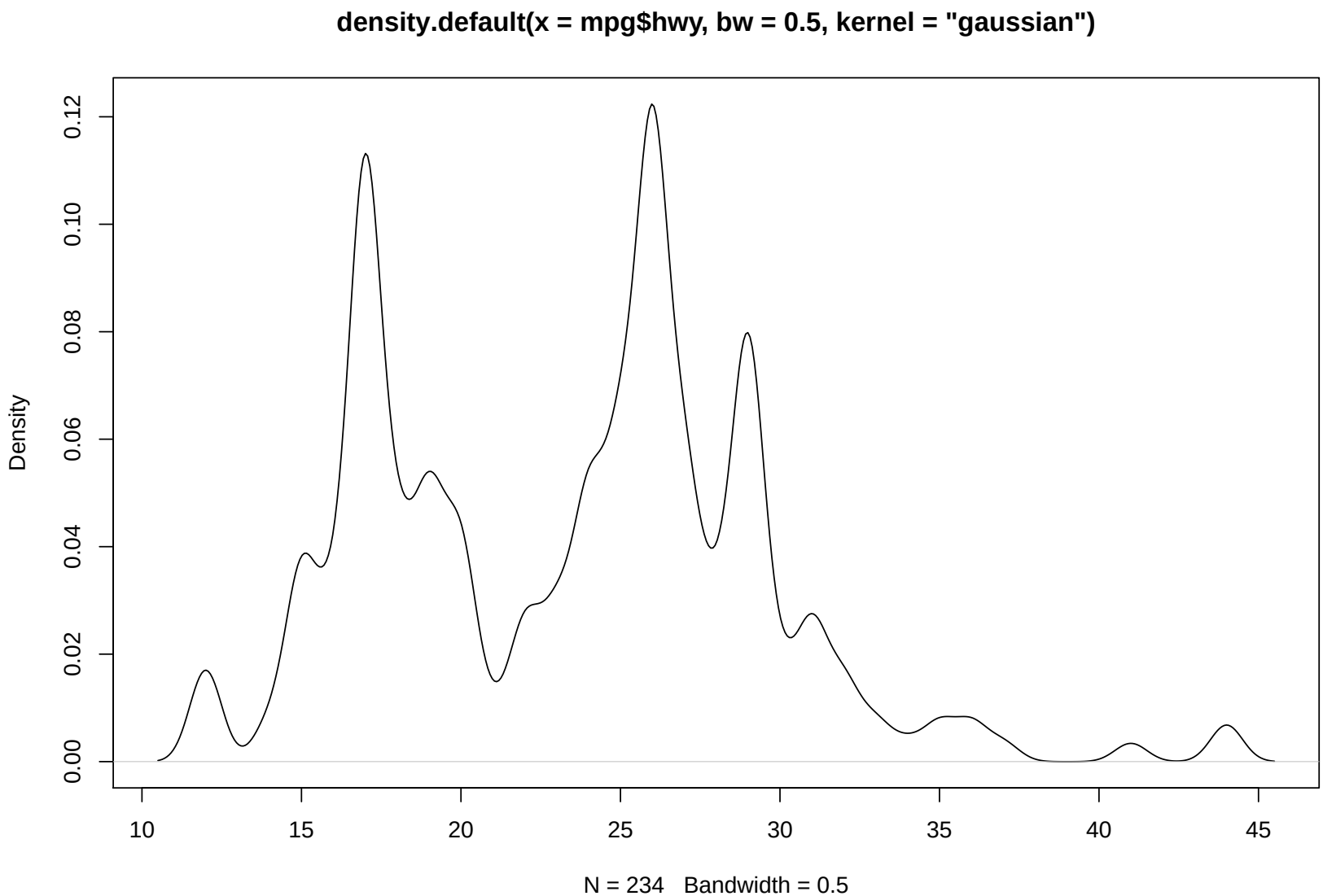
```
hist(mpg$hwy, breaks = seq(10, 44, by = 2),
     col = "lightblue", labels = TRUE)
```



Réaliser des graphiques avec graphics

Histogrammes et densités

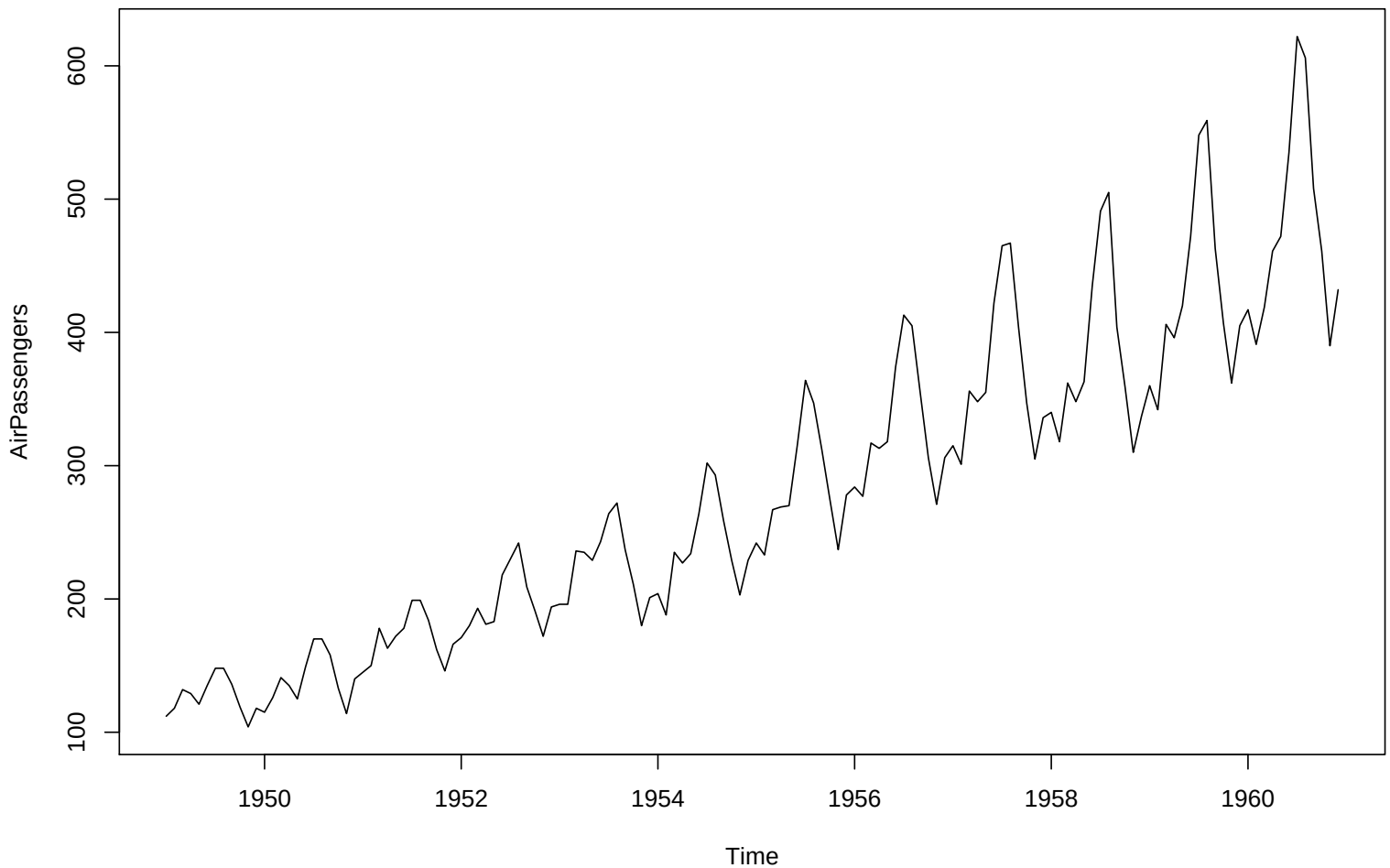
```
plot(density(mpg$hwy, bw = 0.5, kernel = "gaussian"))
```



Réaliser des graphiques avec graphics

Séries chronologiques avec plot(ts)

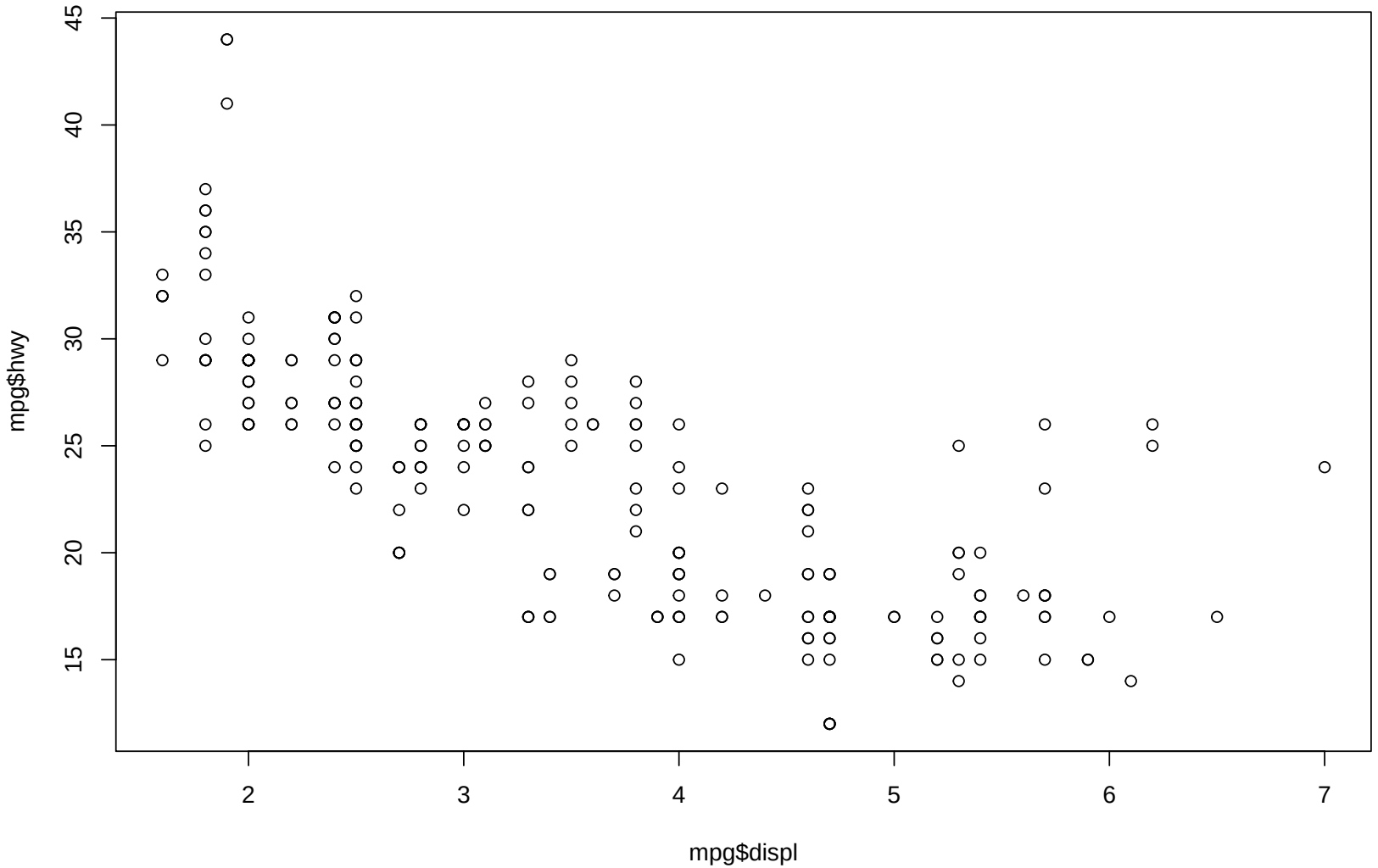
```
class(AirPassengers)
  ## [1] "ts"
plot(AirPassengers)
```



Réaliser des graphiques avec graphics

Nuages de points avec plot(x, y)

```
plot(mpg$displ, mpg$hwy)
```



Diagrammes en bâtons et circulaires

La fonction `table()` permet de calculer les statistiques utilisées ensuite par `barplot()` et `pie()` pour construire les graphiques.

Arguments spécifiques à `barplot()` :

- `horiz` : construit le graphique horizontalement ;
- `names.arg` : nom à afficher près des barres.

Arguments spécifiques à `pie()` :

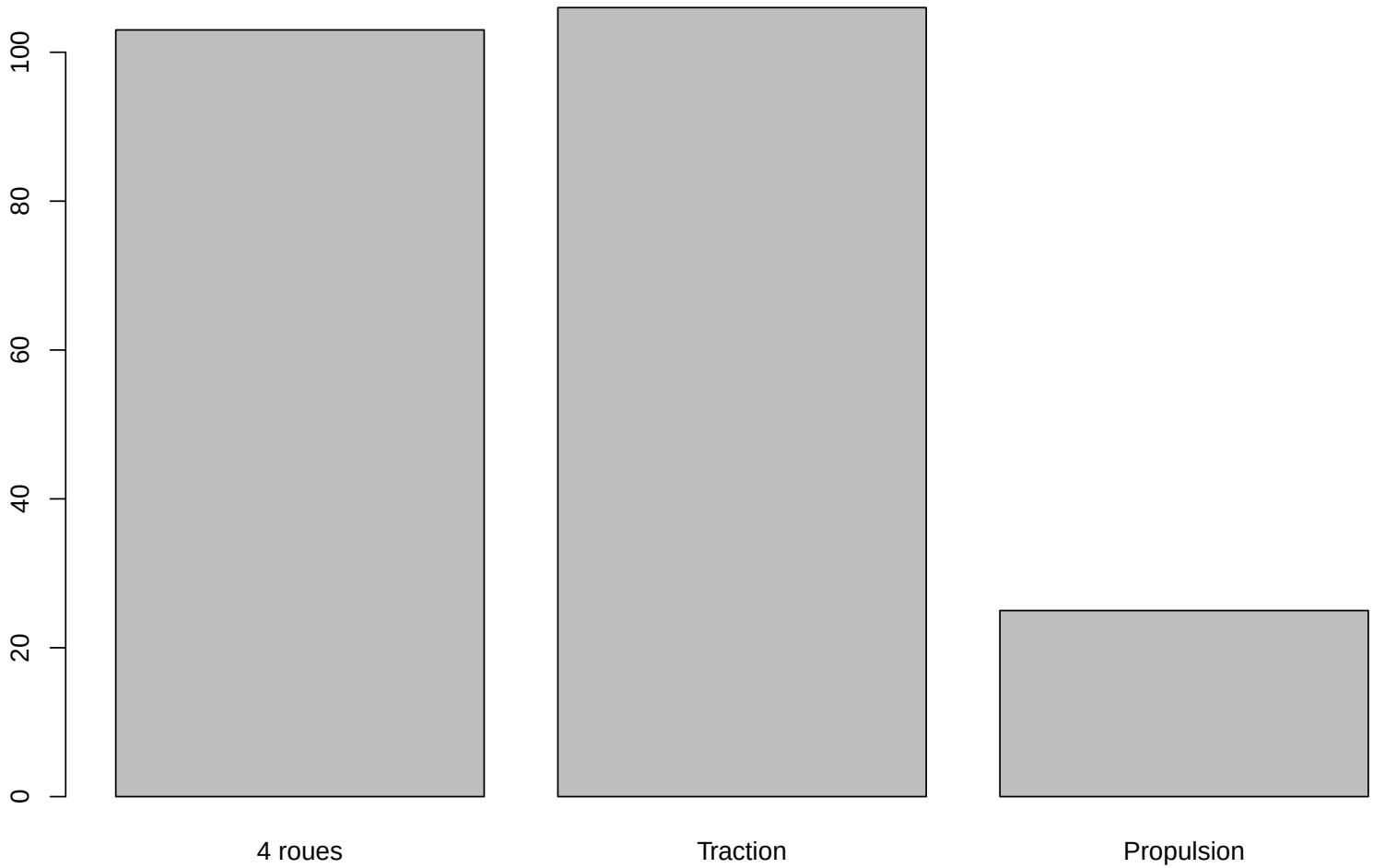
- `labels` : noms à afficher à côté des portions de disque ;
- `clockwise` : sens dans lequel sont représentées les modalités ;
- `init.angle` : point de départ en degrés.

Remarque Quand `barplot()` est appliqué à un tri croisé, la couleur des barres varie et les paramètres deviennent utiles :

- `beside` : position des barres ;
- `legend.text` : ajoute une légende avec le texte indiqué.

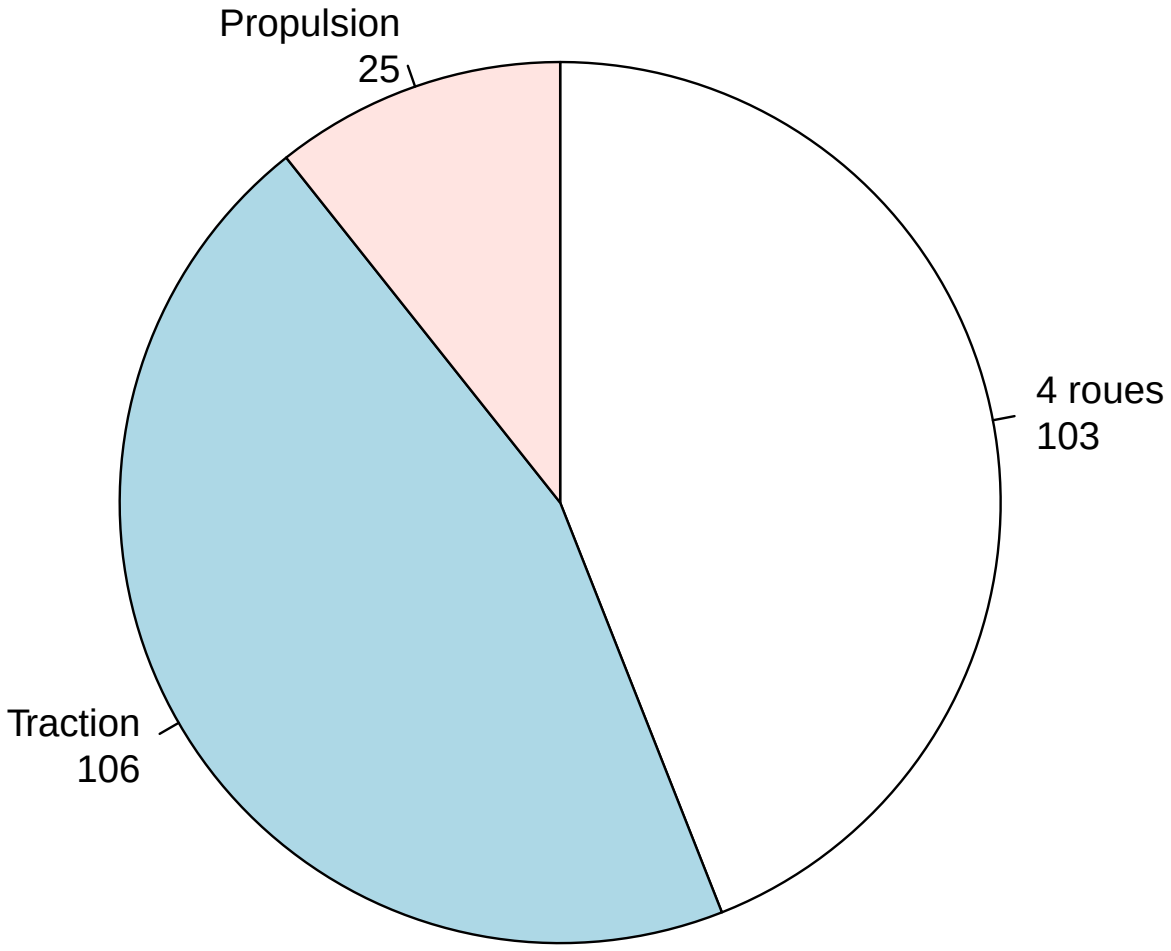
Diagrammes en bâtons et circulaires

```
uni <- table(mpg$drv)
lab <- c("4 roues", "Traction", "Propulsion")
barplot(uni, names.arg = lab)
```



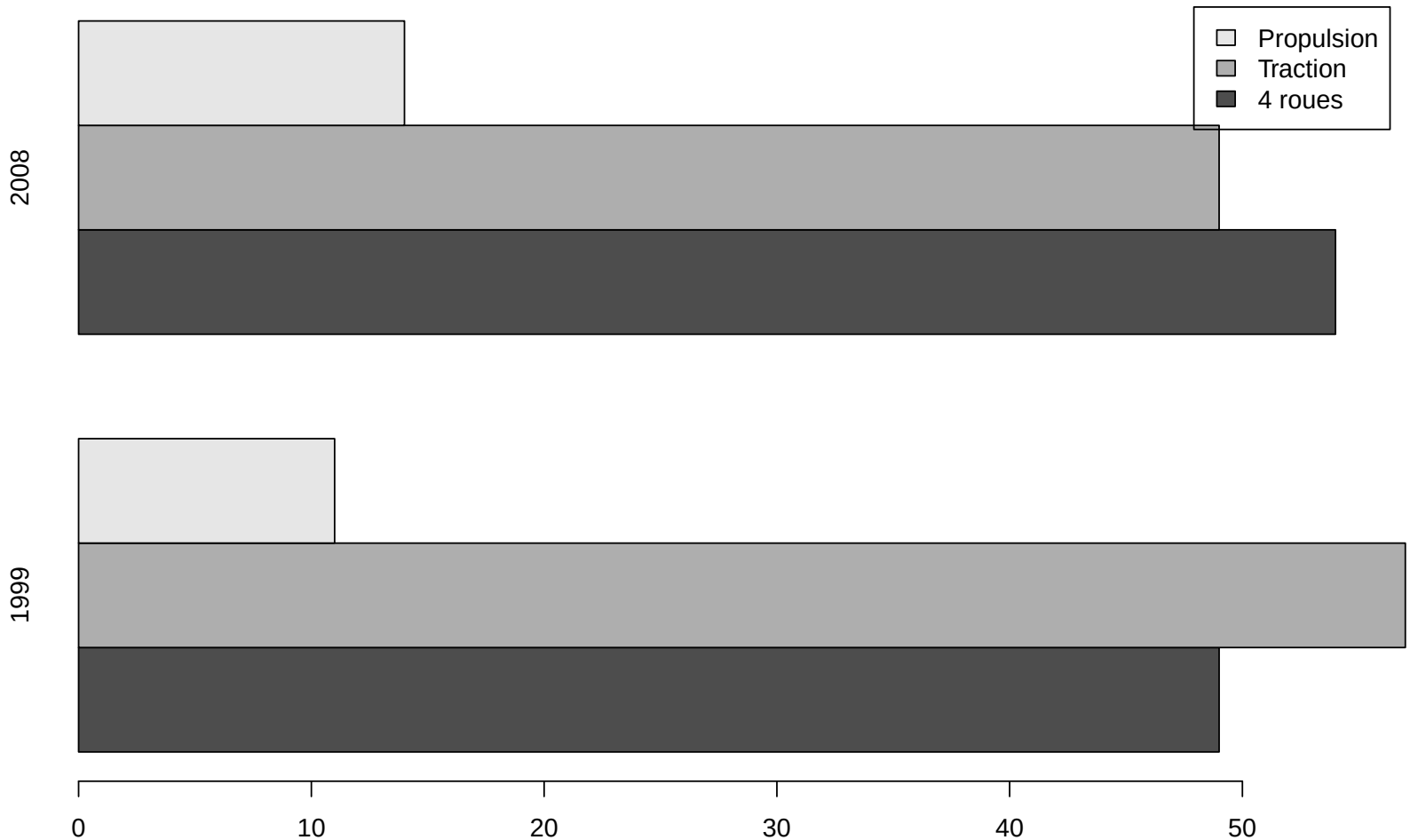
Diagrammes en bâtons et circulaires

```
pie(uni, labels = paste0(lab, "\n", uni)
    , init.angle = 90, clockwise = TRUE)
```



Diagrammes en bâtons et circulaires

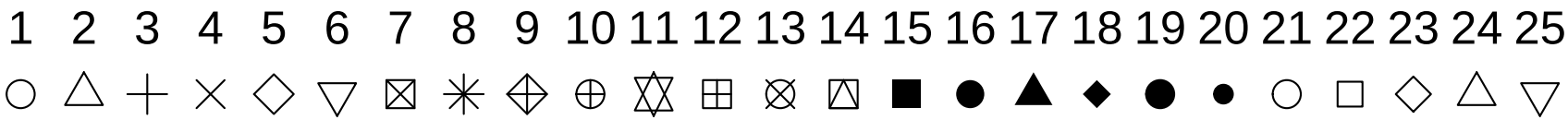
```
bi <- table(mpg$drv, mpg$year)
barplot(bi, horiz = TRUE, beside = TRUE, legend.text = lab)
```



Couleur, forme et taille des objets

Plusieurs paramètres permettent de modifier la couleur, la forme ou la taille des éléments qui composent un graphique :

- `pch` : entier ou caractère spécial indiquant la forme des points à représenter.



- `col` : valeur indiquant la couleur du contour des formes utilisées. Peut être un entier (recyclé au-delà de 8), un nom ou un code RGB hexadécimal (du type `"#FF1111"`).

Pour certaines formes (`pch` entre 21 et 25), il est également possible de modifier la couleur de remplissage avec `bg`.

Couleur, forme et taille des objets

Remarque : la palette de couleurs accessibles en utilisant des entiers est réduite. Il est possible de l'étendre considérablement *via* la fonction `colors()`.

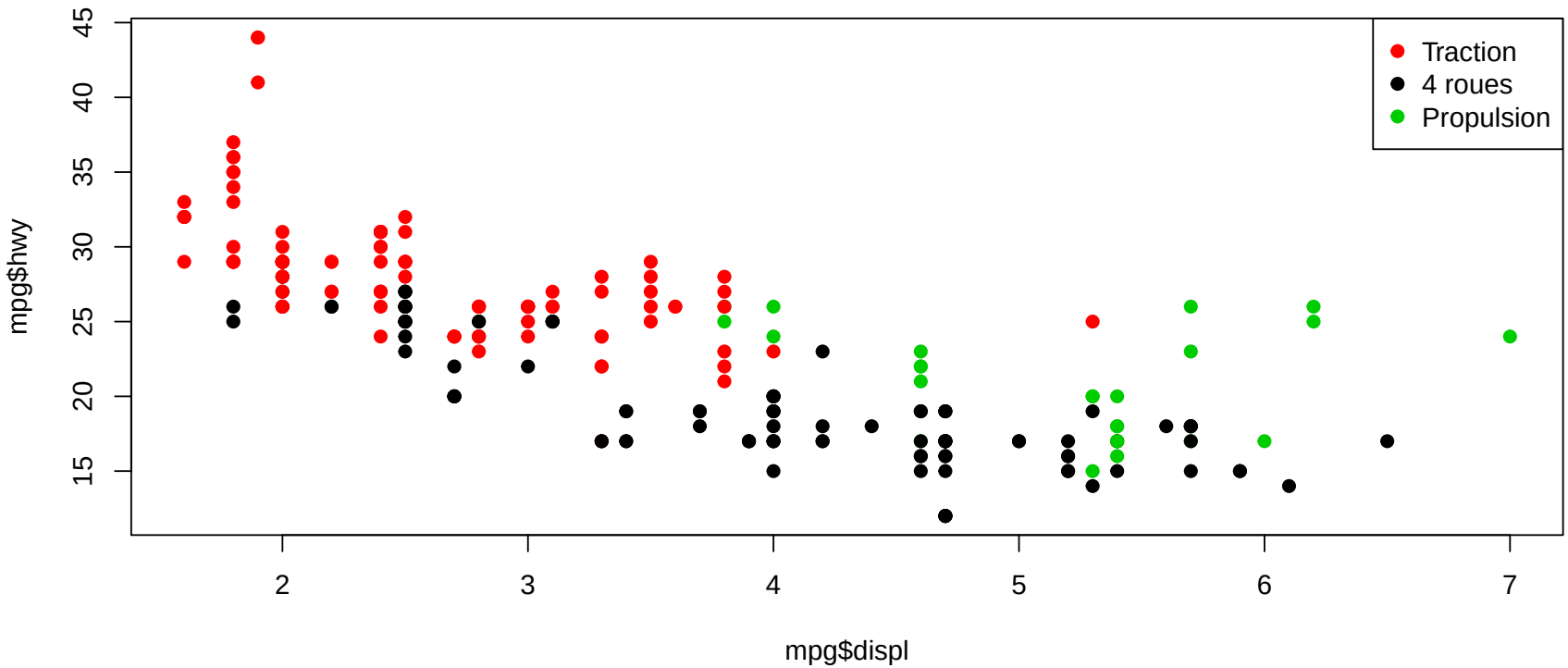
```
colors()[1:3]
  ## [1] "white"          "aliceblue"    "antiquewhite"
length(colors())
  ## [1] 657
grep("blue", colors(), value = TRUE)[1:3]
  ## [1] "aliceblue" "blue"      "blue1"
```

- `cex` : utilisé dans une fonction `plot()`, `cex` permet d'ajuster la taille des points qui le composent.

Couleur, forme et taille des objets

La fonction `legend()` permet d'ajouter une légende.

```
t <- factor(mpg$drv
  , labels = c("4 roues", "Traction", "Propulsion"))
plot(mpg$displ, mpg$hwy, pch = 21, col = t, bg = t)
legend("topright", legend = unique(t), pch = 21
  , col = unique(t), pt.bg = unique(t))
```



Titres, texte et axes

Les titres sont paramétrés à l'aide des fonctions suivantes :

- `main` pour ajouter le titre principal ;
- `xlab` et `ylab` pour ajouter des titres aux axes.

La fonction `text()` permet d'ajouter du texte sur le graphique en le positionnant par ses coordonnées, éventuellement avec un décalage (pour nommer des points par exemple). Il est également possible de paramétrer les axes :

- `xlim` et `ylim` spécifient les valeurs minimales et maximales de chaque axe ;
- `axis()` est une fonction qui permet d'ajouter un axe personnalisé.

Remarque Pour produire un graphique sans axe et les rajouter après, utiliser l'option `axes = FALSE` de la fonction `plot()`.

Combinaison de plusieurs graphiques

Par défaut l'utilisation de la fonction `plot()` produit un nouveau graphique.

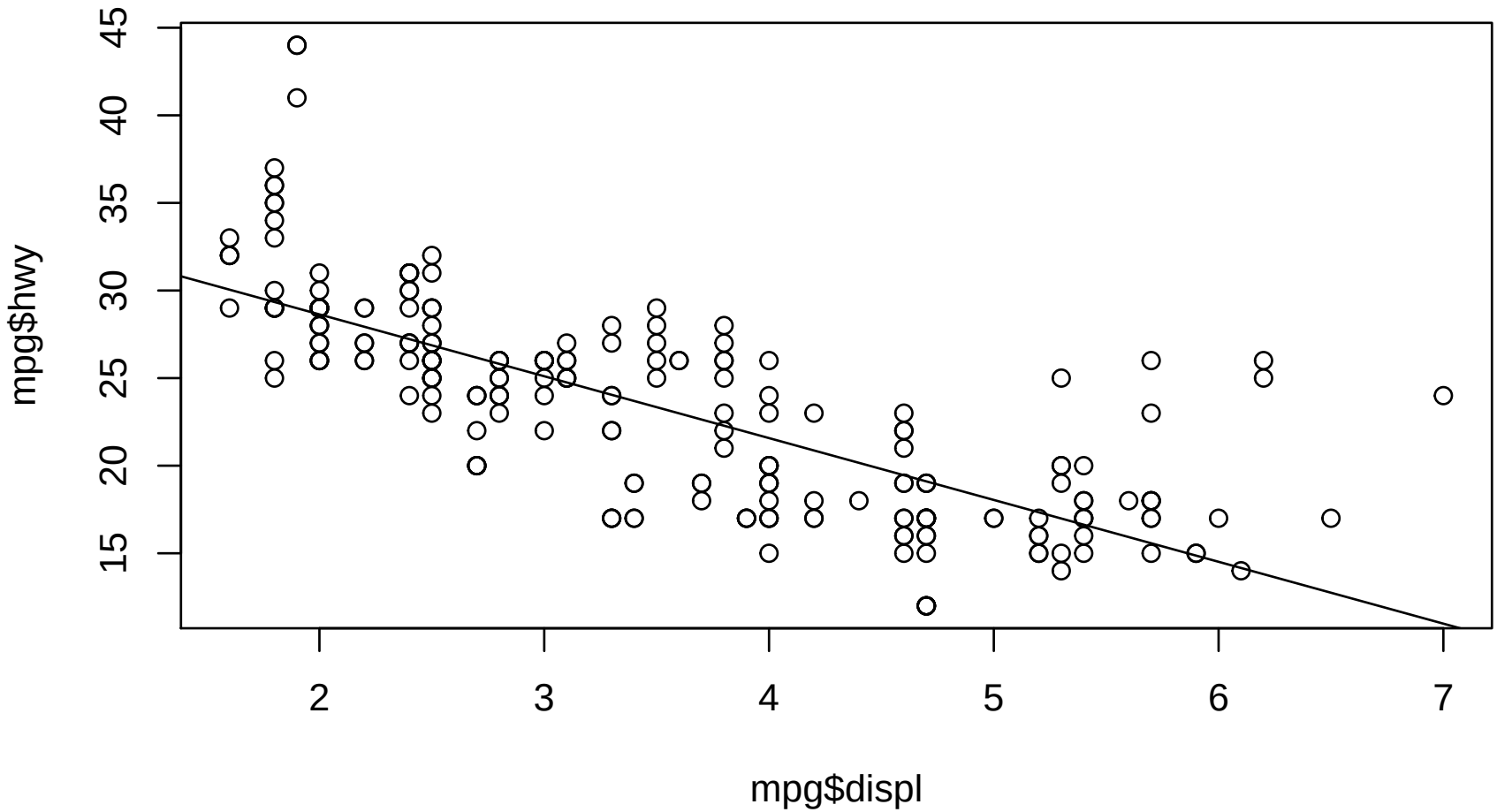
Pour superposer différents graphiques, le plus simple est de commencer par une instruction `plot()` puis de la compléter :

- avec `points()` pour ajouter des points ;
- avec `lines()` pour ajouter des lignes ;
- avec `abline()` pour ajouter des lignes d'après une équation ;
- avec `curve()` pour ajouter des courbes d'après une équation.

Exemple Ajout d'une droite de régression au graphique de `hwy` par `displ`.

Combinaison de plusieurs graphiques

```
reg <- lm(hwy ~ displ, data = mpg)
plot(mpg$displ, mpg$hwy)
abline(a = reg$coefficients[1], b = reg$coefficients[2])
```



Paramètres généraux et disposition (1)

Utilisée en dehors de la fonction `plot()`, la fonction `par()` permet de définir l'ensemble des paramètres graphiques globaux.

Ses mots-clés les plus importants sont :

- `mfrow` : permet de disposer plusieurs graphiques côte-à-côte.

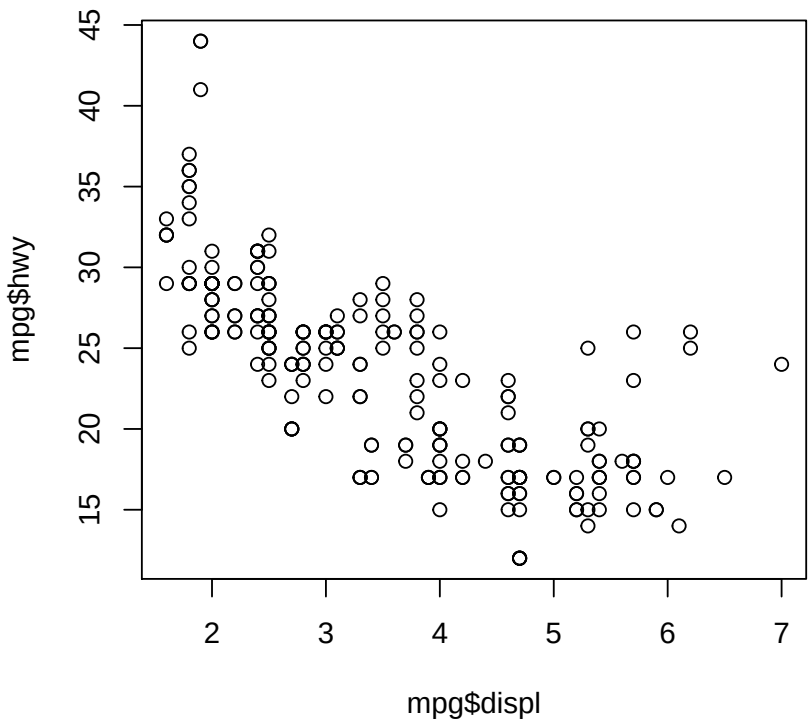
```
par(mfrow = c(1, 2)) # 1 ligne et 2 colonnes
par(mfrow = c(3, 2)) # 3 lignes et 2 colonnes
par(mfrow = c(1, 1)) # 1 ligne et 1 colonne
```

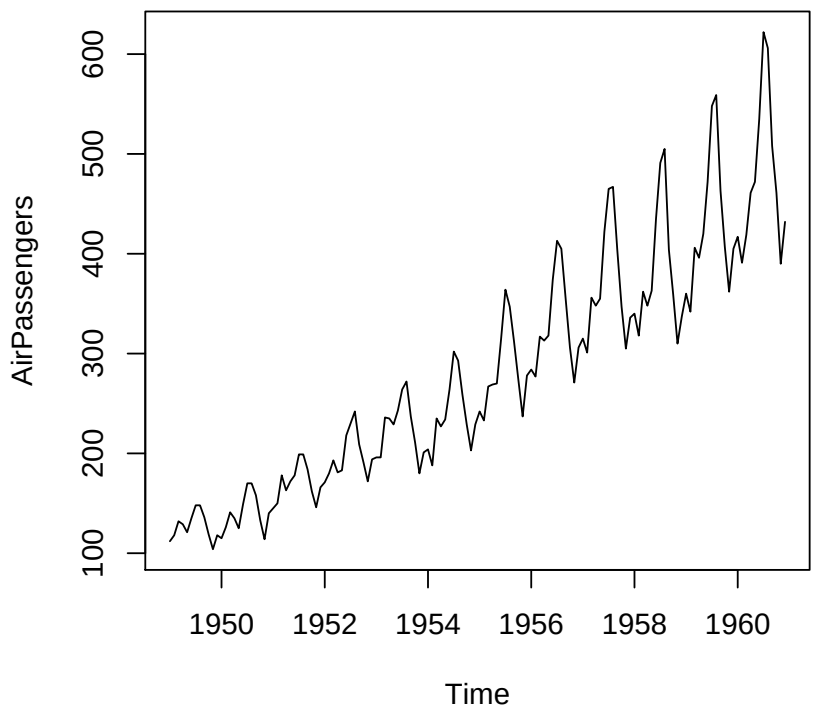
- `cex` : coefficient multiplicatif pour modifier la taille de l'ensemble des textes et symboles utilisés dans les graphiques (1 par défaut).

Pour en savoir plus La page d'aide de la fonction `par()` détaille toutes ces options.

Paramètres généraux et disposition (2)

```
par(mfrow = c(1, 2))
plot(mpg$displ, mpg$hwy)
plot(AirPassengers)
```





Exportation

Pour exporter des graphiques depuis R, la démarche consiste à rediriger le flux de production du graphique vers un fichier à l'aide d'une fonction du *package* grDevices. Par exemple :

```
png("monGraphique.png", width = 10, height = 8
    , unit = "cm", res = 600)
plot(mpg$displ, mpg$hwy)
dev.off()
```

Dans ce contexte, les fonctions les plus utiles sont : png(), jpeg() et pdf(). En particulier, pdf() permet de conserver le caractère vectoriel des graphiques produits par R.

Remarque Les graphiques peuvent également facilement être exportés depuis RStudio en utilisant les menus prévus à cet effet.

L'implémentation d'une grammaire des graphiques

Le *package* `graphics` permet de réaliser une grande quantité de graphiques mais présente deux limites importantes :

- les fonctions qui le composent forment une casuistique complexe ;
- il n'est pas possible d'inventer de nouvelles représentations à partir des fonctions existantes.

Ce sont ces limites que tente de dépasser le *package* `ggplot2` en implémentant une **grammaire des graphiques**.

Comme les éléments du langage, les **composants élémentaires** d'un graphique doivent pouvoir être **réassemblés** pour produire de **nouvelles représentations.**

Pour aller plus loin Wilkinson L. (2005) *The Grammar of Graphics*, Springer, ggplot2: elegant graphics for data analysis

Les trois composants essentiels d'un graphique

La construction d'un graphique avec ggplot2 fait intervenir trois composants essentiels (d'après Wickham, *ibid.*, 2.3) :

- le `data.frame` dans lequel sont stockées les données à représenter ;
- des correspondances esthétiques (*aesthetic mappings*) entre des variables et des propriétés visuelles ;
- au moins une couche (*layer*) décrivant comment représenter les observations.

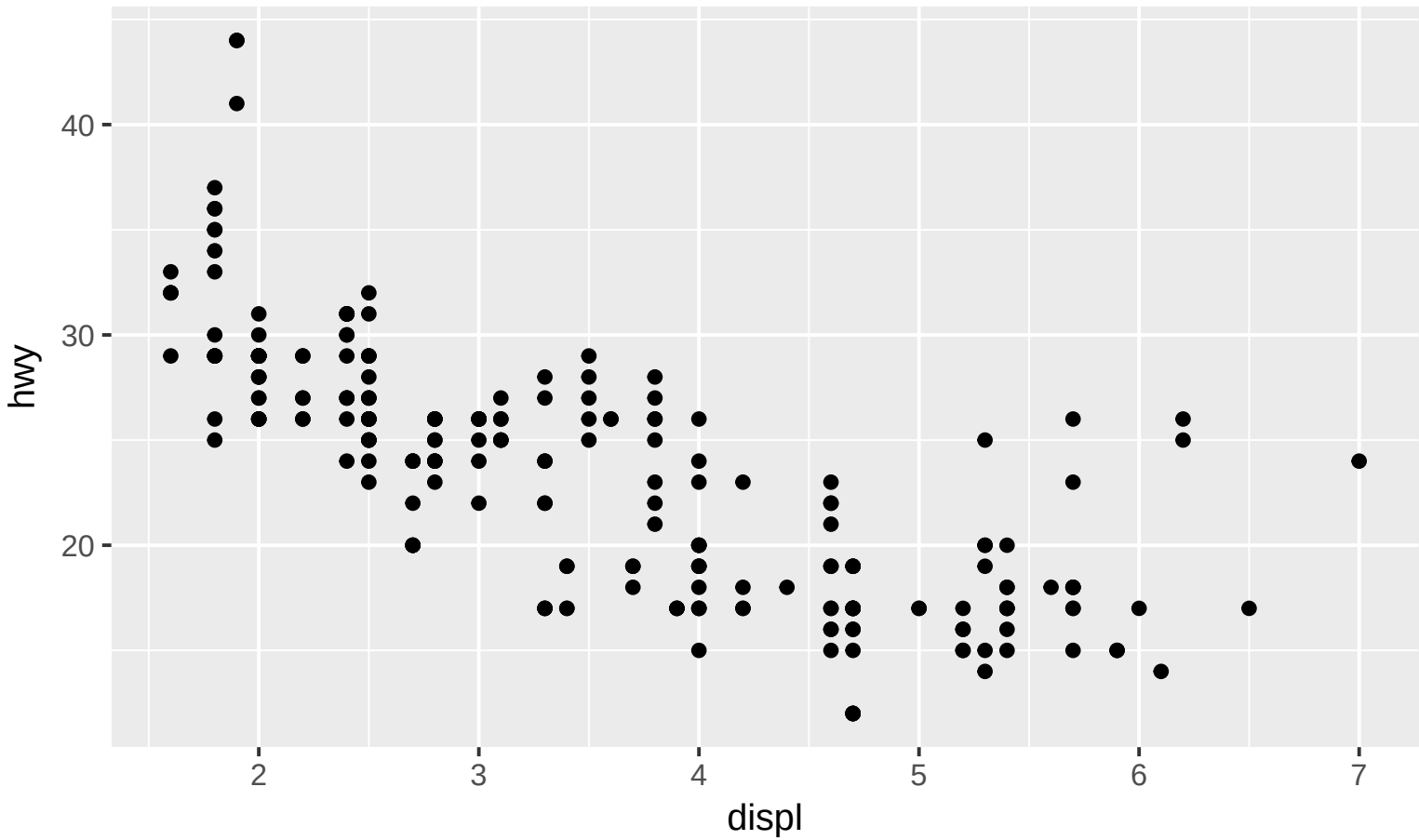
Exemple *Miles per gallon* sur l'autoroute en fonction de la cylindrée.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point()
```

Réaliser des graphiques avec ggplot2

Les trois composants essentiels d'un graphique

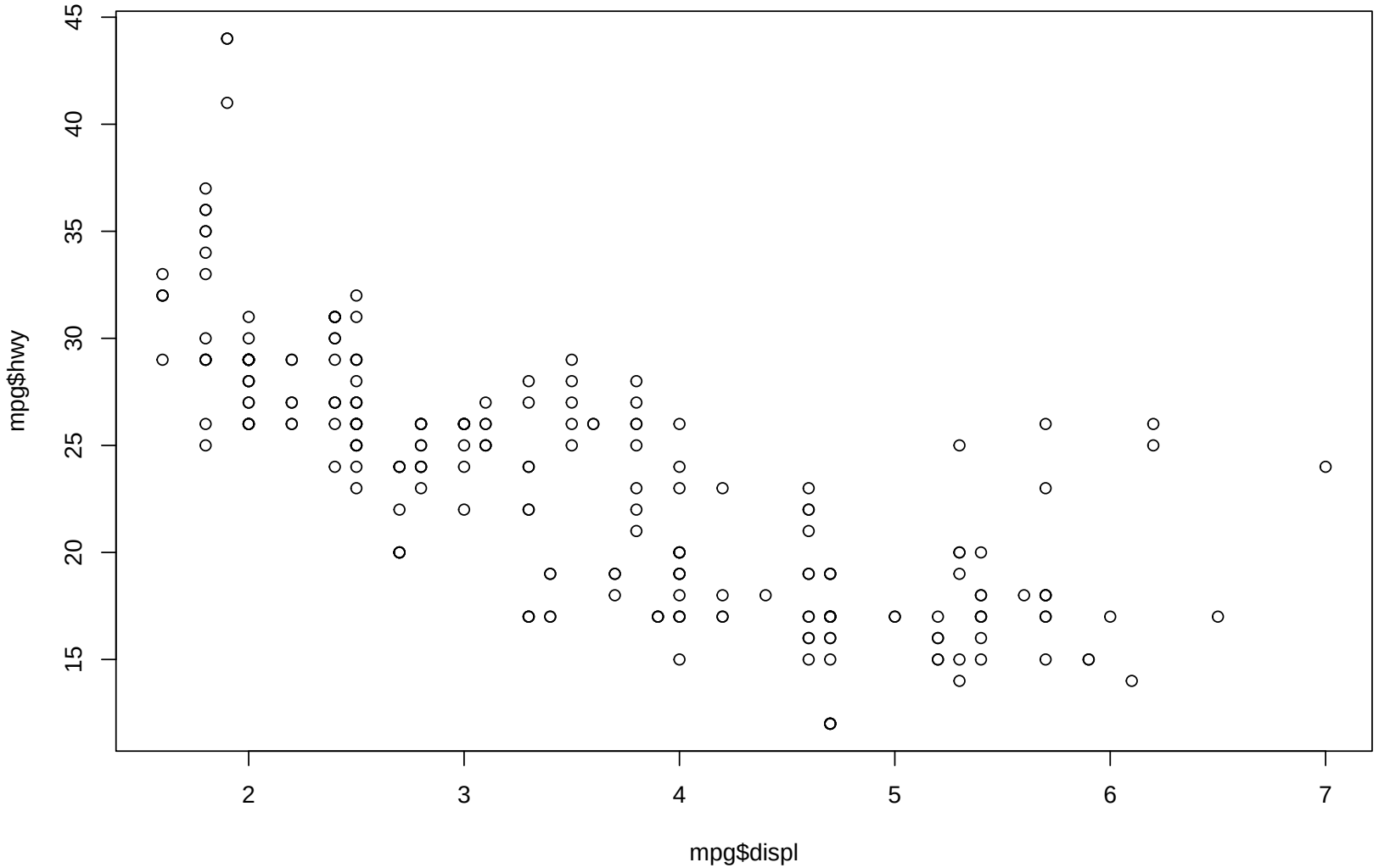
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point()
```



Réaliser des graphiques avec ggplot2

Rappel : le même graphique avec base R

```
plot(mpg$displ, mpg$hwy)
```

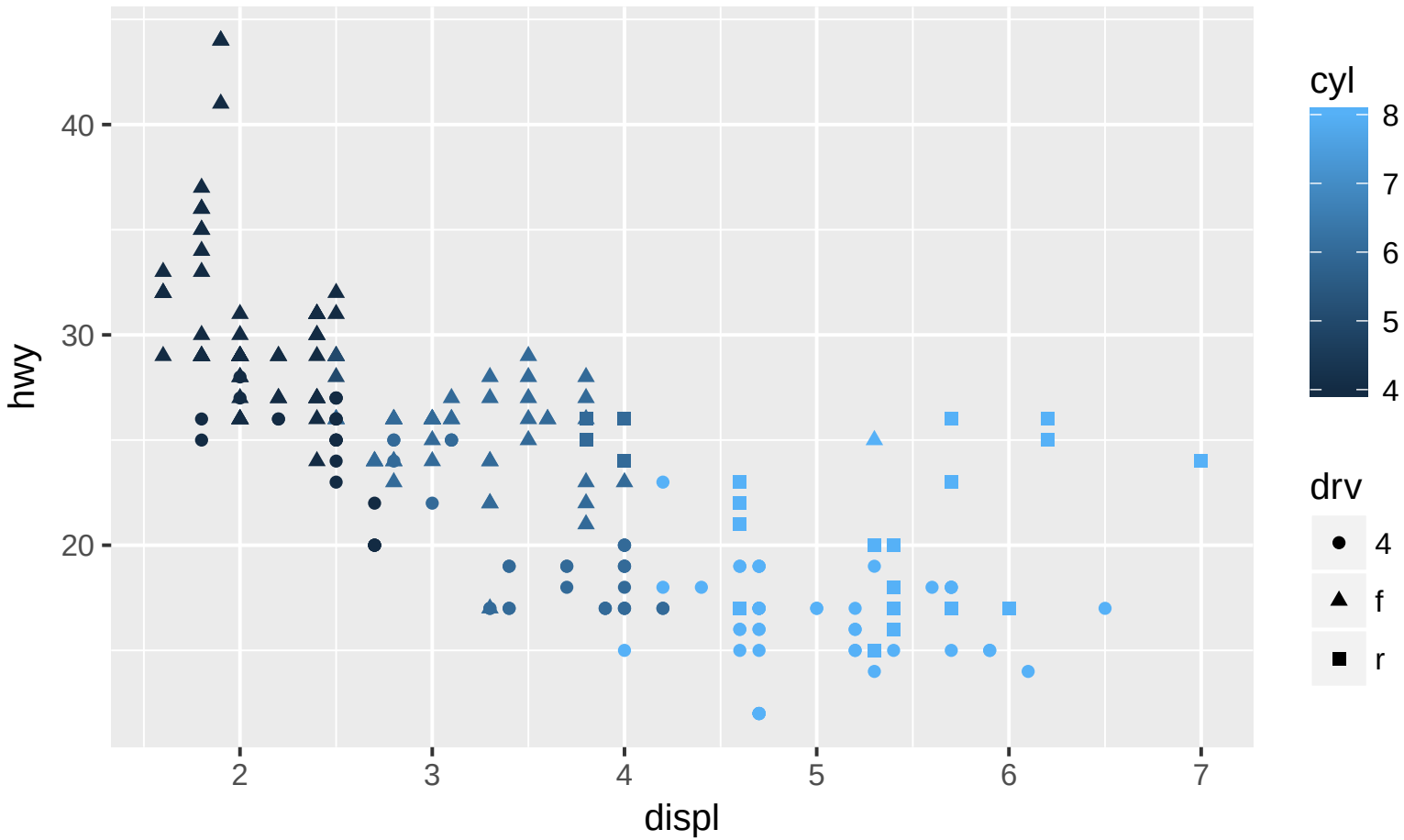


Couleur, forme et taille des objets

Pour faire varier l'aspect visuel des éléments représentés en fonction de données, il suffit d'**associer une variable à l'attribut de couleur, de taille ou de forme** dans la fonction `aes()`.

Couleur, forme et taille des objets

```
ggplot(mpg, aes(displ, hwy, colour = cyl, shape = drv)) +
  geom_point()
```



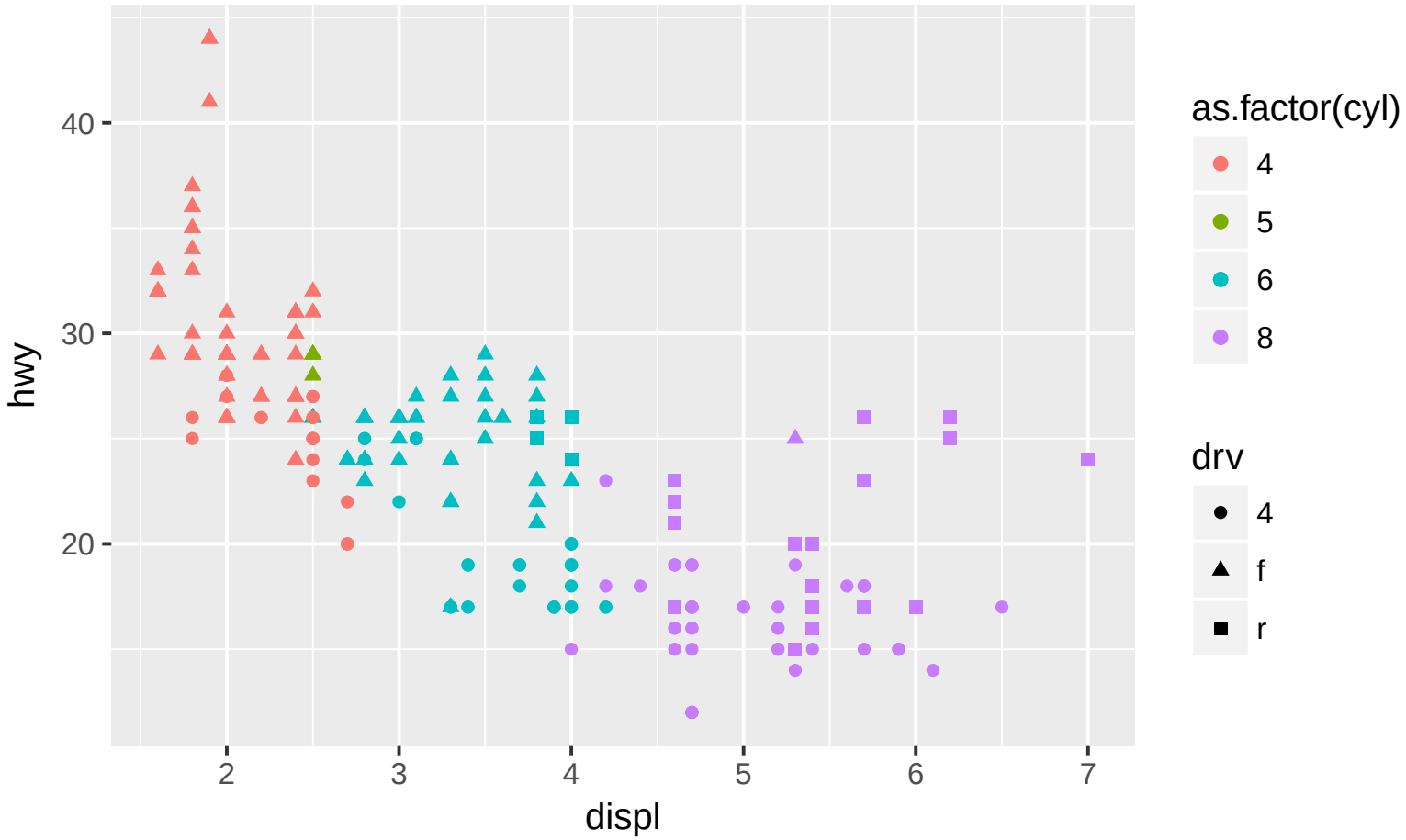
Couleur, forme et taille des objets

Pour faire varier l'aspect visuel des éléments représentés en fonction de données, il suffit d'**associer une variable à l'attribut de couleur, de taille ou de forme** dans la fonction `aes()`.

Selon le type des variables utilisées pour les correspondances esthétiques, **les échelles sont continues ou discrètes**.

Couleur, forme et taille des objets

```
ggplot(mpg, aes(displ, hwy, colour = as.factor(cyl)
  , shape = drv)) +
  geom_point()
```



Couleur, forme et taille des objets

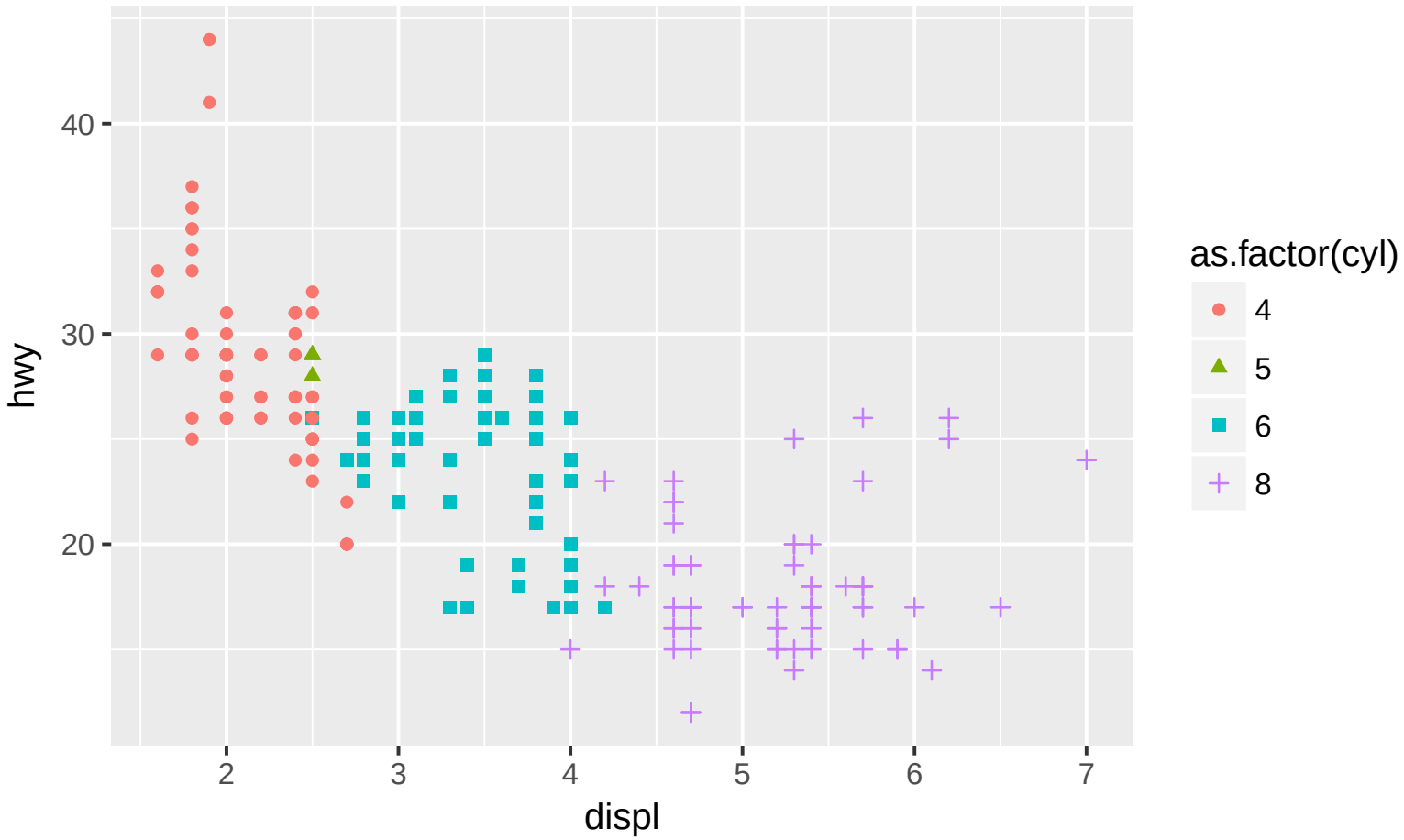
Pour faire varier l'aspect visuel des éléments représentés en fonction de données, il suffit d'**associer une variable à l'attribut de couleur, de taille ou de forme** dans la fonction `aes()`.

Selon le type des variables utilisées pour les correspondances esthétiques, **les échelles sont continues ou discrètes**.

Quand la même variable est utilisée dans plusieurs correspondances esthétiques, **les échelles qui lui correspondent sont fusionnées**.

Couleur, forme et taille des objets

```
ggplot(mpg, aes(displ, hwy, colour = as.factor(cyl)
  , shape = as.factor(cyl))) +
  geom_point()
```



Couleur, forme et taille des objets

Pour faire varier l'aspect visuel des éléments représentés en fonction de données, il suffit d'**associer une variable à l'attribut de couleur, de taille ou de forme** dans la fonction `aes()`.

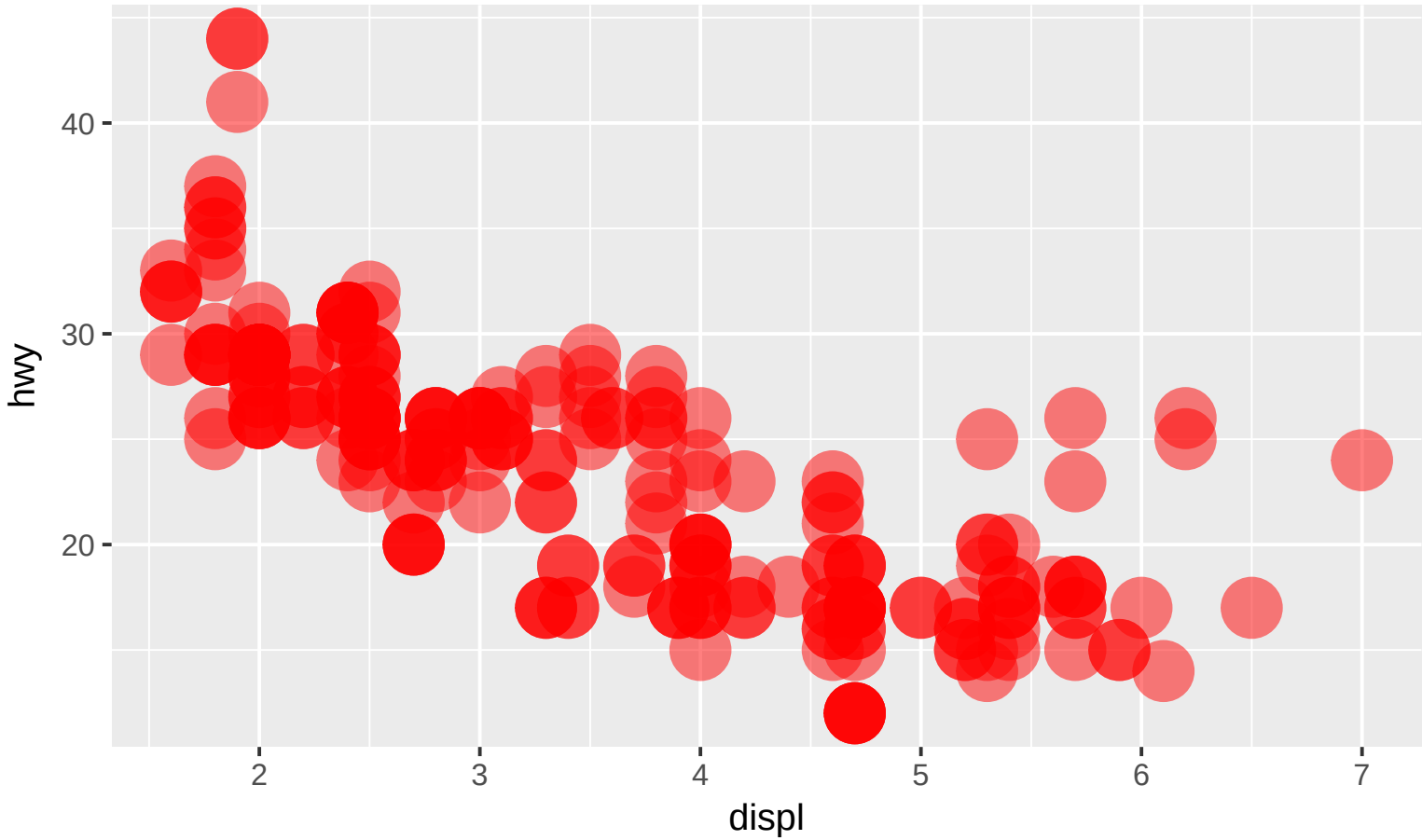
Selon le type des variables utilisées pour les correspondances esthétiques, **les échelles sont continues ou discrètes**.

Quand la même variable est utilisée dans plusieurs correspondances esthétiques, **les échelles qui lui correspondent sont fusionnées**.

Au-delà des correspondances esthétiques dans la fonction `aes()`, **l'aspect visuel peut être ajusté directement dans la fonction `geom_*`**.

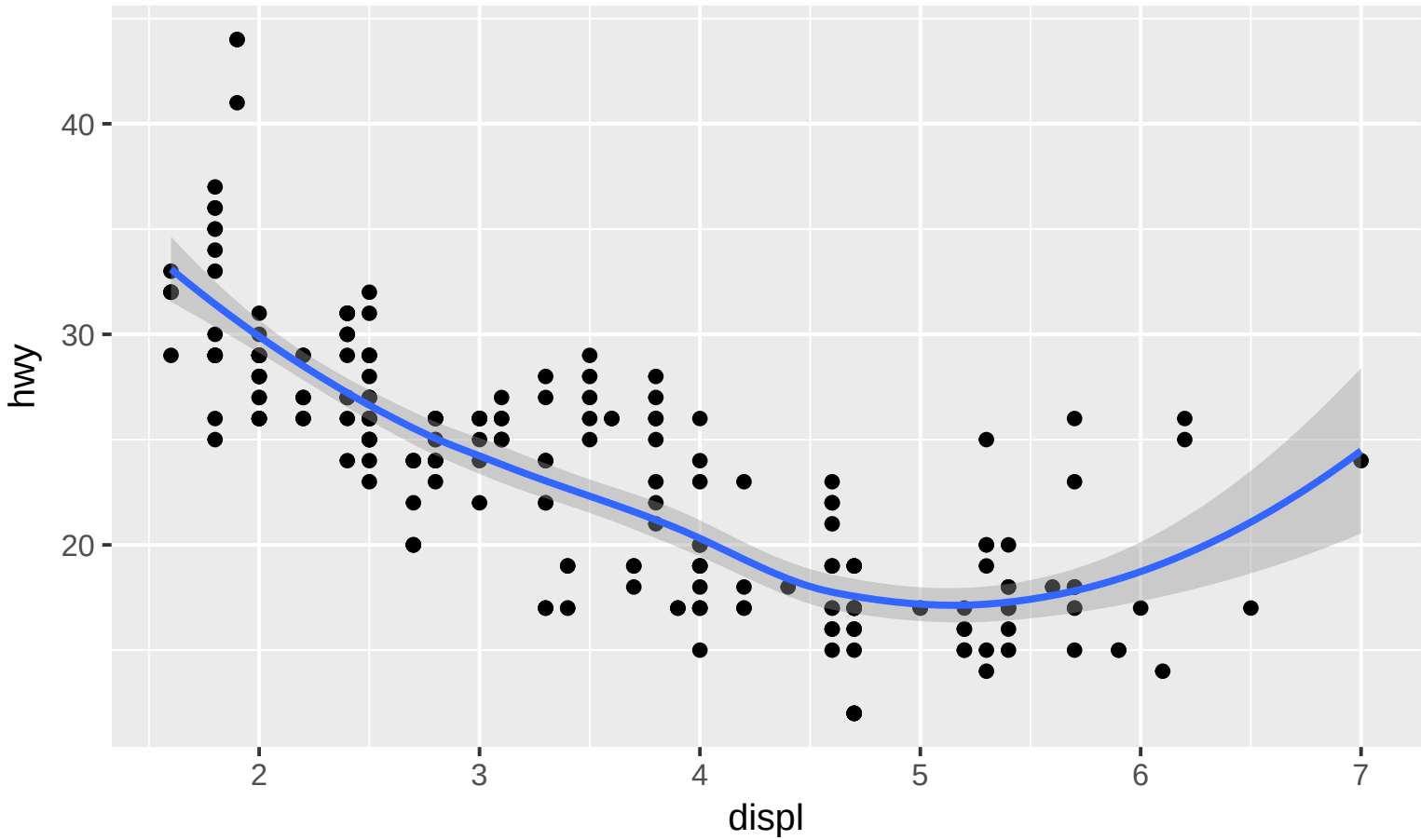
Couleur, forme et taille des objets

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(colour = "red", size = 8, alpha = 0.5)
```



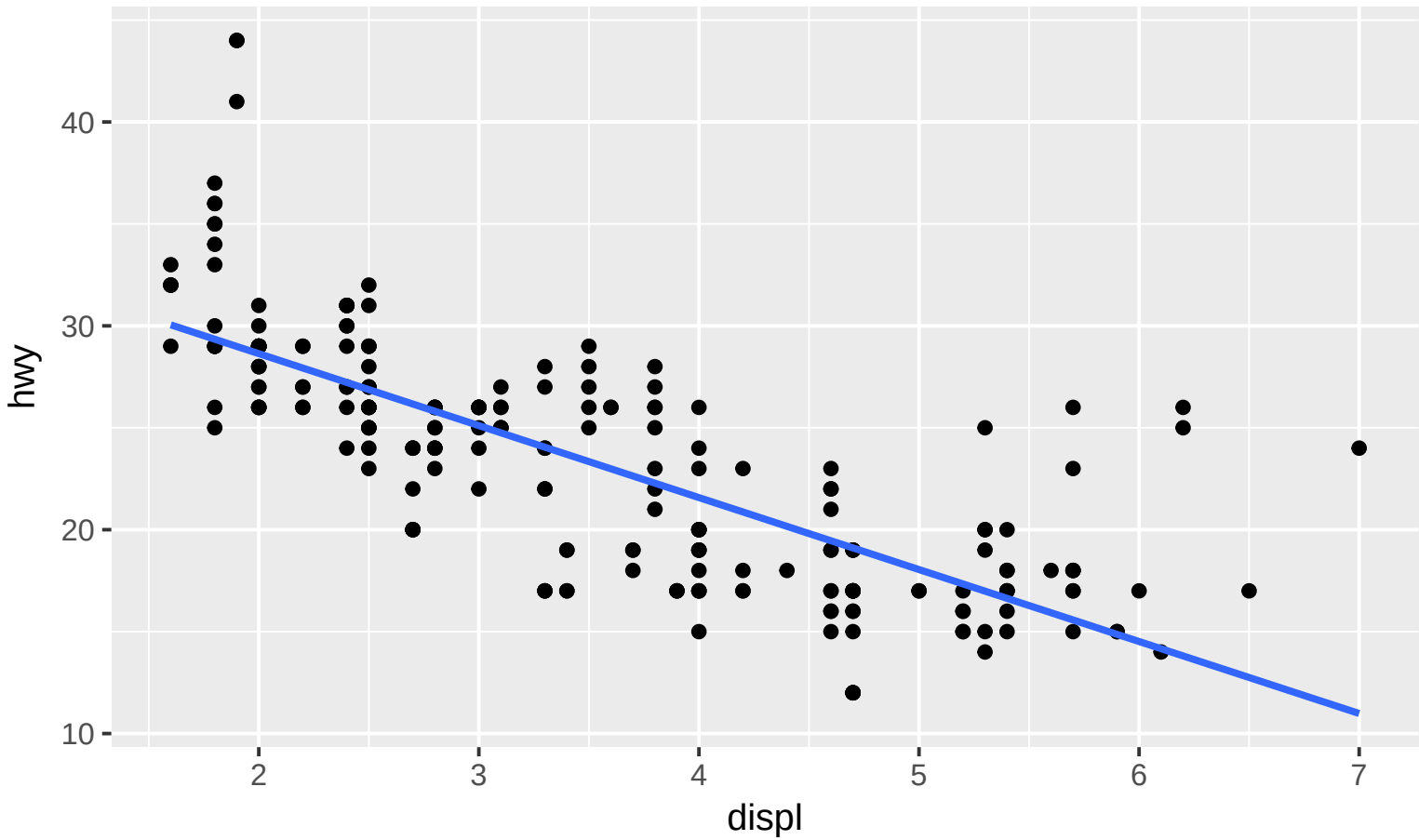
Combinaison de plusieurs graphiques

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() + geom_smooth()
  ## `geom_smooth()` using method = 'loess'
```



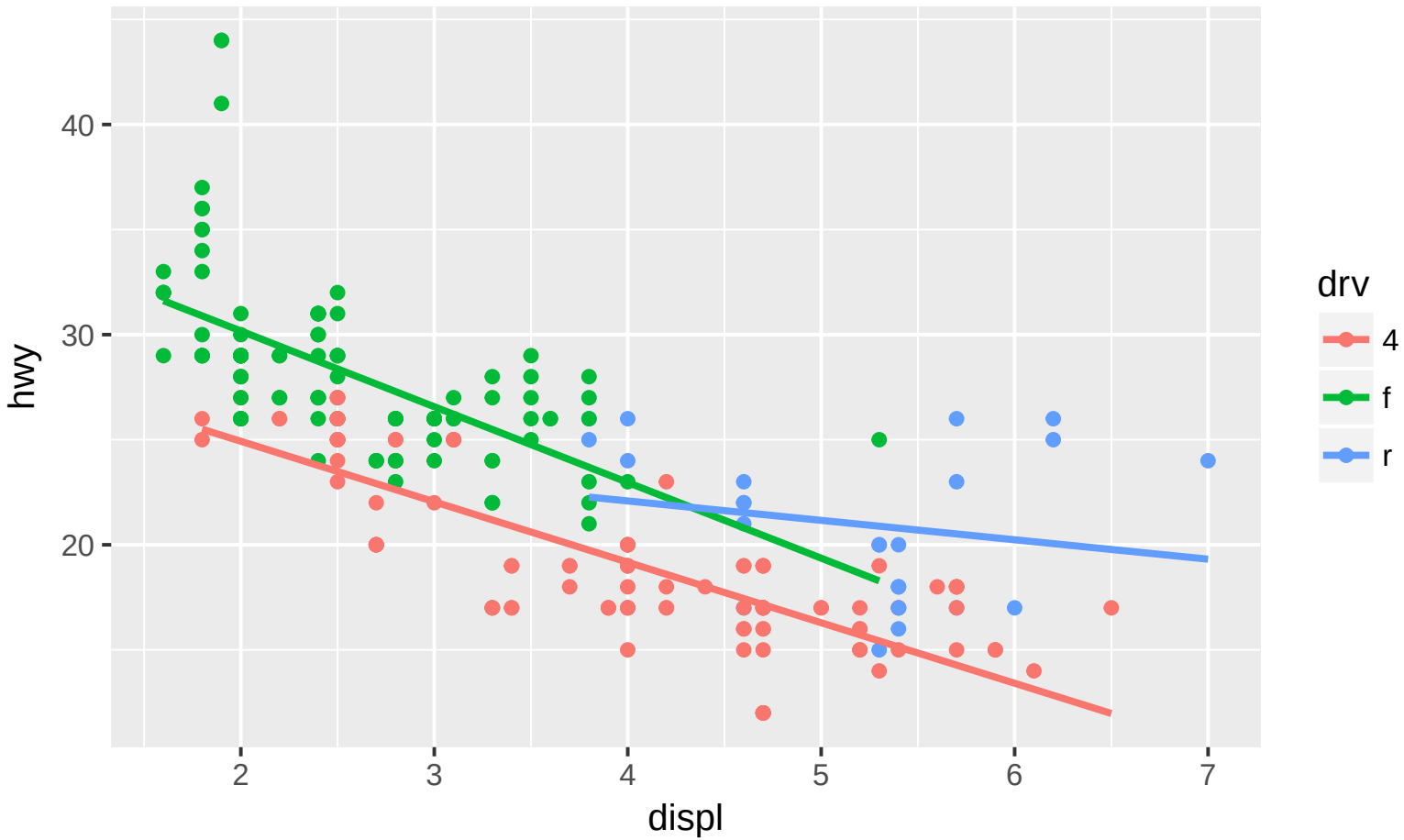
Combinaison de plusieurs graphiques

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE)
```



Combinaison de plusieurs graphiques

```
ggplot(mpg, aes(displ, hwy, colour = drv)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE)
```



Le fonctionnement en « couches » de ggplot2

La construction d'un graphique dans ggplot2 repose sur la superposition de couches (*layer*) **conçues indépendamment** mais **réconciliées en fin d'opération**.

Chaque couche est composée de cinq éléments :

- un `data.frame` (`data`) ;
- une ou plusieurs correspondances esthétiques (`mapping`) ;
- une transformation statistique (`stat`) ;
- un objet géométrique (`geom`) ;
- un paramètre d'ajustement de la position (`position`).

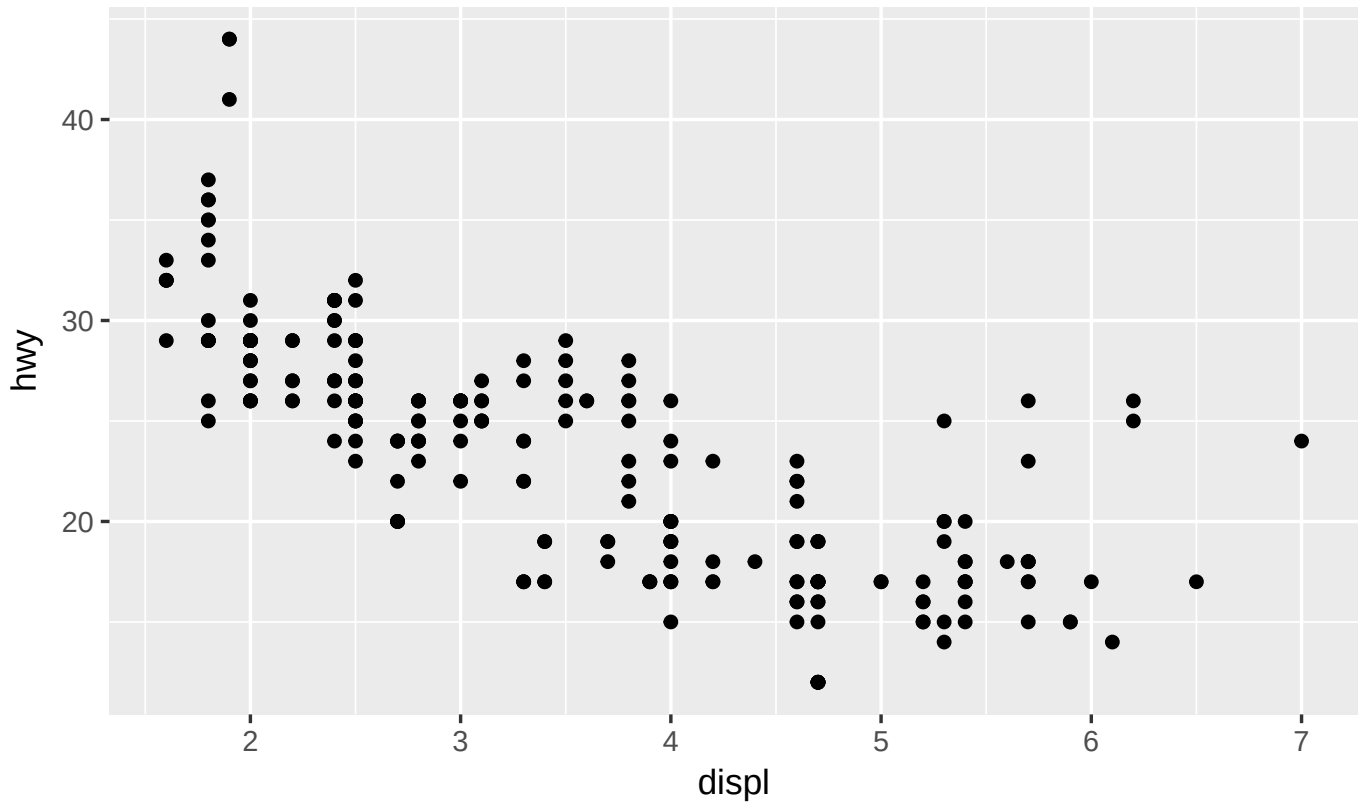
C'est la **fonction `layer()`** qui articule ces cinq éléments.

Les fonctions `geom_*` vues précédemment sont des appels pré-paramétrées de `layer()`.

Le fonctionnement en « couches » de ggplot2

Un graphique à une couche

```
ggplot() + layer(
  data = mpg, mapping = aes(displ, hwy), stat = "identity"
  , geom = "point", position = "identity"
)
```



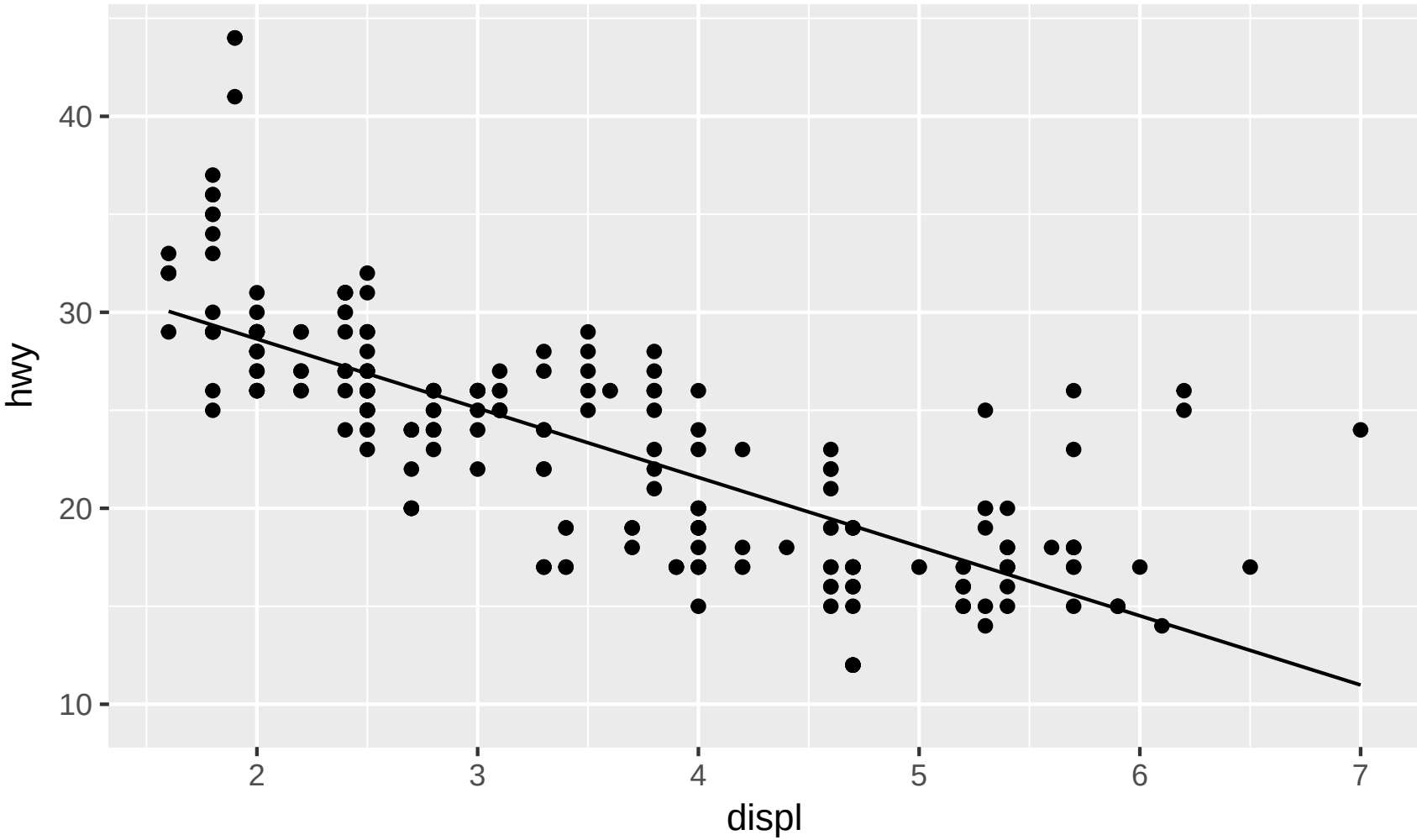
Le fonctionnement en « couches » de ggplot2

Un graphique à deux couches

```
ggplot() + layer(
  data = mpg, mapping = aes(displ, hwy), stat = "identity"
  , geom = "point", position = "identity"
) + layer(
  data = mpg, mapping = aes(displ, hwy), stat = "smooth"
  , geom = "line", position = "identity"
  , params = list(method = "lm", formula = y ~ x)
)
```

Le fonctionnement en « couches » de ggplot2

Un graphique à deux couches



Le fonctionnement en « couches » de ggplot2

Mise en facteur dans `ggplot()` de `data` et `mapping`

```
ggplot(data = mpg, mapping = aes(displ, hwy)) + layer(
  stat = "identity", geom = "point", position = "identity"
) + layer(
  stat = "smooth", geom = "line", position = "identity"
  , params = list(method = "lm", formula = y ~ x)
)
```

Remplacement de `layer()` par des alias pré-paramétrés

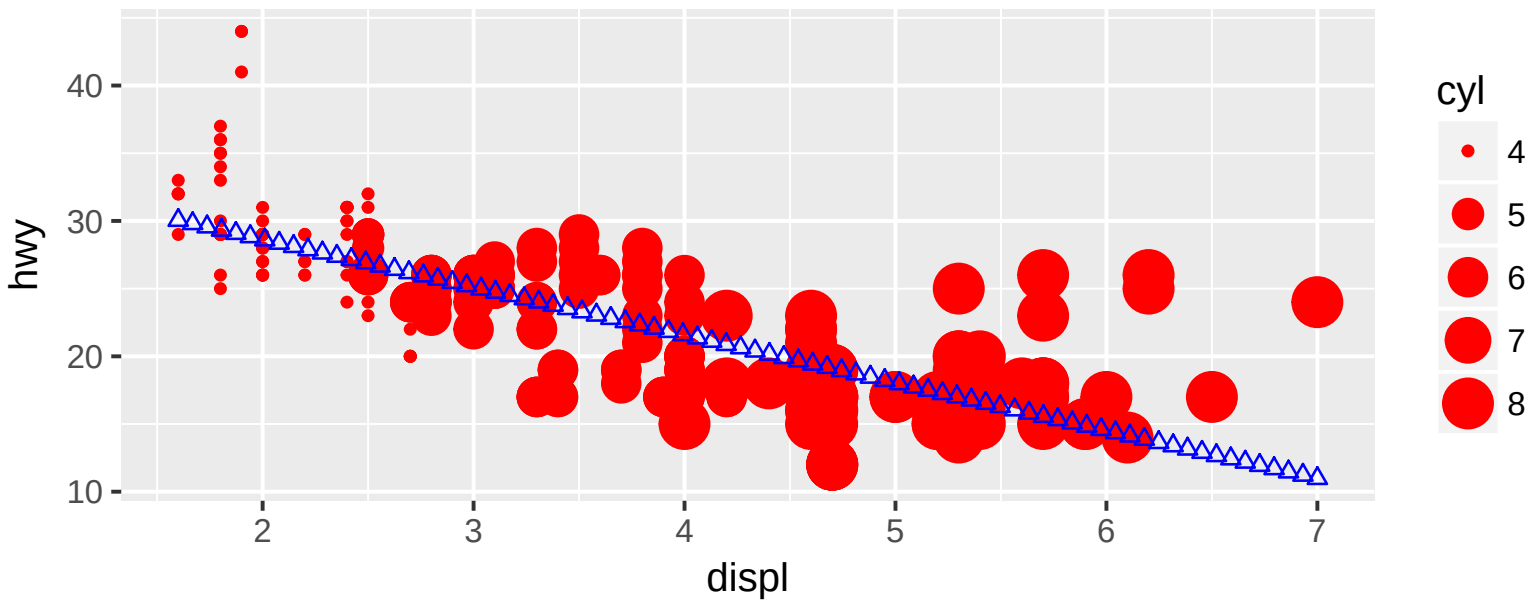
```
ggplot(data = mpg, mapping = aes(displ, hwy)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE)
```

```
ggplot(data = mpg, mapping = aes(displ, hwy)) +
  geom_point() + stat_smooth(method = "lm", se = FALSE)
```

Le fonctionnement en « couches » de ggplot2

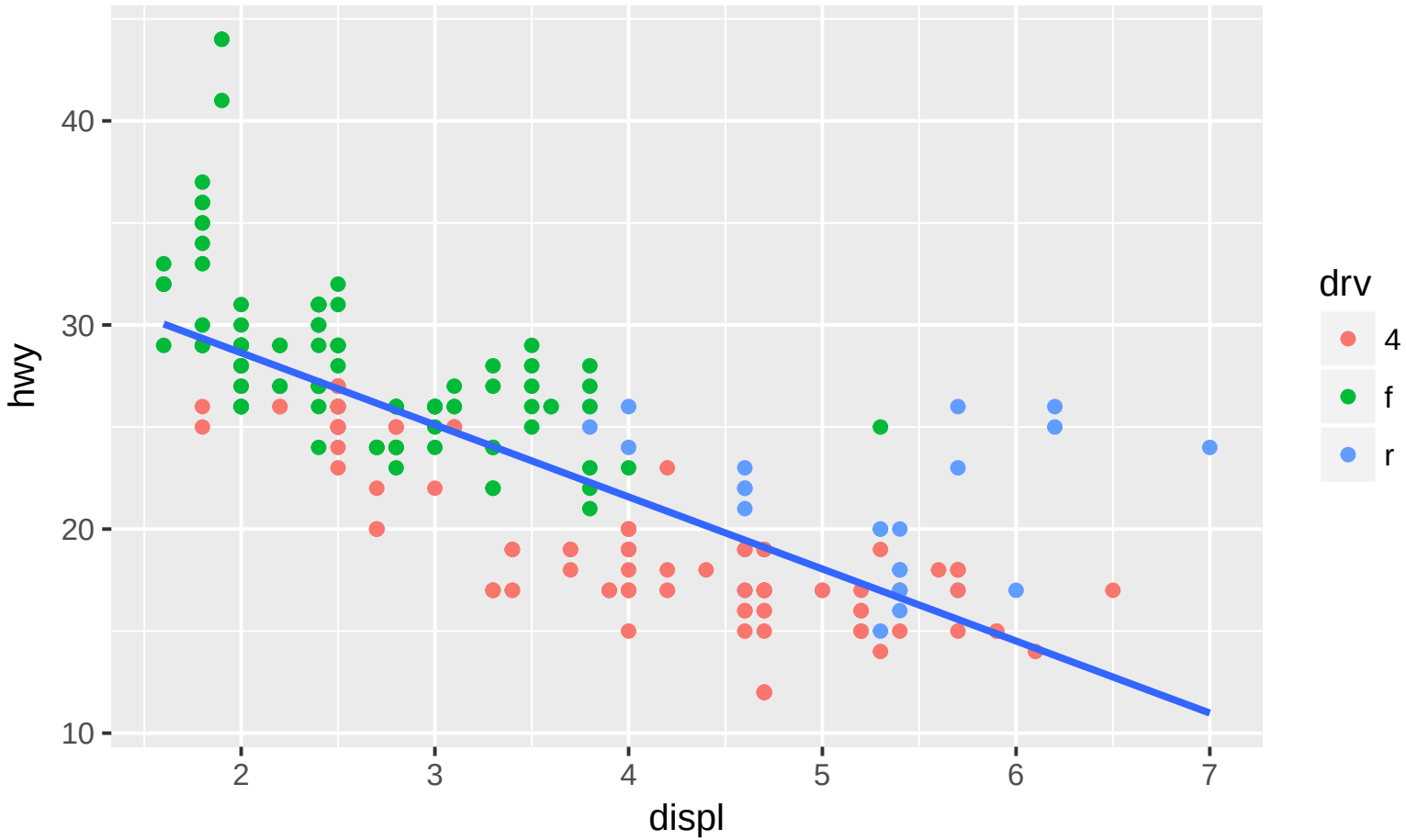
À chaque fonction `geom_*()` est associée un paramètre `stat` par défaut, et à chaque fonction `stat_*()` un `geom` par défaut.

```
ggplot(data = mpg, mapping = aes(displ, hwy)) +
  geom_point(colour = "red", aes(size = cyl)) +
  stat_smooth(geom = "point", method = "lm", se = FALSE
    , colour = "blue", shape = 2)
```



Le fonctionnement en « couches » de ggplot2

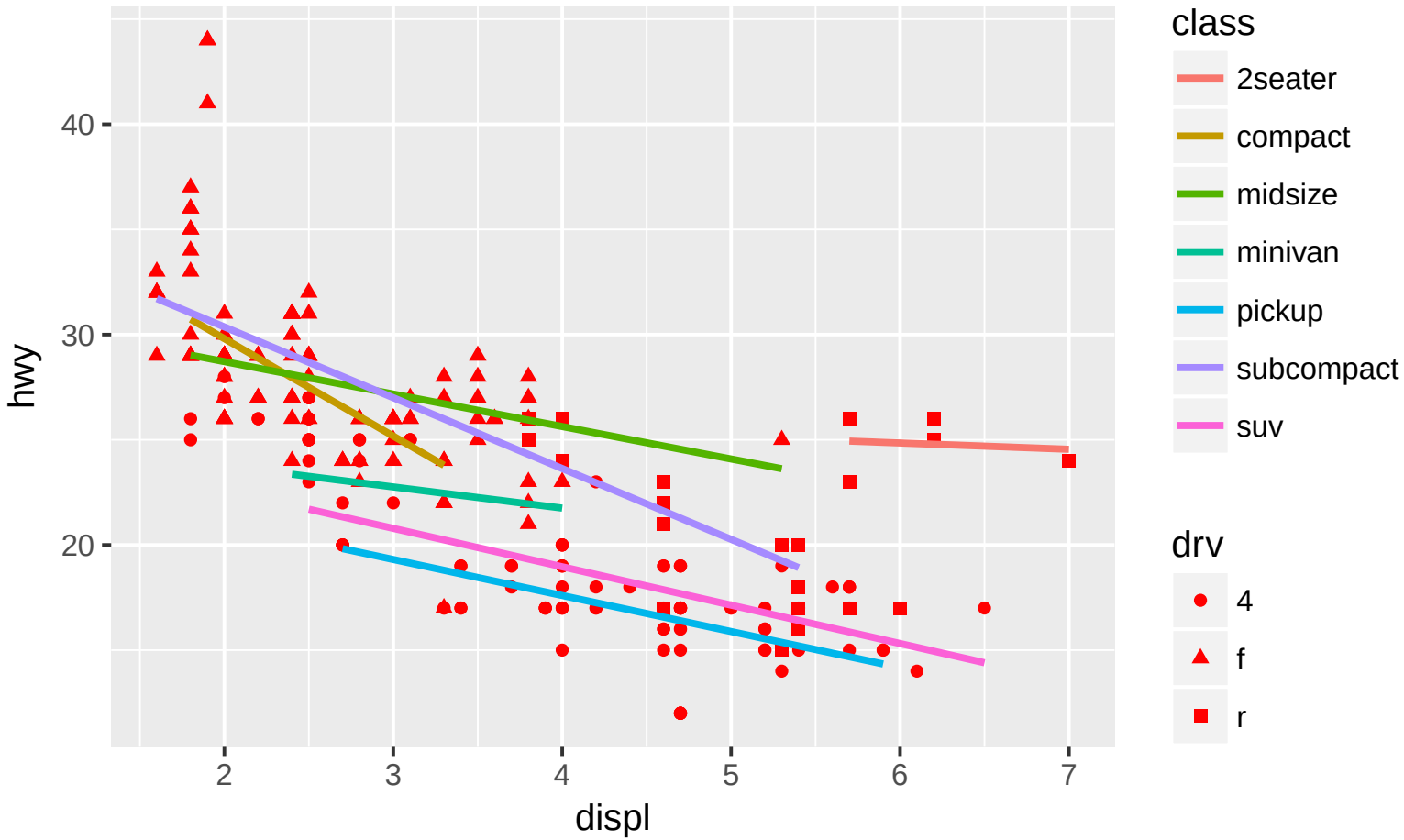
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = drv)) +
  stat_smooth(method = "lm", se = FALSE)
```



Réaliser des graphiques avec ggplot2

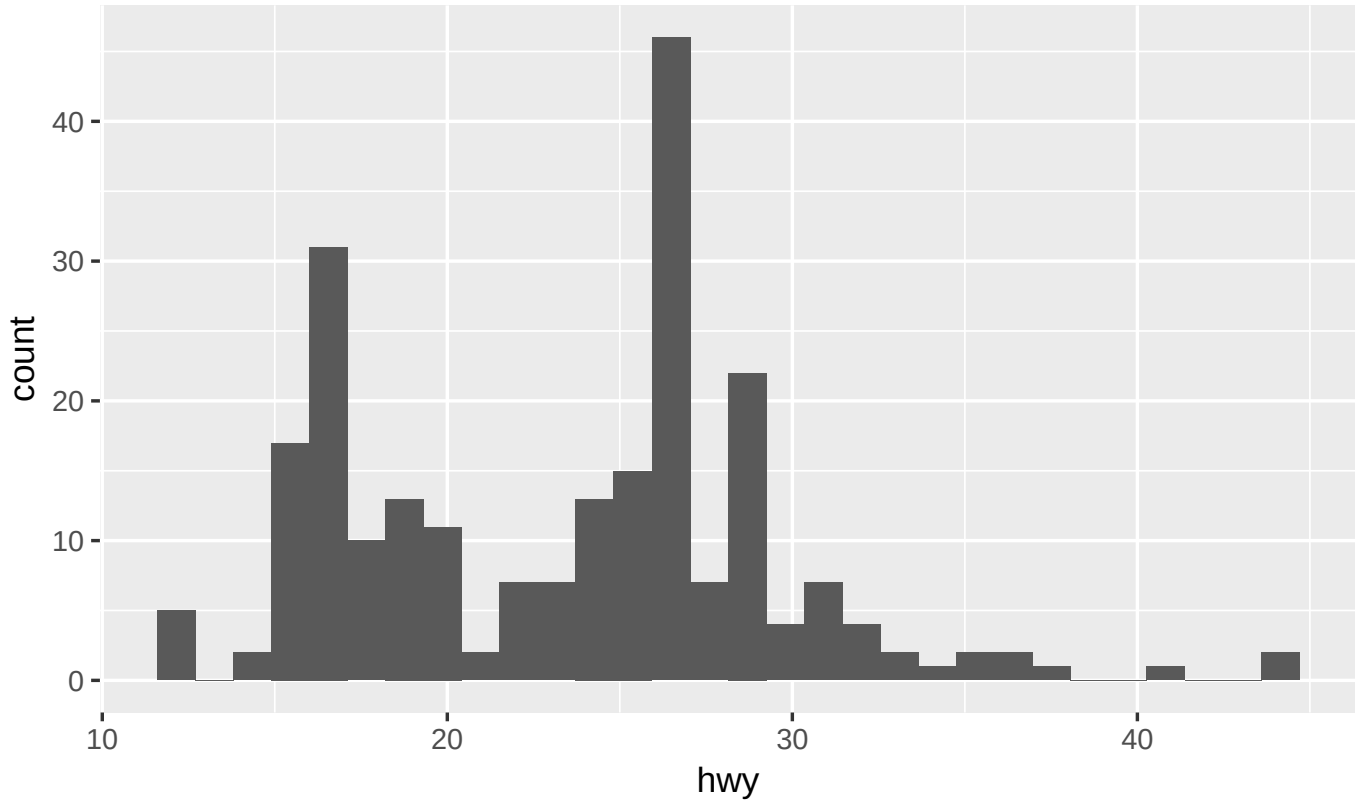
Le fonctionnement en « couches » de ggplot2

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(shape = drv), colour = "red") +
  stat_smooth(aes(colour = class), method = "lm", se = FALSE)
```



Histogrammes et densités

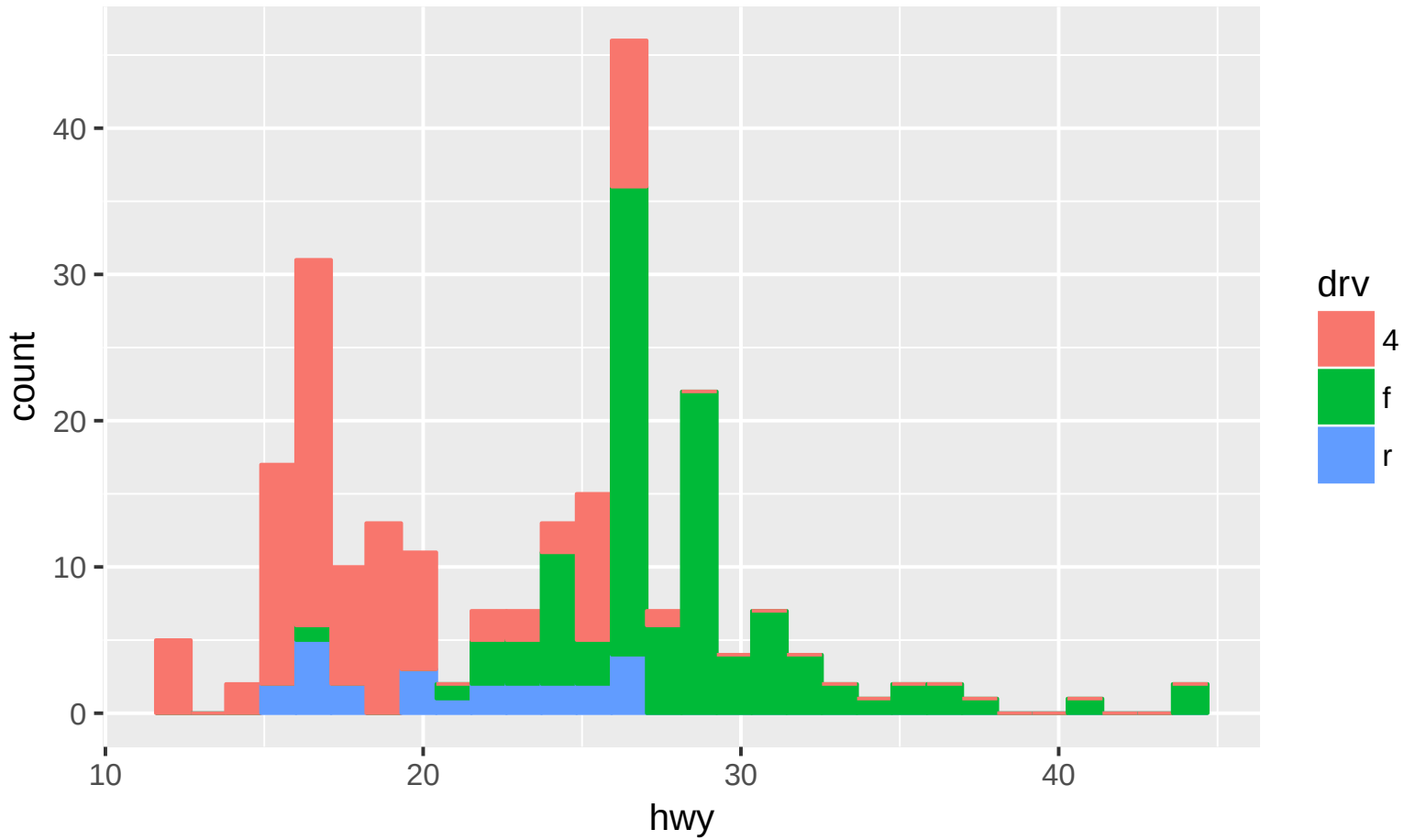
```
ggplot(mpg, aes(hwy)) + geom_histogram()
```



Remarque Le positionnement des classes des histogrammes semble perturbé dans les dernières versions de `ggplot2` : le paramètre `boundary` permet de corriger ce problème (*cf.* cette discussion).

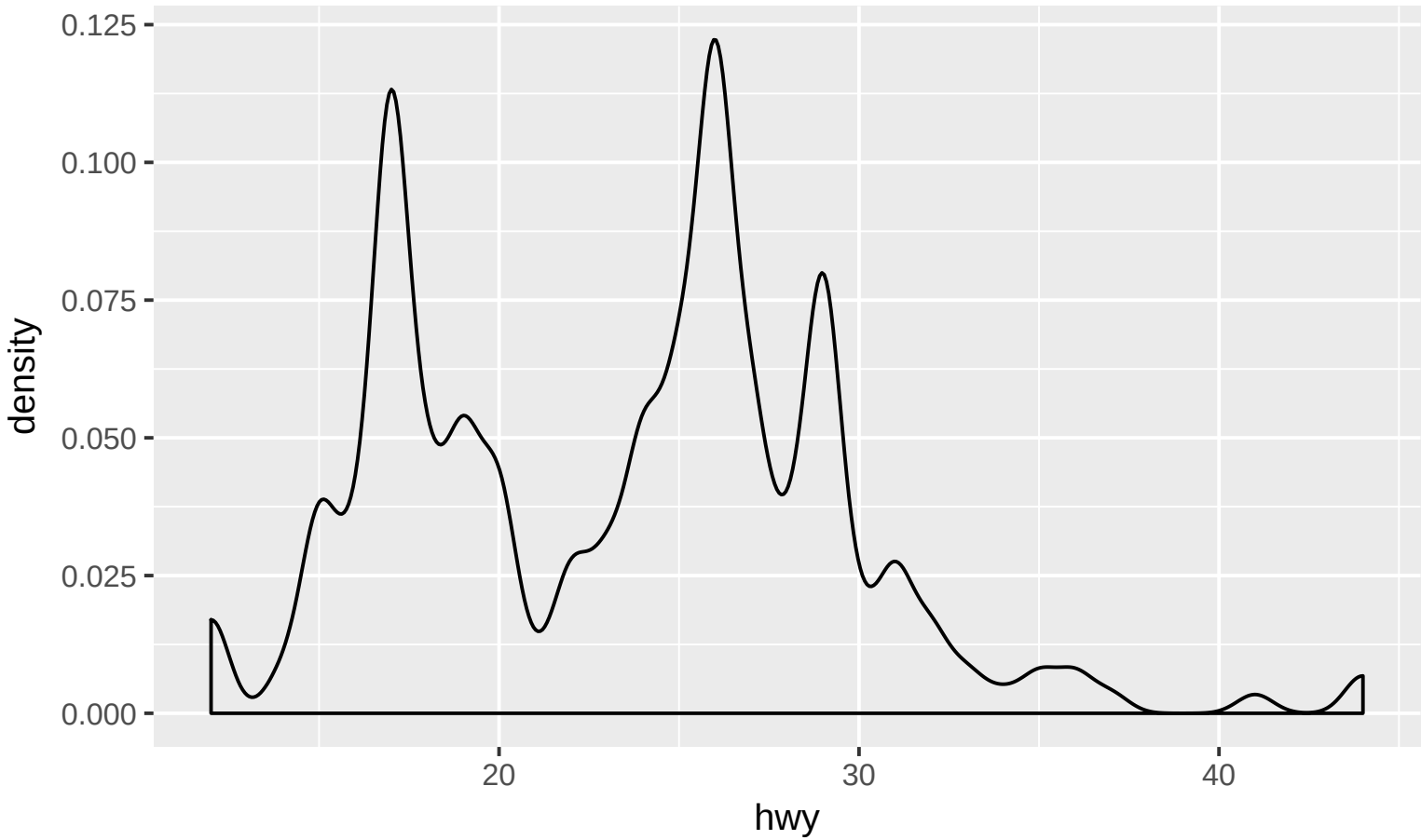
Histogrammes et densités

```
ggplot(mpg, aes(hwy, colour = drv, fill = drv)) +
  geom_histogram()
```



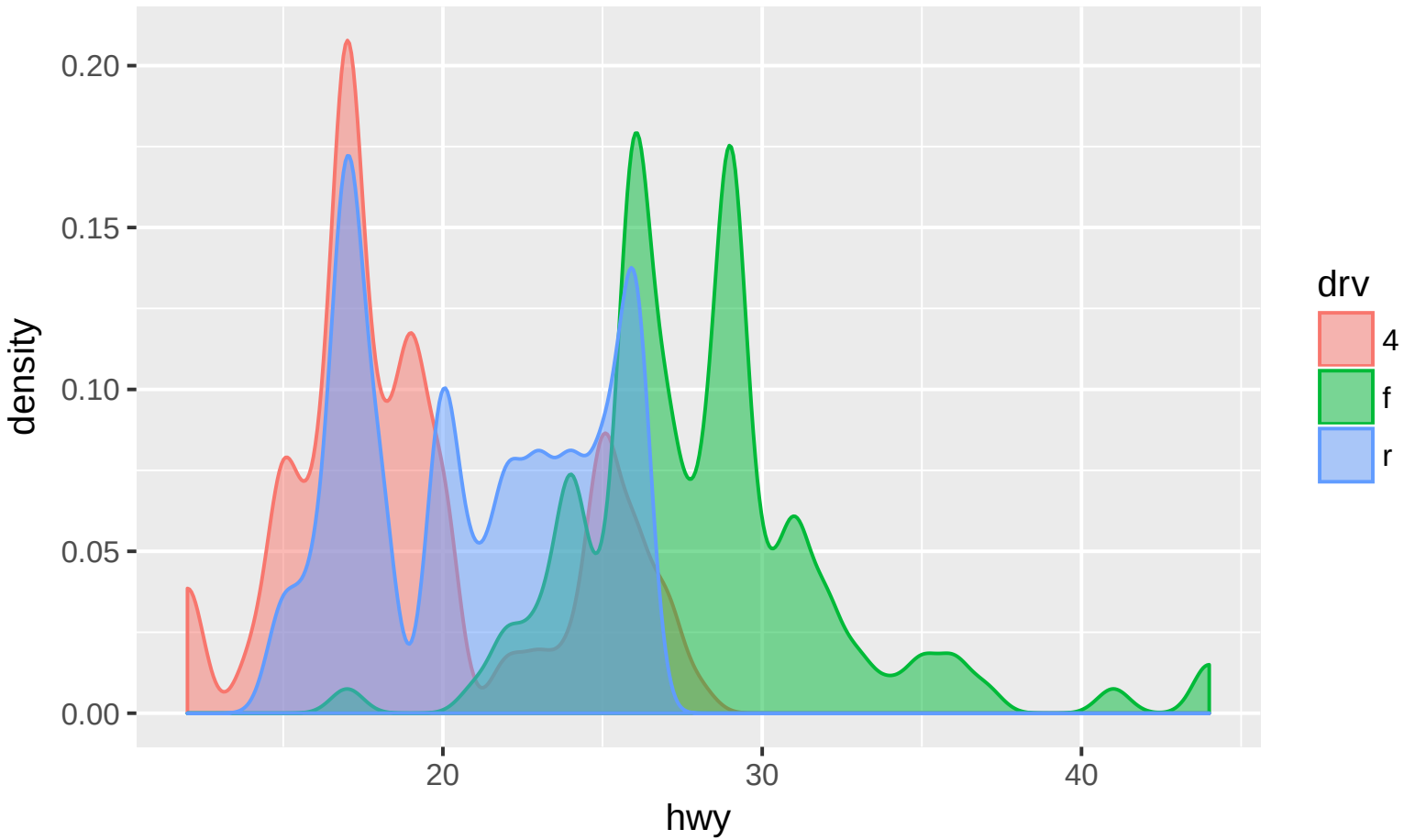
Histogrammes et densités

```
ggplot(mpg, aes(hwy)) + geom_density(bw = 0.5)
```



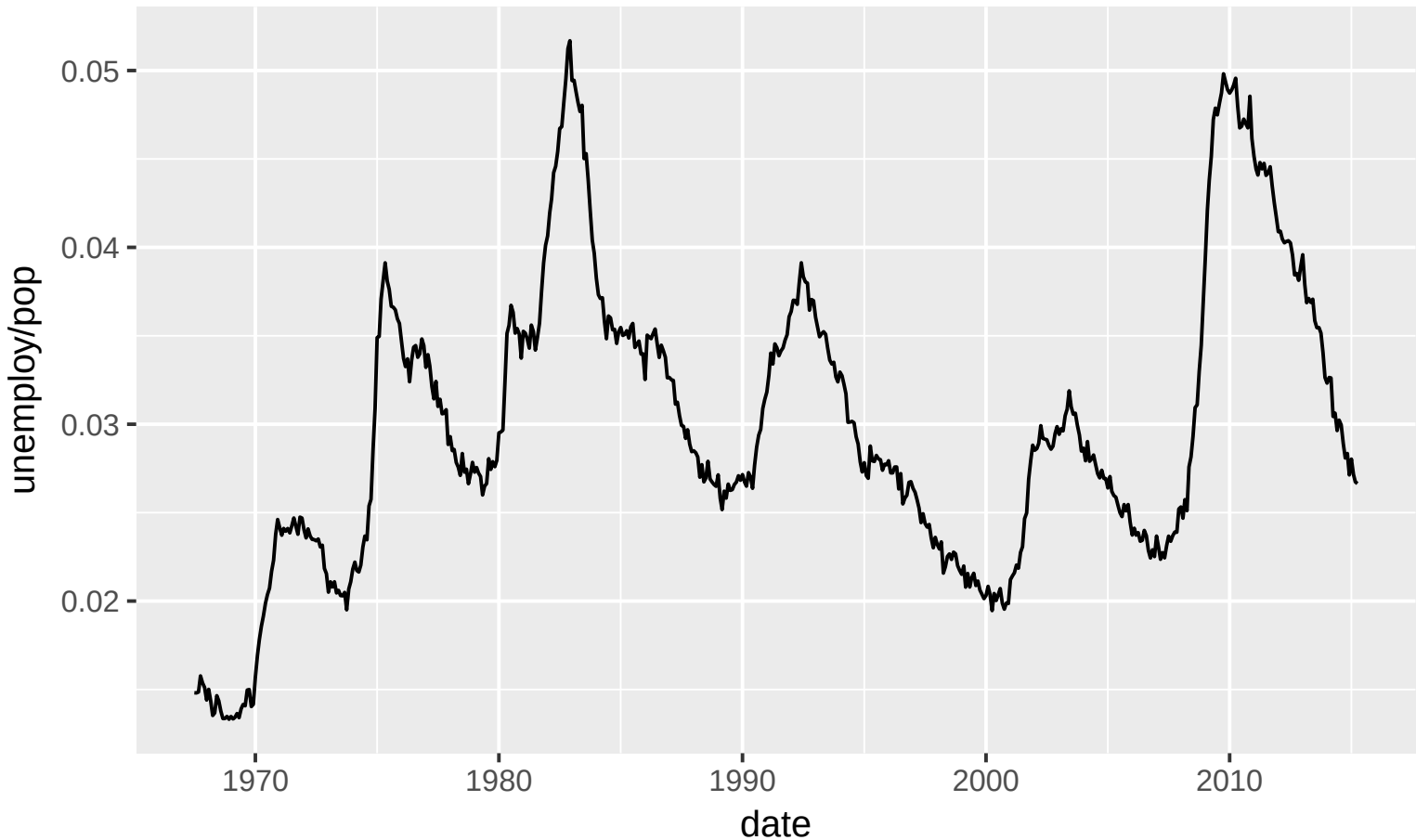
Histogrammes et densités

```
ggplot(mpg, aes(hwy, colour = drv, fill = drv)) +
  geom_density(bw = 0.5, alpha = 0.5)
```



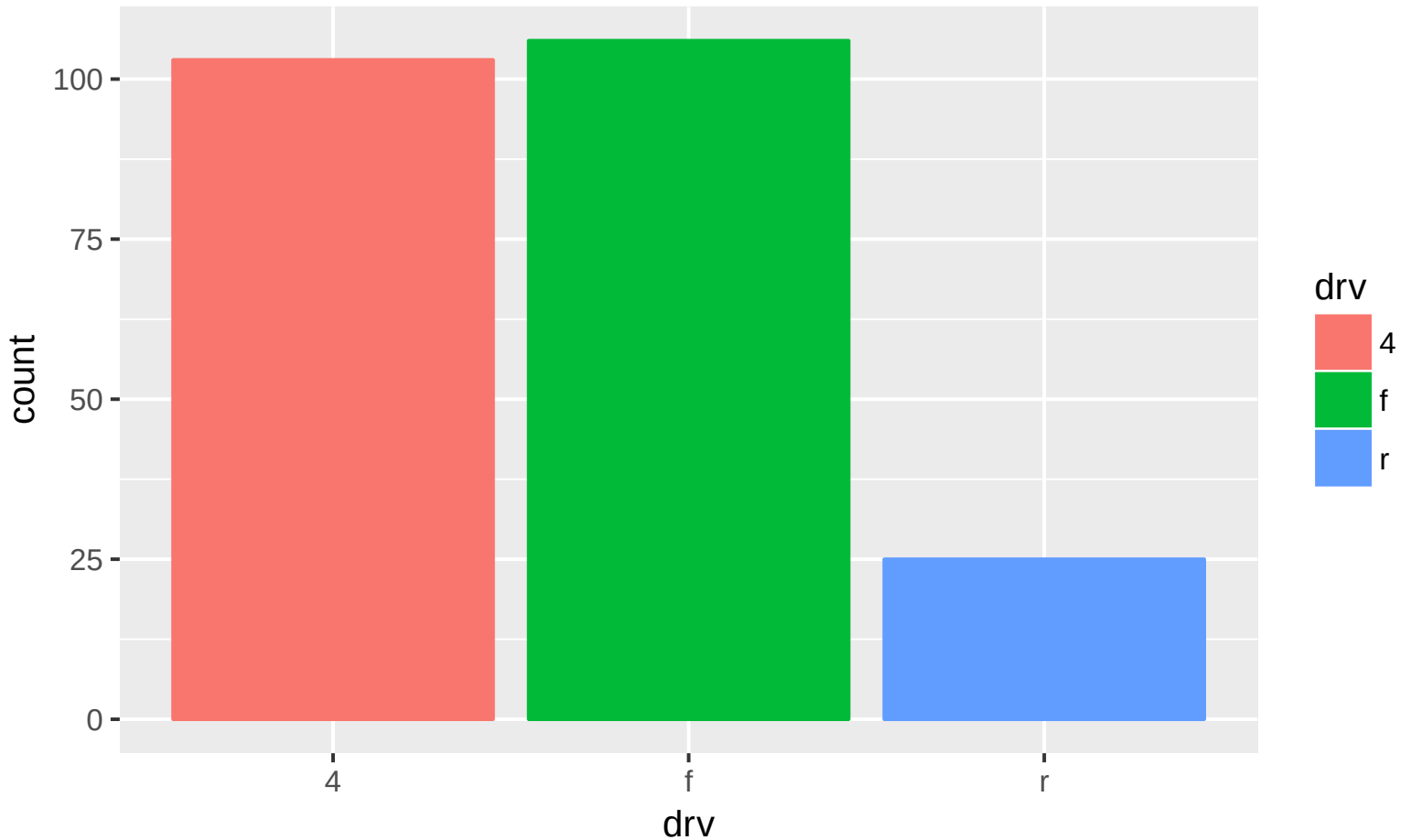
Séries temporelles

```
ggplot(economics, aes(date, unemploy / pop)) +
  geom_line()
```



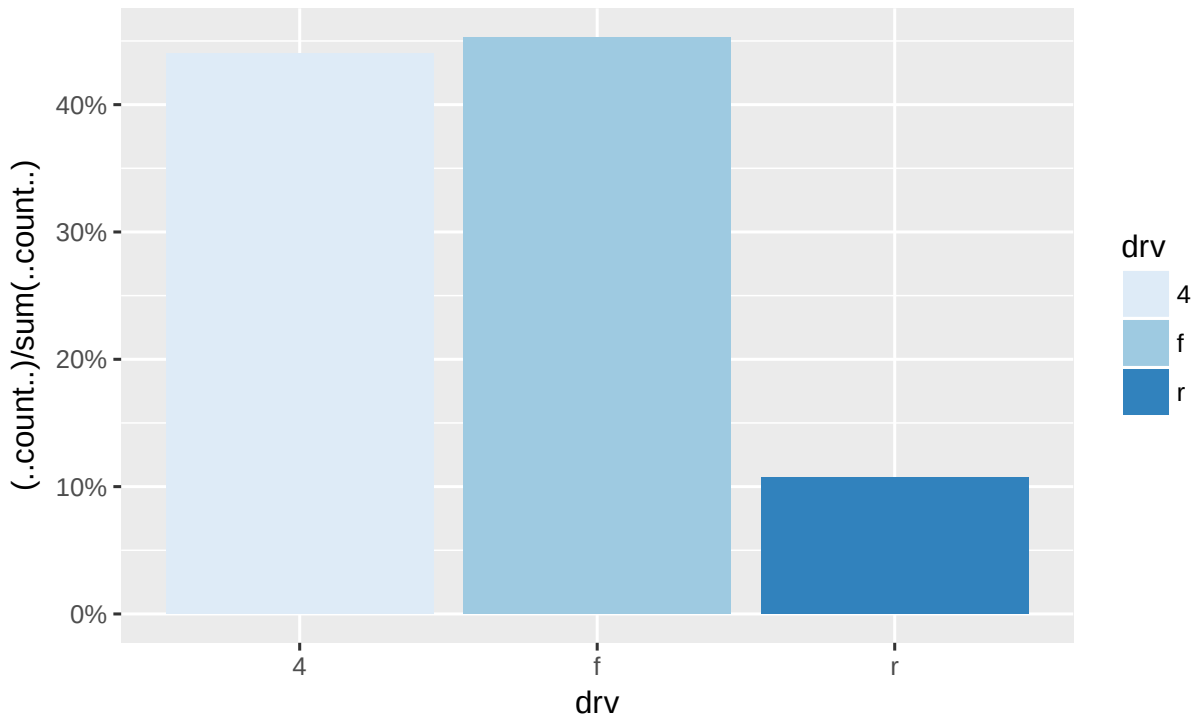
Diagrammes en bâtons et circulaires

```
ggplot(mpg, aes(drv, colour = drv, fill = drv)) +
  geom_bar()
```



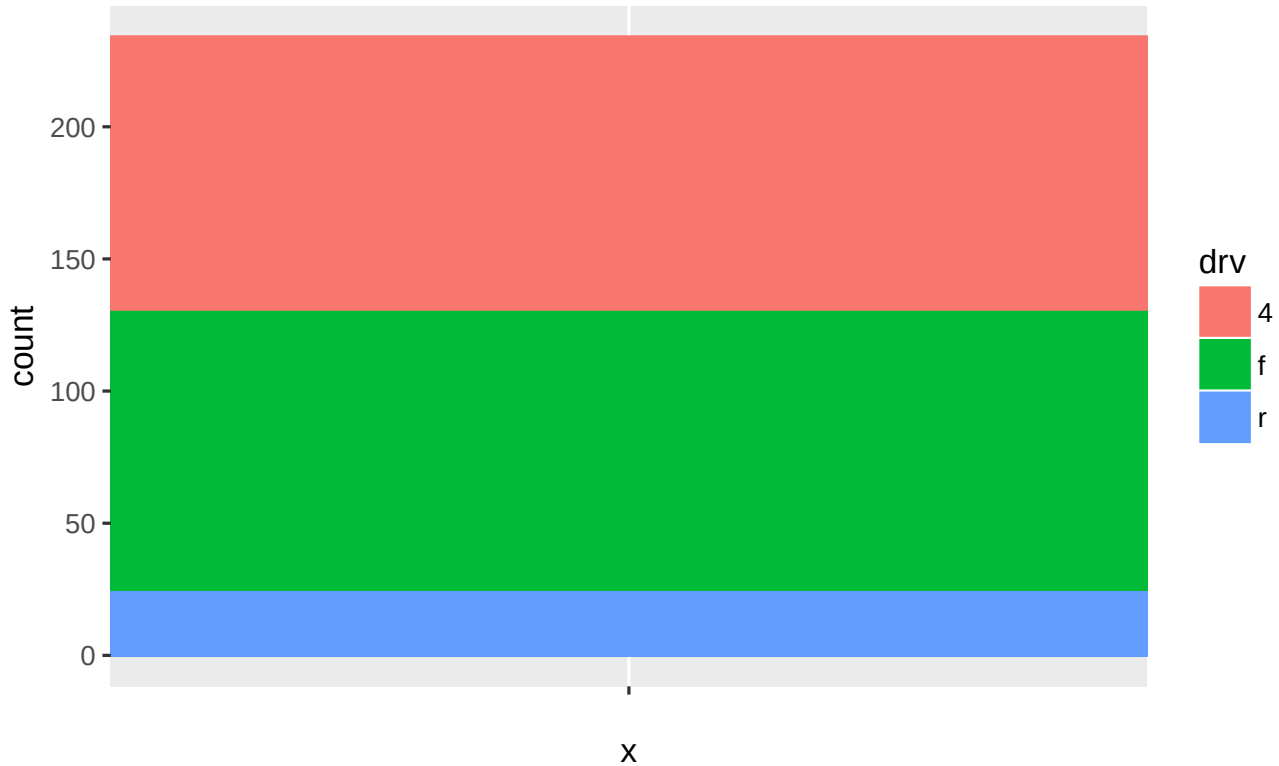
Diagrammes en bâtons et circulaires

```
library(scales)
ggplot(mpg, aes(drv, fill = drv)) +
  geom_bar(aes(y = (..count..)/sum(..count..))) +
  scale_y_continuous(labels=percent) +
  scale_fill_brewer(palette="Blues")
```



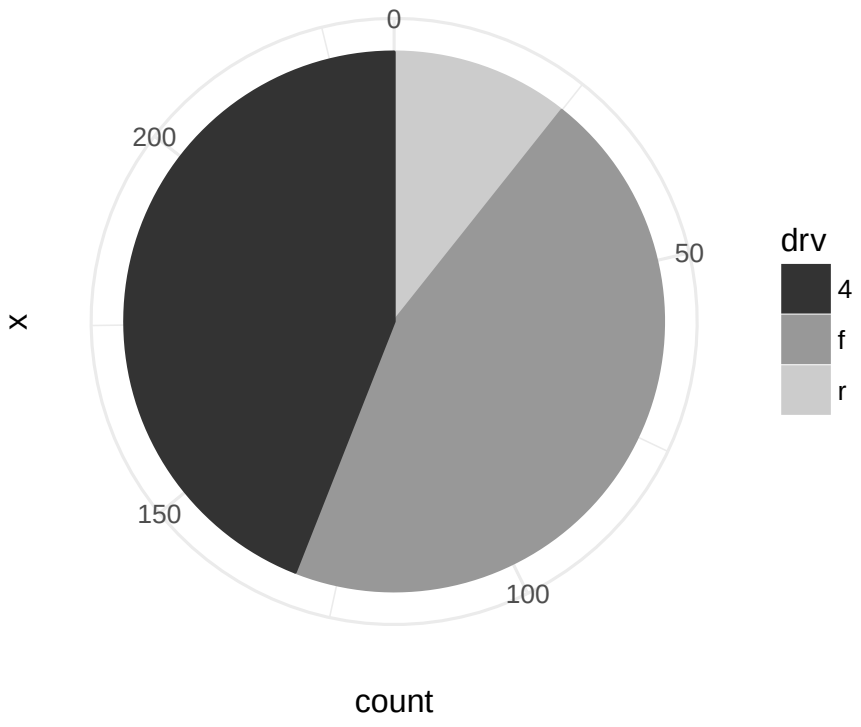
Diagrammes en bâtons et circulaires

```
g <- ggplot(mpg, aes(x = "", fill = drv, colour = drv)) +
  geom_bar(width = 1)
g
```



Diagrammes en bâtons et circulaires

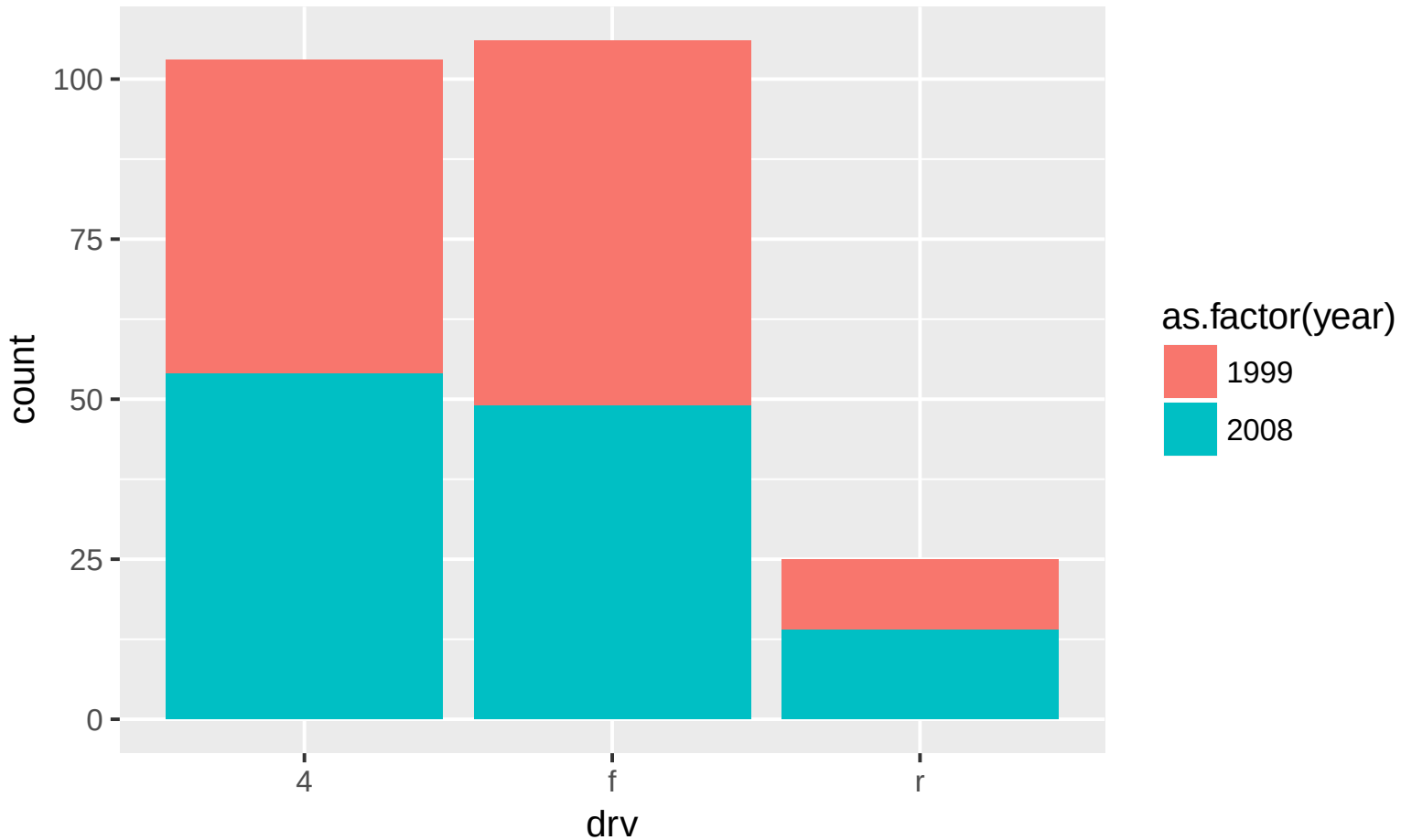
```
g + coord_polar(theta = "y") + theme_minimal() +
  scale_fill_grey() + scale_colour_grey()
```



Pour aller plus loin Une page du site sthda.com explique (en français) comment produire un diagramme circulaire complet avec ggplot2.

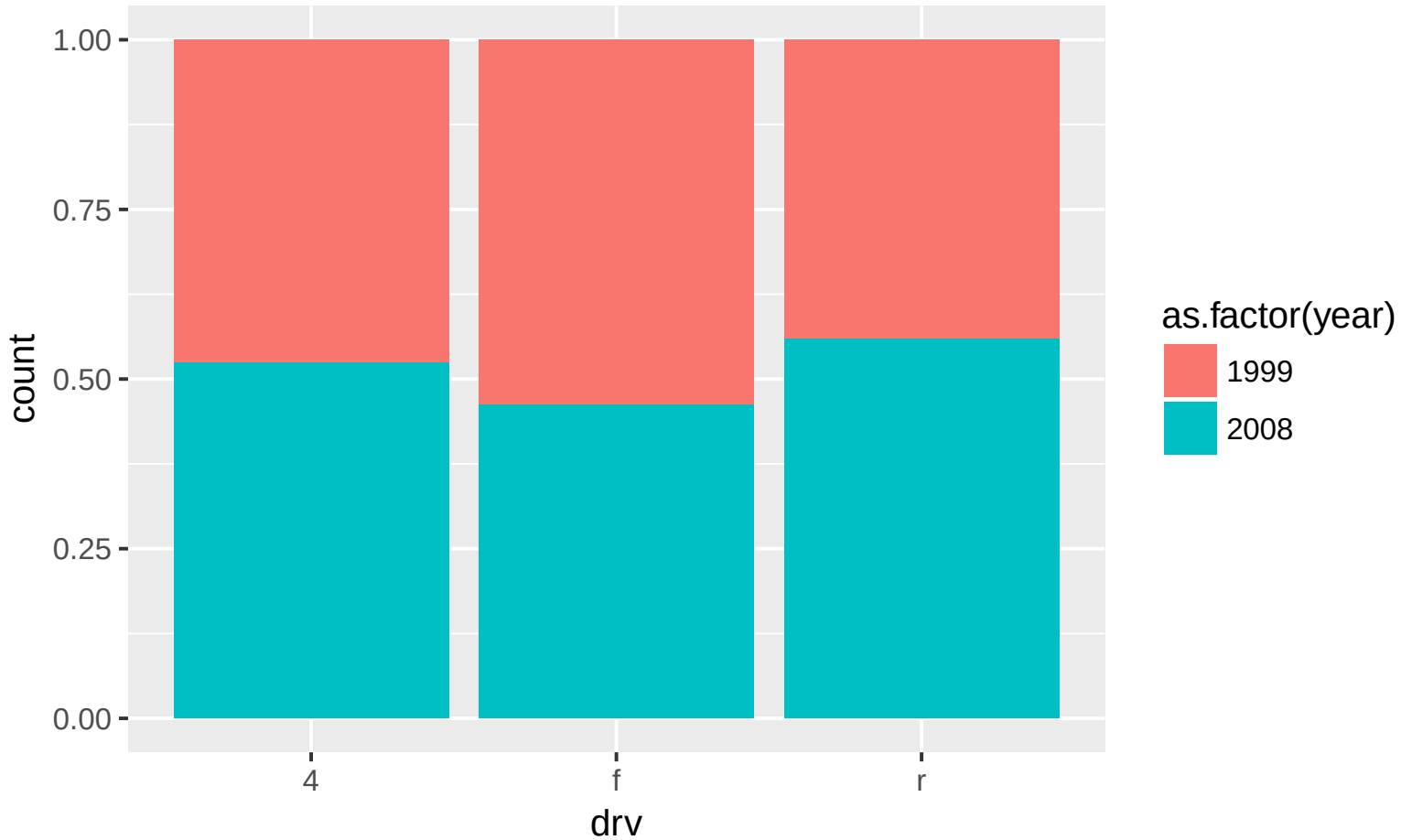
Diagrammes en bâtons et circulaires

```
ggplot(mpg, aes(drv, fill = as.factor(year))) +
  geom_bar()
```



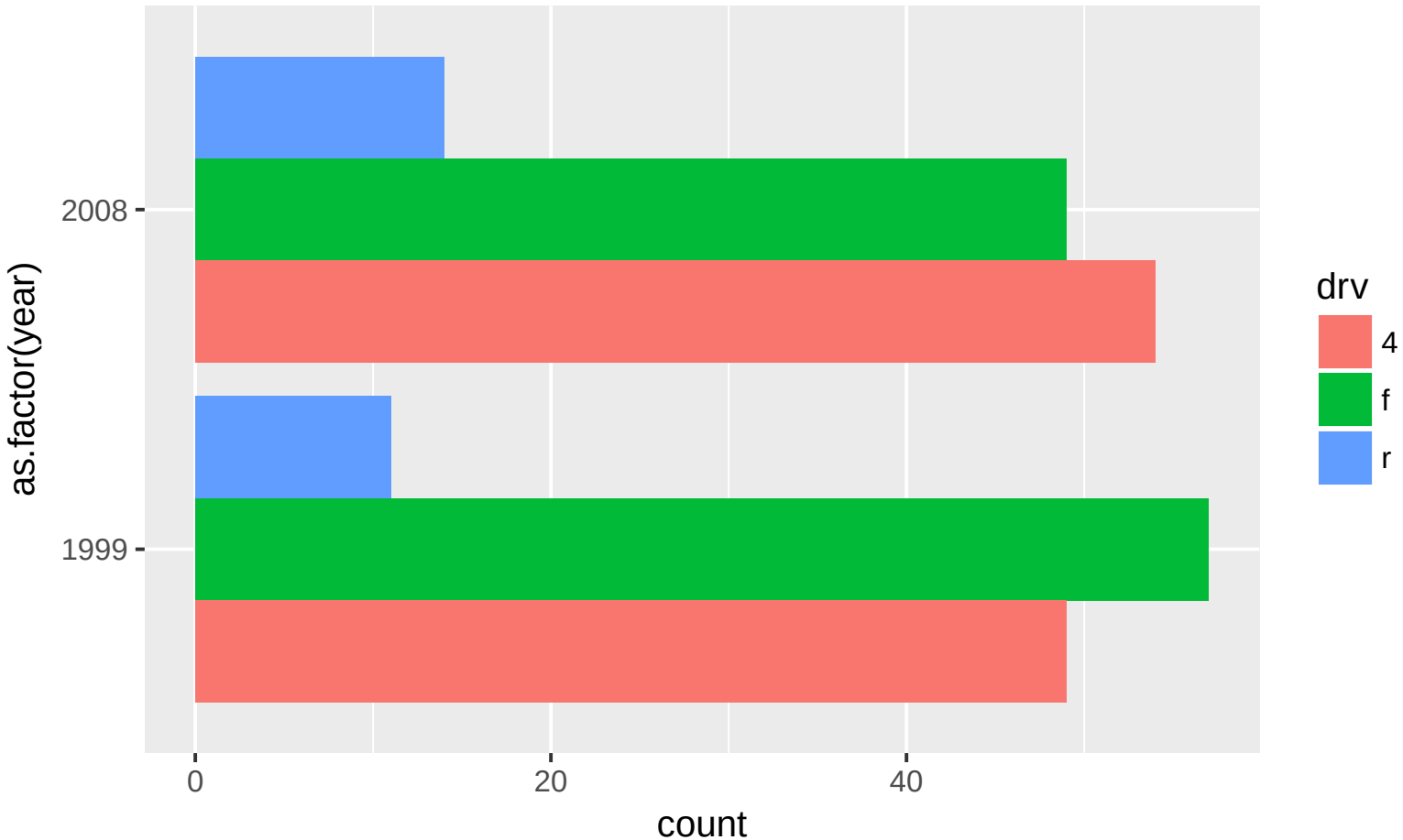
Diagrammes en bâtons et circulaires

```
ggplot(mpg, aes(drv, fill = as.factor(year))) +
  geom_bar(position = "fill")
```



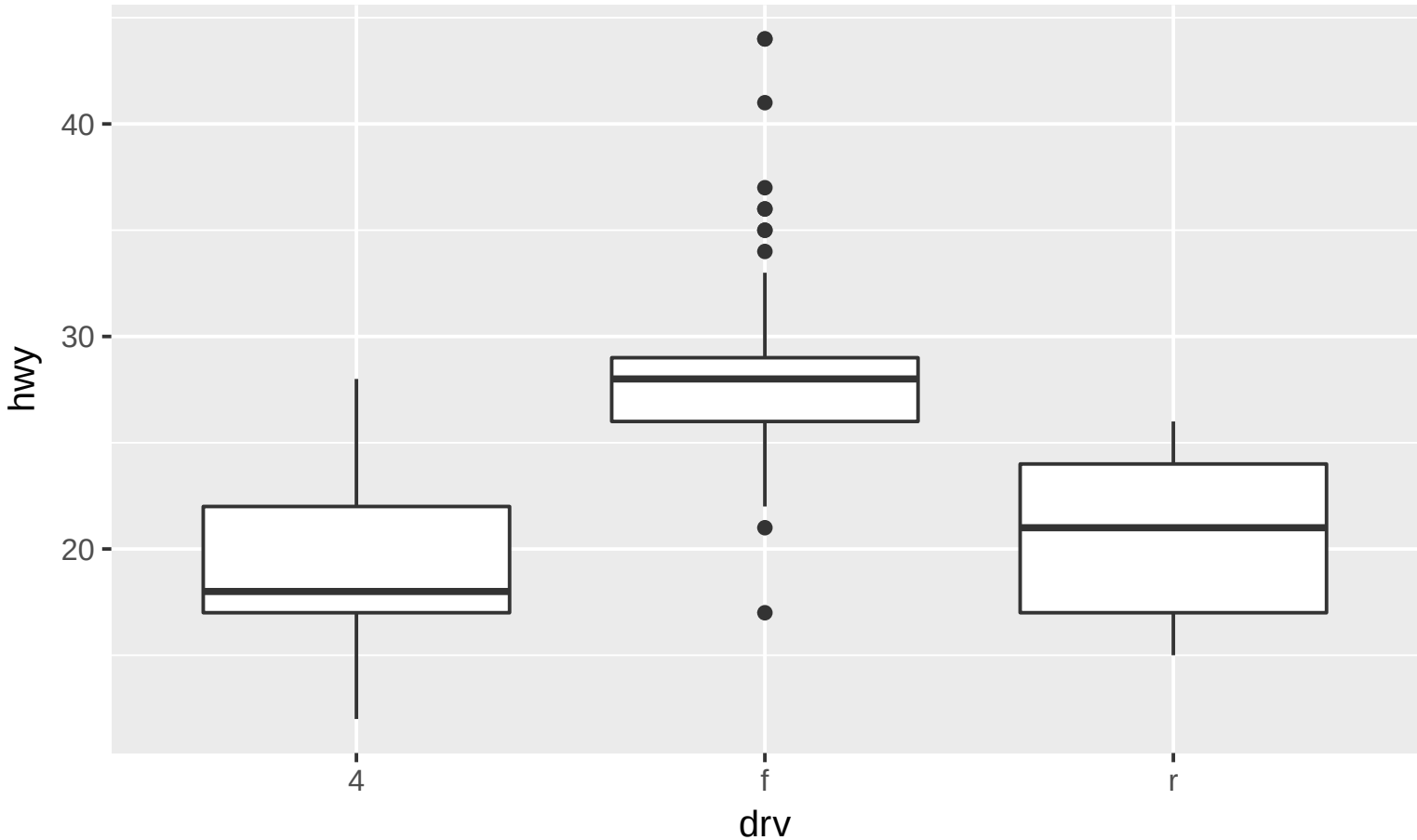
Diagrammes en bâtons et circulaires

```
ggplot(mpg, aes(as.factor(year), fill = drv)) +
  geom_bar(position = "dodge") +
  coord_flip()
```



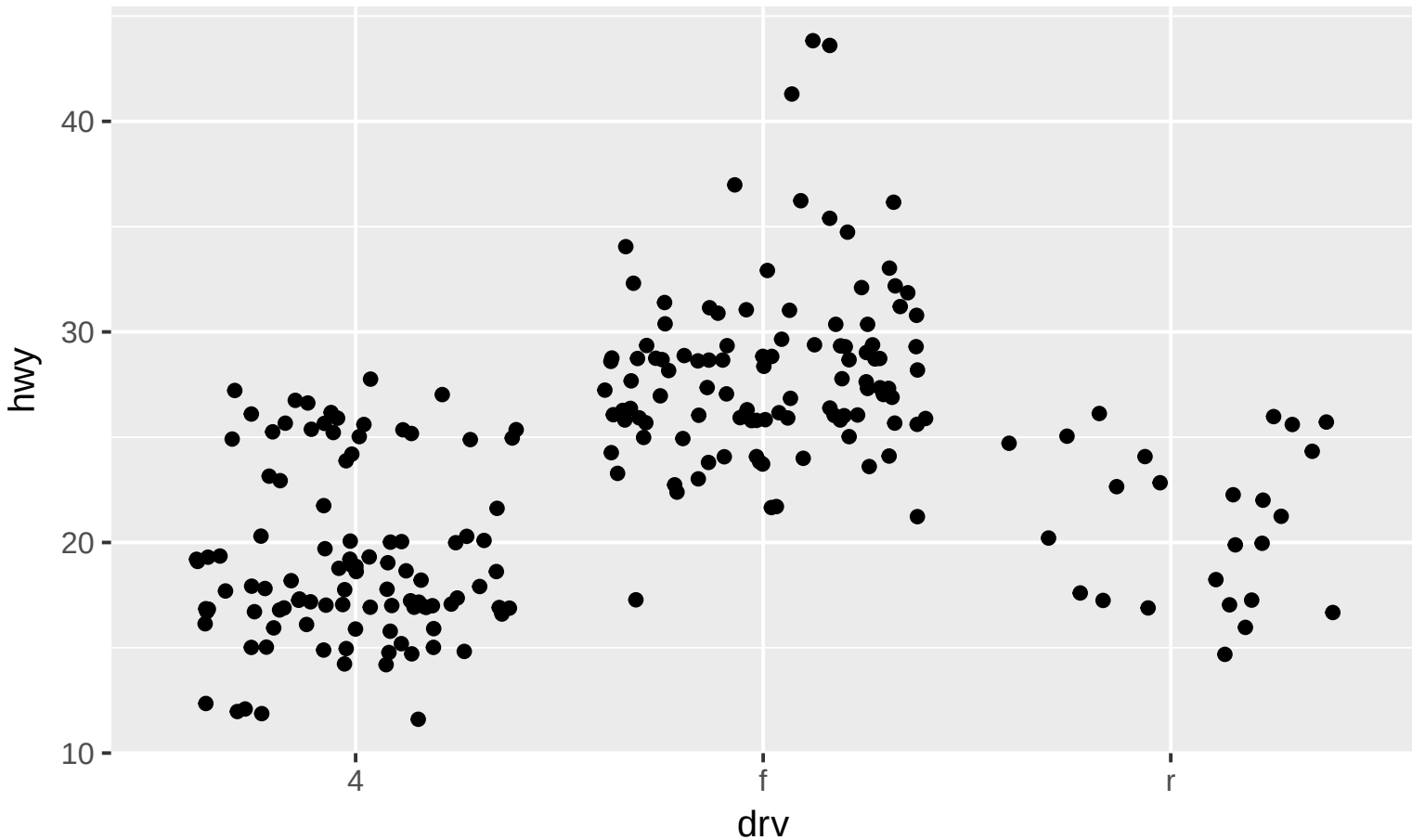
Boîtes à moustaches et assimilés

```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_boxplot(coef = 1.5)
```



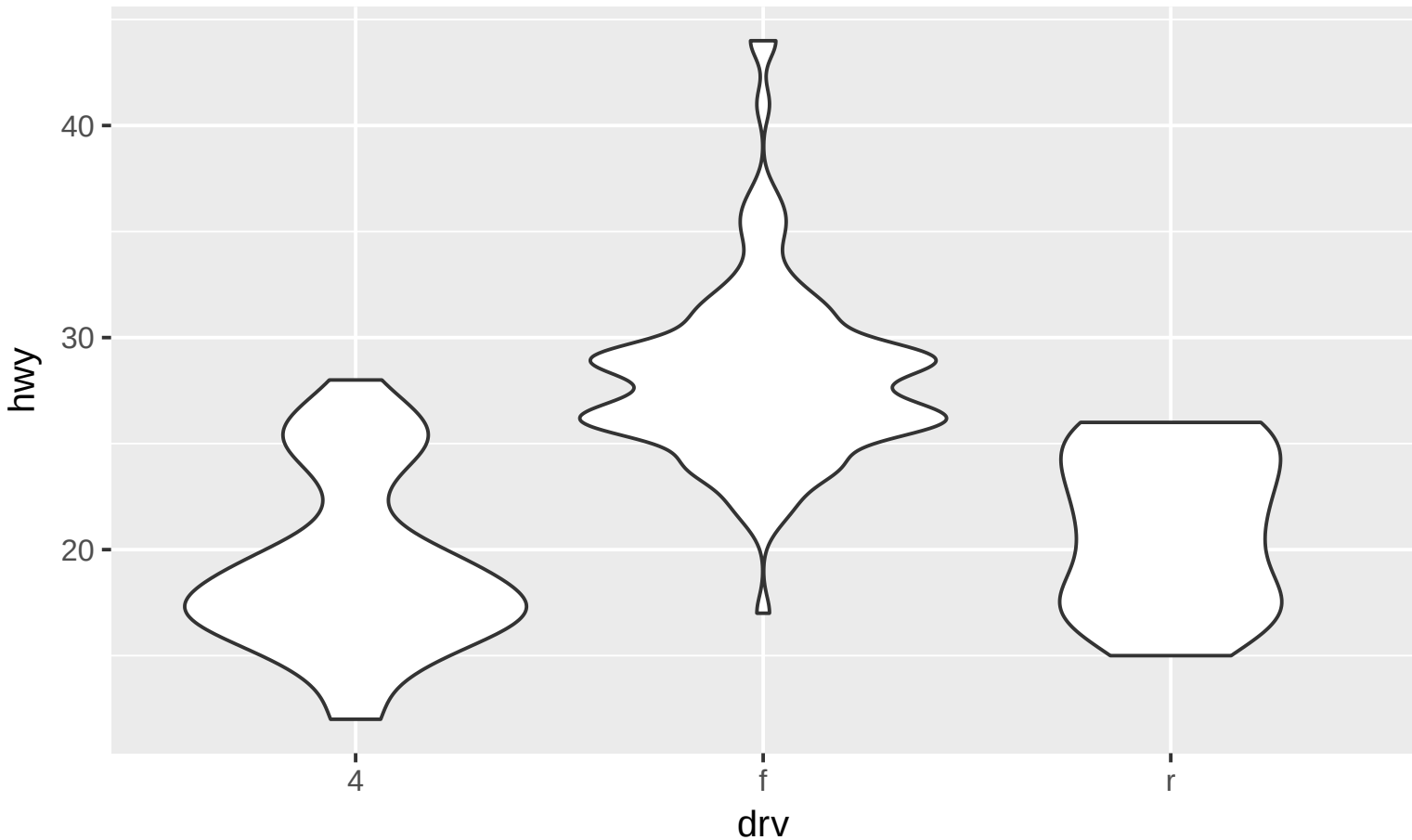
Boîtes à moustaches et assimilés

```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_jitter()
```



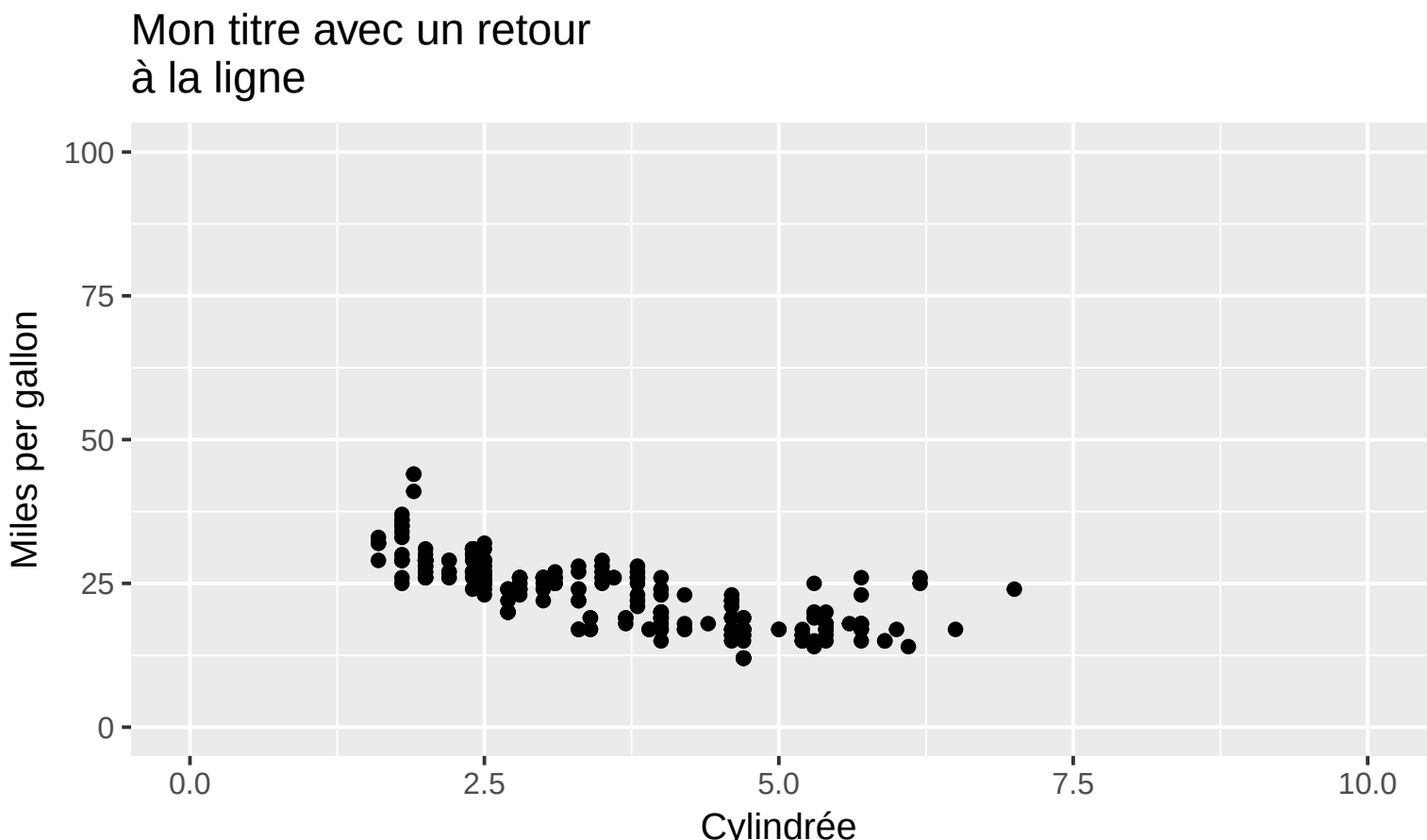
Boîtes à moustaches et assimilés

```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_violin()
```



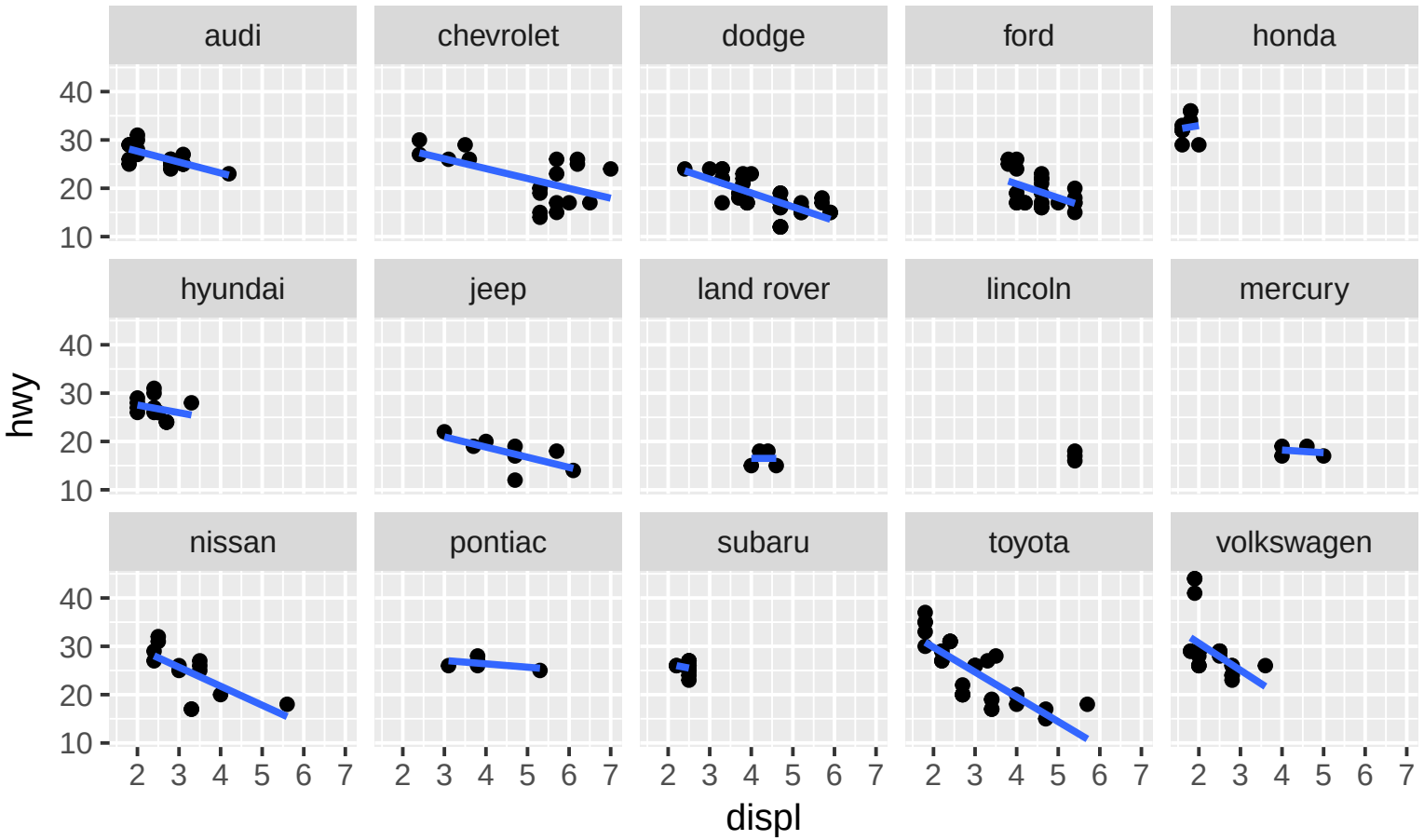
Titres et axes

```
ggplot(mpg, aes(displ, hwy)) + geom_point() +
  ggtitle("Mon titre avec un retour \nà la ligne") +
  xlab("Cylindrée") + ylab("Miles per gallon") +
  coord_cartesian(xlim = c(0,10), ylim = c(0, 100))
```



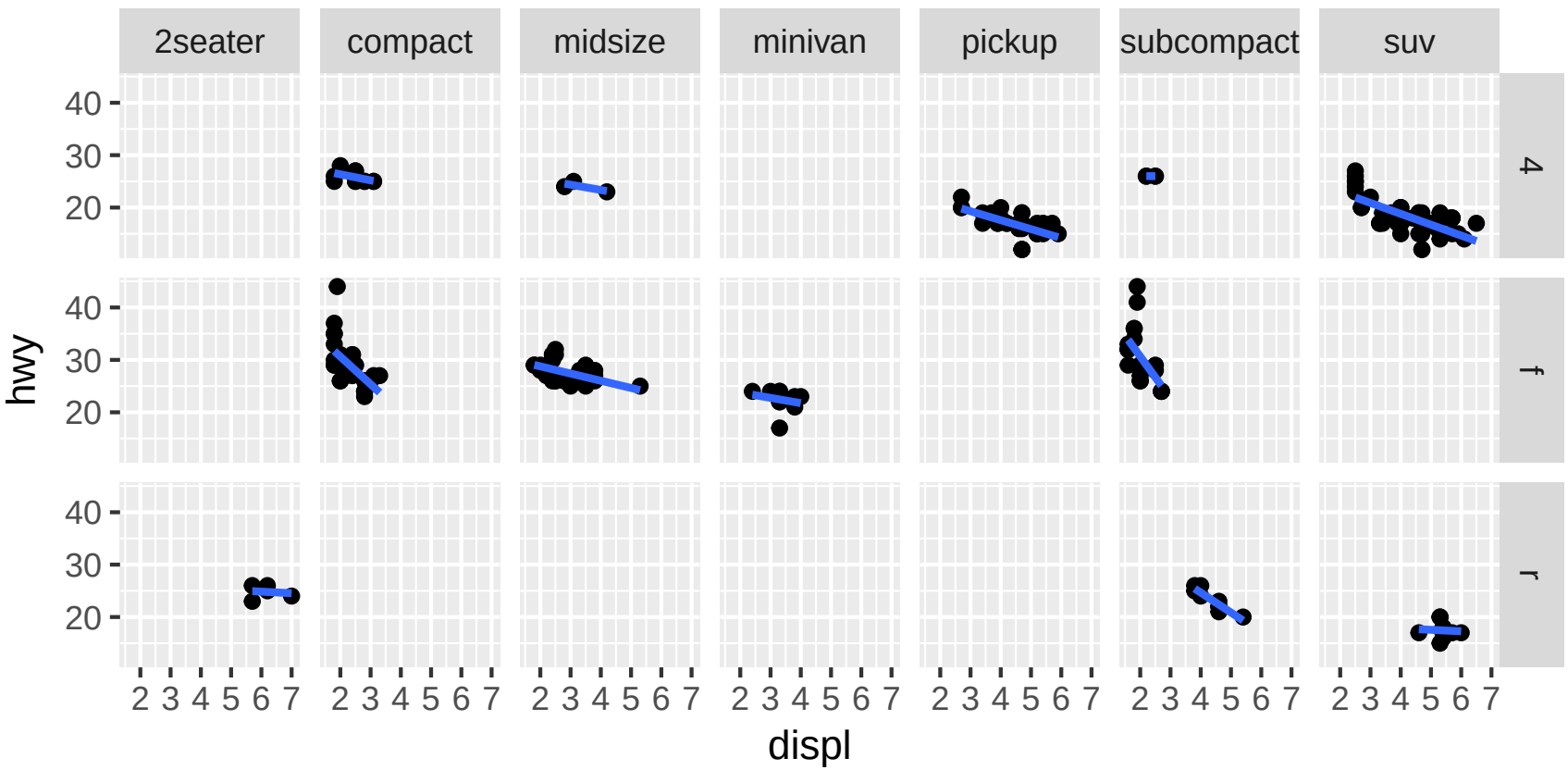
Disposition : le *facetting*

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~manufacturer, nrow = 3)
```



Disposition : le *facetting*

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE) +
  facet_grid(drv~class)
```



Sauvegarde et exportation

Le résultat de la fonction `ggplot()` pouvant être stocké dans un objet R, il est possible de le sauvegarder tel quel avec `save()` ou `saveRDS()` et de le réutiliser par la suite dans R.

```
g <- ggplot(mpg, aes(displ, hwy)) + geom_point()
saveRDS(g, file = "g.rds")
```

La fonction `ggsave()` simplifie l'export de graphiques en dehors de R. Par défaut, elle sauvegarde le dernier graphique produit.

```
g + geom_smooth(method = "lm", se = FALSE)
ggsave("monGraphique.pdf")
ggsave("monGraphique.png")
```

Générer automatiquement des documents depuis R

Pourquoi générer automatiquement des documents ?

- Exporter et documenter des **traitements** en vue d'une réutilisation future : statistiques pour une étude, traitements réalisés lors d'une réunion de travail, etc.

  Remarque Utilisation analogue à celle permise par les instructions `ODS RTF` ou `ODS PDF` de SAS.

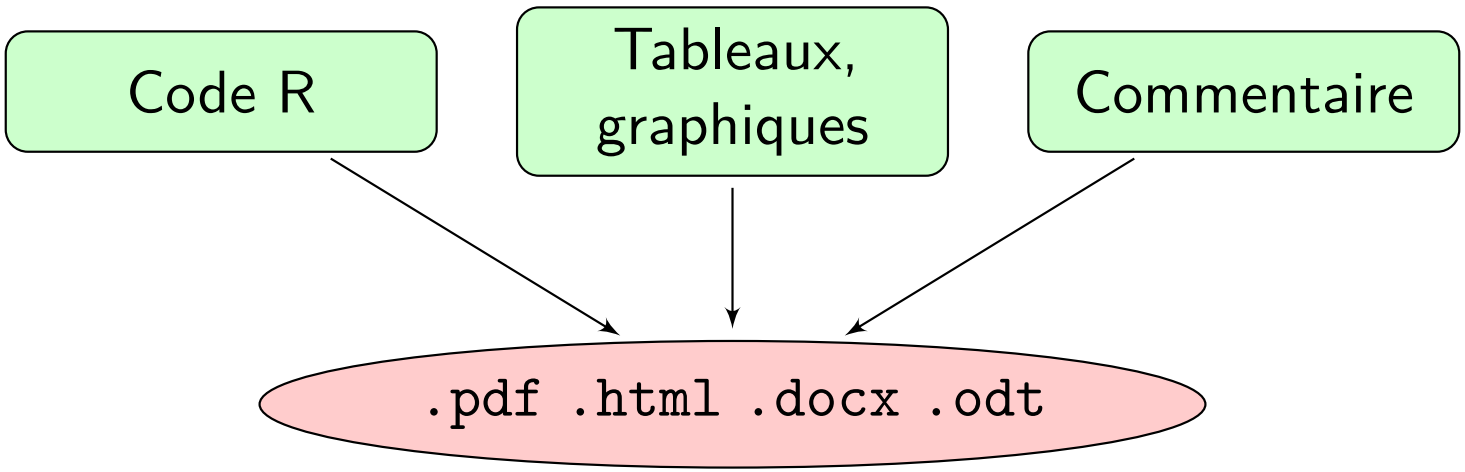
- Construire des **rapports complets et automatisés** pour des tâches répétitives : rapports d'utilisation, tests de la cohérence ou de la qualité de nouvelles données, etc.

- Produire des publications **reproductibles** sur différents supports : notes, documentation, articles de revues, etc.

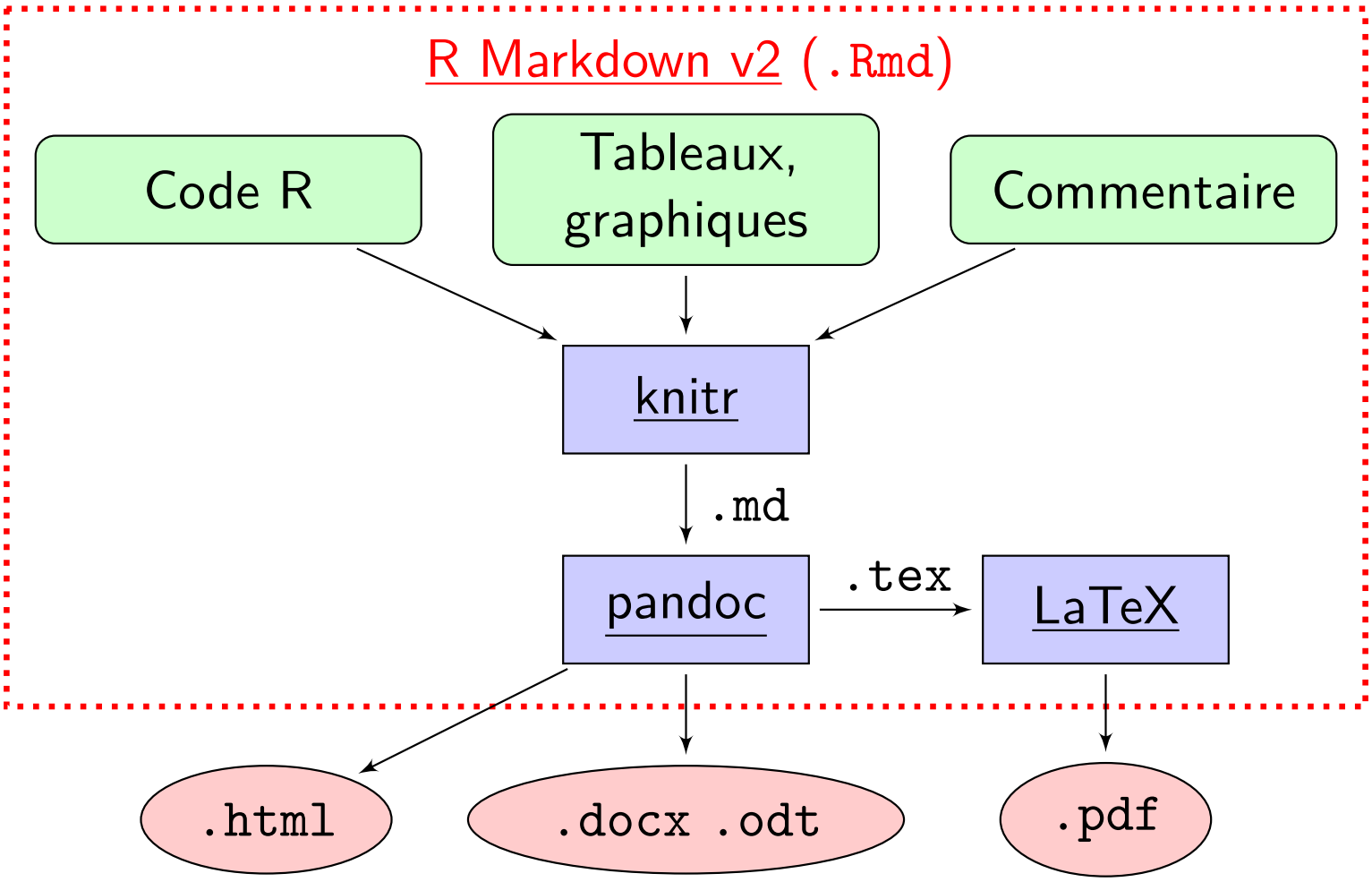
Principe de la génération automatique de documents

La génération automatique de documents complets repose sur deux éléments :

1. Articuler le code, les résultats et le commentaire dans un **même document** : garantir la cohérence et faciliter les mises à jour ;
2. Formater de façon standardisée le document vers **plusieurs sorties** : `.html`, `.pdf`, `.docx`, `.odt`.



Etapes de la génération automatique de documents



Note `rmarkdown` et `knitr` sont des *packages* R (avec plusieurs dépendances) ; pandoc et LaTeX sont des programmes autonomes.

Préparer et tester l'environnement de travail

1. Travailler sous RStudio
   - RStudio facilite l'édition et la compilation de fichier `.Rmd` ;
   - pandoc est embarqué par défaut dans RStudio.

2. Installer les *packages* nécessaires
   - installer le *package* `rmarkdown` et ses dépendances ;
   - installer le *package* `knitr` et ses dépendances.

3. Pour produire des fichiers `.pdf`, installer LaTeX (MiKTeX sous Windows) et s'assurer que ses programmes figurent dans le *path* de Windows.

4. Créer un nouveau fichier R Markdown (`.Rmd`), installer les *packages* complémentaires demandés, choisir le type de document et compiler le fichier d'exemple (`Ctrl + K`).

Ecrire du texte dans R Markdown

Pour écrire du texte dans un document R Markdown, il suffit de le **taper dans le fichier `.Rmd`** (sans le commenter ni l'échapper d'aucune manière).

Des **balises** spéciales permettent de mettre en forme le document :

- les signes `*` et `_` permettent de mettre des mots en `*italique*` ou en `**gras**` ;
- les six niveaux de titres sont préfixés par les signes # (premier niveau), ## (deuxième niveau), etc.
- des listes sont automatiquement créées à partir de successions de - ou de séquences de nombres ou de lettres séparées par un retour à la ligne.

Note Pour une présentation synthétique de R Markdown, se référer à l'aide-mémoire (*cheat sheet*) sur le site de RStudio.

Ecrire du code dans R Markdown

Les blocs de code R sont intégrés dans R Markdown de la façon suivante :

````
```{r}
2 + 2
```
````

Par défaut **le code est évalué**, et **lui-même ainsi que ses résultats sont affichés** dans le document en sortie :

```
2 + 2
```

```
## [1] 4
```

Ecrire du code dans R Markdown

Les **options** saisies en début de bloc permettent de préciser à `knitr` la manière de le prendre en compte, par exemple :

- `eval=FALSE` : le bloc n'est pas évalué ;
- `echo=FALSE` : le bloc n'est pas affiché ;
- `collapse=TRUE` : code et résultats sont affichés à la suite.

````
```{r, echo=FALSE}
2 + 2
```
````

```
## [1] 4
```

Note Toutes les options de `knitr` relatives aux blocs de code (*chunk options*) sont présentées sur la page du créateur du *package*, Yihui Xie.

Ecrire du code dans R Markdown

Il est également possible d'intégrer le résultat d'un traitement R dans le corps d'un paragraphe avec la syntaxe :

```
`r   `
```

Exemple Pour intégrer dans le texte la date de compilation du document, utiliser

```
Document compilé le `r Sys.Date()`.
```

Document compilé le 2018-01-26.

Intégrer des graphiques dans R Markdown

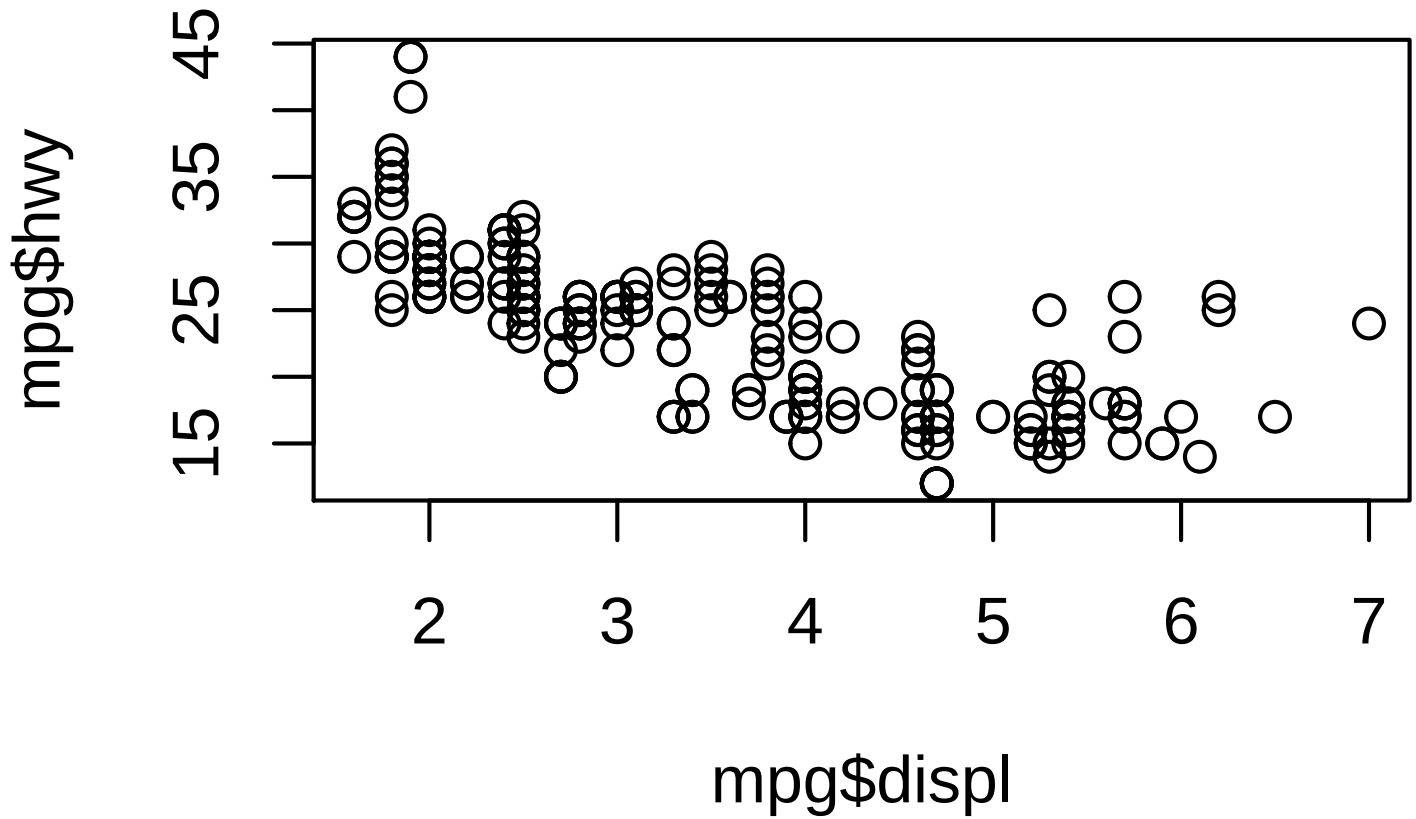
Tous les graphiques produits par les blocs de code sont **automatiquement intégrés au fichier final**.

Un **grand nombre d'options** sont consacrées au paramétrage des graphiques, notamment :

- `fig.width`, `fig.height` : largeur et hauteur utilisées pour produire le graphique, en pouces ;
- `fig.asp` : rapport hauteur/largeur (`fig.height` est neutralisé quand `fig.asp` est renseigné) ;
- `out.width`, `out.height` : largeur et hauteur du graphique dans la sortie finale ;
- `fig.align` : alignement du grahique (`"left"`, `"right"` ou `"center"`) ;
- `dpi` (72 par défaut) : résolution (utile uniquement pour HTML).

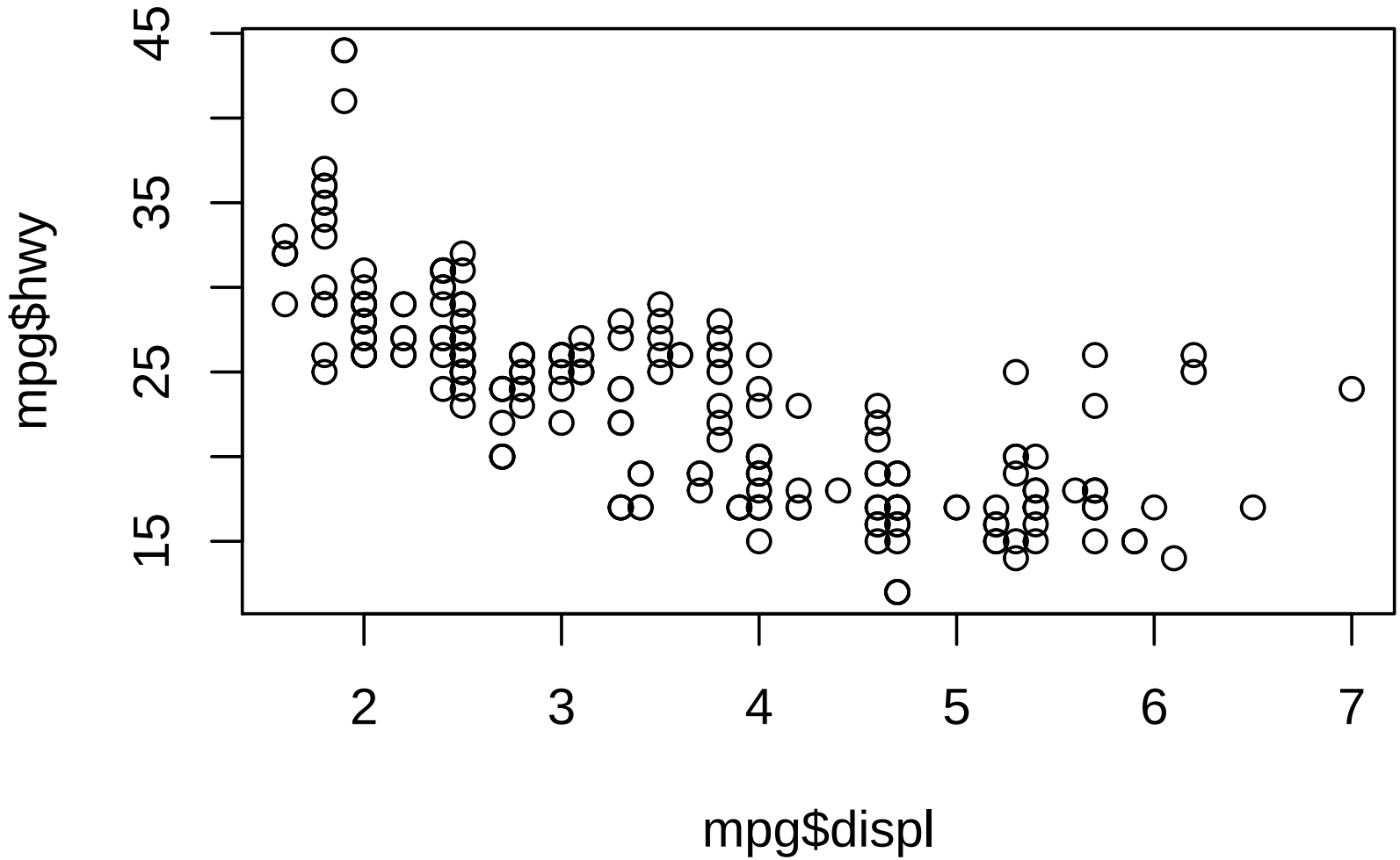
Intégrer des graphiques dans R Markdown

```
```{r, fig.asp = 3/4, fig.width = 4}
plot(mpg$displ, mpg$hwy)
```
```



Intégrer des graphiques dans R Markdown

````
```{r, fig.asp = 3/4, fig.width = 6, out.width = "4in"}
plot(mpg$displ, mpg$hwy)
```
````



Intégrer des tableaux dans R Markdown

Pour construire un tableau dans R Markdown, il suffit de le « dessiner » avec les signes - et | :

```
Colonne 1 | Colonne 2 | Colonne 3
--------: | :-------- | :-------:
1         | a         | `TRUE`
2         | b         | `FALSE`
```

Colonne 1	Colonne 2	Colonne 3
1	a	`TRUE`
2	b	`FALSE`

Les : permettent de spécifier l'alignement des colonnes.

Intégrer des tableaux dans R Markdown

En règle générale cependant, les tableaux à intégrer sont générés automatiquement à partir des données.

````
```{r}
resultat <- data.table(mpg)[
    , list(hwy=mean(hwy), cty=mean(cty)), by = drv
]
resultat
```
````

```
##    drv      hwy     cty
## 1:   f 28.16038 19.9717
## 2:   4 19.17476 14.3301
## 3:   r 21.00000 14.0800
```

La fonction `knitr::kable()` permet de **transformer un objet R en tableau formaté pour R Markdown**.

Intégrer des tableaux dans R Markdown

````
```{r, results = "asis"}
knitr::kable(resultat)
```
````

```
|drv |      hwy|      cty|
|:---|--------:|--------:|
|f   | 28.16038| 19.9717|
|4   | 19.17476| 14.3301|
|r   | 21.00000| 14.0800|
```

Ce qui donne une fois formaté par R Markdown :

drv	hwy	cty
f	28.16038	19.9717
4	19.17476	14.3301
r	21.00000	14.0800

Paramétrer un document R Markdown

La plupart des paramètres généraux du documents sont à indiquer dans son en-tête (désigné par l'acronyme YAML) :

```
---
title: "Formation R Perfectionnement"
author: "Martin Chevalier (Insee)"
output:
  html_document:
    highlight: haddock
    toc: yes
    toc_depth: 2
    toc_float: yes
---
```

Pour en savoir plus Le site de RStudio documente le paramétrage de l'en-tête YAML selon les formats de sortie souhaités (`html`, `pdf`).