

# Travaux Pratiques de Biométrie 2

Benoît Simon-Bouhet

2021-11-06

## Table des matières

<b>1 Introduction</b>	<b>2</b>
1.1 Objectifs . . . . .	2
1.2 Organisation . . . . .	3
<b>2 R et RStudio : les bases</b>	<b>4</b>
2.1 Que sont R et RStudio . . . . .	5
2.2 Comment exécuter du code R? . . . . .	6
2.3 Les packages additionels . . . . .	17
2.4 Exercices . . . . .	19
<b>3 Explorez votre premier jeu de données</b>	<b>19</b>
3.1 Le package nycflights13 . . . . .	19
3.2 Le data frame flights . . . . .	20
3.3 Explorer un data.frame . . . . .	21
3.4 Exercices . . . . .	25
<b>4 Visualiser des données avec ggplot2</b>	<b>25</b>
4.1 Prérequis . . . . .	26
4.2 La grammaire des graphiques . . . . .	27
4.3 Les nuages de points . . . . .	30
4.4 Les graphiques en lignes . . . . .	42
4.5 Les histogrammes . . . . .	50
4.6 Les facets . . . . .	56
4.7 Les boîtes à moustaches ou boxplots . . . . .	59

4.8	Les diagrammes bâtons . . . . .	63
4.9	De l'exploration à l'exposition . . . . .	74
4.10	Exercices . . . . .	98
<b>5</b>	<b>(Ar)ranger des données avec tidyverse</b>	<b>99</b>
5.1	Prérequis . . . . .	100
5.2	C'est quoi des "tidy data"? . . . . .	101
5.3	Importer des données depuis un tableau . . . . .	112
<b>6</b>	<b>Tripatouiller les données avec dplyr</b>	<b>123</b>
6.1	Pré-requis . . . . .	123
6.2	Le pipe %>% . . . . .	123
6.3	Les verbes du tripatouillage de données . . . . .	125
6.4	Filtrer des lignes avec filter() . . . . .	126
6.5	Créer des résumés avec summarise() et group_by() . . . . .	131
6.6	Sélectionner des variables avec select() . . . . .	144
6.7	Créer de nouvelles variables avec mutate() . . . . .	148
6.8	Trier des lignes avec arrange() . . . . .	152
6.9	Associer plusieurs tableaux avec left_join() et inner_join() . . . . .	154
6.10	Exercices . . . . .	159
	<b>Références</b>	<b>160</b>

## I Introduction

### I.1 Objectifs

Ce livre contient l'ensemble du matériel (contenus, exemples, exercices...) nécessaire à la réalisation des travaux pratiques de biométrie 2.

Ces travaux pratiques ont essentiellement 3 objectifs :

1. Vous faire (re)découvrir les logiciels R et RStudio dans lesquels vous allez passer beaucoup de temps en L3 puis en master. Si vous choisissez une spécialité de master qui implique de traiter des données (c'est-à-dire à peu près toutes les spécialités !) et/ou de communiquer des résultats d'analyses statistiques, alors R et RStudio devraient être les logiciels vers lesquels vous vous tournerez naturellement.
2. Vous faire prendre conscience de l'importance des visualisations graphiques :

- d'une part, pour comprendre à quoi ressemblent les données en votre possession,
- d'autre part, pour vous permettre de formuler des hypothèses pertinentes et intéressantes concernant les systèmes que vous étudiez,
- et enfin, pour communiquer efficacement vos trouvailles à un public qui ne connaît pas vos données aussi bien que vous (cela inclut évidemment vos enseignants à l'issue de vos stages).

Les données que vous serez amenés à traiter lors de vos stages, ou plus tard, lorsque vous serez en poste, ont souvent été acquises à grands frais, et au prix d'efforts importants. Il est donc de votre responsabilité d'en tirer le maximum. Et ça commence toujours (ou presque), par la réalisation de visualisations graphiques parlantes.

3. Vous apprendrez comment calculer des statistiques descriptives simples, sur plusieurs types de variables, afin de vous mettre dans les meilleures conditions possibles pour aborder d'une part les cours de biométrie 3 du second semestre, et d'autre part les comptes-rendus de TP que vous aurez à produire dans d'autres EC. Vos enseignants attendent de vous la plus grande rigueur lorsque vous analysez et présentez des résultats d'analyses statistiques. Ces TP ont pour objectifs de vous fournir les bases nécessaires pour satisfaire ce niveau d'exigence.
- 

## 1.2 Organisation

Les séances de travaux pratiques de biométrie 2 durent 1h30 et ont lieu en salle informatique, soit au Pôle Communication Multimédia Réseaux (PCM), soit à la MSI (Maison des Sciences pour l'Ingénieur, salle 217). Vous pourrez utiliser les ordinateurs de l'université si vous n'avez pas d'ordinateur personnel, mais je ne peux que vous encourager à utiliser ou vos propres machines : lors de vos stages et pour rédiger vos comptes-rendus de TP, vous utiliserez le plus souvent vos propres ordinateurs, autant prendre dès maintenant de bonnes habitudes.

Seules 6 heures de TP sont prévues dans la maquette, soit 4 séances de TP. C'est très peu ! En fait, c'est très insuffisant pour couvrir correctement la totalité du contenu de cet enseignement. C'est la raison pour laquelle chacune des 3 premières séances de TP est suivie d'**une séance de TEA de 90 minutes**.

Au début de chaque séance de TP, j'indiquerai un objectif que vous devriez être en mesure d'atteindre en l'espace de 3 heures. Certains iront probablement plus vite et d'autres plus lentement. Lors des séances de TP, je serai disponible pour répondre à chacune de vos questions. À l'issue des 90 minutes de TP, et compte tenu de la situation sanitaire, vous serez libérés et vous pourrez ainsi soit trouver une autre salle à l'université, soit rentrer chez vous pour effectuer les 90 minutes de TEA prévues.

Pendant le TEA, vous devrez continuer la lecture de ce livre et terminer les exercices demandés. Les exercices devront être déposés sur l'ENT à l'issue de chaque séance de TEA, qu'ils soient terminés ou non. Si la séance de TEA ne vous suffit pas pour aller au bout des exercices, vous êtes vivement encouragés à les terminer en dehors des heures de cours. En effet, une correction rapide sera faite lors de la séance de travaux pratiques suivante et nous n'auront que très peu

de temps à consacrer à ces corrections. Si vous n'y avez pas sérieusement réfléchi en amont, cela ne vous servira absolument à rien. Pour apprendre à utiliser un logiciel comme R, il faut en effet faire les choses soi-même, “mettre les mains dans le cambouis”, se confronter à la difficulté, ne pas avoir peur des messages d'erreurs (il faut d'ailleurs apprendre à les déchiffrer pour comprendre d'où viennent les problèmes), essayer maintes fois, se tromper beaucoup, recommencer, et surtout, ne pas se décourager. Même si ce livre est conçu pour vous faciliter la tâche, l'apprentissage de R est souvent perçu comme une lutte. Mais quand ça fonctionne enfin, c'est extrêmement gratifiant et il n'existe pas à l'heure actuelle de meilleur logiciel pour analyser des données et produire des graphiques de qualité. Vous pouvez me croire sur parole : j'utilise ce logiciel presque quotidiennement depuis plus de 15 ans.

En cas de blocage pendant les TEA ou en dehors des heures de cours, n'hésitez pas à [me contacter par email](#) en indiquant clairement à quel endroit se situe votre problème. Toutefois, ce document est fait pour vous permettre d'avancer en autonomie et vous ne devriez normalement pas avoir beaucoup besoin de moi si votre lecture de ce document est attentive. L'expérience montre en effet que la plupart du temps, il suffit de lire correctement les paragraphes précédents et/ou suivants pour obtenir la réponse à ses questions. Je vous encourage également à vous **entraider** : c'est très formateur pour celui qui explique, et celui qui rencontre une difficulté a plus de chance de comprendre si c'est quelqu'un d'autre qui lui explique plutôt que la personne qui a rédigé les instructions mal comprises.

Enfin, les exercices demandés ne seront pas notés, mais tout ce que nous voyons en TP devra être acquis le jour de l'examen. Utilisez donc le temps du TEA pour vous préparer au mieux. L'apprentissage prend du temps, autant vous y mettre sérieusement dès maintenant.

## 2 R et RStudio : les bases

Avant de commencer à explorer des données dans R, il y a plusieurs concepts clés qu'il faut comprendre en premier lieu :

1. Que sont R et RStudio ?
2. Comment s'y prend-on pour coder dans R ?
3. Que sont les “packages” ?

Même si vous pensez être déjà à l'aise avec ces concepts, lisez attentivement ce chapitre et faites les exercices demandés. Cela vous rafraîchira probablement la mémoire, et il n'est pas impossible que vous appreniez une chose ou deux au passage. Une bonne maîtrise des éléments présentés dans ce chapitre est en effet nécessaire pour aborder sereinement les chapitres suivants, à commencer par le chapitre 3, qui présente quelques jeux de données que nous explorerons en détail dans cet ouvrage.

Ce chapitre est en grande partie basé sur les 3 ressources suivantes que je vous encourage à consulter si vous souhaitez obtenir plus de détails :

- I. L'ouvrage intitulé [ModernDive](#), de Chester Ismay et Albert Y. Kim. Une bonne partie de ce livre est très largement inspirée de cet ouvrage. C'est en anglais, mais c'est un très bon texte d'introduction aux statistiques sous R et RStudio.

2. L'ouvrage intitulé [Getting used to R, RStudio, and R Markdown](#) de Chester Ismay, comprend des podcasts (en anglais toujours) que vous pouvez suivre en apprenant.
  3. Les tutoriels en ligne de DataCamp. DataCamp est une plateforme de e-learning accessible depuis n'importe quel navigateur internet et dont la priorité est l'enseignement des “data sciences”. Leurs tutoriels vous aideront à apprendre certains des concepts de développés dans ce livre. Avant d'aller plus loin, rendez-vous sur [le site de DataCamp](#) et créez-vous un compte gratuit.
- 

## 2.1 Que sont R et RStudio

Pour l'ensemble de ces TP, j'attends de vous que vous utilisez R via RStudio. Les utilisateurs novices confondent souvent les deux. Pour tenter une analogie simple :

- R est le moteur d'une voiture
- RStudio est l'habitacle, le tableau de bord, les pédales

Si vous n'avez pas de moteur, vous n'irez nulle part. En revanche, un moteur sans tableau de bord est difficile à manœuvrer. Il est en effet beaucoup plus simple de faire avancer une voiture depuis l'habitacle, plutôt qu'en actionnant à la main les câbles et leviers du moteur.

En l'occurrence, R est un langage de programmation capable de produire des graphiques et de réaliser des analyses statistiques, des plus simples aux plus complexes. RStudio est un “emballage” qui rend l'utilisation de R plus aisée. RStudio est ce qu'on appelle un IDE : Integrated Development Environment. On peut utiliser R sans RStudio, mais c'est nettement plus compliqué, nettement moins pratique.

### 2.1.1 Installation de R et RStudio

*Si vous travaillez exclusivement sur les ordinateurs de l'Université, vous pouvez passer cette section. Si vous souhaitez utiliser R et RStudio sur votre ordinateur personnel, alors suivez le guide !*

Avant tout, vous devez télécharger et installer R et RStudio, dans cet ordre :

#### 1. Téléchargez et installez R

- Vous devez installer ce logiciel en premier.
- Cliquez sur le lien de téléchargement qui correspond à votre système d'exploitation, puis, sur “base”, et suivez les instructions.

#### 2. Téléchargez et installez RStudio

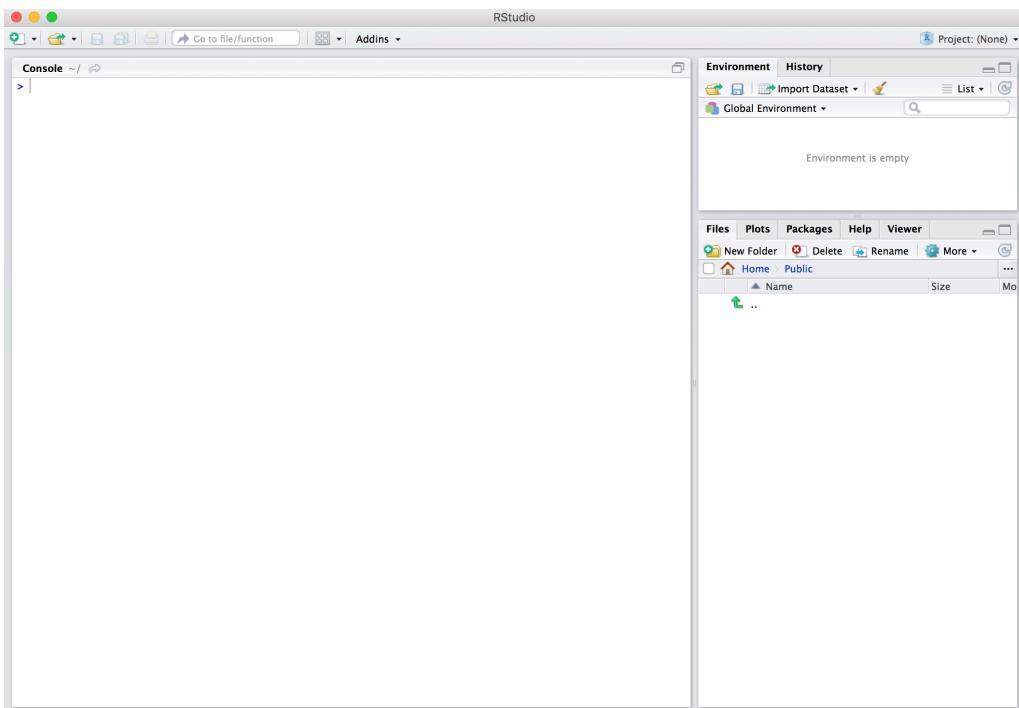
- Cliquez sur “Download RStudio Desktop”.
- Choisissez la version gratuite et cliquez sur le lien de téléchargement qui correspond à votre système d'exploitation.

### 2.1.2 Utiliser R depuis RStudio

Puisqu'il est beaucoup plus facile d'utiliser Rstudio pour interagir avec R, nous utiliserons exclusivement l'interface de RStudio. Après l'installation des 2 logiciels, vous disposez de 2 nouveaux logiciels sur votre ordinateur. RStudio ne peut fonctionner sans R, mais nous travaillerons exclusivement dans RStudio :

- R, ne pas ouvrir ceci : 
- RStudio, ouvrir ça : 

À l'université, vous trouverez RStudio dans le menu Windows. Quand vous ouvrez RStudio pour la première fois, vous devriez obtenir une fenêtre qui ressemble à ceci :



Prenez le temps d'explorer cette interface, cliquez sur les différents onglets, ouvrez les menus, allez faire un tour dans les préférences du logiciel pour découvrir les différents panneaux de l'application, en particulier la Console dans laquelle nous exécuterons très bientôt du code R.

---

## 2.2 Comment exécuter du code R ?

Maintenant que vous avez installé R et RStudio, vous vous demandez probablement comment on s'en sert... La première chose à noter est que, contrairement à d'autres logiciels comme Excel, STATA ou SAS qui fournissent des interfaces où tout se fait en cliquant avec sa souris, R est un langage interprété, ce qui signifie que vous devez taper des commandes R, écrites en code R. C'est-à-dire que vous devez **programmer** en R (j'utilise les termes "coder" et "programmer" de manière interchangeable dans ce livre).

Il n'est pas nécessaire d'être un programmeur pour utiliser R, néanmoins, il est nécessaire de programmer ! Il existe en effet un ensemble de concepts de programmation de base que les utilisateurs R doivent comprendre et maîtriser. Par conséquent, bien que ce livre ne soit pas un livre sur la programmation, vous en apprendrez juste assez sur ces concepts de programmation de base pour explorer et analyser efficacement des données.

### 2.2.1 La console

La façon la plus simple d'interagir avec RStudio (mais pas du tout la meilleure !) consiste à taper directement des commandes que R pourra comprendre dans la Console.

Cliquez dans la console (après le symbole >) et tapez ceci, sans oublier de valider en tapant sur la touche Entrée :

```
3 + 8
```

```
[1] 11
```

Félicitations, vous venez de taper votre première instruction R : vous savez maintenant faire des additions !

### 2.2.2 Le répertoire de travail

La première commande que vous devriez connaître quand vous travaillez dans R ou RStudio est la suivante :

```
getwd()
```

Si vous tapez cette commande dans la console, RStudio doit vous afficher un emplacement sur votre ordinateur. Cet emplacement est appelé “Répertoire de travail”, ou “Working Directory” en anglais (getwd() est l'abréviation de “Get Working Directory”).

Ce répertoire de travail est important : c'est là que seront stockés les tableaux et graphiques que vous déciderez de sauvegarder. C'est là aussi que vous sauvegarderez vos scripts (voir plus bas) qui vous permettront de garder la trace de votre travail et de le reprendre là où vous l'aviez laissé la dernière fois. Enfin, lorsque vous souhaiterez importer des tableaux de données contenus dans des fichiers externes (par exemple, des fichiers Excel), c'est également dans ce répertoire que R tentera de trouver vos données.

Avant d'aller plus loin je vous conseille donc vivement de :

1. Créer un nouveau dossier intitulé “Rstats” sur votre espace personnel (généralement, sur le disque “W :” des ordinateurs de l'Université)
2. Indiquez à RStudio que vous souhaitez travailler dans ce nouveau répertoire de travail.

Pour cela vous avez 3 solutions au choix :

1. Dans RStudio, cliquez dans le menu “Session > Set Working Directory > Choose Directory...” puis naviguez jusqu'au dossier que vous venez de créer

2. Dans le panneau “Files”, naviguez jusqu’au dossier “Rstats” que vous venez de créer, puis cliquez sur le bouton “More > Set As Working Directory”
3. En ligne de commande, dans la console, utilisez la fonction `setwd()` pour spécifier le chemin de votre nouveau dossier, par exemple :

```
# Attention à bien respecter les majuscules et à utiliser les guillemets.
setwd("W:/Rstats")
```

Il ne vous reste plus qu’à vérifier que le changement a bien été pris en compte en tapant à nouveau `getwd()` dans la console. Attention, vous devrez vous assurer d’être dans le bon répertoire de travail **à chaque nouvelle session !**

### 2.2.3 Les scripts

Taper du code directement dans la console est probablement la pire façon de travailler dans RStudio. Cela est parfois utile pour faire un rapide calcul, ou pour vérifier qu’une commande fonctionne correctement. Mais la plupart du temps, vous devriez taper vos commandes dans un script.

Un script est un fichier au format “texte brut” (cela signifie qu’il n’y a pas de mise en forme et que ce fichier peut-être ouvert par n’importe quel éditeur de texte, y compris les plus simples comme le bloc notes de Windows), dans lequel vous pouvez taper :

1. des instructions qui seront comprises par R comme si vous les tapiez directement dans la console
2. des lignes de commentaires, qui doivent obligatoirement commencer par le symbole #.

Les avantages de travailler dans un script sont nombreux :

1. Vous pouvez sauvegarder votre script à tout moment (vous devriez prendre l’habitude de le sauvegarder très régulièrement). Vous gardez ainsi la trace de toutes les commandes que vous avez tapées.
2. Vous pouvez aisément partager votre script pour collaborer avec vos collègues de promo et enseignants.
3. Vous pouvez documenter votre démarche et les différentes étapes de vos analyses. Vous devez ajouter autant de commentaires que possible. Cela permettra à vos collaborateurs de comprendre ce que vous avez fait. Et dans 6 mois, cela vous permettra de comprendre ce que vous avez fait. Si votre démarche vous paraît cohérente aujourd’hui, il n’est en effet pas garanti que vous vous souviendrez de chaque détail quand vous vous re-plongerez dans vos analyses dans quelques temps. Donc aidez-vous vous même en commentant vos scripts dès maintenant.
4. Un script bien structuré, bien indenté (avec les bons retours à la ligne, des sauts de lignes, des espaces, bref, de l’air) et clair permet de rendre vos analyses répétables. Si vous passez 15 heures à analyser un tableau de données précis, il vous suffira de quelques secondes pour analyser un nouveau jeu de données similaire : vous n’auriez que quelques lignes à modifier dans votre script original pour l’appliquer à de nouvelles données.

Vous pouvez créer un script en cliquant dans le menu “File > New File > R Script”. Un nouveau panneau s’ouvre dans l’application. Pensez à sauvegarder immédiatement votre nouveau script. Il faut pour cela lui donner un nom. Vous noterez que par défaut, RStudio propose d’enregistrer votre script dans votre répertoire de travail.

À partir de maintenant, vous ne devriez plus taper de commande directement dans la console. Tapez systématiquement vos commandes dans un script et sauvegardez-le régulièrement.

Pour exécuter les commandes du script dans la console, il suffit de placer le curseur sur la ligne contenant la commande et de presser les touches **ctrl + enter** (ou **command + enter** sous macOS). Si un message d’erreur s’affiche dans la console, c’est que votre instruction était erronée. Modifiez la directement dans votre script et pressez à nouveau les touches **ctrl + enter** (ou **command + enter** sous macOS) pour tenter à nouveau votre chance. Idéalement, votre script ne devrait contenir que des commandes qui fonctionnent et des commentaires expliquant à quoi servent ces commandes.

À la fin de chaque séance de TEA, vous devrez déposer sur l’ENT le script que vous avez créé durant la séance. Ce script devra porter votre nom de famille et se terminer par l’extension **.R**. Ainsi, si Jean-Claude Van Damme faisait des statistiques, il devrait déposer sur l’ENT un fichier intitulé **VanDamme.R** à la fin de chaque séance de TEA.

Ci-dessous, un exemple de script :

```
# Penser à installer le package ggplot2 si besoin
# install.packages("ggplot2")

# Chargement du package
library(ggplot2)

# Mise en mémoire des données de qualité de l'air à New-York de mai à
# septembre 1973
data(airquality)

# Affichage des premières lignes du tableau de données
head(airquality)

# Quelle est la structure de ce tableau ?
str(airquality)

# Réalisation d'un graphique présentant la relation entre la concentration
# en ozone atmosphérique en ppb et la température en degrés Farenheit
ggplot(data = airquality, mapping = aes(x = Temp, y = Ozone)) +
  geom_point() +
  geom_smooth(method = "loess")

# On constate une augmentation importante de la concentration d'ozone
# pour des températures supérieures à 75°F
```

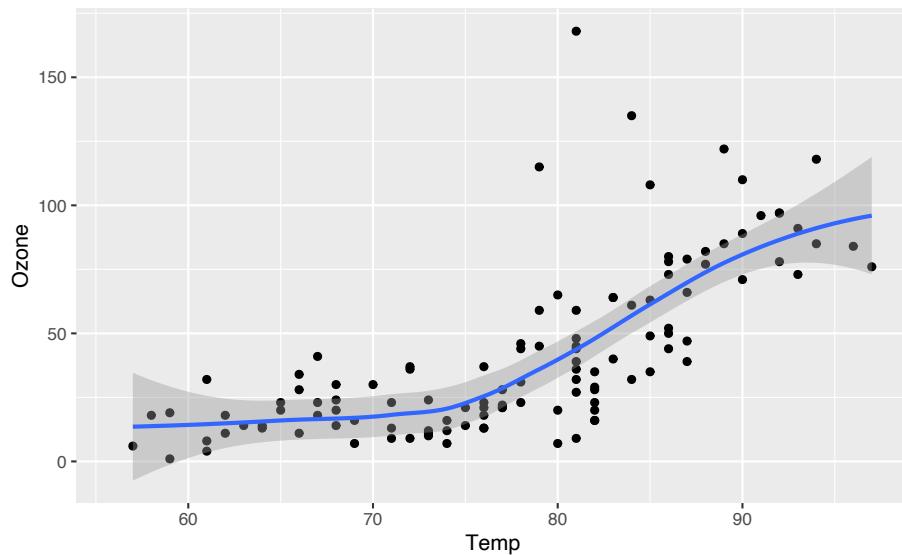
Même si vous ne comprenez pas encore les commandes qui figurent dans ce script (ça viendra!), voici ce que vous devez en retenir :

1. Le script contient plus de lignes de commentaires que de commandes R.
2. Chaque étape de l'analyse est décrite en détail.
3. On peut ajouter des commentaires afin de décrire les résultats.
4. Seules les commandes pertinentes et qui fonctionnent ont été conservées dans ce script.
5. Chaque ligne de commentaire commence par #. Il est ainsi possible de conserver certaines commandes R dans le script, "pour mémoire", sans pour autant qu'elle ne soient exécutées. C'est le cas pour la ligne `# install.packages("ggplot2")`.

Si j'exécute ce script dans la console de RStudio (en sélectionnant toutes les lignes et en pressant les touches `ctrl + enter` ou `command + enter` sous macOS), voilà ce qui est produit :

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

```
'data.frame': 153 obs. of 6 variables:
$ Ozone : int 41 36 12 18 NA 28 23 19 8 NA ...
$ Solar.R: int 190 118 149 313 NA NA 299 99 19 194 ...
$ Wind : num 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
$ Temp : int 67 72 74 62 56 66 65 59 61 69 ...
$ Month : int 5 5 5 5 5 5 5 5 5 5 ...
$ Day : int 1 2 3 4 5 6 7 8 9 10 ...
```



## 2.2.4 Concepts de base en programmation et terminologie

Pour vous présenter les concepts de base et la terminologie de la programmation dont nous aurons besoin dans R, vous allez suivre des tutoriels en ligne sur le site de DataCamp. Pour chacun des tutoriels, j'indique une liste des concepts de programmation couverts. Notez que dans ce livre, nous utiliserons une police différente pour distinguer le texte normal et les commandes-informatiques.

Il est important de noter que, bien que ces tutoriels sont d'excellentes introductions, une seule lecture est insuffisante pour un apprentissage en profondeur et une rétention à long terme. Les outils ultimes pour l'apprentissage et la rétention à long terme sont "l'apprentissage par la pratique" et "la répétition". Outre les exercices demandés dans DataCamp, que vous devez effectuer directement dans votre navigateur, je vous encourage donc à multiplier les essais, directement dans la console de RStudio, ou, de préférence, dans un script que vous annoterez, pour vous assurer que vous avez bien compris chaque partie.

**2.2.4.1 Objets, types, vecteurs, facteurs et tableaux de données** Dans [le cours d'introduction à R](#) sur DataCamp, suivez les chapitres suivants. Au fur et à mesure de votre travail, notez les termes importants et ce à quoi ils font référence.

— [Chapitre 1 : introduction](#)

- La console : l'endroit où vous tapez des commandes.
- Les objets : où les valeurs sont stockées, comment assigner des valeurs à des objets.
- Les types de données : entiers, doubles/numériques, caractères et logiques.

— [Chapitre 2 : vecteurs](#)

- Les vecteurs : des collections de valeurs du même type.

— [Chapitre 4 : les facteurs](#)

- Des données catégorielles (et non pas *numériques*) représentées dans R sous forme de *factors*.

— [Chapitre 5 : les jeux de données ou data.frame](#)

- Les *data.frames* sont similaires aux feuilles de calcul rectangulaires que l'on peut produire dans un tableur. Dans R, ce sont des objets rectangulaires (des tableaux !) contenant des jeux de données : les lignes correspondent aux observations et les colonnes aux variables décrivant les observations. La plupart du temps, c'est le format de données que nous utiliserons. Plus de détails dans le chapitre [3](#).

Avant de passer à la suite, il nous reste 2 grandes notions à découvrir dans le domaine du code et de la syntaxe afin de pouvoir travailler efficacement dans R : les opérateurs de comparaison d'une part, et les fonctions d'autre part.

**2.2.4.2 Opérateurs de comparaison** Comme leur nom l'indique, ils permettent de comparer des valeurs ou des objets. Les principaux opérateurs de comparaison sont :

- $=$  : égal à

- $\neq$  : différent de
- $>$  : supérieur à
- $<$  : inférieur à
- $\geq$  : supérieur ou égal à
- $\leq$  : inférieur ou égal à

Ainsi, on peut tester si 3 est égal à 5 :

```
3 == 5
```

```
[1] FALSE
```

La réponse est bien entendu FALSE. Est-ce que 3 est inférieur à 5 ?

```
3 < 5
```

```
[1] TRUE
```

La réponse est maintenant TRUE. Lorsque l'on utilise un opérateur de comparaison, la réponse est toujours soit vrai (TRUE), soit faux (FALSE).

Il est aussi possible de comparer des chaînes de caractères :

```
"Bonjour" == "Au revoir"
```

```
[1] FALSE
```

```
"Bonjour" >= "Au revoir"
```

```
[1] TRUE
```

Manifestement, “Bonjour” est supérieur ou égal à “Au revoir”. En fait, R utilise l'ordre alphabétique pour comparer les chaînes de caractères. Puisque dans l'alphabet, le “B” de “Bonjour” arrive après le “A” de “Au revoir”, pour R, “Bonjour” est supérieur à “Au revoir”.

Il est également possible d'utiliser ces opérateurs pour comparer un chiffre et un vecteur :

```
tailles_pop1 <- c(112, 28, 86, 14, 154, 73, 63, 48)
tailles_pop1 > 80
```

```
[1] TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE
```

Ici, l'opérateur nous permet d'identifier quels éléments du vecteur taille\_pop1 sont supérieurs à 80. Il s'agit des éléments placés en première, troisième et cinquième positions.

Il est aussi possible de comparer 2 vecteurs qui contiennent le même nombre d'éléments :

```
tailles_pop2 <- c(114, 27, 38, 91, 54, 83, 33, 68)  
tailles_pop1 > tailles_pop2
```

```
[1] FALSE TRUE TRUE FALSE TRUE FALSE TRUE FALSE
```

Les comparaisons sont ici faites élément par élément. Ainsi, les observations 2, 3, 5 et 7 du vecteur `tailles_pop1` sont supérieures aux observations 2, 3, 5 et 7 du vecteur `tailles_pop2` respectivement.

Ces vecteurs de vrais/faux sont très utiles car ils peuvent permettre de compter le nombre d'éléments répondant à une certains condition :

```
sum(tailles_pop1 > tailles_pop2)
```

```
[1] 4
```

Lorsque l'on effectue une opération arithmétique (comme le calcul d'une somme ou d'une moyenne) sur un vecteur de vrais/faux, les TRUE sont remplacés par 1 et les FALSE par 0. La somme nous indique donc le nombre de vrais dans un vecteur de vrais/faux, et la moyenne nous indique la proportion de vrais :

```
mean(tailles_pop1 > tailles_pop2)
```

```
[1] 0.5
```

**Note** : Attention, si les vecteurs comparés n'ont pas la même taille, un message d'avertissement est affiché :

```
tailles_pop3 <- c(43, 56, 92)  
tailles_pop1
```

```
[1] 112 28 86 14 154 73 63 48
```

```
tailles_pop3
```

```
[1] 43 56 92
```

```
tailles_pop3 > tailles_pop1
```

```
Warning in tailles_pop3 > tailles_pop1: longer object length is not a  
multiple of shorter object length
```

```
[1] FALSE TRUE TRUE TRUE FALSE TRUE FALSE TRUE
```

Dans un cas comme celui là, R va *recycler* l'objet le plus court, ici tailles\_pop3 pour qu'une comparaison puisse être faite avec chaque élément de l'objet le plus long (ici, tailles\_pop1). Ainsi, 43 est comparé à 112, 56 est comparé à 28 et 92 est comparé à 86. Puisque tailles\_pop3 ne contient plus d'éléments, ils sont recyclés, dans le même ordre : 43 est comparé à 14, 56 est comparé à 154, et ainsi de suite jusqu'à ce que tous les éléments de tailles\_pop1 aient été passés en revue.

Ce type de recyclage est très risqué car il est difficile de savoir ce qui a été comparé avec quoi. En travaillant avec des tableaux plutôt qu'avec des vecteurs, le problème est généralement évité puisque toutes les colonnes d'un `data.frame` contiennent le même nombre d'éléments.

Dernière chose concernant les opérateurs de comparaison : la question des données manquantes. Dans R les données manquantes sont symbolisées par cette notation : NA, abréviation de “Not Available”. Le symbole NaN (comme “Not a Number”) est parfois aussi observé lorsque des opérations ont conduit à des indéterminations. Mais c'est plus rare et la plupart du temps, les NaNs peuvent être traités comme les NAs. L'un des problèmes des données manquantes est qu'il est nécessaire de prendre des précautions pour réaliser des comparaisons les impliquants :

```
3 == NA
```

```
[1] NA
```

On s'attend logiquement à ce que 3 ne soit pas considéré comme égal à NA, et donc, on s'attend à obtenir FALSE. Pourtant, le résultat est NA. La comparaison d'un élément quelconque à une donnée manquante fournit toujours une donnée manquante : la comparaison ne peut pas se faire, R n'a donc rien à retourner. C'est également le cas aussi lorsque l'on compare deux valeurs manquantes :

```
NA == NA
```

```
[1] NA
```

C'est en fait assez logique. Imaginons que j'ignore l'âge de Pierre et l'âge de Marie. Il n'y a aucune raison pour que leur âge soit le même, mais il est tout à fait possible qu'il le soit. C'est impossible à déterminer :

```
age_Pierre <- NA  
age_Marie <- NA  
age_Pierre == age_Marie
```

```
[1] NA
```

Mais alors comment faire pour savoir si une valeur est manquante puisqu'on ne peut pas utiliser les opérateurs de comparaison ? On utilise la fonction `is.na()` :

```
is.na(age_Pierre)
```

```
[1] TRUE
```

```
is.na(tailles_pop3)
```

```
[1] FALSE FALSE FALSE
```

D'une façon générale, le point d'exclamation permet de signifier à R que nous souhaitons obtenir le contraire d'une expression :

```
!is.na(age_Pierre)
```

```
[1] FALSE
```

```
!is.na(tailles_pop3)
```

```
[1] TRUE TRUE TRUE
```

Cette fonction nous sera très utile plus tard pour éliminer toutes les lignes d'un tableau contenant des valeurs manquantes.

**2.2.4.3 L'utilisation des fonctions** Dans R, les fonctions sont des objets particuliers qui permettent d'effectuer des tâches très variées. Du calcul d'une moyenne à la création d'un graphique, en passant par la réalisation d'analyses statistiques complexes ou simplement l'affichage du chemin du répertoire de travail, tout, dans R, repose sur l'utilisation de fonctions. Vous en avez déjà vu un certain nombre :

Fonction	Pour quoi faire ?
c()	Créer des vecteurs
class()	Afficher ou modifier la classe d'un objet
factor()	Créer des facteurs
getwd()	Afficher le chemin du répertoire de travail
head()	Afficher les premiers éléments d'un objet
is.na()	Tester si un objet contient des valeurs manquantes
mean()	Calculer une moyenne
names()	Afficher ou modifier le nom des éléments d'un vecteur
order()	Ordonner les éléments d'un objet
setwd()	Modifier le chemin du répertoire de travail
subset()	Extraire une partie des éléments d'un objet
sum()	Calculer une somme
tail()	Afficher les derniers éléments d'un objet

Cette liste va très rapidement s'allonger au fil des séances. Je vous conseille donc vivement de tenir à jour une liste des fonctions décrites, avec une explication de leur fonctionnement et éventuellement un exemple de syntaxe.

Certaines fonctions ont besoin d'arguments (par exemple, la fonction `factor()`), d'autres peuvent s'en passer (par exemple, la fonction `getwd()`). Pour apprendre comment utiliser une fonction particulière, pour découvrir quels sont ses arguments possibles, quel est leur rôle et leur intérêt, la meilleure solution est de consulter l'aide de cette fonction. Il suffit pour cela de taper un `? suivi du nom de la fonction :`

```
?factor()
```

Toutes les fonctions et jeux de données disponibles dans R disposent d'un fichier d'aide similaire. Cela peut faire un peu peur au premier abord (tout est en anglais !), mais ces fichiers d'aide ont l'avantage d'être très complets, de fournir des exemples d'utilisation, et ils sont tous construits sur le même modèle. Vous avez donc tout intérêt à vous familiariser avec eux. Vous devriez d'ailleurs prendre l'habitude de consulter l'aide de chaque fonction qui vous pose un problème. Par exemple, le logarithme (en base 10) de 100 devrait faire 2, car 100 est égal à  $10^2$ . Pourtant :

```
log(100)
```

```
[1] 4.60517
```

Que se passe-t'il ? Pour le savoir, il faut consulter l'aide de la fonction `log` :

```
?log()
```

Ce fichier d'aide nous apprend que par défaut, la syntaxe de la fonction `log()` est la suivante :

```
log(x, base = exp(1))
```

Par défaut, la base du logarithme est fixée à `exp(1)`. Nous avons donc calculé un logarithme népérien (en base e). Cette fonction prend donc 2 arguments :

1. `x` ne possède pas de valeur par défaut : il nous faut obligatoirement fournir quelque chose (la rubrique "Argument" du fichier d'aide nous indique que `x` doit être un vecteur numérique ou complexe) afin que la fonction puisse calculer un logarithme
2. `base` possède un argument par défaut. Si nous ne spécifions pas nous même la valeur de `base`, elle sera fixée à sa valeur par défaut, c'est à dire `exp(1)`.

Pour calculer le logarithme de 100 en base 10, il faut donc taper, au choix, l'une de ces 3 expressions :

```
log(x = 100, base = 10)
```

```
[1] 2
```

```
log(100, base = 10)
```

```
[1] 2
```

```
log(100, 10)
```

```
[1] 2
```

Le nom des arguments d'une fonction peut être omis tant que ses arguments sont indiqués dans l'ordre attendu par la fonction (cet ordre est celui qui est précisé à la rubrique "Usage" du fichier d'aide de la fonction). Il est possible de modifier l'ordre des arguments d'une fonction, mais il faut alors être parfaitement explicite et utiliser les noms des arguments tels que définis dans le fichier d'aide.

Ainsi, pour calculer le logarithme de 100 en base 10, on ne peut pas taper :

```
log(10, 100)
```

```
[1] 0.5
```

car cela revient à calculer le logarithme de 10 en base 100. On peut en revanche taper :

```
log(base = 10, x = 100)
```

```
[1] 2
```

---

## 2.3 Les packages additionnels

Une source de confusion importante pour les nouveaux utilisateurs de R est la notion de package. Les packages étendent les fonctionnalités de R en fournissant des fonctions, des données et de la documentation supplémentaires et peuvent être téléchargés gratuitement sur Internet. Ils sont écrits par une communauté mondiale d'utilisateurs de R. Par exemple, parmi près de 15000 packages disponibles à l'heure actuelle, nous utiliseront fréquemment :

- Le package `ggplot2` pour la visualisation des données dans le chapitre 4
- Le package `dplyr` pour manipuler des tableaux de données dans le chapitre 6

Une bonne analogie pour les packages R : ils sont comme les apps que vous téléchargez sur un téléphone portable. R est comme un nouveau téléphone mobile. Il est capable de faire certaines choses lorsque vous l'utilisez pour la première fois, mais il ne sait pas tout faire. Les packages sont comme les apps que vous pouvez télécharger dans l'App Store et Google Play. Pour utiliser un package, comme pour utiliser Instagram, vous devez :

1. Le télécharger et l'installer. Vous ne le faites qu'une fois.
2. Le charger (en d'autres termes, l'ouvrir) en utilisant la commande `library()` à chaque nouvelle session de travail.

Donc, tout comme vous ne pouvez commencer à partager des photos avec vos amis sur Instagram que si vous installez d'abord l'application et que vous l'ouvrez, vous ne pouvez accéder aux données et fonctions d'un package R que si vous installez d'abord le package et le chargez avec la fonction `library()`. Passons en revue ces 2 étapes.

### 2.3.1 Installation d'un package

Il y a deux façons d'installer un package. Par exemple, pour installer le package `ggplot2` :

1. **Le plus simple** : Dans le panneau "Files" de Rstudio :
  - a) Cliquez sur l'onglet "Packages"
  - b) Cliquez sur "Install"
  - c) Tapez le nom du package dans le champ "Packages (separate multiple with space or comma) :" Pour notre exemple, tapez `ggplot2`
  - d) Cliquez "Install"
2. **Méthode alternative** : Dans la console, tapez `install.packages("ggplot2")` (vous devez inclure les guillemets).

En procédant de l'une ou l'autre façon, installez également les packages suivants : `tidyverse` et `nycflights13`.

**Note** : un package doit être installé une fois seulement, sauf si une version plus récente est disponible et que vous souhaitez mettre à jour ce package.

### 2.3.2 Charger un package en mémoire

Après avoir installé un package, vous pouvez le charger en utilisant la fonction `library()`. Par exemple, pour charger `ggplot2` et `dplyr` tapez ceci dans la console :

```
library(ggplot2)
library(dplyr)
```

**Note** : Vous devez charger à nouveau chaque package que vous souhaitez utiliser **à chaque fois que vous ouvrez une nouvelle session de travail dans RStudio**. Ça peut être un peu pénible et c'est une source d'erreur fréquente pour les débutants. Quand vous voyez un message d'erreur commençant par :

Error: could not find function ...

rappelez-vous que c'est probablement parce que vous tentez d'utiliser une fonction qui fait partie d'un package que vous n'avez pas chargé. Pour corriger l'erreur, il suffit donc de charger le package approprié avec la commande `library()`.

## 2.4 Exercices

Créez un nouveau script que vous nommerez `VotreNomDeFamille.R`. Vous prendrez soin d'ajouter autant de commentaires que nécessaire dans votre script afin de le structurer correctement.

1. Téléchargez (si besoin) et chargez le package `ggplot2`
2. Chargez le jeu de données `diamonds` grâce à la commande `data(diamonds)`
3. Déterminez le nombre de lignes et de colonnes de ce tableau nommé `diamonds`
4. Créez un nouveau tableau que vous nommerez `diamants_chers` qui contiendra uniquement les informations des diamants dont le prix est supérieur ou égal à \$15000.
5. Combien de diamants coûtent \$15000 ou plus?
6. Cela représente quelle proportion du jeu de données de départ?
7. Triez ce tableau par ordre de prix décroissants et affichez les informations des 20 diamants les plus chers.

## 3 Explorez votre premier jeu de données

Mettons en pratique tout ce que nous avons appris pour commencer à explorer un jeu de données réel. Les données nous parviennent sous différents formats, des images au texte en passant par les chiffres. Tout au long de ce document, nous nous concentrerons sur les ensembles de données qui peuvent être stockés dans une feuille de calcul, car il s'agit de la manière la plus courante de collecter des données dans de nombreux domaines. N'oubliez pas ce que nous avons appris dans la section [2.2.4.1](#) : ces ensembles de données de type “tableurs” sont appelés `data.frame` dans R, et nous nous concentrerons sur l'utilisation de ces objets tout au long de ce livre.

Commençons par charger les packages nécessaires pour ce chapitre (cela suppose que vous les ayez déjà installés; relisez la section [2.3](#) pour plus d'informations sur l'installation et le chargement des packages R si vous ne l'avez pas déjà fait). Au début de chaque chapitre, nous aurons systématiquement besoin de charger quelques packages. Donc n'oubliez pas de les installer au préalable si besoin.

```
# Pensez à installer ces packages avant de les charger si besoin
library(dplyr)
library(nycflights13)
```

---

### 3.1 Le package `nycflights13`

Nous avons probablement déjà presque tous pris l'avion. Les grands aéroports contiennent de nombreuses portes d'embarquement, et pour chacune d'elles, des informations sur les vols en partance sont affichées. Par exemple, le numéro du vol, les heures de décollage et d'atterrissement prévues, les retards etc. Dans la mesure du possible, on aime arriver à destination à l'heure.

Dans la suite de ce document, on examinera les jeux de données du package nycflights13, notamment afin d'en apprendre plus sur les causes de retard les plus fréquentes.

Ce package contient 5 “tableaux” contenant des informations sur chaque vol intérieur ayant quitté New York en 2013, soit depuis l'aéroport de Newark Liberty International (EWR), soit depuis l'aéroport John F. Kennedy International (JFK), soit depuis l'aéroport LaGuardia (LGA) :

1. flights : informations sur chacun des 336776 vols
  2. airlines : traduction entre les codes IATA à 2 lettres des compagnies aériennes et leur nom complet (il y en a 16 au total)
  3. planes : informations constructeurs pour chacun des 3322 avions utilisés en 2013
  4. weather : données météorologiques heure par heure (environ 8705 observations) pour chacun des 3 aéroports de New York
  5. airports : noms et localisations géographiques des aéroports desservis (1458 aéroports)
- 

### 3.2 Le data frame flights

Nous allons commencer par explorer le jeu de données flights qui est inclus avec le package nycflights13 afin de nous faire une idée de sa structure. Dans votre script, tapez la commande suivante et exécutez la dans la console (selon les réglages de RStudio et la largeur de votre console, l'affichage peut varier légèrement) :

```
flights
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     1     1      517            515       2        830
2 2013     1     1      533            529       4        850
3 2013     1     1      542            540       2        923
4 2013     1     1      544            545      -1       1004
5 2013     1     1      554            600      -6        812
6 2013     1     1      554            558      -4        740
7 2013     1     1      555            600      -5        913
8 2013     1     1      557            600      -3        709
9 2013     1     1      557            600      -3        838
10 2013    1     1      558            600      -2        753
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

Essayons de déchiffrer cet affichage :

- A tibble: 336,776 x 19 : un tibble est un data.frame amélioré. Il a toutes les caractéristiques d'un data.frame, (tapez `class(flights)` pour vous en convaincre), mais en plus, il a quelques propriétés intéressantes sur lesquelles nous reviendrons plus tard. Ce tibble possède donc :
  - 336776 lignes
  - 19 colonnes, qui correspondent aux variables. Dans un tibble, les observations sont toujours en ligne et les variables en colonnes.
- year, month, day, dep\_time, sched\_dep\_time... sont les noms des colonnes, c'est à dire les variables de ce jeu de données.
- Nous avons ensuite les 10 premières lignes du tableau qui correspondent à 10 vols.
- ... with 336,766 rows, and 12 more variables, nous indique que 336766 lignes et 12 variables ne logent pas à l'écran. Ces données font toutefois partie intégrante du tableau `flights`.
- le nom et le type de chaque variable qui n'a pas pu être affichée à l'écran

Cette façon d'afficher les tableaux est spécifique des tibbles. Vous noterez que le type de chaque variable est indiqué entre < ... >. Les types que vous pourrez rencontrer sont les suivants :

- `<int>` : nombres entiers (“integers”)
- `<dbl>` : nombres réels (“doubles”)
- `<chr>` : caractères (“characters”)
- `<fct>` : facteurs (“factors”)
- `<ord>` : facteurs ordonnés (“ordinals”)
- `<lgl>` : logiques (colonne de vrais/faux : “logical”)
- `<date>` : dates
- `<time>` : heures
- `<dttm>` : combinaison de date et d'heure (“date time”)

Cette façon d'afficher le contenu d'un tableau permet d'y voir (beaucoup) plus clair que l'affichage classique d'un data.frame. Malheureusement, ce n'est pas toujours suffisant. Voyons quelles sont les autres méthodes permettant d'explorer un data.frame.

---

### 3.3 Explorer un data.frame

Parmi les nombreuses façons d'avoir une idée des données contenues dans un data.frame tel que `flights`, on présente ici 2 fonctions qui prennent le nom du data.frame en guise d'argument et un opérateur :

- la fonction `View()` intégrée à RStudio. C'est celle que vous utiliserez le plus souvent. Attention, elle s'écrit avec un “V” majuscule.
- la fonction `glimpse()` chargée avec le package `dplyr`. Elle est très similaire à la fonction `str()` découverte dans les tutoriels de DataCamp.

- l'opérateur `$` permet d'accéder à une unique variable d'un `data.frame`.
  - la fonction `skim()` du package `skimr` permet d'obtenir un résumé complet mais très synthétique et visuel des variables d'un `data.frame`.

### 3.3.1 View()

Tapez `View(flights)` dans votre script et exécutez la commande. Un nouvel onglet contenant ce qui ressemble à un tableauur doit s'ouvrir.

**Question** : à quoi correspondent chacune des lignes de ce tableau ?

- A. aux données d'une compagnie aérienne
  - B. aux données d'un vol
  - C. aux données d'un aéroport
  - D. aux données de plusieurs vols

Ici, vous pouvez donc explorer la totalité du tableau, passer chaque variable en revue, et même appliquer des filtres pour ne visualiser qu'une partie des données. Par exemple, essayez de déterminer combien de vols ont décollé de l'aéroport JFK le 12 février.

Ce tableau n'est pas facile à manipuler. Il est impossible de corriger des valeurs, et lorsque l'on applique des filtres, il est impossible de récupérer uniquement les données filtrées. Nous verrons plus tard comment les obtenir en tapant des commandes simples dans un script. La seule utilité de ce tableau est donc l'exploration visuelle des données.

### 3.3.2 glimpse()

La seconde façon d'explorer les données contenues dans un tableau est d'utiliser la fonction `glimpse()` après avoir chargé le package `dplyr` :

```
glimpse(flights)
```

```
Rows: 336,776
Columns: 19

$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ~
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ dep_time        <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, ~
$ dep_delay       <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, ~
$ arr_time        <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, ~
```

```

$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846,~
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2,~
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV~
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, ~
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668~
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EW~
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FL~
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 1~
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, ~
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, ~
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0~
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 20~

```

Ici, les premières observations sont présentées en lignes pour chaque variable du jeu de données. Là encore, le type de chaque variable est précisé. Essayez d'identifier 3 variables catégorielles. À quoi correspondent-elles ? En quoi sont-elles différentes des variables numériques ?

### 3.3.3 L'opérateur \$

Enfin, l'opérateur \$ permet d'accéder à une unique variable grâce à son nom. Par exemple le tableau `airlines` contient seulement 2 variables :

```

airlines

# A tibble: 16 x 2
  carrier name
  <chr>   <chr>
1 9E      Endeavor Air Inc.
2 AA      American Airlines Inc.
3 AS      Alaska Airlines Inc.
4 B6      JetBlue Airways
5 DL      Delta Air Lines Inc.
6 EV      ExpressJet Airlines Inc.
7 F9      Frontier Airlines Inc.
8 FL      AirTran Airways Corporation
9 HA      Hawaiian Airlines Inc.
10 MQ     Envoy Air
11 OO     SkyWest Airlines Inc.
12 UA     United Air Lines Inc.
13 US     US Airways Inc.
14 VX     Virgin America
15 WN     Southwest Airlines Co.
16 YV     Mesa Airlines Inc.

```

Nous pouvons accéder à ces variables grâce à leur nom :

```
airlines$name
```

```
[1] "Endeavor Air Inc."      "American Airlines Inc."
[3] "Alaska Airlines Inc."   "JetBlue Airways"
[5] "Delta Air Lines Inc."   "ExpressJet Airlines Inc."
[7] "Frontier Airlines Inc." "AirTran Airways Corporation"
[9] "Hawaiian Airlines Inc." "Envoy Air"
[11] "SkyWest Airlines Inc."  "United Air Lines Inc."
[13] "US Airways Inc."       "Virgin America"
[15] "Southwest Airlines Co." "Mesa Airlines Inc."
```

Cela nous permet de récupérer les données sous la forme d'un vecteur. Attention toutefois, le tableau `flights` contient tellement de lignes, que récupérer une variable grâce à cet opérateur peut rapidement saturer la console. Si, par exemple, vous souhaitez extraire les données relatives aux compagnies aériennes (colonne `carrier`) du tableau `flights`, vous pouvez taper ceci :

```
flights$carrier
```

Le résultat est pour le moins indigeste ! Lorsqu'un tableau contient de nombreuses lignes, c'est rarement une bonne idée de transformer l'une de ses colonnes en vecteur. Dans la mesure du possible, les données d'un tableau doivent rester dans le tableau.

### 3.3.4 `skim()`

Pour utiliser la fonction `skim()`, vous devez au préalable installer le package `skimr` :

```
install.packages("skimr")
```

Ce package est un peu “expérimental” et il se peut que l’installation pose problème. Si un message d’erreur apparaît lors de l’installation, procédez comme suit :

1. Quittez RStudio (sans oublier de sauvegarder votre travail au préalable)
2. Relancez RStudio et dans la console, tapez ceci :

```
install.packages("rlang")
```

3. Tentez d’installer `skimr` à nouveau.
4. Exécutez à nouveau votre script afin de retrouver votre travail dans l’état où il était avant de quitter RStudio.

Si l’installation de `skimr` s’est bien passée, vous pouvez maintenant taper ceci :

```
library(skimr)
skim(flights)
```

### 3.3.5 Les fichiers d'aide

Une fonctionnalité particulièrement utile de R est son système d'aide. On peut obtenir de l'aide au sujet de n'importe quelle fonction et de n'importe quel jeu de données en tapant un “?” immédiatement suivi du nom de la fonction ou de l'objet.

Par exemple, examinez l'aide du jeu de données flights :

```
?flights
```

Vous devriez absolument prendre l'habitude d'examiner les fichiers d'aide des fonctions ou jeux de données pour lesquels vous avez des questions. Ces fichiers sont très complets, et même s'il peuvent paraître impressionnantes au premier abord, ils sont tous structurés sur le même modèle et vous aideront à comprendre comment utiliser les fonctions, quels sont les arguments possibles, à quoi ils servent et comment les utiliser.

Prenez le temps d'examiner le fichier d'aide du jeu de données flights. Avant de passer à la suite, assurez-vous d'avoir compris à quoi correspondent chacune des 19 variables de ce tableau.

---

## 3.4 Exercices

Consultez l'aide du jeu de données diamonds du package ggplot2.

- Quel est le code de la couleur la plus prisée ?
- Quel est le code de la moins bonne clarté ?
- À quoi correspond la variable z ?
- En quoi la variable depth est-elle différente de la variable z ?

Consultez l'aide du package nycflights13 en tapant `help(package="nycflights13")`.

- Consultez l'aide des 5 jeux de données de ce package.
- À quoi correspond la variable visib ?
- Dans quel tableau se trouve-t'elle ?
- Combien de lignes possède ce tableau ?

## 4 Visualiser des données avec ggplot2

Dans les chapitres 2 et 3, nous avons vu ce qui me semble être les concepts essentiels avant de commencer à explorer en détail des données dans R. Les éléments de syntaxe abordés dans la section 2.2 sont nombreux et vous n'avez probablement pas tout retenu. C'est pourquoi je vous conseille de garder les tutoriels de DataCamp à portée de main afin de pouvoir refaire les parties que vous maîtrisez le moins. Ce n'est qu'en répétant plusieurs fois ces tutoriels que les choses seront vraiment comprises et que vous les retiendrez. Ainsi, si des éléments de code présentés

ci-dessous vous semblent obscurs, revenez en arrière : toutes les réponses à vos questions se trouvent probablement dans les chapitres précédents.

Après la découverte des bases du langage R, nous abordons maintenant les parties de ce livre qui concernent la “science des données” (ou “Data Science” pour nos amis anglo-saxons). Nous allons voir dans ce chapitre qu’outre les fonctions `View()` et `glimpse()`, l’exploration visuelle via la représentation graphique des données est un moyen indispensable et très puissant pour comprendre ce qui se passe dans un jeu de données. **La visualisation de vos données devrait toujours être un préambule à toute analyse statistique.**

La visualisation des données est en outre un excellent point de départ quand on découvre la programmation sous R, car ses bénéfices sont clairs et immédiats : vous pouvez créer des graphiques élégants et informatifs qui vous aident à comprendre les données. Dans ce chapitre, vous allez donc plonger dans l’art de la visualisation des données, en apprenant la structure de base des graphiques réalisés avec `ggplot2` qui permettent de transformer des données numériques et catégorielles en graphiques.

Toutefois, la visualisation seule ne suffit généralement pas. Il est en effet souvent nécessaire de transformer les données pour produire des représentations plus parlantes. Ainsi, dans les chapitres 5 et 6, vous découvrirez les verbes clés qui vous permettront de sélectionner des variables importantes, de filtrer des observations, de créer de nouvelles variables, de calculer des résumés, d’associer des tableaux ou de les remettre en forme.

Ce n’est qu’en combinant les transformations de données et représentations graphiques d’une part, avec votre curiosité et votre esprit critique d’autre part, que vous serez véritablement en mesure de réaliser une analyse exploratoire utile de vos données. C’est la seule façon d’identifier des questions intéressantes et pertinentes sur vos données, afin de tenter d’y répondre par les analyses statistiques et la modélisation par la suite.

---

## 4.1 Prérequis

Dans ce chapitre, nous aurons besoin des packages suivants :

```
library(ggplot2)
library(nycflights13)
library(dplyr)
```

Si ce n'est pas déjà fait, pensez à les installer avant de les charger en mémoire.

Au niveau le plus élémentaire, les graphiques permettent de comprendre comment les variables se comparent en termes de tendance centrale (à quel endroit les valeurs ont tendance à être localisées, regroupées) et leur dispersion (comment les données varient autour du centre). La chose la plus importante à savoir sur les graphiques est qu’ils doivent être créés pour que votre public (le professeur qui vous évalue, le collègue avec qui vous collaborez, votre futur employeur, etc.) comprenne bien les résultats et les informations que vous souhaitez transmettre. Il s’agit d’un exercice d’équilibrisme : d’une part, vous voulez mettre en évidence

autant de relations significatives et de résultats intéressants que possible, mais de l'autre, vous ne voulez pas trop en inclure, afin d'éviter de rendre votre graphique illisible ou de submerger votre public. Tout comme n'importe quel paragraphe de document écrit, un graphique doit permettre de **communiquer un message** (une idée forte, un résultat marquant, une hypothèse nouvelle, etc.).

Comme nous le verrons, les graphiques nous aident également à repérer les tendances extrêmes et les valeurs aberrantes dans nos données. Nous verrons aussi qu'une façon de faire, assez classique, consiste à comparer la distribution d'une variable quantitative pour les différents niveaux d'une variable catégorielle.

---

## 4.2 La grammaire des graphiques

Les lettres gg du package `ggplot2` sont l'abréviation de “grammar of graphics” : la grammaire des graphiques. De la même manière que nous construisons des phrases en respectant des règles grammaticales précises (usage des noms, des verbes, des sujets et adjectifs...), la grammaire des graphiques établit un certain nombre de règles permettant de construire des graphiques : elle précise les composants d'un graphique en suivant le cadre théorique défini par Wilkinson (2005).

### 4.2.1 Éléments de la grammaire

En bref, la grammaire des graphiques nous dit que :

Un graphique est l'association (mapping) de données/variables (data) à des attributs esthétiques (aesthetics) d'objets géométriques (geometric objects).

Pour clarifier, on peut disséquer un graphique en 3 éléments essentiels :

1. data : le jeu de données contenant les variables que l'on va associer à des objets géométriques
2. geom : les objets géométriques en question. Cela fait référence aux types d'objets que l'on peut observer sur le graphique (des points, des lignes, des barres, etc.)
3. aes : les attributs esthétiques des objets géométriques présents sur le graphique. Par exemple, la position sur les axes x et y, la couleur, la taille, la transparence, la forme, etc. Chacun de ces attributs esthétiques peut-être associé à une variable de notre jeu de données.

Examinons un exemple pour bien comprendre.

### 4.2.2 Gapminder

En février 2006, un statisticien du nom de Hans Rosling a donné un TED Talk intitulé “[The best stats you've ever seen](#)”. Au cours de cette conférence, Hans Rosling présente des données

sur l'économie mondiale, la santé et le développement des pays du monde. Les données sont disponibles [sur ce site](#) et dans [le package gapminder](#).

Pour l'année 2007, le jeu de données contient des informations pour 142 pays. Examinons les premières lignes de ce jeu de données :

Table 2 – Les 6 premières lignes du jeu de données ‘gapminder’ pour l’année 2007.

Country	Continent	Life Expectancy	Population	GDP per Capita
Afghanistan	Asia	43.828	31889923	974.5803
Albania	Europe	76.423	3600523	5937.0295
Algeria	Africa	72.301	33333216	6223.3675
Angola	Africa	42.731	12420476	4797.2313
Argentina	Americas	75.320	40301927	12779.3796
Australia	Oceania	81.235	20434176	34435.3674

Pour chaque ligne, les variables suivantes sont décrites :

- Country : le pays
- Continent : le continent
- Life Expectancy : espérance de vie à la naissance
- Population : nombre de personnes vivant dans le pays
- GDP per Capita : produit intérieur brut (PIB) par habitant en dollars américains. GDP est l’abréviation de “Growth Domestic Product”. C’est un indicateur de l’activité économique d’un pays, parfois utilisé comme une approximation du revenu moyen par habitant.

Examinons maintenant la figure 1 qui représente ces variables pour chacun des 142 pays de ce jeu de données (notez l’utilisation de la notation scientifique dans la légende).

Si on décrypte ce graphique du point de vue de la grammaire des graphiques, on voit que :

- la variable GDP per Capita est associée à l'aesthetic x de la position des points
- la variable Life Expectancy est associée à l'aesthetic y de la position des points
- la variable Population est associée à l'aesthetic size (taille) des points
- la variable Continent est associée à l'aesthetic color (couleur) des points

Ici, l’objet géométrique (ou geom) qui représente les données est le point. Les données (ou data) sont contenues dans le tableau gapminder et chacune de ces variables est associée (mapping) aux caractéristiques esthétiques des points.

#### 4.2.3 Autres éléments de la grammaire des graphiques

Outre les éléments indispensables évoqués ici (data, mapping, aes, et geom), il existe d’autres aspects de la grammaire des graphiques qui permettent de contrôler l’aspect des graphiques. Ils ne sont pas toujours indispensables. Nous en verrons néanmoins quelque-uns particulièrement utiles :

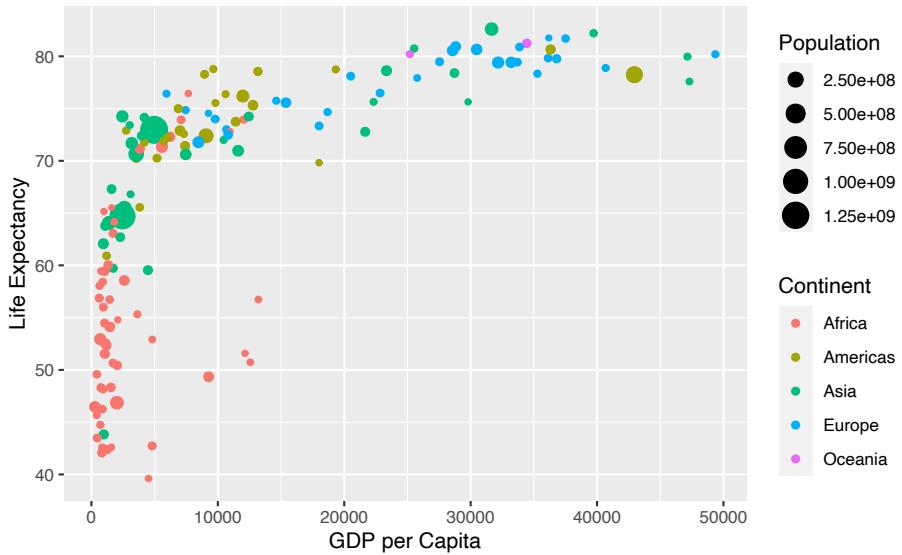


Figure 1 – Espérance de vie en fonction du PIB par habitant en 2007.

- facet : c'est un moyen très pratique de scinder le jeu de données en plusieurs sous-groupes et de produire automatiquement un graphique pour chacun d'entre eux.
- position : permet notamment de modifier la position des barres d'un barplot.
- labs : permet de définir les titres, sous-titres et légendes des axes d'un graphique
- theme : permet de modifier l'aspect général des graphiques en appliquant des thèmes prédéfinis ou en modifiant certains aspects de thèmes existants

#### 4.2.4 Le package ggplot2

Comme indiqué plus haut, le package `ggplot2` (Wickham et al., 2021) permet de réaliser des graphiques dans R en respectant les principes de la grammaire des graphiques. Vous avez probablement remarqué que depuis le début de la section 4.2, beaucoup de termes sont écrits dans la police réservée au code informatique. C'est parce que les éléments de la grammaire des graphiques sont tous précisés dans la fonction `ggplot()` qui demande, au grand minimum, que les éléments suivants soient spécifiés :

- le nom du `data.frame` contenant les variables qui seront utilisées pour le graphique. Ce nom correspond à l'argument `data` de la fonction `ggplot()`.
- l'association des variables à des attributs esthétiques. Cela se fait grâce à l'argument `mapping` et la fonction `aes()`

Après avoir spécifié ces éléments, on ajoute des couches supplémentaires au graphique grâce au signe `+`. La couche la plus essentielle à ajouter à un graphique, est une couche contenant un élément géométrique, ou `geom` (par exemple des points, des lignes ou des barres). D'autres couches peuvent s'ajouter pour spécifier des titres, des facets ou des modifications des axes et des thèmes du graphique.

Dans le cadre de ce cours, nous nous limiterons aux 5 types de graphiques suivants :

1. les nuages de points
  2. les graphiques en lignes
  3. les boîtes à moustaches ou boxplots
  4. les histogrammes
  5. les diagrammes bâtons
- 

### 4.3 Les nuages de points

C'est probablement le plus simple des 5 types de graphiques cités plus haut. Il s'agit de graphiques bi-variés pour lesquels une variable est associée à l'axe des abscisses, et une autre est associée à l'axe des ordonnées. Comme pour le graphique présenté à la figure 1 ci-dessus, d'autres variables peuvent être associées à des caractéristiques esthétiques des points (transparence, taille, couleur, forme...).

Ici, dans le jeu de données `flights`, nous allons nous intéresser à la relation qui existe entre :

1. `dep_delay` : le retard des vols au décollage, que nous placerons sur l'axe des "x"
2. `arr_delay` : le retard des mêmes vols à l'atterrissement, que nous placerons sur l'axe des "y"

Afin d'avoir un jeu de données plus facile à utiliser, nous nous contenterons de visualiser les vols d'Alaska Airlines, dont le code de compagnie aérienne est "AS".

```
alaska_flights <- flights %>%
  filter(carrier == "AS")
```

Il est normal que vous ne compreniez pas encore les commandes ci-dessus : elles seront décrites dans le chapitre 6. Retenez juste que nous avons créé un nouveau tableau, nommé `alaska_flights`, qui contient toutes les informations des vols d'Alaska Airlines. Commencez par examiner ce tableau avec la fonction `View()`. En quoi est-il différent du tableau `flights` ?

#### 4.3.1 La couche de base : la fonction `ggplot()`

La fonction `ggplot()` permet d'établir la première base du graphique. C'est grâce à cette fonction que l'on précise quel jeu de données utiliser et quelles variables placer sur les axes :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay))
```

Ce graphique est pour le moins vide : c'est normal, nous n'avons pas encore spécifié la couche contenant l'objet géométrique que nous souhaitons utiliser.

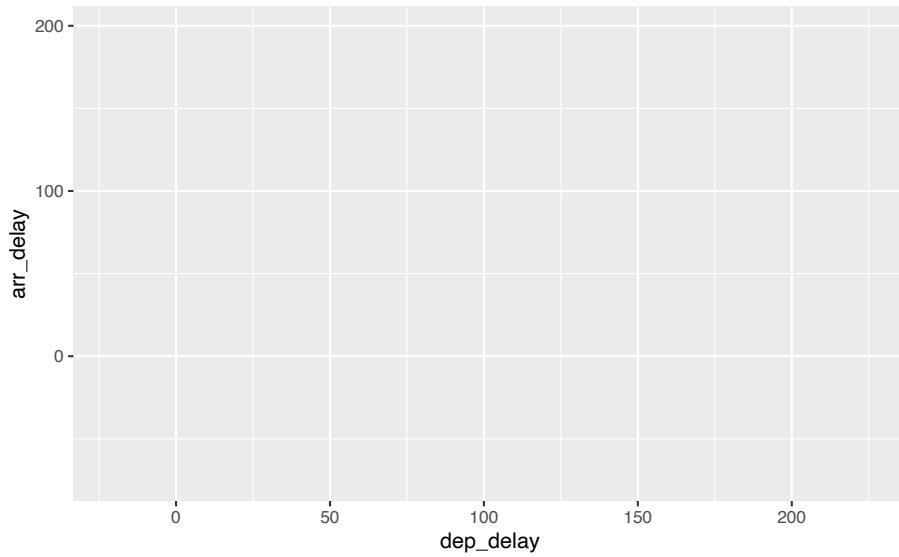


Figure 2 – Un graphique sans ‘geom’.

#### 4.3.2 Ajout d'une couche supplémentaire : l'objet géométrique

Les nuages de points sont créés par la fonction `geom_point()`:

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

`Warning: Removed 5 rows containing missing values (geom_point).`

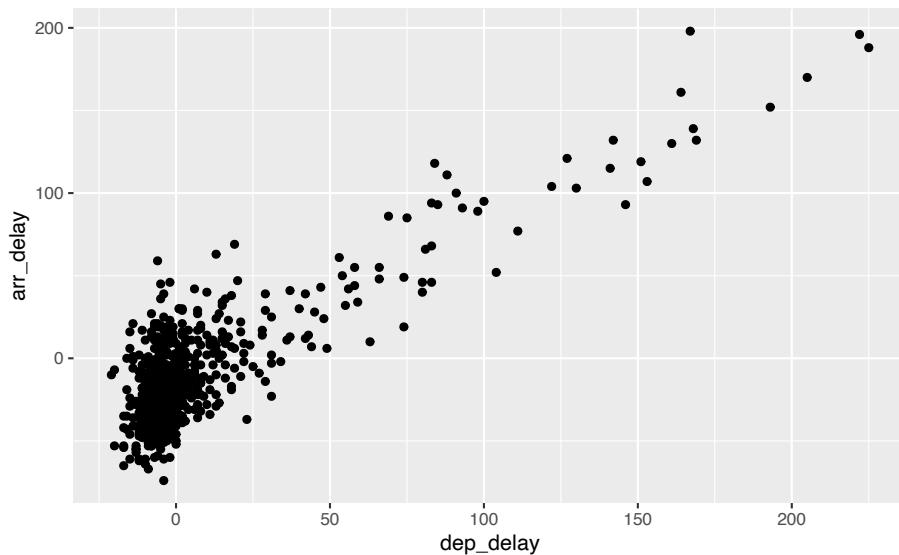


Figure 3 – Retards à l'arrivée en fonction des retards au décollage pour les vols d'Alaska Airlines au départ de New York City en 2013.

Plusieurs choses importantes sont à remarquer sur la figure 3 :

1. le graphique présente maintenant une couche supplémentaire constituée de points.
2. la fonction `geom_point()` nous prévient que 5 lignes contenant des données manquantes n'ont pas été intégrées au graphique. Les données manquent soit pour une variable, soit pour l'autre, soit pour les 2. Il est donc impossible de les faire apparaître sur le graphique.
3. il existe une relation positive entre `dep_delay` et `arr_delay` : quand le retard d'un vol au décollage augmente, le retard de ce vol augmente aussi à l'arrivée.
4. Enfin, il y a une grande majorité de points centrés près de l'origine (0,0).

Si je résume cette syntaxe :

- Au sein de la fonction `ggplot()`, on spécifie 2 composants de la grammaire des graphiques :
  1. le nom du tableau contenant les données grâce à l'argument `data = alaska_flights`
  2. l'association (`mapping`) des variables à des caractéristiques esthétiques (`aes()`) en précisant `aes(x = dep_delay, y = arr_delay)` :
    - la variable `dep_delay` est associée à l'esthétique de position `x`
    - la variable `arr_delay` est associée à l'esthétique de position `y`
- On ajoute une couche au graphique `ggplot()` grâce au symbole `+`. La couche en question précise le troisième élément indispensable de la grammaire des graphiques : l'objet géométrique. Ici, les objets sont des points. On le spécifie grâce à la fonction `geom_point()`.

Quelques remarques concernant les couches :

- Notez que le signe `+` est placé à *la fin de la ligne*. Vous recevrez un message d'erreur si vous le placez au début.
- Quand vous ajoutez une couche à un graphique, je vous encourage vivement à presser la touche `enter` de votre clavier juste après le symbole `+`. Ainsi, le code correspondant à chaque couche sera sur une ligne distincte, ce qui augmente considérablement la lisibilité de votre code.
- Comme indiqué dans la section [2.2.4.3](#), tant que les arguments d'une fonction sont spécifiés dans l'ordre, on peut se passer d'écrire leur nom. Ainsi, les deux blocs de commande suivants produisent exactement le même résultat :

```
# Le nom des arguments est précisé
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()

# Le nom des arguments est omis
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

#### 4.3.3 Exercices

1. Donnez une raison pratique expliquant pourquoi les variables `dep_delay` et `arr_delay` ont une relation positive

2. Quelles variables (pas nécessairement dans le tableau alaska\_flights) pourraient avoir une corrélation négative (relation négative) avec dep\_delay ? Pourquoi ? Rappelez-vous que nous étudions ici des variables numériques.
3. Selon vous, pourquoi tant de points sont-ils regroupés près de (0, 0) ? À quoi le point (0,0) correspond-il pour les vols d'Alaska Airlines ?
4. Citez les éléments de ce graphique/de ces données qui vous sautent le plus aux yeux ?
5. Créez un nouveau nuage de points en utilisant d'autres variables du jeu de données alaska\_flights

#### 4.3.4 Over-plotting

L'over-plotting est la superposition importante d'une grande quantité d'information sur une zone restreinte d'un graphique. Dans notre cas, nous observons un over-plotting important autour de (0,0). Cet effet est gênant car il est difficile de se faire une idée précise du nombre de points accumulés dans cette zone. La façon la plus simple de régler le problème est de modifier la transparence des points grâce à l'argument alpha de la fonction geom\_point(). Par défaut, cette valeur est fixée à 1, pour une opacité totale. Une valeur de 0 rend les points totalement transparents, et donc invisibles. Trouver la bonne valeur peut demander de tâtonner un peu. Le code suivant produit la figure 4 :

```
ggplot(data = alaska_flights,
       mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.2)
```

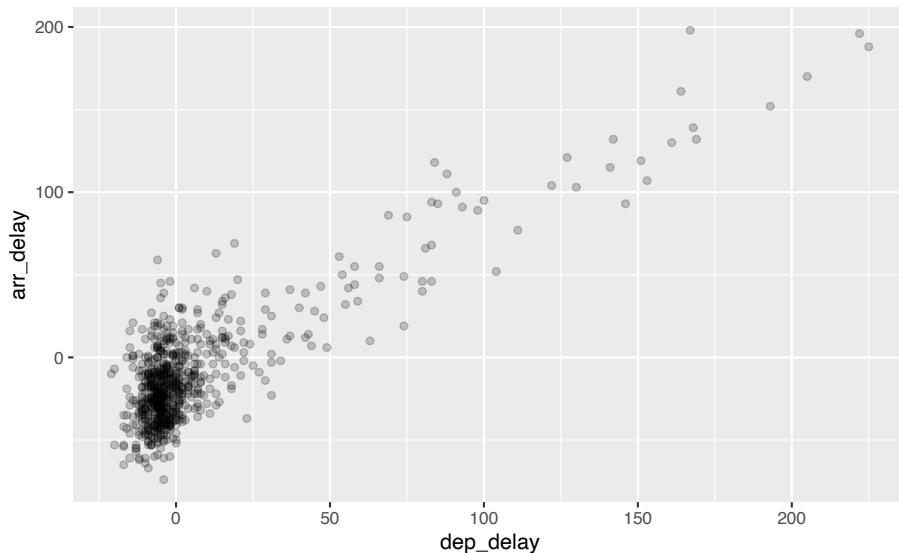


Figure 4 – La même figure, avec des points semi-transparents.

Sur cette figure, notez que :

- la transparence est additive : plus il y a de points, plus la zone est foncée car les points se superposent et rendent la zone plus opaque.

- l'argument `alpha` = n'est pas intégré à l'intérieur d'une fonction `aes()` car ici, il n'est pas associé à une variable : c'est un simple paramètre.

L'over-plotting est souvent rencontré lorsque l'on représente plusieurs nuages de points pour les différentes valeurs d'une variable catégorielle. Par exemple, si on transforme la variable `month` en facteur (`factor(month)`), on peut regarder s'il existe une relation entre les retards à l'atterrissement et le mois de l'année :

```
ggplot(data = alaska_flights,
       mapping = aes(x = factor(month), y = arr_delay)) +
  geom_point()
```

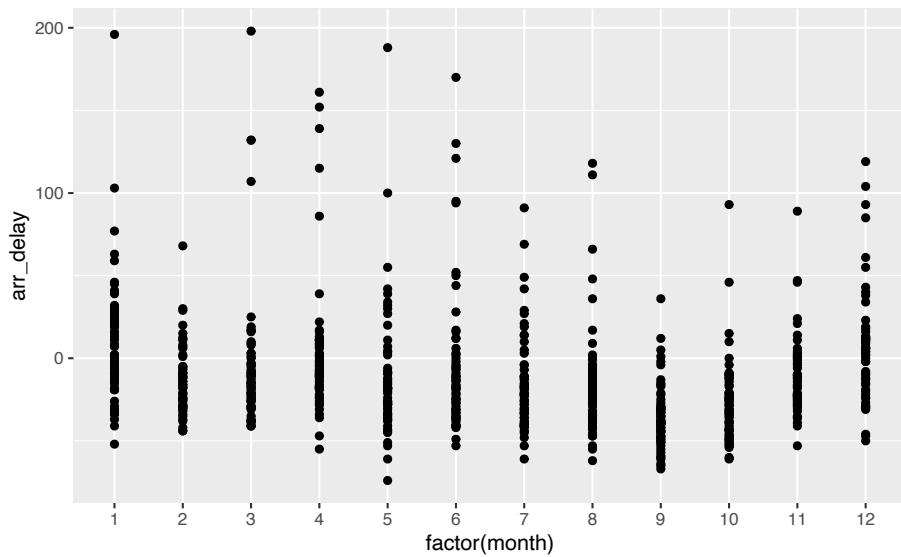


Figure 5 – Retards à l'arrivée pour les 12 mois de l'année 2013.

Ici (figure 5), l'ajout de transparence ne serait pas suffisant. Une autre solution est d'appliquer la méthode dite du “jittering”, ou tremblement. Elle consiste à ajouter un bruit aléatoire horizontal et/ou vertical aux points d'un graphique. Ici, on peut ajouter un léger bruit horizontal afin de disperser un peu les points pour chaque mois de l'année. On n'ajoute pas de bruit vertical car on ne souhaite pas que les valeurs de retard (sur l'axe des y) soient altérées :

```
ggplot(data = alaska_flights,
       mapping = aes(x = factor(month), y = arr_delay)) +
  geom_jitter(width = 0.25)
```

On y voit déjà plus clair. L'argument `width` permet de spécifier l'intensité de la dispersion horizontale. Pour ajouter du bruit vertical (ce qui n'est pas souhaitable ici!), on peut ajouter l'argument `height`. Le graphique de la figure 6 est parfois appelé un “stripchart”. C'est un graphique du type “nuage de points”, mais pour lequel l'une des 2 variables est numérique, et l'autre est catégorielle.

Il est évidemment possible d'ajouter de la transparence :

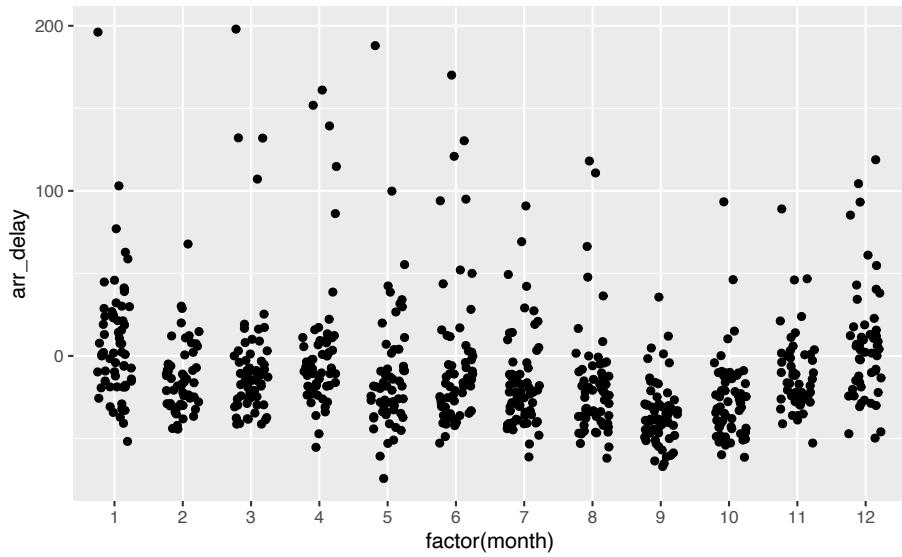


Figure 6 – Retards à l'arrivée pour les 12 mois de l'année 2013.

```
ggplot(data = alaska_flights,
       mapping = aes(x = factor(month), y = arr_delay)) +
  geom_jitter(width = 0.25, alpha = 0.5)
```

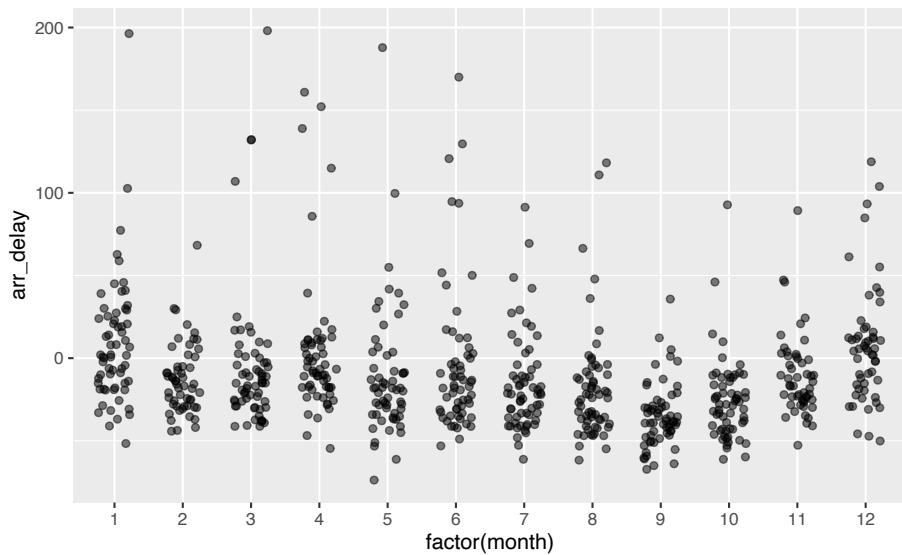


Figure 7 – Retards à l'arrivée pour les 12 mois de l'année 2013.

#### 4.3.5 Couleur, taille et forme

L'argument `color` (ou `colour`, les deux orthographies fonctionnent) permet de spécifier la couleur des points. L'argument `size` permet de spécifier la taille des points. L'argument `shape` permet de spécifier la forme utilisée en guise de symbole. Ces 3 arguments peuvent être utilisés

comme des paramètres, pour modifier l'ensemble des points d'un graphique. Mais ils peuvent aussi être associés à une variable, pour apporter une information supplémentaire.

Comparez les deux graphiques suivants (figures 8 et 9) :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(color = "blue")
```

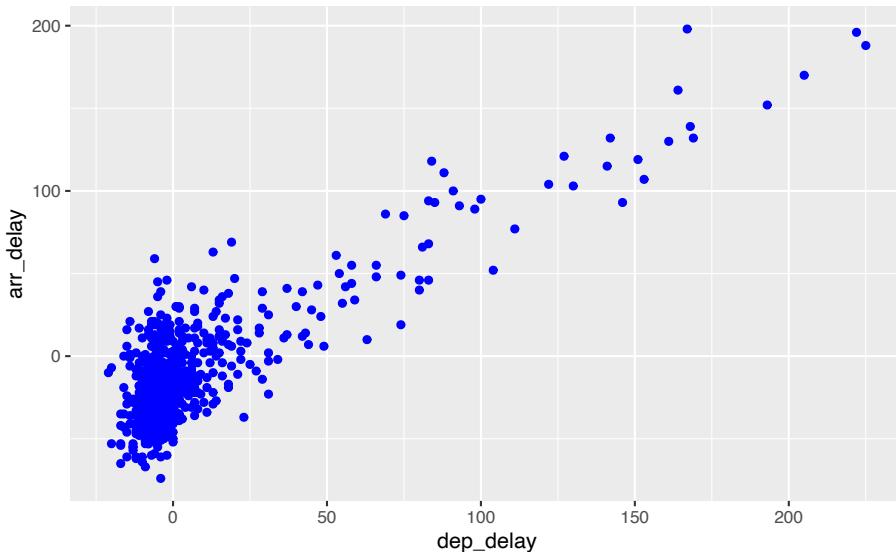


Figure 8 – Utilisation correcte de ‘color’.

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(aes(color = "blue"))
```

Le code qui permet de produire la figure 8 fait un usage correct de l'argument color. On demande des points de couleur bleue, les points apparaissent bleus. La figure 9 en revanche ne produit pas le résultat attendu. Puisque nous avons mis l'argument color à l'intérieur de la fonction aes(), R s'attend à ce que la couleur soit associée à une variable. Puisqu'aucune variable ne s'appelle “blue”, R utilise la couleur par défaut. Pour associer la couleur des points à une variable, nous devons fournir un nom de variable valide :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(aes(color = factor(month)))
```

Ici, l'utilisation de la couleur est correcte. Elle est associée à une variable catégorielle, et chaque valeur possible du vecteur month se voit donc attribuer une couleur différente.

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(aes(color = arr_time))
```

De la même façon, la couleur des points est ici associée à une variable continue (l'heure d'arrivée des vols). Les points se voient donc attribuer une couleur choisie le long d'un gradient.

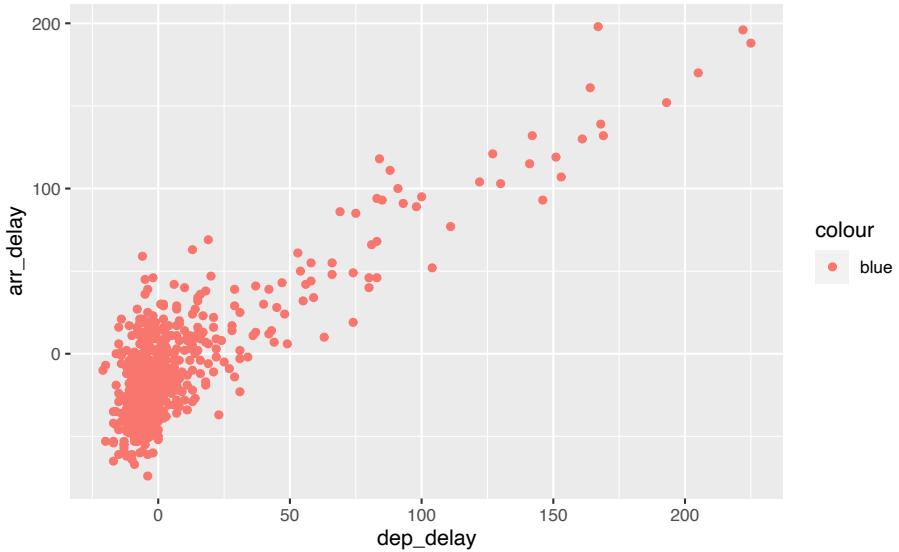


Figure 9 – Utilisation incorrecte de ‘color’.

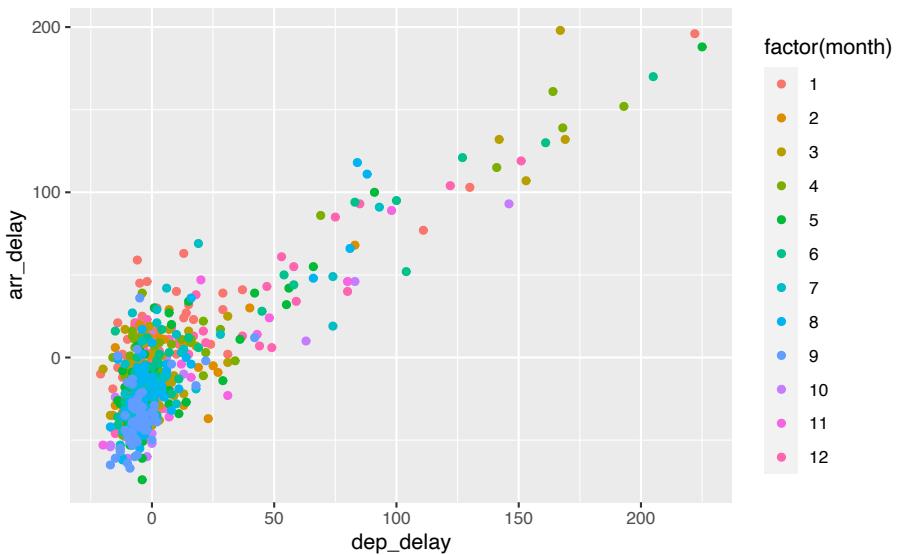


Figure 10 – Association de ‘color’ à une variable catégorielle.

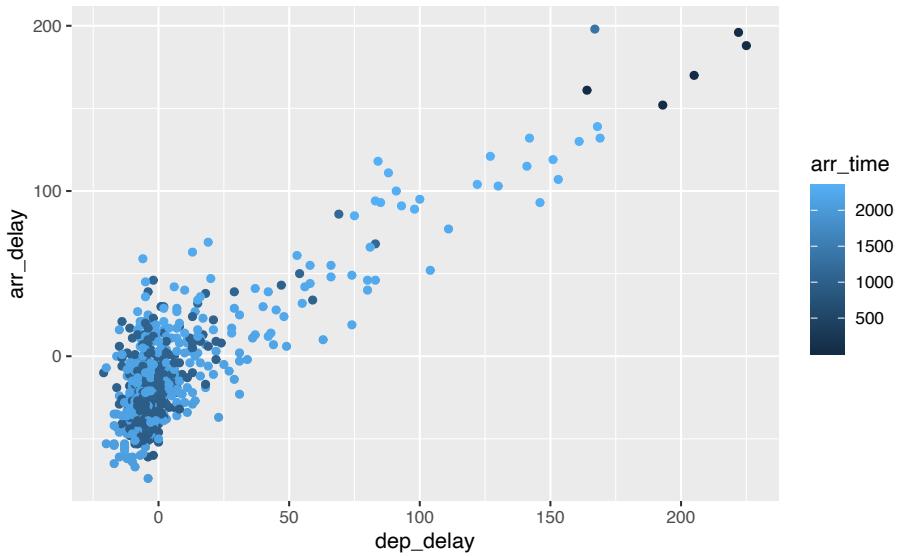


Figure 11 – Association de ‘color’ à une variable numérique.

La même approche peut être utilisée pour spécifier la forme des symboles avec l’argument shape. Attention toutefois : une variable continue ne peut pas être associée à shape

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point(aes(shape = factor(month)))
```

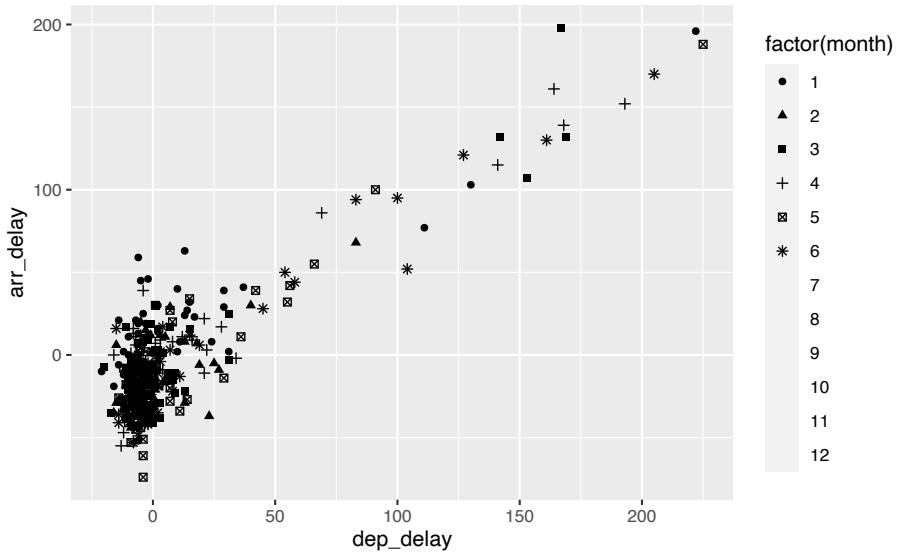


Figure 12 – Association de ‘shape’ à un facteur.

Vous noterez que seuls les 6 premiers niveaux d’un facteur se voient attribuer une forme automatiquement. Au delà de 6 symboles différents sur un même graphique, le résultat est souvent illisible. Il est possible d’ajouter plus de 6 symboles, mais cela demande de modifier la légende manuellement et concrètement nous n’en aurons jamais besoin. Lorsque plus de 6 séries

doivent être distinguées, d'autres solutions bien plus pertinentes (par exemple les factets) devraient être utilisées.

Comme pour la couleur, il est possible d'utiliser l'argument shape en tant que paramètre du graphique sans l'associer à une variable. Il faut alors fournir un code compris entre 0 et 24 :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(shape = 4)
```

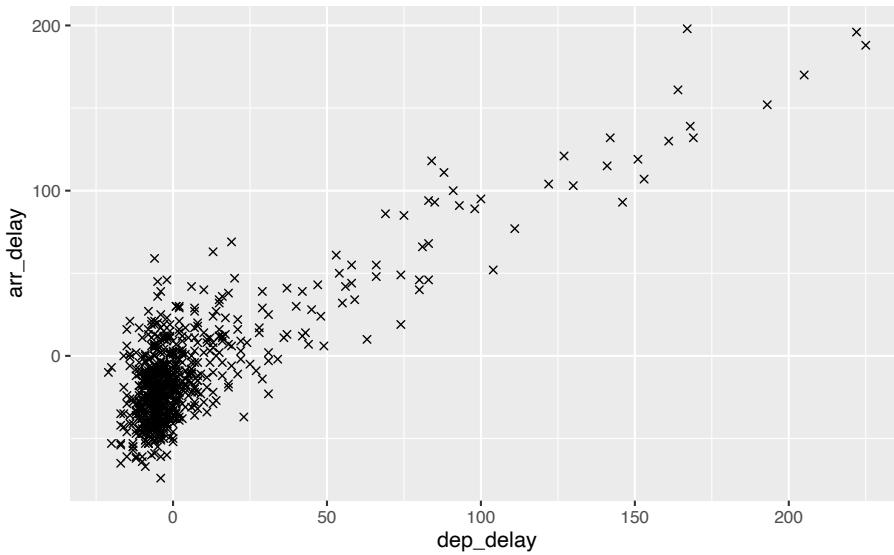


Figure 13 – Utilisation de 'shape' en tant que paramètre.

Notez qu'ici, ggplot() ne crée pas de légende : tous les points ont le même symbole, ce symbole n'est pas associé à une variable, une légende est donc inutile.

Parmi les valeurs possibles pour shape, les symboles 21 à 24 sont des symboles dont on peut spécifier séparément la couleur de contour, avec color et la couleur de fond avec fill :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(shape = 21, fill = "steelblue", color = "orange", alpha = 0.5)
```

N'hésitez pas à zoomer pour bien observer les points et comprendre ce qui se passe. Un conseil, faites des choix raisonnables ! Trop de couleurs n'est pas forcément souhaitable.

Enfin, on peut ajuster la taille des symboles avec l'argument size. Tout comme il n'est pas possible d'associer une variable continue à shape, il n'est pas conseillé d'associer une variable catégorielle nominale (c'est-à-dire un facteur non ordonné) à size. Associer une variable continue est en revanche parfois utile :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point(aes(size = arr_time), alpha = 0.1)
```

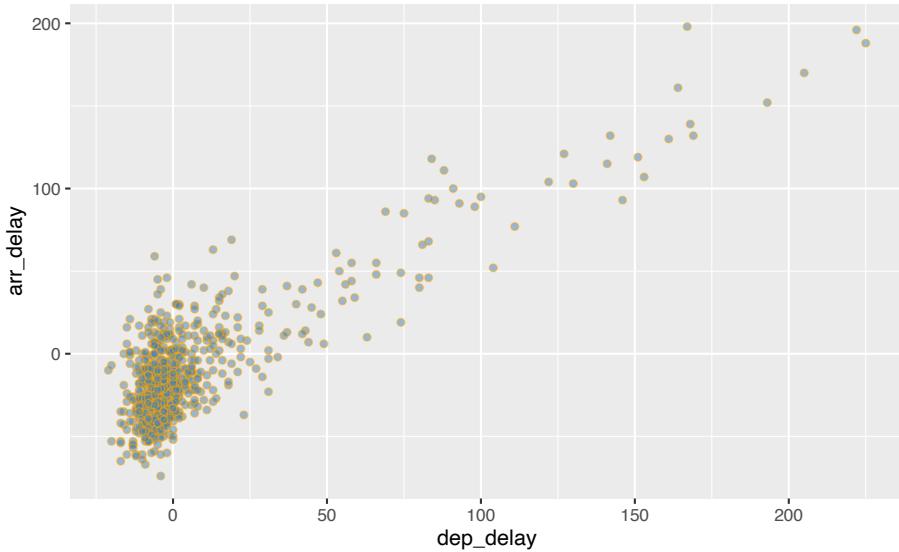


Figure 14 – Utilisation de ‘shape’, ‘color’ et ‘fill’.

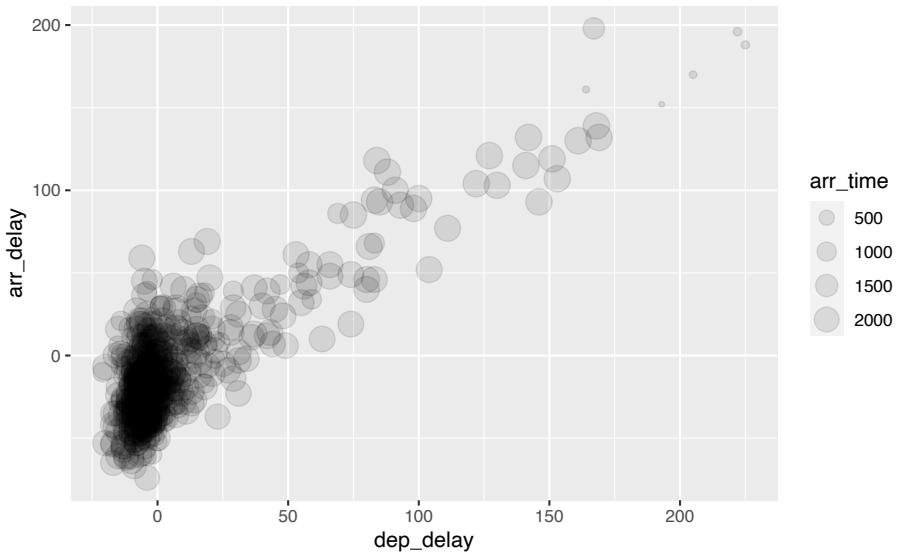


Figure 15 – Association d’une variable continue à la taille des symboles avec l’argument ‘size’.

Si l'over-plotting est ici très important (c'est pourquoi j'ai utilisé alpha), on constate néanmoins que les vols avec les retards les plus importants sont presque tous arrivés très tôt dans la journée ("500" signifie 5h00 du matin). Il s'agit probablement de vols qui devaient arriver dans la nuit, avant minuit, et qui sont finalement arrivés en tout début de journée, entre 00h01 et 5h00 du matin. Comme pour les autres arguments, il est possible d'utiliser size avec une valeur fixe, la même pour tous les symboles, lorsque cet argument n'est pas associé à une variable.

Enfin un conseil : évitez de trop surcharger vos graphiques. En combinant l'ensemble de ces arguments, il est malheureusement très facile d'obtenir des graphiques peu lisibles, ou contenant tellement d'informations qu'ils en deviennent difficiles à déchiffrer. Faites preuve de modération :

```
ggplot(data = alaska_flights,
       mapping = aes(x = dep_delay, y = arr_delay, size = arr_time)) +
  geom_point(alpha = 0.6,
             shape = 22,
             color = "orange",
             fill = "steelblue",
             stroke = 2)
```

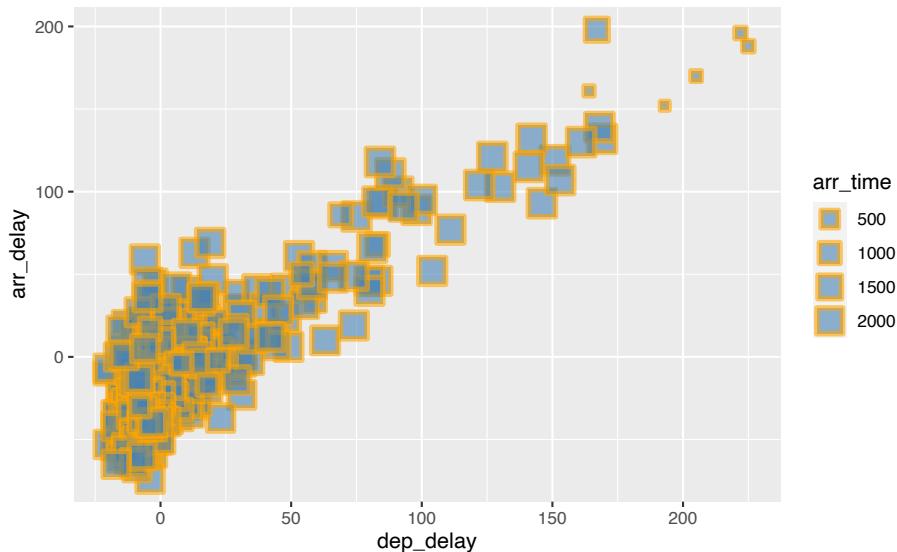


Figure 16 – Sometimes, less is more !

#### 4.3.6 Exercices

1. À quoi sert l'argument stroke ?
2. Avec le jeu de données diamonds, tapez les commandes suivantes pour créer un nouveau tableau diams contenant moins de lignes (5000 au lieu de près de 54000) :

```
library(dplyr)
set.seed(4532) # Afin que tout le monde récupère les mêmes lignes
```

```
diams <- diamonds %>%
  sample_n(5000)
```

3. Avec ce nouveau tableau diams, tapez le code permettant de créer le graphique 17 (Indice : affichez le tableau diams dans la console afin de voir quelles sont les variables disponibles).

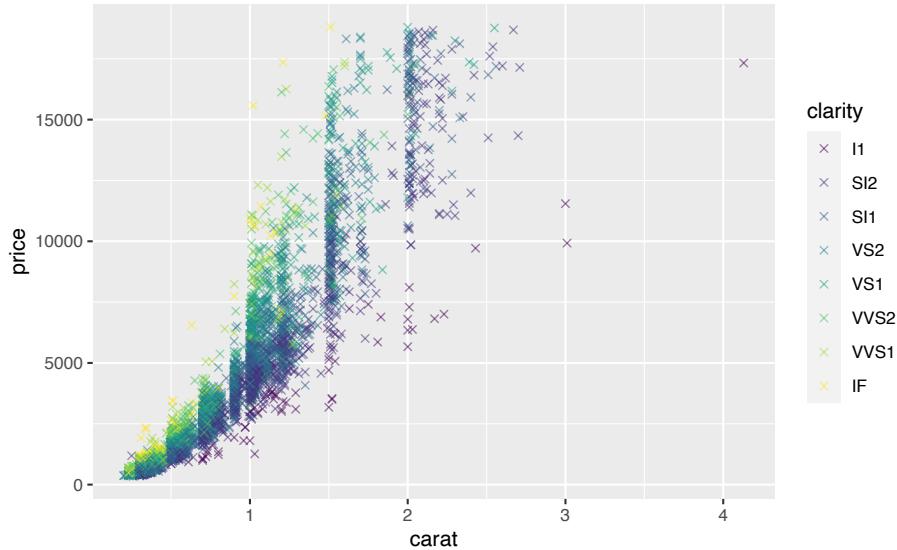


Figure 17 – Prix de 5000 diamants en fonction de leur taille en carats et de leur clarté.

4. Selon vous, à quoi sont dues les bandes verticales que l'on observe sur ce graphique ?
- 

## 4.4 Les graphiques en lignes

### 4.4.1 Un nouveau jeu de données

Les graphiques en ligne, ou “linegraphs” sont généralement utilisés lorsque l’axe des x porte une information **temporelle**, et l’axe des y une autre variable numérique. Le temps est une variable naturellement ordonnée : les jours, semaines, mois, années, se suivent naturellement. Les graphiques en lignes devraient être évités lorsqu'il n'y a pas une organisation séquentielle évidente de la variable portée par l'axe des x.

Concentrons nous maintenant sur le tableau weather du package nycflights13. Explorez ce tableau en appliquant les méthodes vues dans le chapitre 3. N’oubliez pas de consultez l'aide de ce jeu de données.

```
weather
```

```
# A tibble: 26,115 x 15
  origin   year month    day hour   temp  dewp humid wind_dir
  <fct>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```

<chr> <int> <int> <int> <int> <dbl> <dbl> <dbl>     <dbl>
1 EWR      2013     1     1     1 39.0  26.1  59.4    270
2 EWR      2013     1     1     2 39.0  27.0  61.6    250
3 EWR      2013     1     1     3 39.0  28.0  64.4    240
4 EWR      2013     1     1     4 39.9  28.0  62.2    250
5 EWR      2013     1     1     5 39.0  28.0  64.4    260
6 EWR      2013     1     1     6 37.9  28.0  67.2    240
7 EWR      2013     1     1     7 39.0  28.0  64.4    240
8 EWR      2013     1     1     8 39.9  28.0  62.2    250
9 EWR      2013     1     1     9 39.9  28.0  62.2    260
10 EWR     2013     1     1    10 41    28.0  59.6    260
# ... with 26,105 more rows, and 6 more variables: wind_speed <dbl>,
#   wind_gust <dbl>, precip <dbl>, pressure <dbl>, visib <dbl>,
#   time_hour <dttm>

```

Nous allons nous intéresser à la variable `temp`, qui contient un enregistrement de température pour chaque heure de chaque jour de 2013 pour les 3 aéroports de New York. Cela représente une grande quantité de données, aussi, nous nous limiterons aux températures observées entre le premier et le 15 janvier, pour l'aéroport Newark uniquement.

```

small_weather <- weather %>%
  filter(origin == "EWR",
         month == 1,
         day <= 15)

```

La fonction `filter()` fonctionne sur le même principe que la fonction `subset()` découverte dans les tutoriels de DataCamp. Ici, nous demandons à R de créer un nouveau tableau de données, nommé `small_weather`, qui ne contiendra que les lignes correspondant à `origin = "EWR"`, `month = 1` et `day <= 15`, c'est à dire les données météorologiques de l'aéroport de Newark pour les 15 premiers jours de janvier 2013.

#### 4.4.2 Exercice

Avec `View()`, consultez le tableau nouvellement créé. Expliquez pourquoi la variable `time_hour` identifie de manière unique le moment où chaque mesure a été réalisée alors que ce n'est pas le cas de la variable `hour`.

#### 4.4.3 La fonction `geom_line()`

Les line graphs sont produits de la même façon que les nuages de points. Seul l'objet géométrique permettant de visualiser les données change. Au lieu d'utiliser `geom_point()`, on utilisera `geom_line()`:

```
ggplot(data = small_weather, mapping = aes(x = time_hour, y = temp)) +
  geom_line()
```

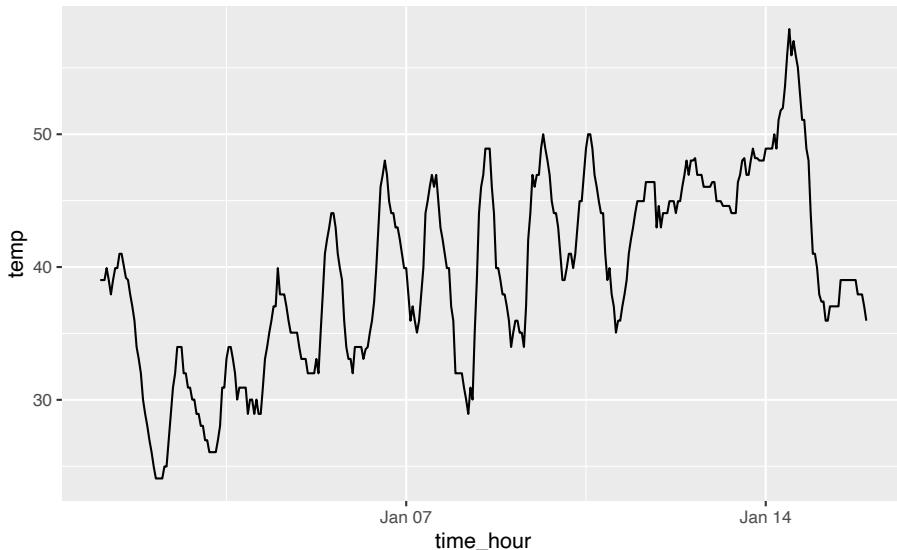


Figure 18 – Températures horaires à l'aéroport de Newark entre le 1er et le 15 janvier 2013.

Très logiquement, on observe des oscillations plus ou moins régulières qui correspondent à l'alternance jour/nuit. Notez l'échelle de l'axe des ordonnées : les températures sont enregistrées en degrés Fahrenheit.

Nous connaissons maintenant 2 types d'objets géométriques : les points et les lignes. Il est tout à fait possible d'ajouter plusieurs couches à un graphique, chacune d'elle correspondant à un objet géométrique différent (voir figure 19) :

```
ggplot(data = small_weather, mapping = aes(x = time_hour, y = temp)) +
  geom_line() +
  geom_point()
```

Enfin, comme pour les points, il est possible de spécifier plusieurs caractéristiques esthétiques des lignes, soit en les associant à des variables, au sein de la fonction `aes()`, soit en les utilisant en guise de paramètres pour modifier l'aspect général. Les arguments les plus classiques sont une fois de plus `color` (ou `colour`) pour modifier la couleur des lignes, `linetype` pour modifier le type de lignes (continues, pointillées, tirets, etc), et `size` pour modifier l'épaisseur des lignes.

Reprendons le jeu de données complet `weather`, et filtrons uniquement les dates comprises entre le premier et le 15 janvier, mais cette fois pour les 3 aéroports de New York :

```
small_weather_airports <- weather %>%
  filter(month == 1,
        day <= 15)
```

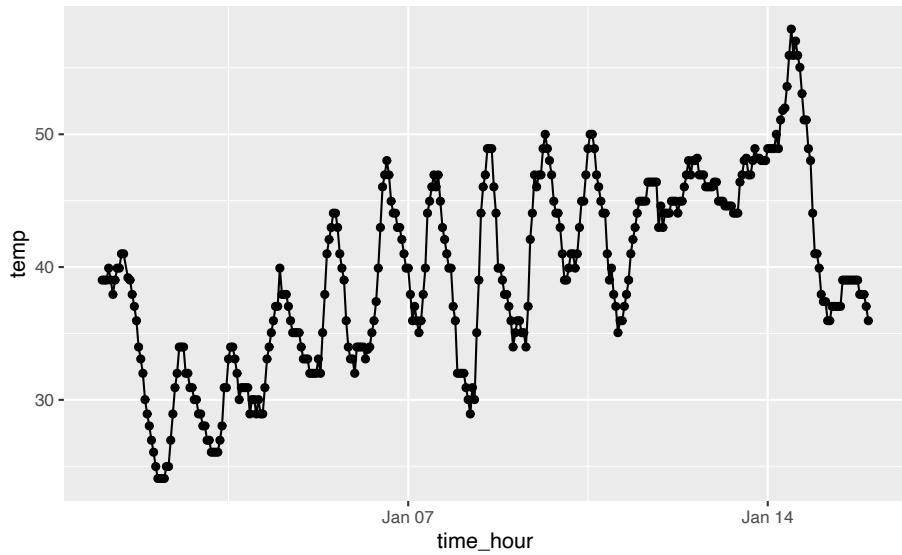


Figure 19 – Températures horaires à l'aéroport de Newark entre le 1er et le 15 janvier 2013.

Nous pouvons maintenant réaliser un “linegraph” sur lequel une courbe apparaîtra pour chaque aéroport. Pour cela, nous devons associer la variable `origin` à un attribut esthétique des lignes. Par exemple (figure 20) :

```
ggplot(data = small_weather_airports,
       mapping = aes(x = time_hour, y = temp)) +
  geom_line(aes(color = origin))
```

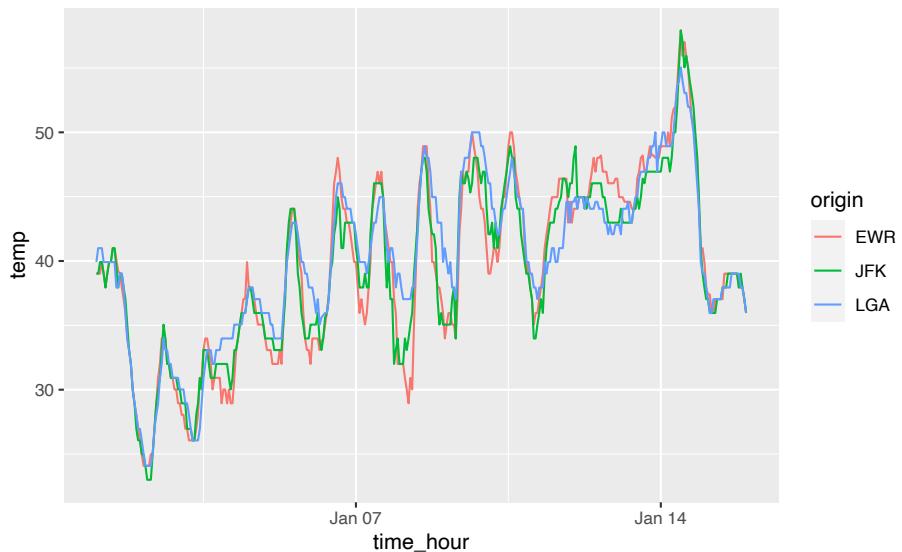


Figure 20 – Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

Ou bien (figure 21) :

```
ggplot(data = small_weather_airports,
       mapping = aes(x = time_hour, y = temp)) +
  geom_line(aes(linetype = origin))
```

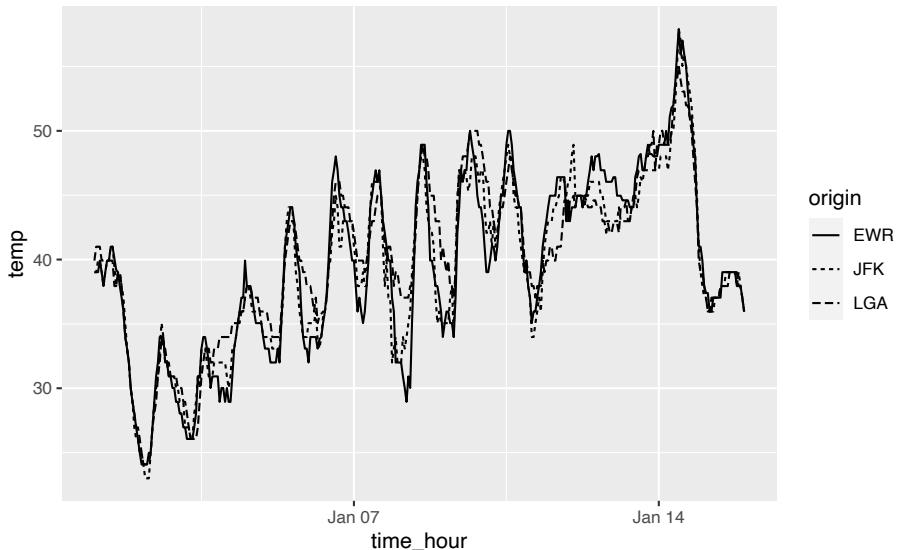


Figure 21 – Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

Ou encore (figure 22) :

```
ggplot(data = small_weather_airports,
       mapping = aes(x = time_hour, y = temp)) +
  geom_line(aes(color = origin, linetype = origin))
```

#### 4.4.4 À quel endroit placer `aes()` et les arguments `color`, `size`, etc.?

Jusqu'à maintenant, pour spécifier les associations entre certaines variables et les caractéristiques esthétiques d'un graphique, nous avons été amenés à utiliser la fonction `aes()` à 2 endroits distincts :

1. au sein de la fonction `ggplot()`
2. au sein des fonctions `geom_XXX()`

Comment choisir l'endroit où renseigner `aes()`? Pour bien comprendre, reprenons l'exemple du graphique 19 sur lequel nous avions ajouté 2 couches contenant chacune un objet géométrique différent (afin de gagner de la place, j'omets volontairement le nom des arguments `data` et `mapping` dans la fonction `ggplot()`):

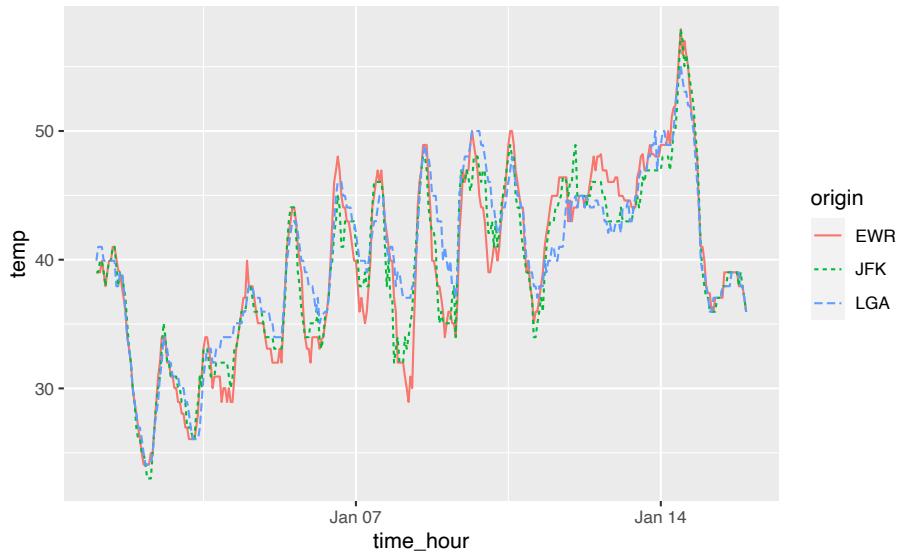


Figure 22 – Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +
  geom_line() +
  geom_point()
```

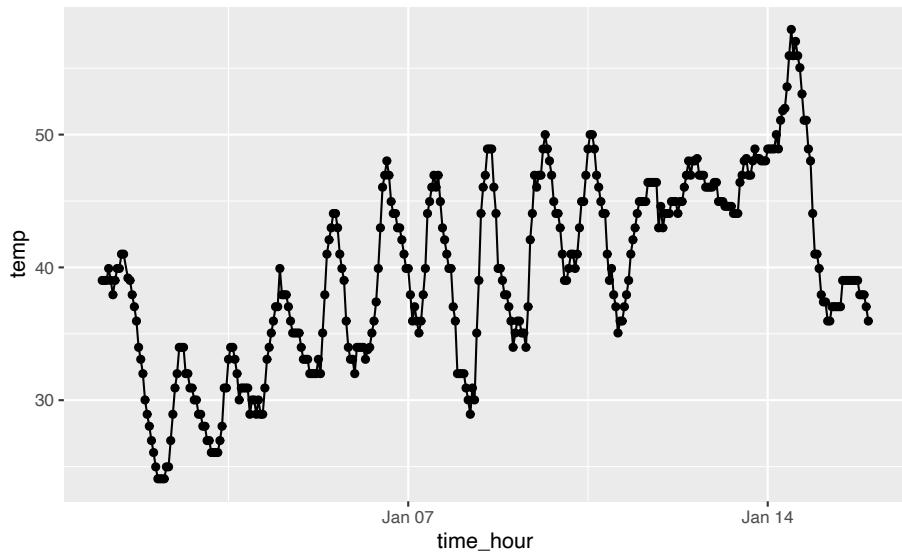


Figure 23 – Températures horaires à l'aéroport de Newark entre le 1er et le 15 janvier 2013.

Voyons ce qui se passe si on associe la variable `wind_speed` à l'esthétique `color`, à plusieurs endroits du code ci-dessus. Comparez les trois syntaxes et observez les différences entre les 3 graphiques obtenus :

```
ggplot(small_weather, aes(x = time_hour, y = temp, color = wind_speed)) +
  geom_line() +
  geom_point()
```

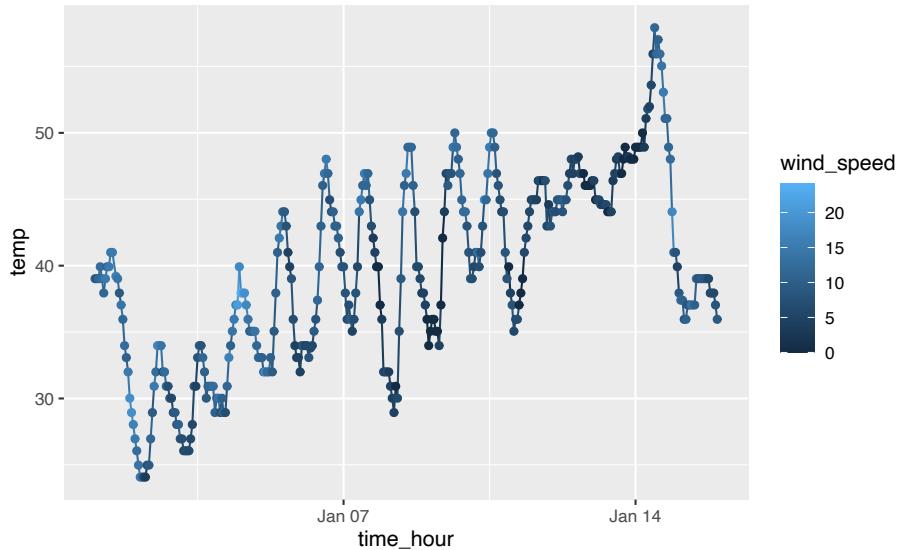


Figure 24 – Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur de la ligne et des points renseigne sur la vitesse du vent.

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +
  geom_line(aes(color = wind_speed)) +
  geom_point()
```

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +
  geom_line() +
  geom_point(aes(color = wind_speed))
```

Vous l'aurez compris, lorsque l'on spécifie `aes()` à l'intérieur de la fonction `ggplot()`, les associations de variables et d'esthétiques sont appliquées à tous les objets géométriques, donc à toutes les autres couches. En revanche, quand `aes()` est spécifié dans une couche donnée, les réglages ne s'appliquent qu'à cette couche spécifique.

En l'occurrence, si le même réglage est spécifié dans la fonction `ggplot()` et dans une fonction `geom_XXX()`, c'est le réglage spécifié dans l'objet géométrique qui l'emporte :

```
ggplot(small_weather, aes(x = time_hour, y = temp, color = wind_speed)) +
  geom_line(color = "orange") +
  geom_point()
```

Il est ainsi possible de spécifier des éléments esthétiques qui s'appliqueront à toutes les couches d'un graphique, et d'autres qui ne s'appliqueront qu'à une couche spécifique, qu'à un objet géométrique particulier.

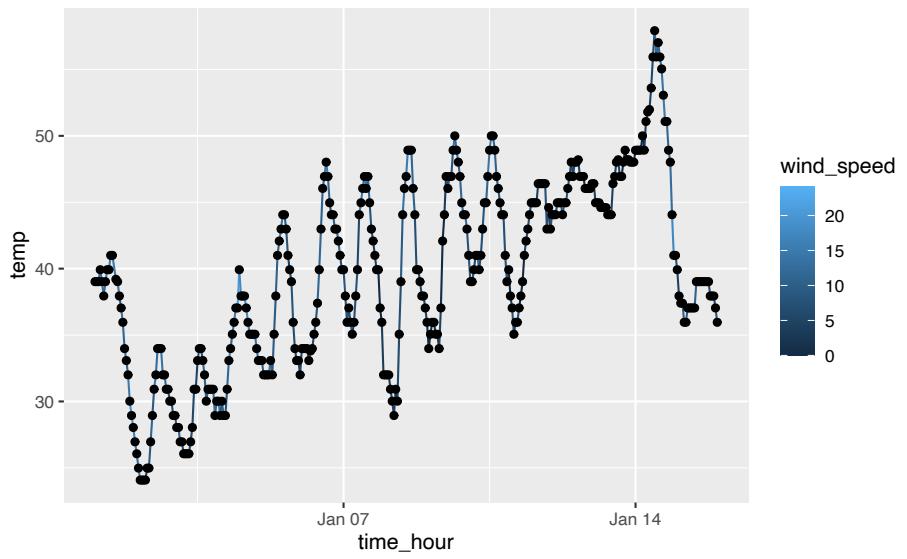


Figure 25 – Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur de la ligne renseigne sur la vitesse du vent.

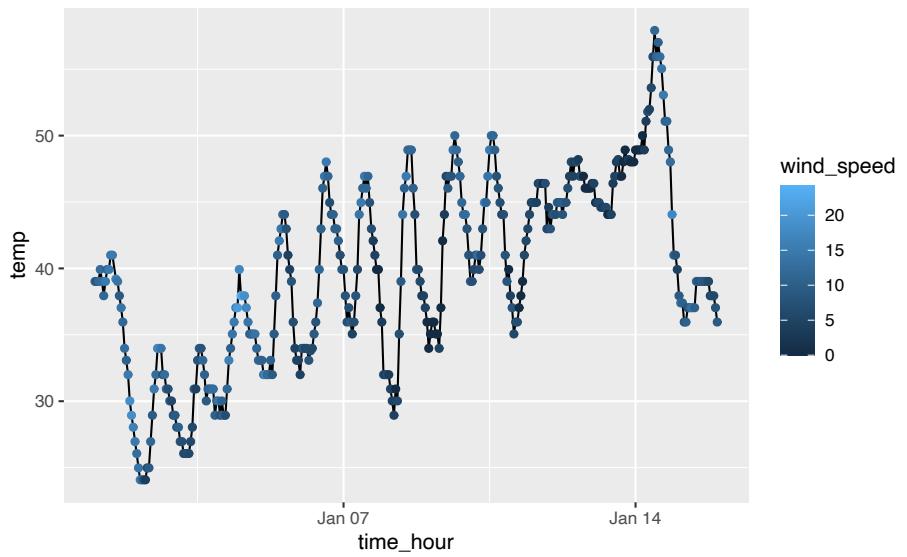


Figure 26 – Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur des points renseigne sur la vitesse du vent.

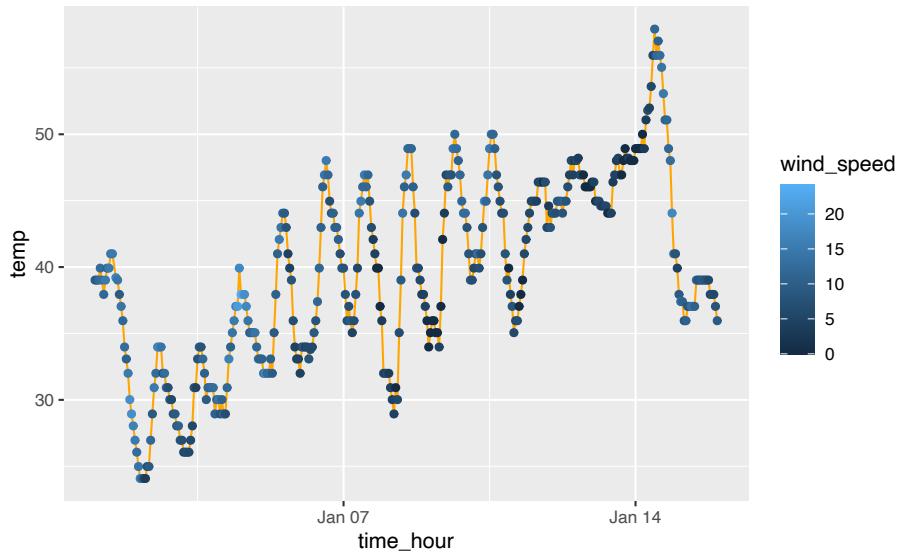


Figure 27 – Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013.

## 4.5 Les histogrammes

Un histogramme permet de visualiser la distribution **d'une variable** numérique continue. Contrairement aux deux types de graphiques vus précédemment, il sera donc inutile de préciser la variable à associer à l'axe des ordonnées : R la calcule automatiquement pour nous lorsque nous faisons appel à la fonction `geom_histogram()` pour créer un objet géométrique “histogramme”.

### 4.5.1 L'objet `geom_histogram()`

Si on reprend le jeu de données `weather`, on peut par exemple s'intéresser à la distribution des températures tout au long de l'année :

```
ggplot(weather, aes(x = temp)) +
  geom_histogram()
```

``stat_bin()` using `bins = 30``. Pick better value with ``binwidth``.

On observe plusieurs choses :

1. La distribution semble globalement bimodale avec un pic autour de 36-37 degrés Farenheit (2 à 3 °C) et un autre autour de 65-70 degrés Farenheit (18-21 °C).
2. Les températures ont varié entre 12 degrés Farenheit (-11°C) et 100 degrés Farenheit (près de 38°C).

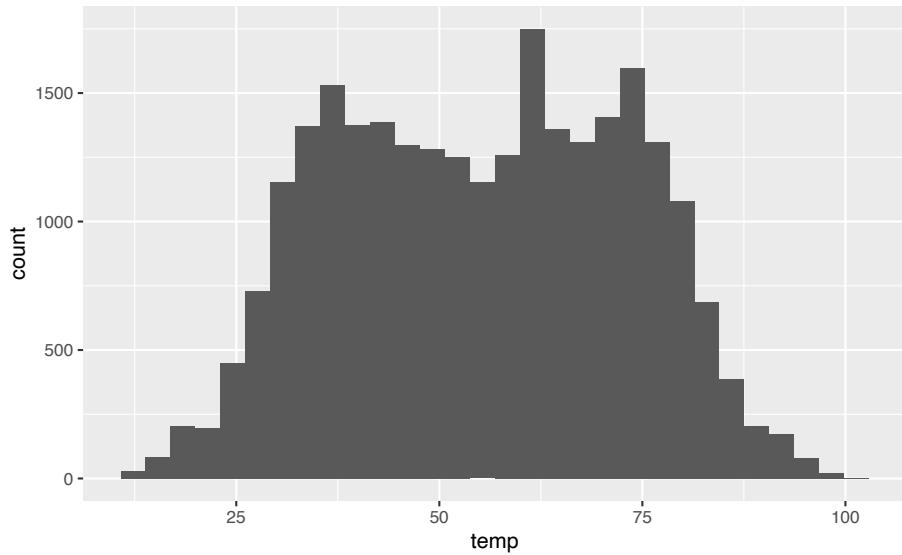


Figure 28 – Histogramme des températures enregistrées en 2013 dans les 3 aéroports de New York.

3. R nous avertit qu'une valeur non finie n'a pas pu être intégrée.
4. R nous indique qu'il a choisi de représenter 30 classes de températures (bins = 30). C'est la valeur par défaut. R nous conseille de choisir une valeur plus appropriée.

Comme pour les nuages de points utilisant les symboles 21 à 24, il est possible de spécifier la couleur de remplissage des barres avec l'argument `fill` et la couleur du contour des barres avec l'argument `color` :

```
ggplot(weather, aes(x = temp)) +
  geom_histogram(fill = "steelblue", color = "grey80")
```

#### 4.5.2 La taille des classes

Par défaut, R choisit arbitrairement de représenter 30 classes. Ce n'est que rarement le bon choix, et il est souvent nécessaire de tâtonner pour trouver le nombre de classes approprié : celui qui permet d'avoir une idée correcte de la distribution des données.

Il est possible d'ajuster les caractéristiques des classes de l'histogramme de l'une des 3 façons suivantes :

1. En ajustant le nombre de classes avec `bins`.
2. En précisant la largeur des classes avec `binwidth`.
3. En fournissant manuellement les limites des classes avec `breaks`.

```
ggplot(weather, aes(x = temp)) +
  geom_histogram(bins = 60, color = "white")
```

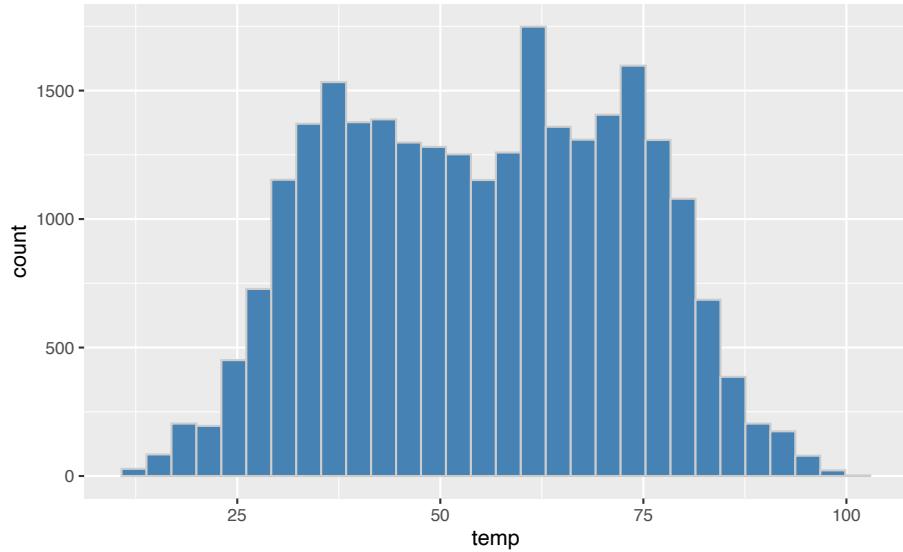


Figure 29 – Utilisation des arguments ‘fill’ et ‘color’ pour modifier l’aspect de l’histogramme.

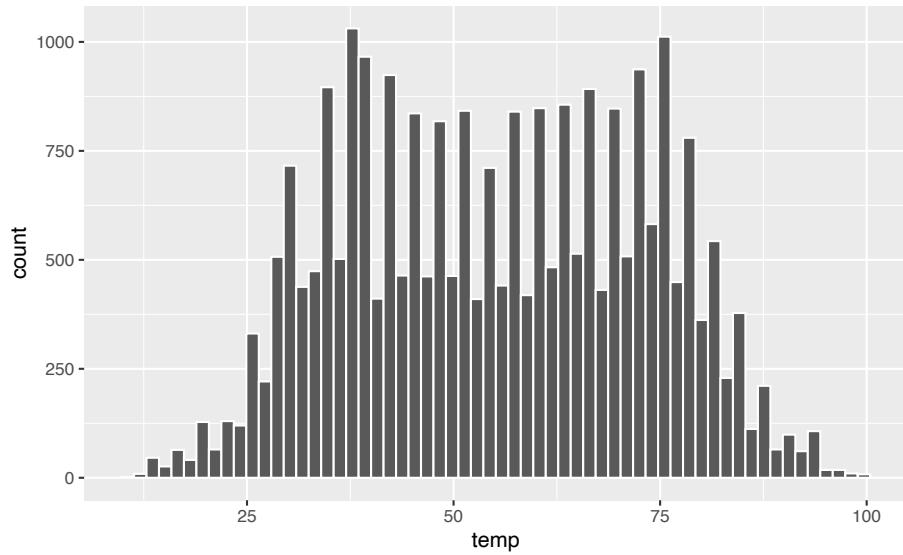


Figure 30 – Modification du nombre de classes.

Ici, augmenter le nombre de classes à 60 permet de prendre conscience que la distribution n'est pas aussi lisse qu'elle en avait l'air. L'ajout d'une couche supplémentaire avec la fonction `geom_rug()` ("a rug" est un tapis en français) permet de prendre conscience que les données de température ne sont pas aussi continues qu'on pouvait le croire (figure 31) :

```
ggplot(weather, aes(x = temp)) +
  geom_histogram(bins = 60, color = "white") +
  geom_rug(alpha = 0.1)
```

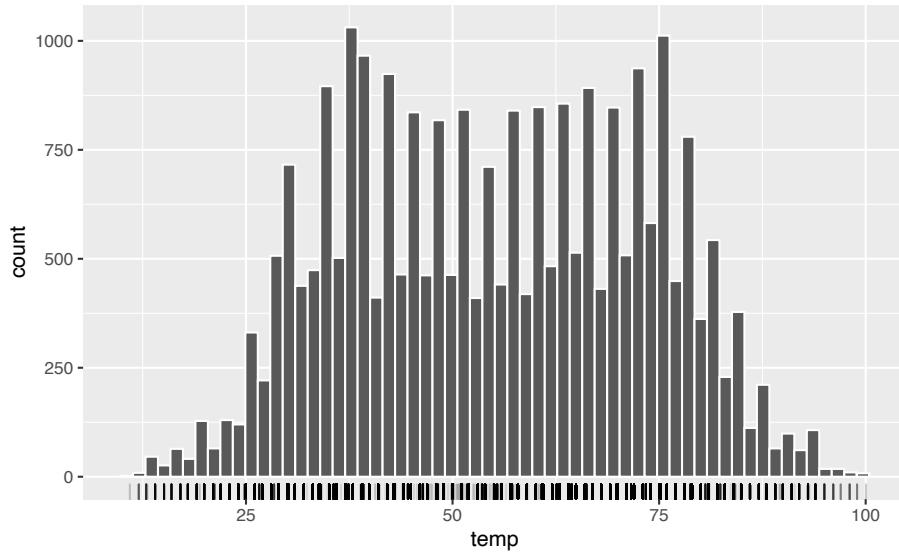


Figure 31 – Ajout des données brutes sous forme de 'tapis' ('rug') sous l'histogramme.

Notez la transparence importante utilisée pour `geom_rug()`. On constate que la précision des relevés de température n'est en fait que de quelques dixièmes de degrés.

On peut également modifier la largeur des classes avec `binwidth` :

```
ggplot(weather, aes(x = temp)) +
  geom_histogram(binwidth = 10, color = "white")
```

Ici, chaque catégorie recouvre 10 degrés Fahrenheit, ce qui est probablement trop large puisque la bimodalité de la distribution est devenue presque invisible.

Enfin, il est possible de déterminer manuellement les limites des classes souhaitées avec l'argument `breaks` (figure 33) :

```
ggplot(weather, aes(x = temp)) +
  geom_histogram(breaks = c(0, 10, 20, 50, 60, 70, 80, 105), color = "white")
```

Vous constatez ici que les choix effectués ne sont pas très pertinents : toutes les classes n'ont pas la même largeur. Cela rend l'interprétation difficile. Il est donc vivement conseillé, pour spécifier `breaks`, de créer des suites régulières, comme avec la fonction `seq()` (consultez son fichier d'aide et les exemples) :

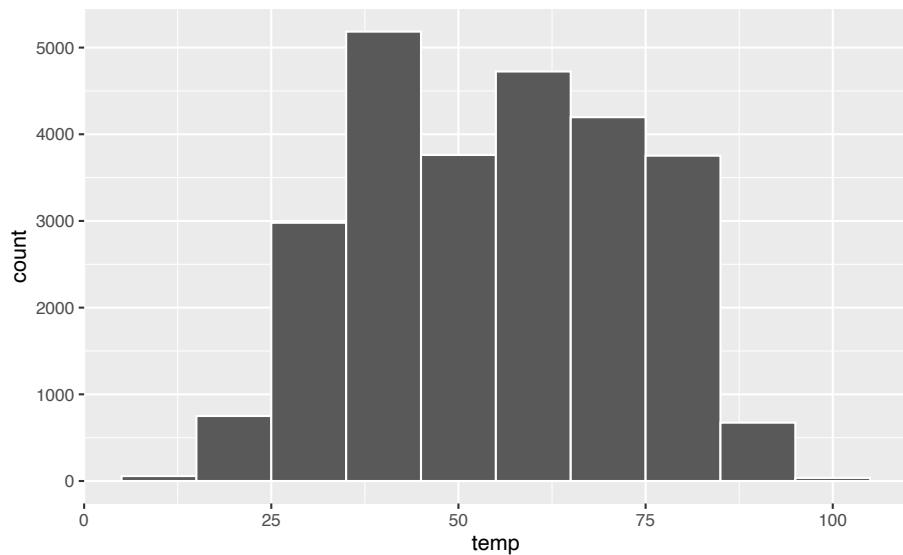


Figure 32 – Modification de la largeur des classes avec ‘binwidth’.

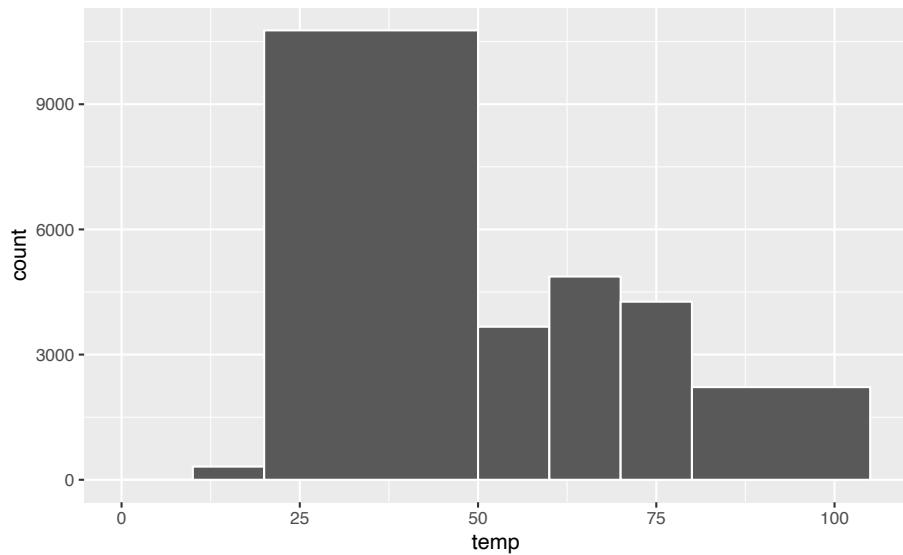


Figure 33 – Spécification manuelle des limites de classes de tailles (classes irrégulières).

```
limits <- seq(from = 10, to = 105, by = 5)  
limits
```

```
[1] 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85  
[17] 90 95 100 105
```

```
ggplot(weather, aes(x = temp)) +  
  geom_histogram(breaks = limits, color = "white")
```

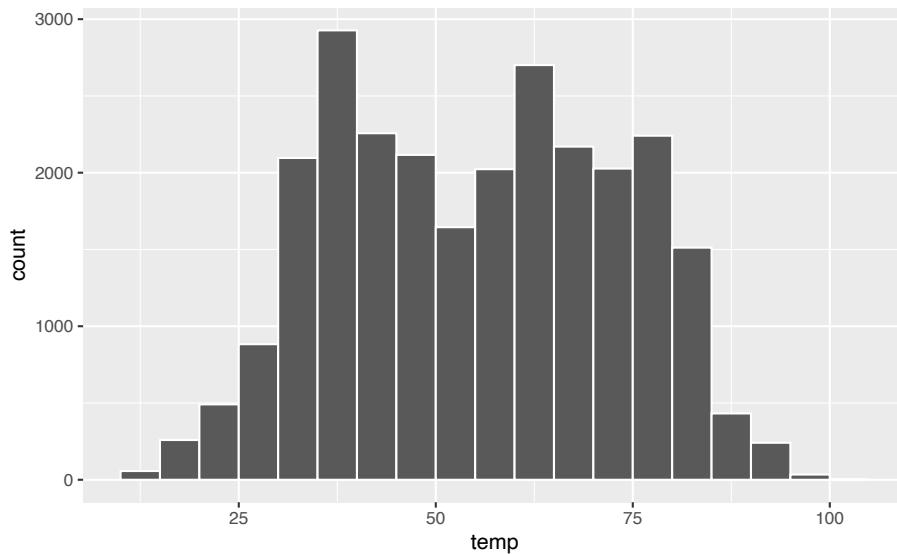


Figure 34 – Un exemple d'utilisation de l'argument ‘breaks’.

Il est important que toute la gamme des valeurs de temp soit couverte par les limites des classes que nous avons définies, sinon, certaines valeurs sont omises et l'histogramme est donc incomplet/incorrect. Une façon de s'en assurer est d'afficher le résumé des données pour la colonne temp du jeu de données weather :

```
summary(weather$temp)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
	10.94	39.92	55.40	55.26	69.98	100.04	1

On voit ici que les températures varient de 10.94 à 100.04 degrés Farenheit. Les classes que nous avons définies couvrent une plage de températures plus large (de 10 à 105). Toutes les données sont donc bien intégrées à l'histogramme.

## 4.6 Les facets

### 4.6.1 facet\_wrap()

Nous l'avons indiqué plus haut, les facets permettent de scinder le jeu de données en plusieurs sous-groupes et de faire un graphique pour chacun des sous-groupes.

Ainsi, si l'on souhaite connaître la distribution des températures pour chaque mois de l'année 2013, plutôt que de faire ceci :

```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30")
```

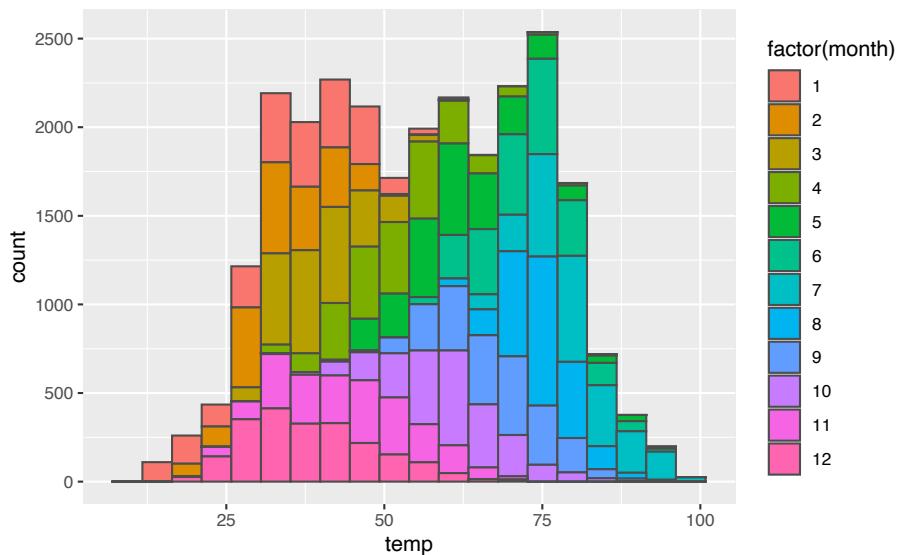


Figure 35 – Distribution des températures avec visualisation des données mensuelles.

qui produit un graphique certes assez joli, mais difficile à interpréter, mieux vaut faire ceci :

```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30") +  
  facet_wrap(~factor(month), ncol = 3)
```

La couche supplémentaire créée avec `facet_wrap()` permet donc de scinder les données en fonction d'une variable. Attention à la syntaxe : il ne faut pas oublier le symbole “~” devant la variable que l'on souhaite utiliser pour scinder les données. Il va sans dire que la variable utilisée doit être catégorielle et non continue, c'est la raison pour laquelle j'utilise la notation `factor(month)` et non simplement `month`.

Avec la fonction `facet_wrap()`, il est possible d'indiquer à R comment les différents graphiques doivent être agencés en spécifiant soit le nombre de colonnes souhaité avec `ncol`, soit le nombre de lignes souhaité avec `nrow`.

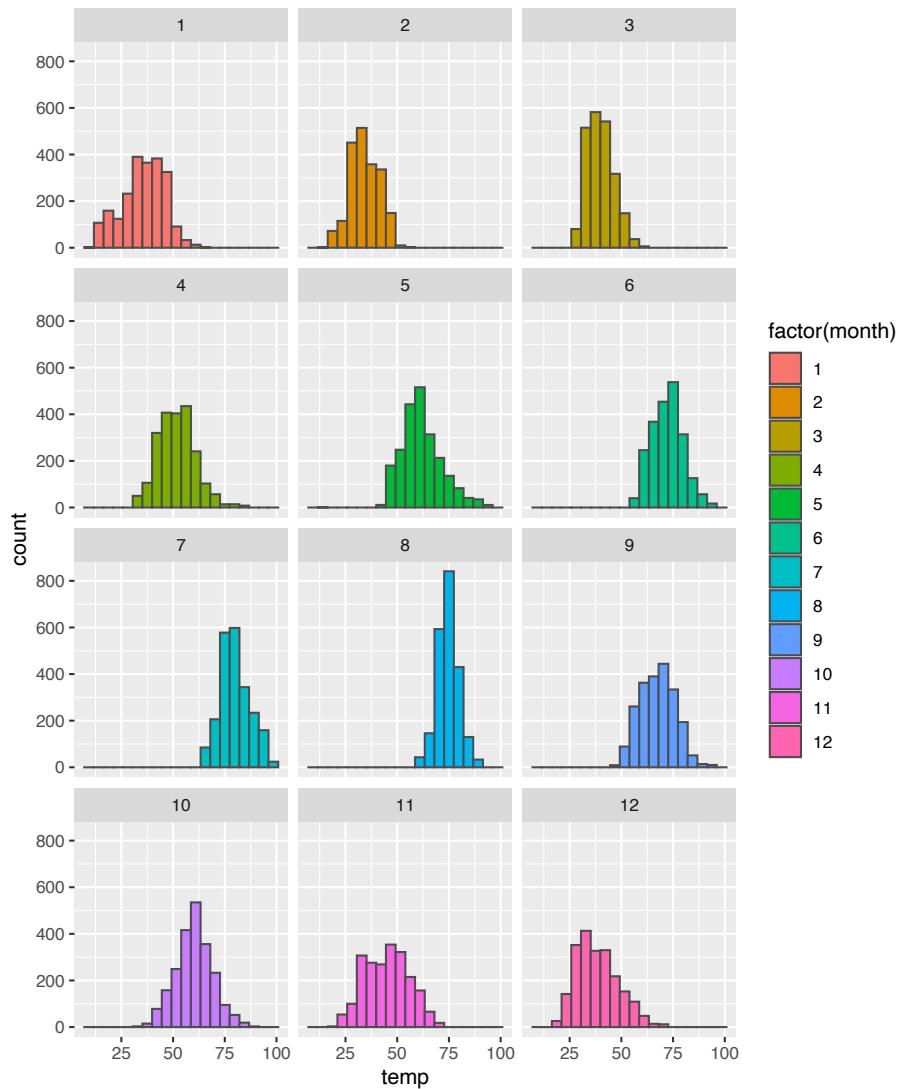


Figure 36 – Un exemple d'utilisation de `facet_wrap()`.

#### 4.6.2 facet\_grid()

Une autre fonction nommée `facet_grid()` permet d'agencer des sous-graphiques selon 2 variables catégorielles. Par exemple :

```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30") +  
  facet_grid(factor(month) ~ origin)
```

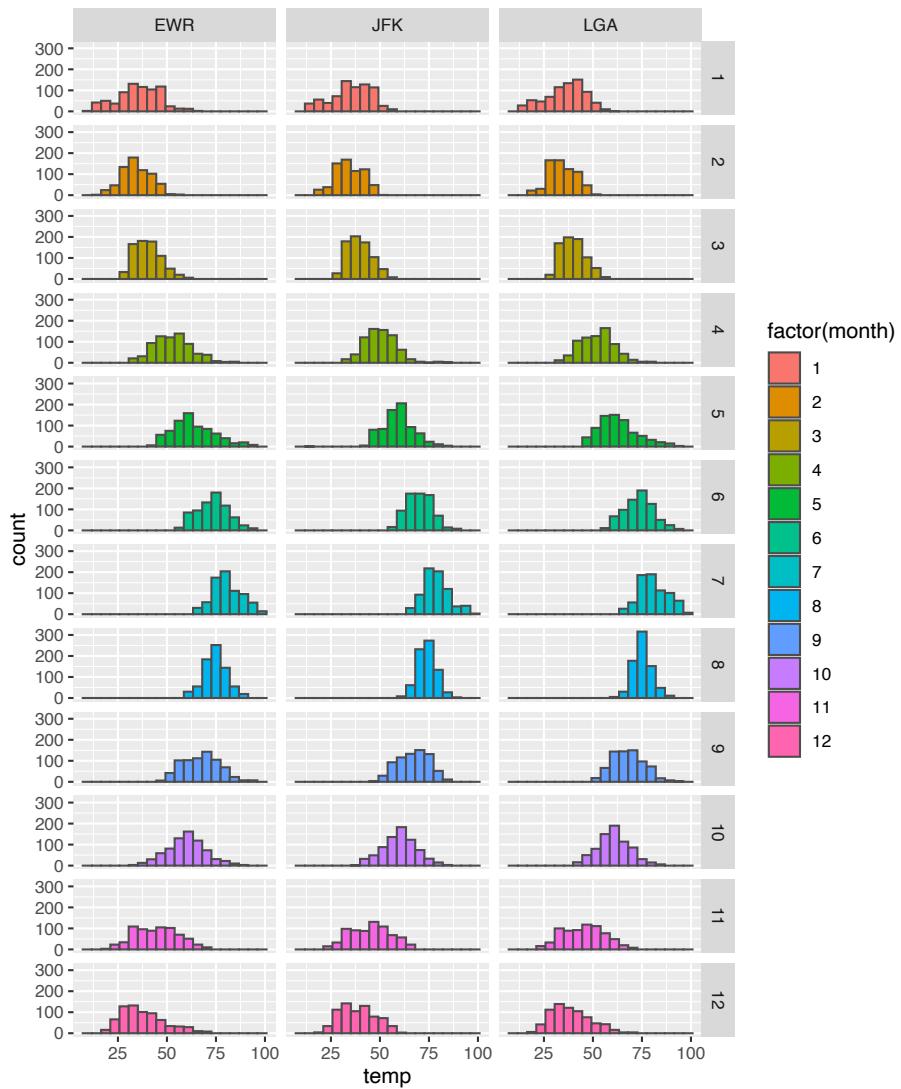


Figure 37 – Un exemple d'utilisation de `facet_grid()`.

Ici, nous avons utilisé la variable `month` (transformée en facteur) et la variable `origin` pour créer un histogramme pour chaque combinaison des modalités de ces 2 variables. Il est donc possible de comparer facilement des températures inter-mensuelles au sein d'un aéroport donné (en colonnes), ou de comparer des températures enregistrées le même mois dans des aéroports distincts (en lignes).

`facet_grid()` doit elle aussi être utilisée avec le symbole “~”. Comme pour les indices d'un tableau, on met à gauche du “~” la variable qui figurera en lignes, et à droite du ~ celle qui figurera en colonnes. Les arguments `nrow` et `ncol` ne peuvent donc pas être utilisés : c'est le nombre de niveaux de chaque variable catégorielle fournie à `facet_grid()` qui détermine le nombre de lignes et de colonnes du graphique.

Vous devriez maintenant être convaincus de la puissance de la grammaire des graphiques. En utilisant un langage standardisé et en ajoutant des couches une à une sur un graphique, il est possible d'obtenir rapidement des visualisations très complexes et néanmoins très claires, qui font apparaître des structures intéressantes dans nos données (des tendances, des groupes, des similitudes, des liaisons, des différences, etc.).

#### 4.6.3 Exercices

Examinez la figure 37.

1. Quels éléments nouveaux ce graphique nous apprend-il par rapport au graphique 34 ci-dessus? Comment le “faceting” nous aide-t'il à visualiser les relations entre 2 (ou 3) variables?
  2. À quoi correspondent les numéros 1 à 12?
  3. À quoi correspondent les chiffres 25, 50, 75, 100?
  4. À quoi correspondent les chiffres 0, 100, 200, 300?
  5. Observez les échelles des axes x et y pour chaque sous graphique. Qu'ont-elles de particulier? En quoi est-ce utile?
  6. La variabilité des températures est-elle plus importante entre les aéroports, entre les mois, ou au sein des mois? Expliquez votre réflexion.
- 

## 4.7 Les boîtes à moustaches ou boxplots

### 4.7.1 Crédit de boxplots et informations apportées

Commençons par créer un boxplot pour comparer les températures mensuelles comme nous l'avons fait plus haut avec des histogrammes :

```
ggplot(weather, aes(x = month, y = temp)) +  
  geom_boxplot()
```

Warning: Continuous x aesthetic -- did you forget aes(group= ... )?

Warning: Removed 1 rows containing non-finite values (stat\_boxplot).

Comme précédemment, R nous avertit qu'une observation n'a pas été intégrée (en raison d'une donnée manquante). Mais il nous dit aussi que x (pour nous, la variable month) est continue, et que nous avons probablement oublié de spécifier des groupes.

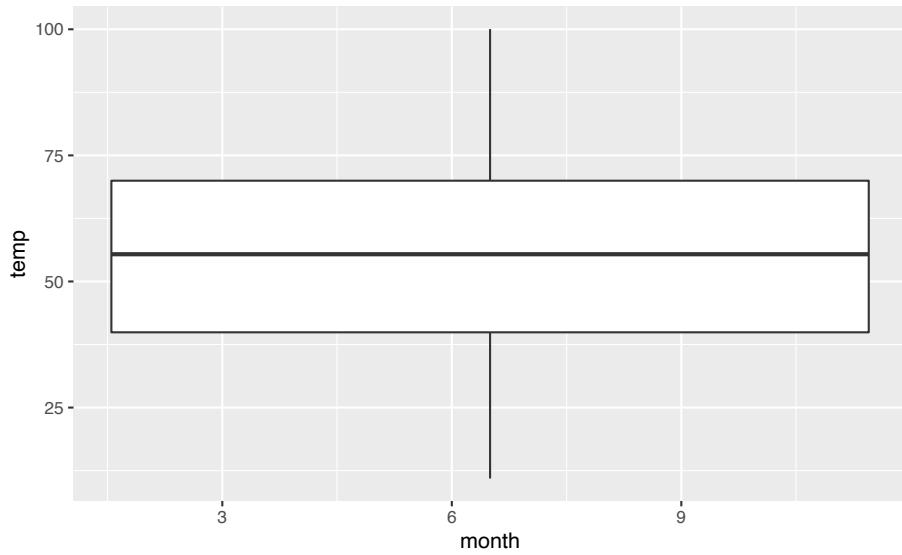


Figure 38 – Un boxplot fort peu utile...

En effet, les boxplots sont généralement utilisés pour examiner la distribution d'une variable numérique pour chaque niveau d'une variable catégorielle (un facteur). Il nous faut donc, ici encore, transformer month en facteur car dans notre tableau de départ, cette variable est considérée comme une variable numérique continue :

```
ggplot(weather, aes(x = factor(month), y = temp)) +
  geom_boxplot()
```

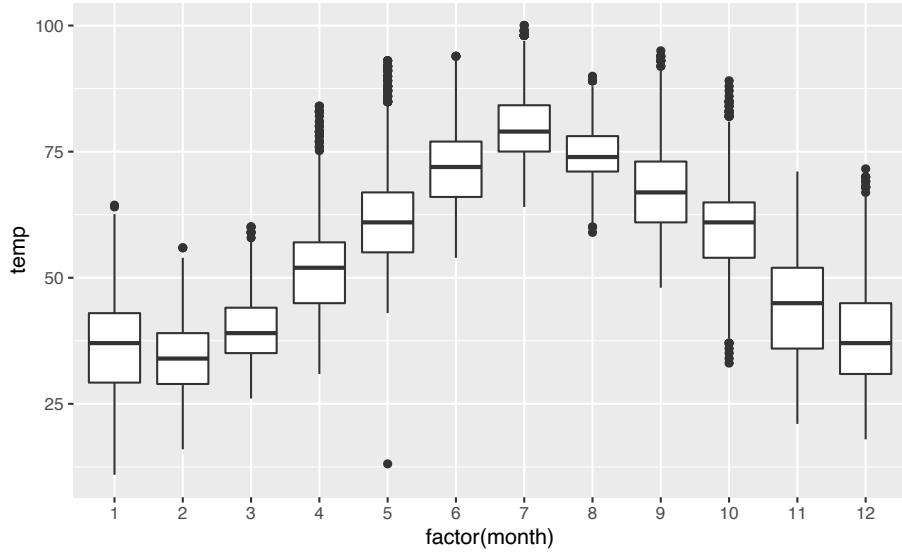


Figure 39 – Boxplot des températures mensuelles.

Les différents éléments d'un boxplot, sont les suivants :

- La limite inférieure de la boîte correspond au premier quartile : 25% des données de

l'échantillon sont situées au-dessous de cette valeur.

- La limite supérieure de la boîte correspond au troisième quartile : 25% des données de l'échantillon sont situées au-dessus de cette valeur.
- Le segment épais à l'intérieur de la boîte correspond au second quartile : c'est la médiane de l'échantillon. 50% des données de l'échantillon sont situées au-dessus de cette valeur, et 50% au-dessous.
- La hauteur de la boîte correspond à ce que l'on appelle l'étendue inter-quartile ou Inter Quartile Range (IQR) en anglais. On trouve dans cette boîte 50% des observations de l'échantillon. C'est une mesure de la dispersion des 50% des données les plus centrales. Une boîte plus allongée indique donc une plus grande dispersion.
- Les moustaches correspondent à des valeurs qui sont en dessous du premier quartile (pour la moustache du bas) et au-dessus du troisième quartile (pour la moustache du haut). La règle utilisée dans R est que ces moustaches s'étendent jusqu'aux valeurs minimales et maximales de l'échantillon, mais elles ne peuvent en aucun cas s'étendre au-delà de 1,5 fois la hauteur de la boîte (1,5 fois l'IQR) vers le haut et le bas. Si des points apparaissent au-delà des moustaches (vers le haut ou le bas), ces points sont appelés "outliers". Ce sont des points qui s'éloignent du centre de la distribution de façon importante puisqu'ils sont au-delà de 1,5 fois l'IQR de part et d'autre du premier ou du troisième quartile. Il peut s'agir d'anomalies de mesures, d'anomalies de saisie des données, ou tout simplement, d'enregistrements tout à fait valides mais extrêmes. J'attire votre attention sur le fait que la définition de ces outliers est relativement arbitraire. Nous pourrions faire le choix d'étendre les moustaches jusqu'à 1,8 fois l'IQR (ou 2, ou 2,5). Nous observerions alors beaucoup moins d'outliers. D'une façons générale, la longueur des moustaches renseigne sur la variabilité des données en dehors de la zone centrale. Plus elles sont longues, plus la variabilité est importante. Et dans tous les cas, l'examen attentif des outliers est utile car il nous permet d'en apprendre plus sur le comportement extrême de certaines observations.

#### 4.7.2 L'intervalle de confiance à 95% de la médiane

On peut également ajouter une encoche autour de la valeur de médiane en ajoutant l'argument `notch = TRUE` à la fonction `geom_boxplot()` :

```
ggplot(weather, aes(x = factor(month), y = temp)) +  
  geom_boxplot(notch = TRUE)
```

Comme l'indique la légende de la figure 40, cette encoche correspond à l'étendue de l'intervalle de confiance à 95% de la médiane. Pour chaque échantillon, nous espérons que la médiane calculée soit le reflet fidèle de la vraie valeur de médiane de la population. Mais il sera toujours impossible d'en avoir la certitude absolue. Le mieux que l'on puisse faire, c'est quantifier l'incertitude. L'intervalle de confiance nous indique qu'il y a de bonnes chances que la vraie valeur de médiane de la population générale (qui restera à jamais inconnue) se trouve dans cet intervalle. Ici, les encoches sont très étroites car les données sont abondantes. Il y a donc peu d'incertitude, ce qui est une bonne chose.

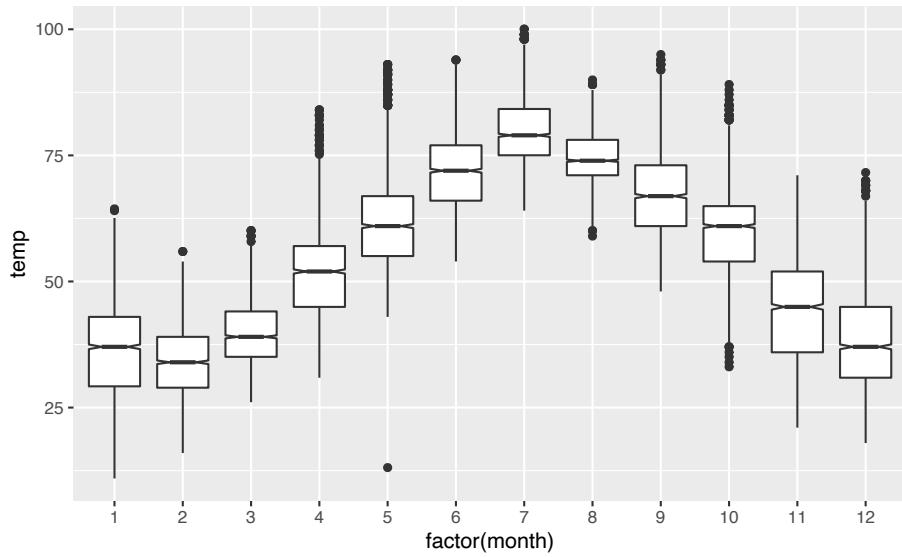


Figure 40 – Boxplot des températures mensuelles. Les intervalles de confiance à 95% de la médiane sont affichés.

Nous reviendrons sur cette notion importante plus tard dans le cursus, car ce type de graphique nous permettra d'anticiper sur les résultats des tests de comparaison de moyennes.

#### 4.7.3 Une autre façon d'examiner des distributions

Dernière chose concernant les boxplots : il s'agit d'une représentation graphique très proche de l'histogramme. Pour vous en convaincre, je représente à la figure 41 ci-dessous uniquement les températures du mois de novembre, avec 3 types d'objets géométriques différents : un histogramme, un boxplot, et un nuage de points.

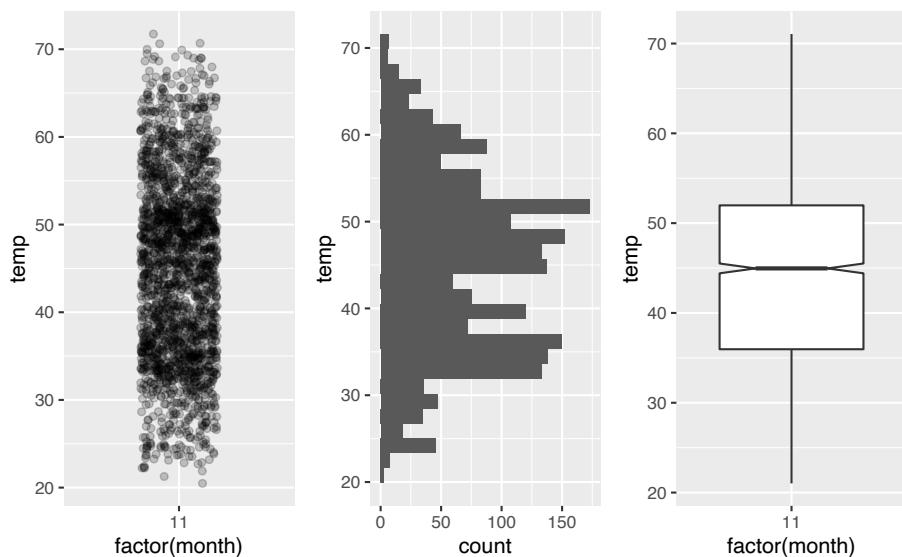


Figure 41 – Distribution des températures de Novembre 2013.

Nous avons donc, à gauche les données brutes, sous la forme d'un nuage de points créé avec `geom_jitter()`, au centre, un histogramme pour les températures de novembre, créé avec `geom_histogram()` (j'ai permué les axes pour que y porte la température pour les 3 graphiques) et à droite, un boxplot pour ces mêmes données, créé avec `geom_boxplot()`. On voit bien que ces 3 représentations graphiques sont similaires. Toutes rendent compte du fait que les températures de Novembre sont majoritairement comprises entre 35 et 52 degrés Farenheit. Au-delà de cette fourchette (au-dessus comme en dessous) les observations sont plus rares.

Le nuage de points affiche toutes les données. C'est donc lui le plus complet mais pas forcément le plus lisible. Les points sont en effet très nombreux et la lecture du graphique peut s'en trouver compliquée. L'histogramme simplifie les données en les regroupant dans des classes. C'est une sorte de résumé des données. On constate cependant toujours la présence de 2 pics qui correspondent aux zones plus denses du nuage de points. Le boxplot enfin synthétise encore plus ces données. Elles sont résumées par 7 valeurs seulement : le minimum, le maximum, les 3 quartiles, et les bornes de l'intervalle de confiance à 95% de la médiane. C'est une représentation très synthétique qui nous permet de comparer beaucoup de catégories côté à côté (voir la figure 40 un peu plus haut), mais qui est forcément moins précise qu'un histogramme. Vous noterez toutefois que la boîte du boxplot recouvre en grande partie la zone des 2 pics de l'histogramme. En outre, sur la figure 40, la tendance générale est très visible : il fait plus chaud en été qu'en hiver (étonnant non?).

#### 4.7.4 Pour conclure

Les boîtes à moustaches permettent donc de comparer et contraster la distribution d'**une variable quantitative** pour plusieurs niveaux d'**une variable catégorielle**. On peut voir où la médiane tombe dans les différents groupes en observant la position de la ligne centrale dans la boîte. Pour avoir une idée de la dispersion de la variable au sein de chaque groupe, regardez à la fois la hauteur de la boîte et la longueur des moustaches. Quand les moustaches s'étendent loin de la boîte mais que la boîte est petite, cela signifie que la variabilité des valeurs proches du centre de la distribution est beaucoup plus faible que la variabilité des valeurs extrêmes. Enfin, les valeurs extrêmes ou aberrantes sont encore plus faciles à détecter avec une boîte à moustaches qu'avec un histogramme.

---

### 4.8 Les diagrammes bâtons

Comme nous venons de le voir, les histogrammes et les boîtes à moustaches permettent de visualiser la distribution d'une **variable numérique continue**. Nous aurons aussi souvent besoin de visualiser la distribution d'une **variable catégorielle**. C'est une tâche plus simple qui consiste à compter combien d'éléments tombent dans chacune des catégories de la variable catégorielle. Le meilleur moyen de visualiser de telles données de comptage (*aka* fréquences) est de réaliser un diagramme bâtons, autrement appelé **barplot** ou **barchart**.

Une difficulté, toutefois, concerne la façon dont les données sont présentées : est-ce que la variable d'intérêt est “pré-comptée” ou non? Par exemple, le code ci-dessous crée 2 `data.frame` qui représentent la même collection de fruits : 3 pommes et 2 oranges :

```
fruits <- tibble(  
  fruit = c("pomme", "pomme", "pomme", "orange", "orange")  
)  
fruits
```

```
# A tibble: 5 x 1  
  fruit  
  <chr>  
1 pomme  
2 pomme  
3 pomme  
4 orange  
5 orange
```

```
fruits_counted <- tibble(  
  fruit = c("pomme", "orange"),  
  nombre = c(3, 2)  
)  
fruits_counted
```

```
# A tibble: 2 x 2  
  fruit  nombre  
  <chr>   <dbl>  
1 pomme     3  
2 orange     2
```

#### 4.8.1 Représentation graphique avec `geom_bar` et `geom_col`

Pour visualiser les données non pré-comptées, on utilise `geom_bar()`:

```
ggplot(data = fruits, mapping = aes(x = fruit)) +  
  geom_bar()
```

Pour visualiser les données déjà pré-comptées, on utilise `geom_col()`:

```
ggplot(data = fruits_counted, mapping = aes(x = fruit, y = nombre)) +  
  geom_col()
```

Notez que les figures 42 et 43 sont absolument identiques (à l'exception du titre de l'axe des ordonnées), mais qu'elles ont été créées à partir de 2 tableaux de données différents. En particulier, notez que :

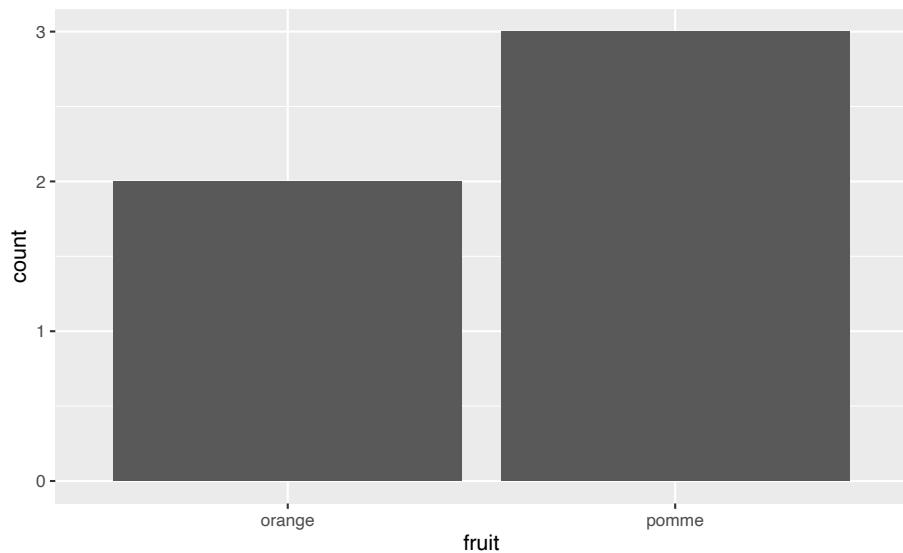


Figure 42 – Barplot pour des données non pré-comptées.

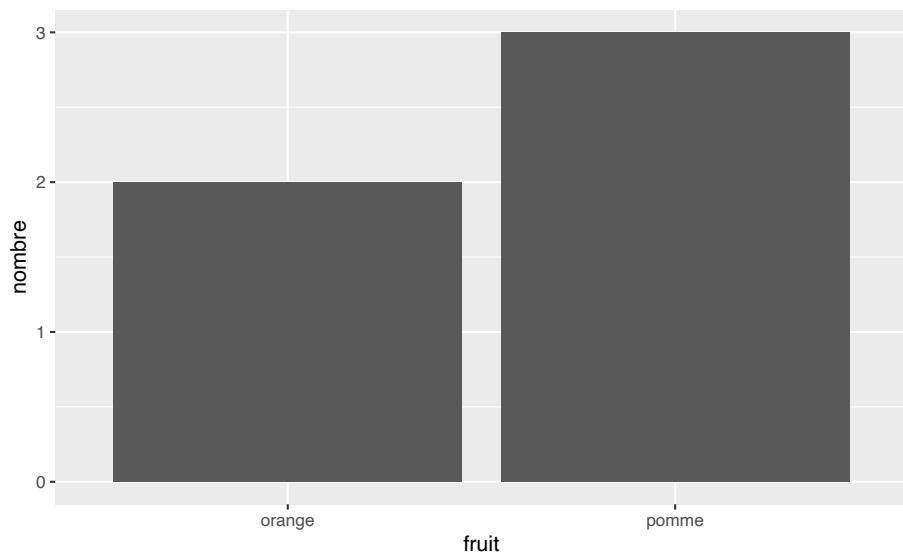


Figure 43 – Barplot pour des données pré-comptées.

- Le code qui génère la figure 42 utilise le jeu de données fruits, et n'associe pas de variable à l'axe des ordonnées : dans la fonction aes(), seule la variable associée à x est précisée. C'est la fonction geom\_bar() qui calcule automatiquement les abondances (ou fréquences) pour chaque catégorie de la variable fruit. La variable count est ainsi générée automatiquement et associée à y.
- Le code qui génère la figure 43 utilise le jeu de données fruits\_counted. Ici, la variable nombre est associée à l'axe des y grâce à la fonction aes(). La fonction geom\_col() a besoin de 2 variables (une variable catégorielle pour l'axe des x et une numérique pour l'axe des y) pour fonctionner.

Autrement dit, lorsque vous souhaiterez créer un diagramme bâtons, il faudra donc au préalable vérifier de quel type de données vous disposez pour choisir l'objet géométrique approprié :

- Si votre variable catégorielle n'est pas pré-comptée dans votre tableau de données, il faut utiliser geom\_bar()
- Si votre variable catégorielle est pré-comptée dans votre tableau de données, il faut utiliser geom\_col() et associer explicitement les comptages à l'aesthétique y du graphique.

#### 4.8.2 Un exemple concret

Revenons à nycflights13. Imaginons que nous souhaitions connaître le nombre de vols affrétés par chaque compagnie aérienne au départ de New York en 2013. Dans le jeu de données flights, la variable carrier nous indique à quelle compagnie aérienne appartiennent chacun des 336776 vols ayant quitté New York en 2013. Une façon simple de représenter ces données est donc la suivante :

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

Ici, geom\_bar() a compté le nombre d'occurrences de chaque compagnie aérienne dans le tableau flights et a automatiquement associé ce nombre à l'axe des ordonnées.

Il est généralement plus utile de trier les catégories par ordre décroissant. Nous pouvons faire cela facilement grâce à la fonction fct\_infreq() du packageforcats. Si vous avez installé le tidyverse, le package forcast doit être disponible sur votre ordinateur. N'oubliez pas de le charger si besoin :

```
library(forcats)
ggplot(data = flights, mapping = aes(x = fct_infreq(carrier))) +
  geom_bar()
```

Ordonner les catégories par ordre décroissant est souvent indispensable afin de faciliter la lecture du graphique et les comparaisons entre catégories.

Si nous souhaitons connaître le nombre de vols précis de chaque compagnie aérienne, il nous faut faire appel à plusieurs fonctions du package dplyr que nous détaillerons dans le chapitre 6.

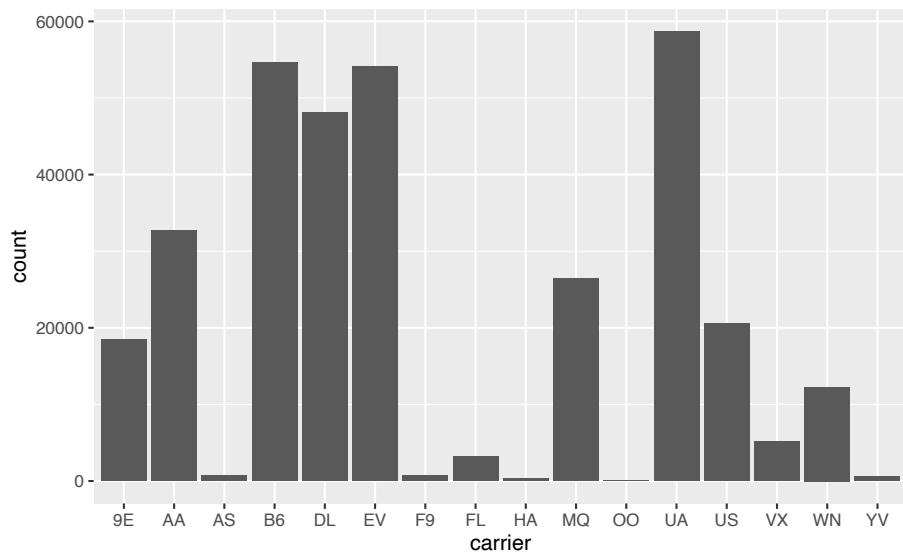


Figure 44 – Nombre de vols par compagnie aérienne au départ de New York en 2013.

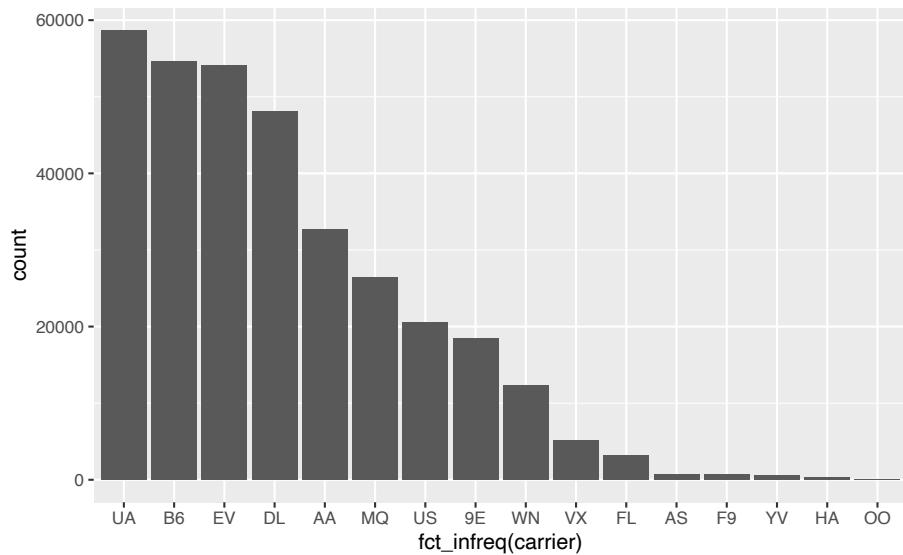


Figure 45 – Nombre de vols par compagnie aérienne au départ de New York en 2013.

Ci-dessous, nous créons un nouveau tableau `carrier_table` contenant le nombre de vols de chaque compagnie aérienne et les compagnies sont ordonnées par nombres de vols décroissants :

```
carrier_table <- flights %>% # On prend flights, puis ...
  group_by(carrier) %>% # On groupe les données par compagnie, puis ...
  summarize(nombre = n()) %>% # On calcule le nb de vols par Cie, puis ...
  arrange(desc(nombre))      # On trie par nb de vols décroissants ...
carrier_table                  # Enfin, on affiche la nouvelle table

# A tibble: 16 x 2
  carrier   nombre
  <chr>     <int>
1 UA        58665
2 B6        54635
3 EV        54173
4 DL        48110
5 AA        32729
6 MQ        26397
7 US        20536
8 9E        18460
9 WN        12275
10 VX       5162
11 FL       3260
12 AS       714
13 F9       685
14 YV       601
15 HA       342
16 OO       32
```

Ici, la table a été triée par nombres de vols décroissants. Mais attention, **les niveaux** du facteur `carrier` n'ont pas été modifiés :

```
factor(carrier_table$carrier)

[1] UA B6 EV DL AA MQ US 9E WN VX FL AS F9 YV HA OO
Levels: 9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
```

Le premier niveau est toujours 9E, puis AA, puis AS, et non l'ordre du tableau nouvellement créé (UA, puis B6, puis EV...) car les niveaux sont toujours triés par ordre alphabétique. La conséquence est que faire un barplot avec ces données et la fonction `geom_col()` ne permet pas d'ordonner les catégories correctement :

```
ggplot(carrier_table, aes(x = carrier, y = nombre)) +
  geom_col()
```

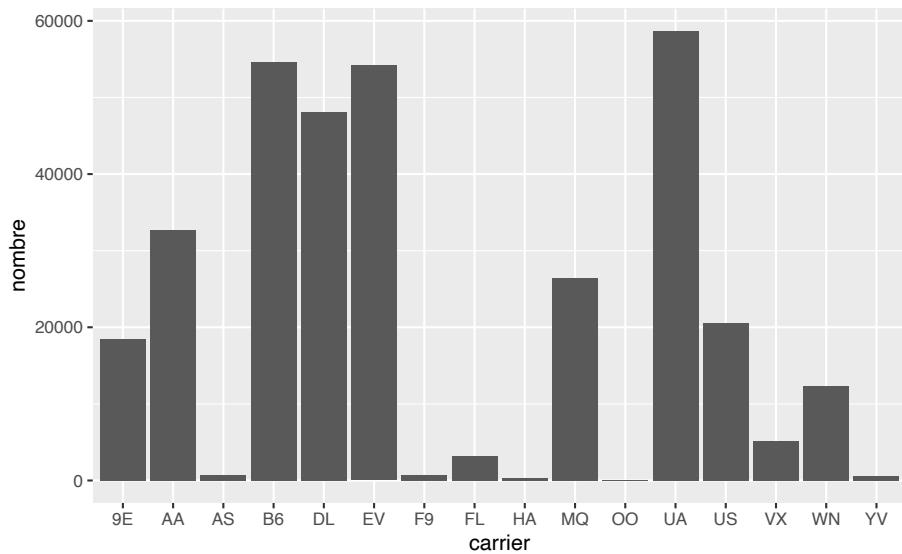


Figure 46 – Nombre de vols par compagnie aérienne au départ de New York en 2013.

Pour parvenir à nos fins, il faut cette fois avoir recours à la fonction `fct_reorder()` pour ordonner correctement les catégories. Cette fonction prend 3 arguments :

1. La variable catégorielle dont on souhaite réordonner les niveaux (ici, la variable `carrier` du tableau `carrier_table`).
2. Une variable numérique qui permet d'ordonner les catégories (ici, la variable `nombre` du même tableau).
3. L'argument optionnel `.desc` qui permet de préciser si le tri doit être fait en ordre croissant (c'est le cas par défaut) ou décroissant.

```
ggplot(carrier_table, aes(x = fct_reorder(carrier, nombre, .desc = TRUE),
                           y = nombre)) +
  geom_col()
```

Vous voyez donc que selon le type de données dont vous disposez (soit un tableau comme `flights`, avec toutes les observations, soit un tableau beaucoup plus compact comme `carrier_table`), la démarche permettant de produire un diagramme bâtons, dans lequel les catégories seront triées, sera différente.

#### 4.8.3 Exercices

1. Quelle est la différence entre un histogramme et un diagramme bâtons ?
2. Pourquoi les histogrammes sont-ils inadaptés pour visualiser des données catégorielles ?
3. Quel est le nom de la compagnie pour laquelle le plus grand nombre de vols ont quitté New York en 2013 (je veux connaître son nom, pas juste son code) ? Où se trouve cette information ?

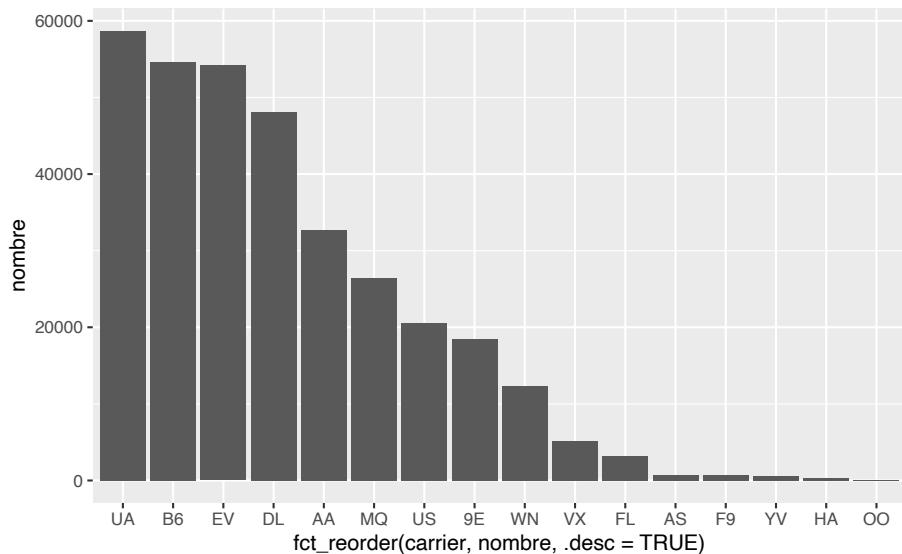


Figure 47 – Nombre de vols par compagnie aérienne au départ de New York en 2013.

- Quel est le nom de la compagnie pour laquelle le plus petit nombre de vols ont quitté New York en 2013 (je veux connaître son nom, pas juste son code) ? Où se trouve cette information ?

#### 4.8.4 Éviter à tout prix les diagrammes circulaires

À mon grand désarroi, l'un des graphiques les plus utilisés pour représenter la distribution d'une variable catégorielle est le diagramme circulaire (ou diagramme camembert, piechart en anglais). C'est presque toujours la plus mauvaise visualisation possible. Je vous demande de l'éviter à tout prix. Notre cerveau n'est en effet pas correctement équipé pour comparer des angles. Ainsi, par exemple, nous avons naturellement tendance à surestimer les angles supérieurs à 90°, et à sous-estimer les angles inférieurs à 90°. En d'autres termes, il est difficile pour les humains de comparer des grandeurs sur des diagrammes circulaires.

À titre d'exemple, examinez ce diagramme, qui reprend les mêmes chiffres que précédemment, et tentez de répondre aux questions suivantes :

- Comparez les compagnies ExpressJet Airlines (EV) et US Airways (US). De combien de fois la part de EV est-elle supérieure à celle d'US ? (2 fois, 3 fois, 1.2 fois?...)
- Quelle est la troisième compagnie aérienne la plus importante en terme de nombre de vols au départ de New York en 2013 ?
- Combien de compagnies aériennes ont moins de vols que United Airlines (UA) ?

Il est difficile (voire impossible) de répondre précisément à ces questions avec le diagramme circulaire de la figure 48, alors qu'il est très simple d'obtenir des réponses précises avec un diagramme bâtons tel que présenté à la figure 47 (vérifiez-le !).

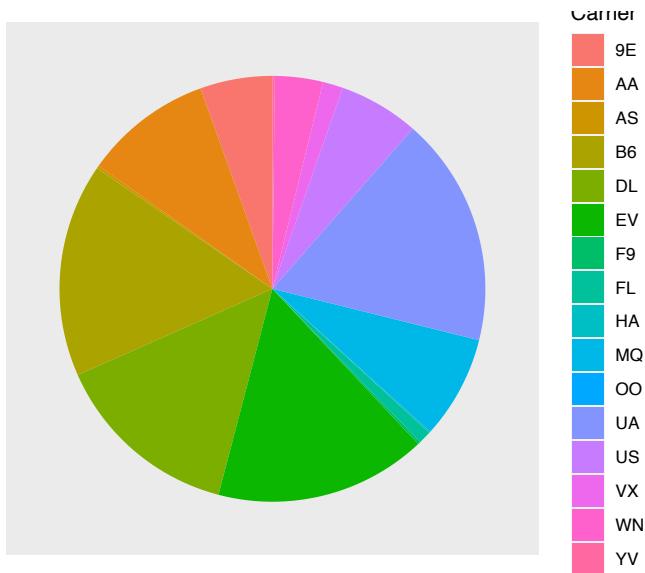


Figure 48 – Nombre de vols par compagnie aérienne au départ de New York en 2013.

#### 4.8.5 Comparer 2 variables catégorielles avec un diagramme bâton

Il y a généralement 3 façons de procéder pour comparer la distribution de 2 variables catégorielles avec un diagramme bâtons :

1. Faire un graphique empilé.
2. Faire un graphique juxtaposé.
3. Utiliser les facets.

Supposons par exemple que nous devions visualiser le nombre de vols de chaque compagnie aérienne, au départ de chacun des 3 aéroports de New York : John F. Kennedy (JFK), Newark (EWR) et La Guardia (LGA). Voyons comment procéder avec chacune des 3 méthodes énoncées ci-dessus.

##### 4.8.5.1 Graphique empilé

La méthode la plus simple est celle du graphique empilé :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar()
```

Notez qu'il s'agit du même code que celui utilisé pour la figure 45, à une différence près : l'ajout de `fill = origin` dans la fonction `aes()`, qui permet d'associer l'aéroport d'origine à la couleur de remplissage des barres. `fill` est associé à une variable (ici, elle est catégorielle), il est donc indispensable de faire figurer cet argument à l'intérieur de la fonction `aes()`. Quand on associe une variable à une caractéristique esthétique du graphique, on fait toujours figurer le code à l'intérieur de la fonction `aes()` (comme quand on associe une variable aux axes du graphique par exemple).

À mon sens, le graphique gagne en lisibilité si on ajoute une couleur pour le contour des barres :

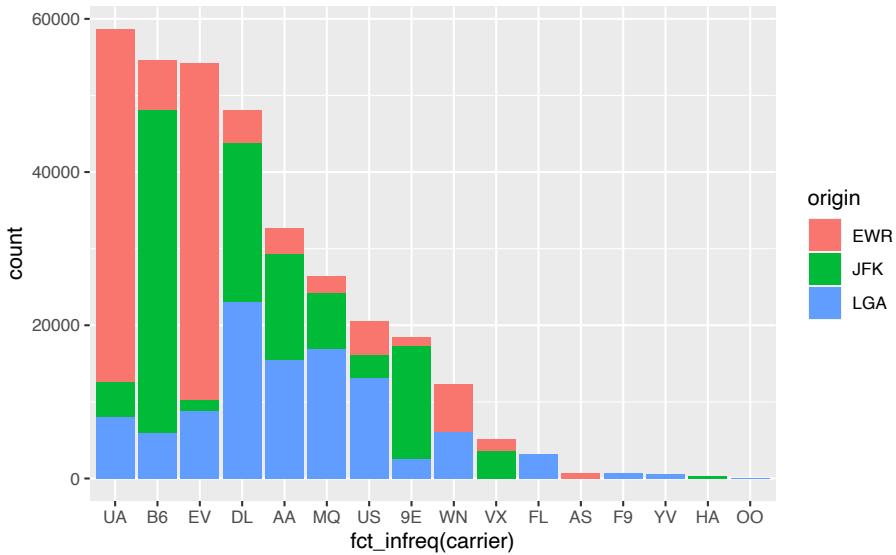


Figure 49 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black")
```

Notez que contrairement à `fill`, cette couleur de contour est un paramètre fixe : elle n'est pas associée à une variable et doit donc être placée en dehors de la fonction `aes()`.

Bien que ces graphiques empilés soient très simples à réaliser, ils sont parfois difficiles à lire. En particulier, il n'est pas toujours aisément de comparer les hauteurs des différentes couleurs (qui correspondent ici aux nombres de vols issus de chaque aéroport) entre barres différentes (qui correspondent ici aux compagnies aériennes).

**4.8.5.2 Graphique juxtaposé** Une variation sur le même thème consiste, non plus à empiler les barres de couleur les unes sur les autres, mais à les juxtaposer :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black", position = "dodge")
```

Passer d'un graphique empilé à un graphique juxtaposé est donc très simple : il suffit d'ajouter l'argument `position = "dodge"` à la fonction `geom_bar()`.

Là encore, la lecture de ces graphiques est souvent difficile car la comparaison des catégories qui figurent sur l'axe des x n'est pas immédiate. Elle est en outre rendue plus difficile par le fait que toutes les barres n'ont pas la même largeur. Par exemple, sur la figure 51, les 8 premières compagnies aériennes desservent les 3 aéroports de New York, mais les 2 suivantes (WN et VX) n'en desservent que 2, et les autres compagnies, qu'un seul. Puisque sur un barplot, seule la hauteur des barres compte, il faut prendre garde à ne pas se laisser influencer par la largeur des barres qui pourraient fausser notre perception.

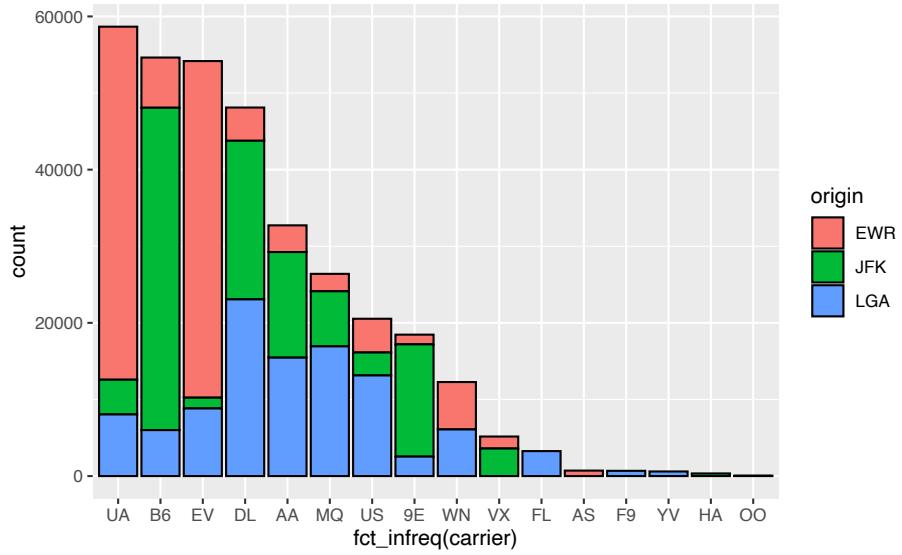


Figure 50 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

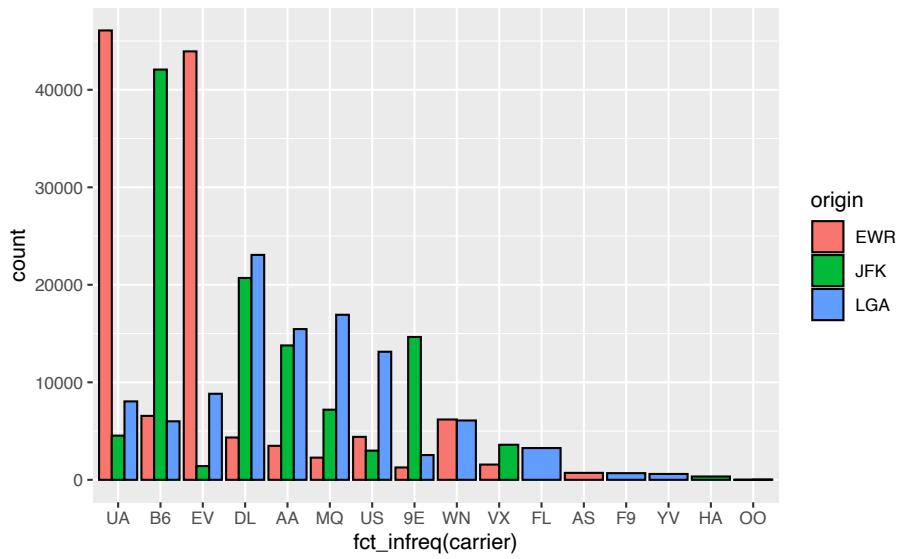


Figure 51 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

**4.8.5.3 Utilisation des facets** La meilleure alternative est probablement l'utilisation de facets que nous avons déjà décrite à la section 4.6 :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1)
```

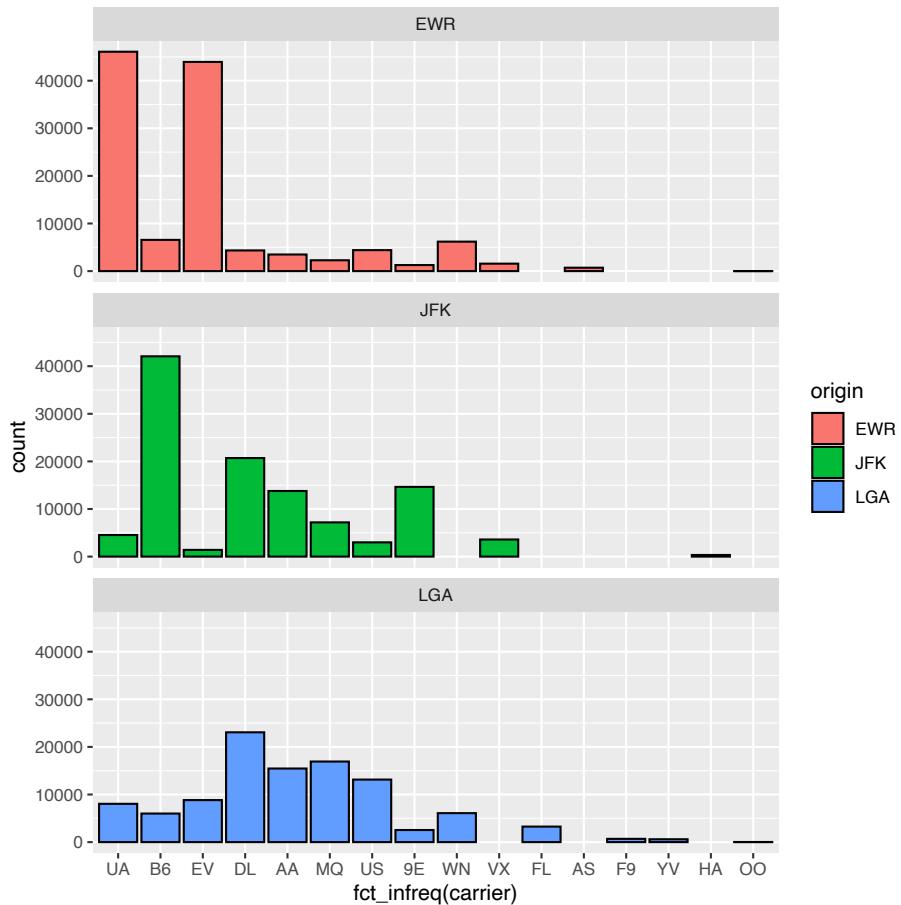


Figure 52 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Ici, chaque graphique permet de comparer les compagnies aériennes au sein de l'un des aéroports de New York, et puisque l'ordre des compagnies aériennes est le même sur l'axe des x des 3 graphiques, une lecture verticale permet de comparer aisément le nombre de vols qu'une compagnie donnée a affrété dans chacun des 3 aéroports de New York.

## 4.9 De l'exploration à l'exposition

Vous savez maintenant comment produire une grande variété de graphiques, permettant d'explorer vos données, de visualiser le comportement d'une ou plusieurs variables, et de

mettre en évidence des tendances, des relations entre variables numériques et/ou catégorielles. Outre les objets géométriques décrits jusqu'ici, ggplot2 contient de nombreuses possibilités supplémentaires pour créer des graphiques parlants et originaux. Je ne peux donc que vous encourager à explorer par vous même les autres possibilités de ce package. Lorsque vous produisez un graphique parlant et permettant de véhiculer un message clair, vous devez ensuite rendre vos graphiques plus présentables afin de les intégrer dans un rapport ou une présentation. Cette section vous permettra de vous familiariser avec quelques fonctions permettant d'annoter correctement vos graphiques et d'en modifier les légendes si nécessaire.

#### 4.9.1 Les labels

Le point de départ le plus évident est d'ajouter des labels de qualité. La fonction `labs()` du package ggplot2 permet d'ajouter plusieurs types de labels sur vos graphiques :

- Un titre : il doit résumer les résultats les plus importants.
- Un sous-titre : il permet de donner quelques détails supplémentaires.
- Une légende : souvent utilisée pour présenter la source des données du graphique.
- Un titre pour chaque axe : permet de préciser les variables portées par les axes et leurs unités.
- Un titre pour les légendes de couleurs, de forme, de taille, etc.

Reprendons par exemple le graphique de la figure 10 :

```
ggplot(alaska_flights,
       aes(x = dep_delay, y = arr_delay, color = factor(month))) +
  geom_point()
```

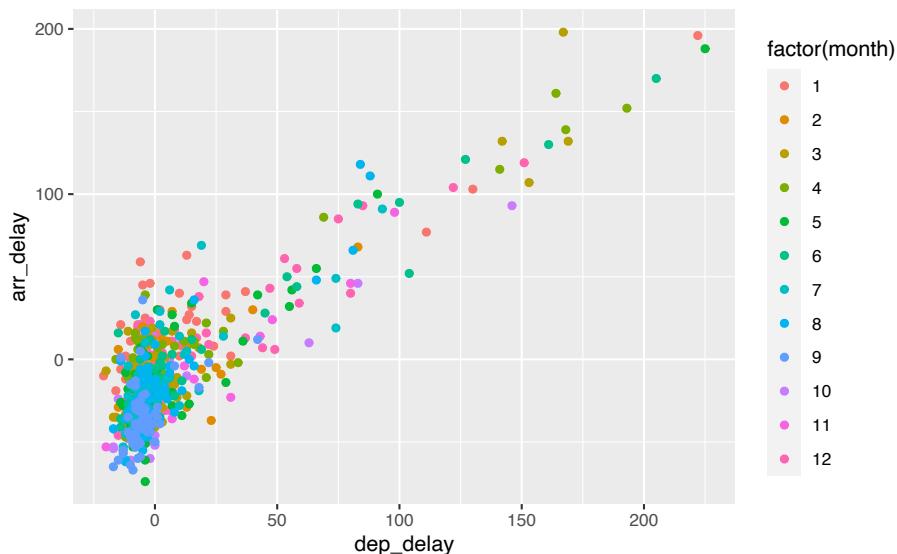


Figure 53 – Association de ‘color’ à une variable catégorielle.

Nous pouvons ajouter sur ce graphique les éléments précisés plus haut en ajoutant la fonction `labs()` sur une nouvelle couche du graphique :

```

ggplot(alaska_flights,
       aes(x = dep_delay, y = arr_delay, color = factor(month))) +
  geom_point() +
  labs(title = "Relation linéaire positive entre retard des vols au départ et à l'arrivée",
       subtitle = "Certains retards dépassent 3 heures",
       caption = "Source : nycflights13",
       x = "Retard au départ de New York (minutes)",
       y = "Retard à l'arrivée à destination (minutes)",
       color = "Mois")

```

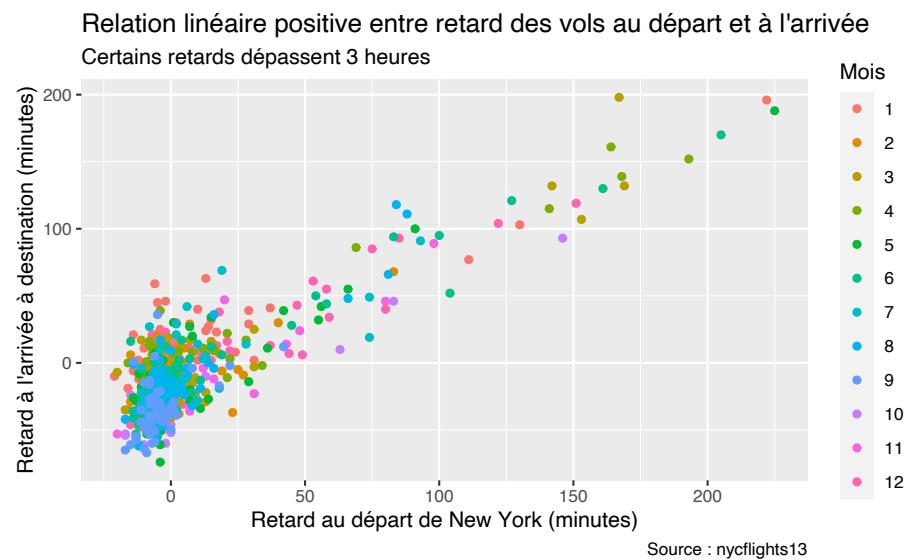


Figure 54 – Exemple d'utilisation de 'labs()'.

À partir de maintenant, vous devriez systématiquement légendier les axes de vos graphiques en n'oubliant pas de préciser les unités, pour tous les graphiques que vous intégrez dans vos rapports, compte-rendus, mémoires, etc.

#### 4.9.2 Les échelles

Tous les détails des graphiques que vous produisez peuvent être édités. C'est notamment le cas des échelles. Qu'il s'agisse de modifier l'étendue des axes, la densité du quadrillage, la position des tirets sur les axes, le nom des catégories figurant sur les axes ou dans les légendes ou encore les couleurs utilisées pour différentes catégories d'objets géométriques, tout est possible dans ggplot2.

Nous n'avons pas le temps ici d'aborder toutes ces questions en détail. Je vous encourage donc à consulter l'ouvrage en ligne intitulé [R for data science](#), et en particulier [son chapitre dédié aux échelles](#), si vous avez besoin d'apporter des modifications à vos graphiques et que vous ne trouvez pas comment faire dans cet ouvrage.

Je vais ici uniquement détailler la façon de procéder pour modifier les couleurs choisies par défaut

par `ggplot2`. Reprenons par exemple la figure 52, en ajoutant au passage des titres corrects pour nos axes

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\\nde New York")
```

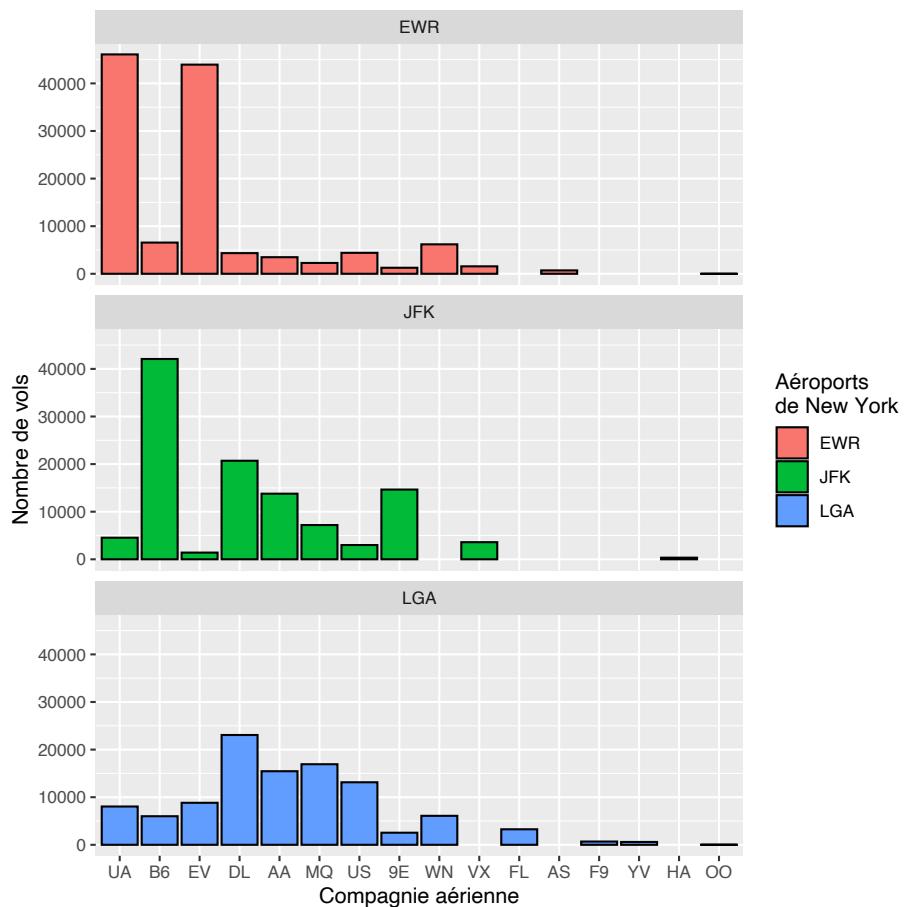


Figure 55 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Notez que le caractère spécial “\n” permet de forcer un retour à la ligne. Ici, les 3 couleurs de remplissage (`fill`) utilisées pour différencier les 3 aéroports de New York ont été choisies par défaut par `ggplot2`. Il est possible de modifier ces couleurs de plusieurs façons :

- En utilisant d'autres palettes de couleurs prédéfinies.
- En utilisant des couleurs choisies manuellement.

Toutes les fonctions permettant d'altérer les légendes commencent par `scale_`. Vient ensuite le nom de l'esthétique que l'on souhaite modifier (ici `fill_`) et enfin, le nom d'une fonction à

appliquer. Les possibilités sont nombreuses et vous pouvez en avoir un aperçu en tapant le début du nom de la fonction et en parcourant la liste proposée par RStudio sous le curseur.

Par exemple, pour utiliser des niveaux de gris plutôt que les couleurs, il suffit d'ajouter une couche à notre graphique :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\nde New York") +
  scale_fill_grey()
```

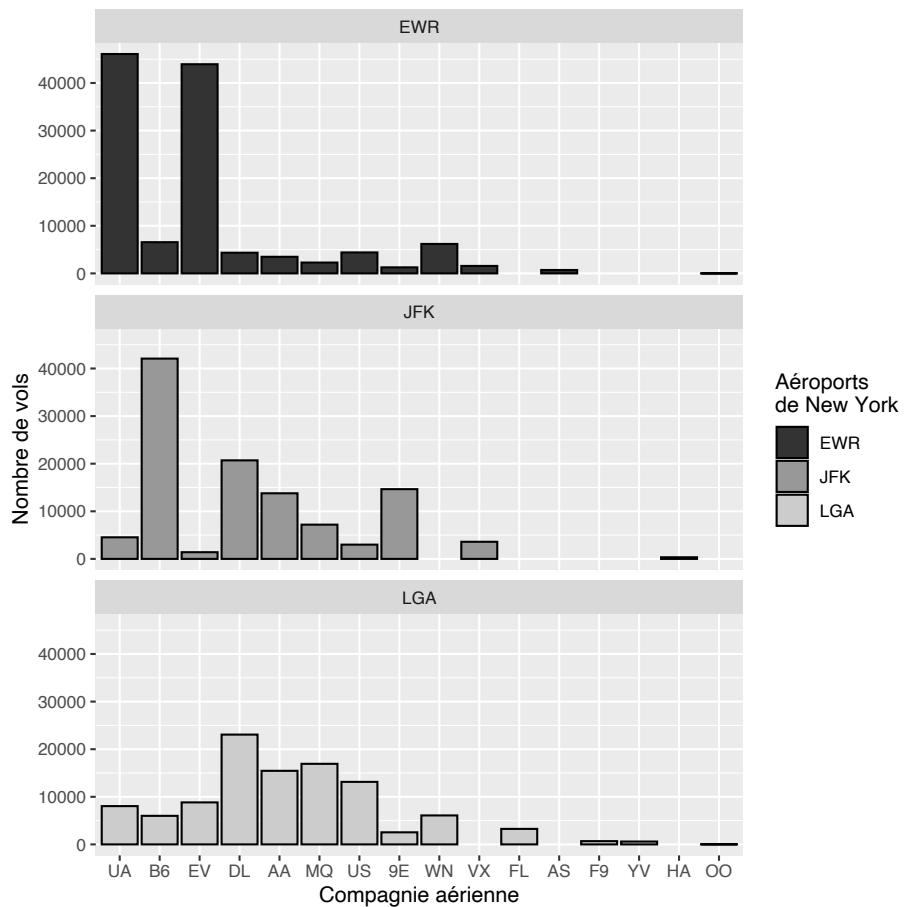


Figure 56 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Le package RColorBrewer propose une large gamme de palettes de couleurs (figure 57) :

ggplot2 permet d'appliquer ces palettes très simplement :

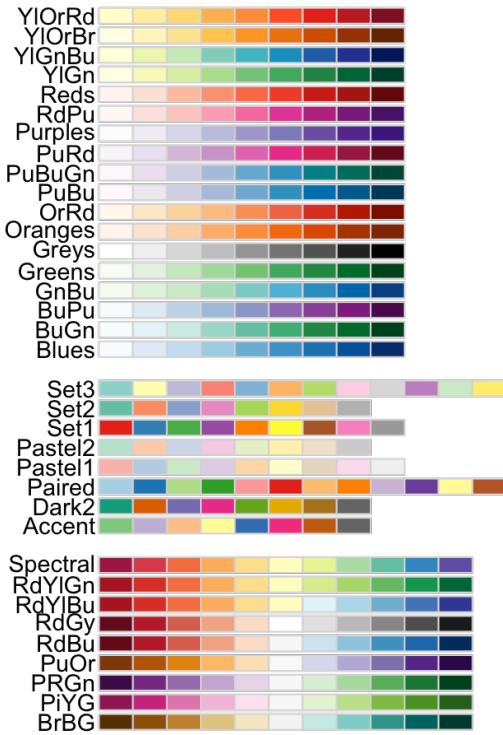


Figure 57 – Toutes les palettes de couleur du package ‘RColorBrewer’.

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\nde New York") +
  scale_fill_brewer(palette = "Accent")
```

De même, le package `viridis` propose une palette de couleurs intéressante qui maximise le contraste et facilite la discrimination des catégories pour les daltoniens. Là encore, `ggplot2` nous donne accès à cette palette :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\nde New York") +
  scale_fill_viridis_d()
```

Enfin, si les palettes de couleurs ne conviennent pas, il est toujours possible de spécifier manuellement les couleurs souhaitées. R propose un accès rapide à 657 noms de couleurs. Pour les afficher, il suffit de taper :

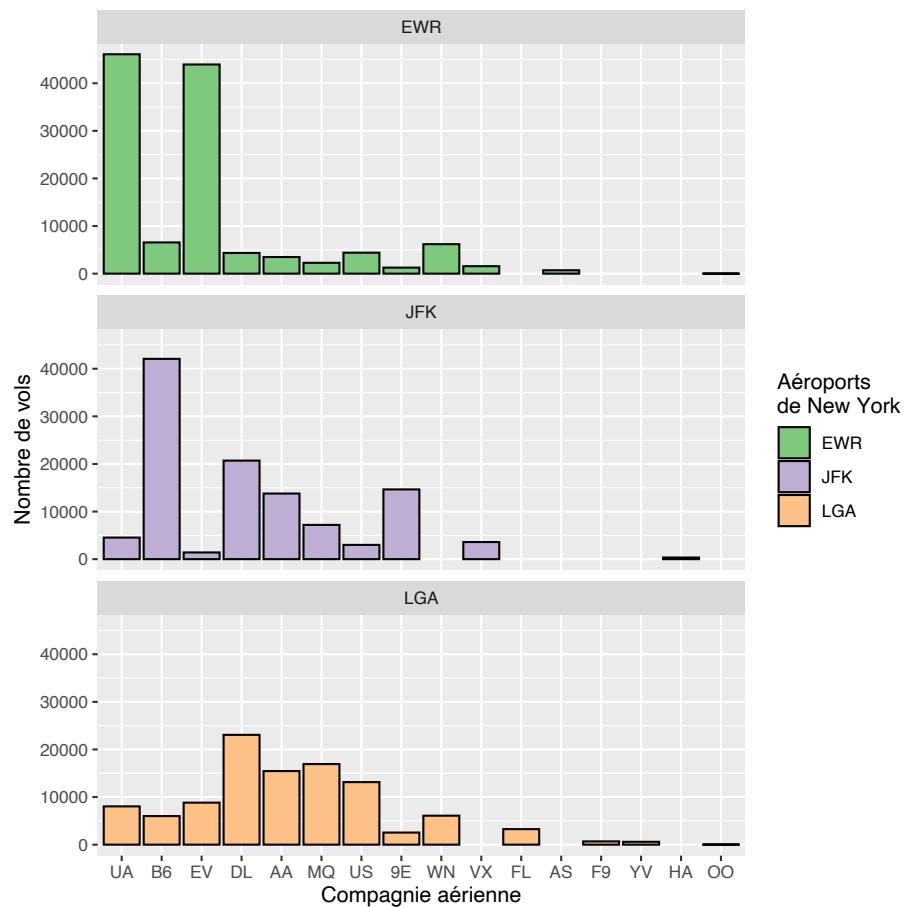


Figure 58 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

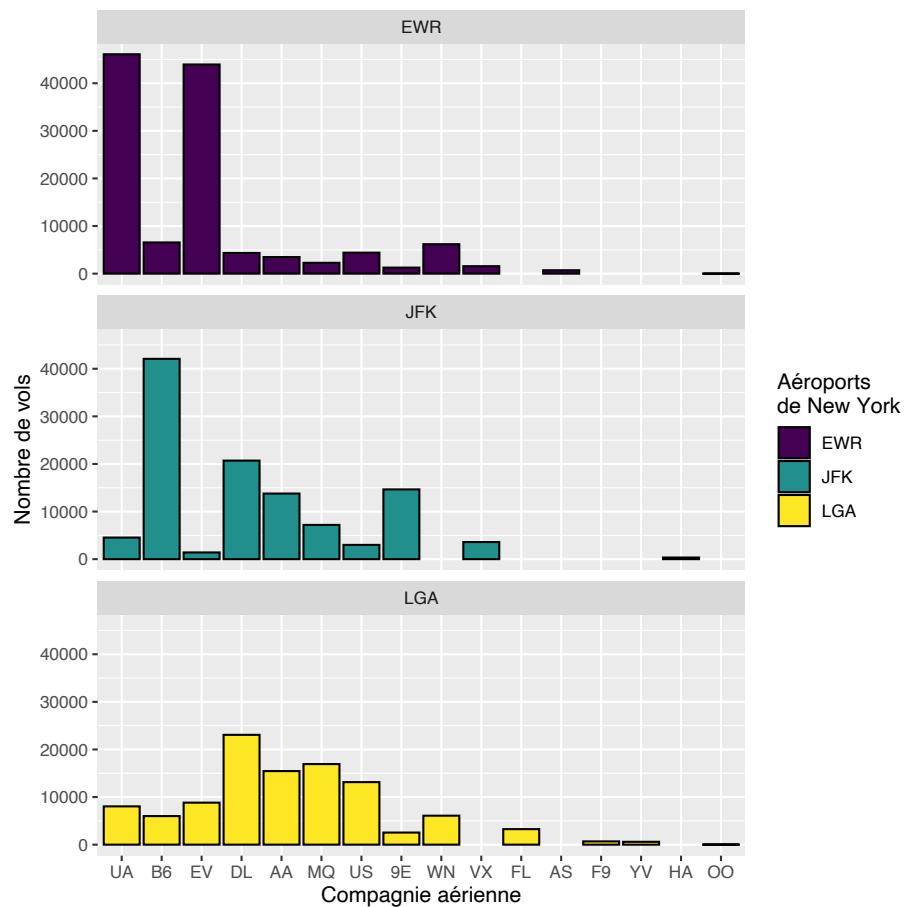


Figure 59 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

```
colors()
```

```
[1] "white"                      "aliceblue"  
[3] "antiquewhite"                "antiquewhite1"  
[5] "antiquewhite2"                "antiquewhite3"  
[7] "antiquewhite4"                "aquamarine"  
[9] "aquamarine1"                  "aquamarine2"  
[11] "aquamarine3"                  "aquamarine4"  
[13] "azure"                       "azure1"  
[15] "azure2"                       "azure3"  
[17] "azure4"                       "beige"  
[19] "bisque"                      "bisque1"  
[21] "bisque2"                      "bisque3"  
[23] "bisque4"                      "black"  
[25] "blanchedalmond"              "blue"  
[27] "blue1"                        "blue2"  
[29] "blue3"                        "blue4"  
[31] "blueviolet"                   "brown"  
[33] "brown1"                      "brown2"  
[35] "brown3"                      "brown4"  
[37] "burlywood"                   "burlywood1"  
[39] "burlywood2"                   "burlywood3"  
[41] "burlywood4"                   "cadetblue"  
[43] "cadetblue1"                   "cadetblue2"  
[45] "cadetblue3"                   "cadetblue4"  
[47] "chartreuse"                   "chartreuse1"  
[49] "chartreuse2"                   "chartreuse3"  
[51] "chartreuse4"                   "chocolate"  
[53] "chocolate1"                   "chocolate2"  
[55] "chocolate3"                   "chocolate4"  
[57] "coral"                         "coral1"  
[59] "coral2"                         "coral3"  
[61] "coral4"                         "cornflowerblue"  
[63] "cornsilk"                      "cornsilk1"  
[65] "cornsilk2"                      "cornsilk3"  
[67] "cornsilk4"                      "cyan"  
[69] "cyan1"                          "cyan2"  
[71] "cyan3"                          "cyan4"  
[73] "darkblue"                      "darkcyan"  
[75] "darkgoldenrod"                 "darkgoldenrod1"  
[77] "darkgoldenrod2"                 "darkgoldenrod3"  
[79] "darkgoldenrod4"                 "darkgray"  
[81] "darkgreen"                     "darkgrey"  
[83] "darkkhaki"                     "darkmagenta"  
[85] "darkolivegreen"                 "darkolivegreen1"
```

[87] "darkolivegreen2"	"darkolivegreen3"
[89] "darkolivegreen4"	"darkorange"
[91] "darkorange1"	"darkorange2"
[93] "darkorange3"	"darkorange4"
[95] "darkorchid"	"darkorchid1"
[97] "darkorchid2"	"darkorchid3"
[99] "darkorchid4"	"darkred"
[101] "darksalmon"	"darkseagreen"
[103] "darkseagreen1"	"darkseagreen2"
[105] "darkseagreen3"	"darkseagreen4"
[107] "darkslateblue"	"darkslategray"
[109] "darkslategray1"	"darkslategray2"
[111] "darkslategray3"	"darkslategray4"
[113] "darkslategrey"	"darkturquoise"
[115] "darkviolet"	"deeppink"
[117] "deeppink1"	"deeppink2"
[119] "deeppink3"	"deeppink4"
[121] "deepskyblue"	"deepskyblue1"
[123] "deepskyblue2"	"deepskyblue3"
[125] "deepskyblue4"	"dimgray"
[127] "dimgrey"	"dodgerblue"
[129] "dodgerblue1"	"dodgerblue2"
[131] "dodgerblue3"	"dodgerblue4"
[133] "firebrick"	"firebrick1"
[135] "firebrick2"	"firebrick3"
[137] "firebrick4"	"floralwhite"
[139] "forestgreen"	"gainsboro"
[141] "ghostwhite"	"gold"
[143] "gold1"	"gold2"
[145] "gold3"	"gold4"
[147] "goldenrod"	"goldenrod1"
[149] "goldenrod2"	"goldenrod3"
[151] "goldenrod4"	"gray"
[153] "gray0"	"gray1"
[155] "gray2"	"gray3"
[157] "gray4"	"gray5"
[159] "gray6"	"gray7"
[161] "gray8"	"gray9"
[163] "gray10"	"gray11"
[165] "gray12"	"gray13"
[167] "gray14"	"gray15"
[169] "gray16"	"gray17"
[171] "gray18"	"gray19"
[173] "gray20"	"gray21"
[175] "gray22"	"gray23"

[177] "gray24"	"gray25"
[179] "gray26"	"gray27"
[181] "gray28"	"gray29"
[183] "gray30"	"gray31"
[185] "gray32"	"gray33"
[187] "gray34"	"gray35"
[189] "gray36"	"gray37"
[191] "gray38"	"gray39"
[193] "gray40"	"gray41"
[195] "gray42"	"gray43"
[197] "gray44"	"gray45"
[199] "gray46"	"gray47"
[201] "gray48"	"gray49"
[203] "gray50"	"gray51"
[205] "gray52"	"gray53"
[207] "gray54"	"gray55"
[209] "gray56"	"gray57"
[211] "gray58"	"gray59"
[213] "gray60"	"gray61"
[215] "gray62"	"gray63"
[217] "gray64"	"gray65"
[219] "gray66"	"gray67"
[221] "gray68"	"gray69"
[223] "gray70"	"gray71"
[225] "gray72"	"gray73"
[227] "gray74"	"gray75"
[229] "gray76"	"gray77"
[231] "gray78"	"gray79"
[233] "gray80"	"gray81"
[235] "gray82"	"gray83"
[237] "gray84"	"gray85"
[239] "gray86"	"gray87"
[241] "gray88"	"gray89"
[243] "gray90"	"gray91"
[245] "gray92"	"gray93"
[247] "gray94"	"gray95"
[249] "gray96"	"gray97"
[251] "gray98"	"gray99"
[253] "gray100"	"green"
[255] "green1"	"green2"
[257] "green3"	"green4"
[259] "greenyellow"	"grey"
[261] "grey0"	"grey1"
[263] "grey2"	"grey3"
[265] "grey4"	"grey5"

[267] "grey6"	"grey7"
[269] "grey8"	"grey9"
[271] "grey10"	"grey11"
[273] "grey12"	"grey13"
[275] "grey14"	"grey15"
[277] "grey16"	"grey17"
[279] "grey18"	"grey19"
[281] "grey20"	"grey21"
[283] "grey22"	"grey23"
[285] "grey24"	"grey25"
[287] "grey26"	"grey27"
[289] "grey28"	"grey29"
[291] "grey30"	"grey31"
[293] "grey32"	"grey33"
[295] "grey34"	"grey35"
[297] "grey36"	"grey37"
[299] "grey38"	"grey39"
[301] "grey40"	"grey41"
[303] "grey42"	"grey43"
[305] "grey44"	"grey45"
[307] "grey46"	"grey47"
[309] "grey48"	"grey49"
[311] "grey50"	"grey51"
[313] "grey52"	"grey53"
[315] "grey54"	"grey55"
[317] "grey56"	"grey57"
[319] "grey58"	"grey59"
[321] "grey60"	"grey61"
[323] "grey62"	"grey63"
[325] "grey64"	"grey65"
[327] "grey66"	"grey67"
[329] "grey68"	"grey69"
[331] "grey70"	"grey71"
[333] "grey72"	"grey73"
[335] "grey74"	"grey75"
[337] "grey76"	"grey77"
[339] "grey78"	"grey79"
[341] "grey80"	"grey81"
[343] "grey82"	"grey83"
[345] "grey84"	"grey85"
[347] "grey86"	"grey87"
[349] "grey88"	"grey89"
[351] "grey90"	"grey91"
[353] "grey92"	"grey93"
[355] "grey94"	"grey95"

[357] "grey96"	"grey97"
[359] "grey98"	"grey99"
[361] "grey100"	"honeydew"
[363] "honeydew1"	"honeydew2"
[365] "honeydew3"	"honeydew4"
[367] "hotpink"	"hotpink1"
[369] "hotpink2"	"hotpink3"
[371] "hotpink4"	"indianred"
[373] "indianred1"	"indianred2"
[375] "indianred3"	"indianred4"
[377] "ivory"	"ivory1"
[379] "ivory2"	"ivory3"
[381] "ivory4"	"khaki"
[383] "khaki1"	"khaki2"
[385] "khaki3"	"khaki4"
[387] "lavender"	"lavenderblush"
[389] "lavenderblush1"	"lavenderblush2"
[391] "lavenderblush3"	"lavenderblush4"
[393] "lawngreen"	"lemonchiffon"
[395] "lemonchiffon1"	"lemonchiffon2"
[397] "lemonchiffon3"	"lemonchiffon4"
[399] "lightblue"	"lightblue1"
[401] "lightblue2"	"lightblue3"
[403] "lightblue4"	"lightcoral"
[405] "lightcyan"	"lightcyan1"
[407] "lightcyan2"	"lightcyan3"
[409] "lightcyan4"	"lightgoldenrod"
[411] "lightgoldenrod1"	"lightgoldenrod2"
[413] "lightgoldenrod3"	"lightgoldenrod4"
[415] "lightgoldenrodyellow"	"lightgray"
[417] "lightgreen"	"lightgrey"
[419] "lightpink"	"lightpink1"
[421] "lightpink2"	"lightpink3"
[423] "lightpink4"	"lightsalmon"
[425] "lightsalmon1"	"lightsalmon2"
[427] "lightsalmon3"	"lightsalmon4"
[429] "lightseagreen"	"lightskyblue"
[431] "lightskyblue1"	"lightskyblue2"
[433] "lightskyblue3"	"lightskyblue4"
[435] "lightslateblue"	"lightslategray"
[437] "lightslategrey"	"lightsteelblue"
[439] "lightsteelblue1"	"lightsteelblue2"
[441] "lightsteelblue3"	"lightsteelblue4"
[443] "lightyellow"	"lightyellow1"
[445] "lightyellow2"	"lightyellow3"

[447] "lightyellow4"	"limegreen"
[449] "linen"	"magenta"
[451] "magenta1"	"magenta2"
[453] "magenta3"	"magenta4"
[455] "maroon"	"maroon1"
[457] "maroon2"	"maroon3"
[459] "maroon4"	"mediumaquamarine"
[461] "mediumblue"	"mediumorchid"
[463] "mediumorchid1"	"mediumorchid2"
[465] "mediumorchid3"	"mediumorchid4"
[467] "mediumpurple"	"mediumpurple1"
[469] "mediumpurple2"	"mediumpurple3"
[471] "mediumpurple4"	"mediumseagreen"
[473] "mediumslateblue"	"mediumspringgreen"
[475] "mediumturquoise"	"mediumvioletred"
[477] "midnightblue"	"mintcream"
[479] "mistyrose"	"mistyrose1"
[481] "mistyrose2"	"mistyrose3"
[483] "mistyrose4"	"moccasin"
[485] "navajowhite"	"navajowhite1"
[487] "navajowhite2"	"navajowhite3"
[489] "navajowhite4"	"navy"
[491] "navyblue"	"oldlace"
[493] "olivedrab"	"olivedrab1"
[495] "olivedrab2"	"olivedrab3"
[497] "olivedrab4"	"orange"
[499] "orange1"	"orange2"
[501] "orange3"	"orange4"
[503] "orangered"	"orangered1"
[505] "orangered2"	"orangered3"
[507] "orangered4"	"orchid"
[509] "orchid1"	"orchid2"
[511] "orchid3"	"orchid4"
[513] "palegoldenrod"	"palegreen"
[515] "palegreen1"	"palegreen2"
[517] "palegreen3"	"palegreen4"
[519] "paleturquoise"	"paleturquoise1"
[521] "paleturquoise2"	"paleturquoise3"
[523] "paleturquoise4"	"palevioletred"
[525] "palevioletred1"	"palevioletred2"
[527] "palevioletred3"	"palevioletred4"
[529] "papayawhip"	"peachpuff"
[531] "peachpuff1"	"peachpuff2"
[533] "peachpuff3"	"peachpuff4"
[535] "peru"	"pink"

[537] "pink1"	"pink2"
[539] "pink3"	"pink4"
[541] "plum"	"plum1"
[543] "plum2"	"plum3"
[545] "plum4"	"powderblue"
[547] "purple"	"purple1"
[549] "purple2"	"purple3"
[551] "purple4"	"red"
[553] "red1"	"red2"
[555] "red3"	"red4"
[557] "rosybrown"	"rosybrown1"
[559] "rosybrown2"	"rosybrown3"
[561] "rosybrown4"	"royalblue"
[563] "royalblue1"	"royalblue2"
[565] "royalblue3"	"royalblue4"
[567] "saddlebrown"	"salmon"
[569] "salmon1"	"salmon2"
[571] "salmon3"	"salmon4"
[573] "sandybrown"	"seagreen"
[575] "seagreen1"	"seagreen2"
[577] "seagreen3"	"seagreen4"
[579] "seashell"	"seashell1"
[581] "seashell2"	"seashell3"
[583] "seashell4"	"sienna"
[585] "sienna1"	"sienna2"
[587] "sienna3"	"sienna4"
[589] "skyblue"	"skyblue1"
[591] "skyblue2"	"skyblue3"
[593] "skyblue4"	"slateblue"
[595] "slateblue1"	"slateblue2"
[597] "slateblue3"	"slateblue4"
[599] "slategray"	"slategray1"
[601] "slategray2"	"slategray3"
[603] "slategray4"	"slategrey"
[605] "snow"	"snow1"
[607] "snow2"	"snow3"
[609] "snow4"	"springgreen"
[611] "springgreen1"	"springgreen2"
[613] "springgreen3"	"springgreen4"
[615] "steelblue"	"steelblue1"
[617] "steelblue2"	"steelblue3"
[619] "steelblue4"	"tan"
[621] "tan1"	"tan2"
[623] "tan3"	"tan4"
[625] "thistle"	"thistle1"

```
[627] "thistle2"           "thistle3"
[629] "thistle4"           "tomato"
[631] "tomato1"            "tomato2"
[633] "tomato3"            "tomato4"
[635] "turquoise"          "turquoise1"
[637] "turquoise2"          "turquoise3"
[639] "turquoise4"          "violet"
[641] "violetred"           "violetred1"
[643] "violetred2"          "violetred3"
[645] "violetred4"          "wheat"
[647] "wheat1"              "wheat2"
[649] "wheat3"              "wheat4"
[651] "whitesmoke"          "yellow"
[653] "yellow1"              "yellow2"
[655] "yellow3"              "yellow4"
[657] "yellowgreen"
```

Pour savoir à quelle couleur correspond chaque nom, le plus simple est probablement de consulter [ce document pdf](#) (n'hésitez pas à le sauvegarder si vous pensez en avoir besoin plus tard).

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\nnde New York") +
  scale_fill_manual(values = c("dodgerblue1", "mediumorchid2", "red2"))
```

Outre ces 657 couleurs qui disposent d'un nom spécifique, il est possible de spécifier les couleurs en utilisant des codes hexadécimaux et des codes rgb (red, green, blue). De nombreux sites permettent de choisir n'importe quelle couleur dans une palette qui en compte des millions et d'obtenir de tels codes. [Ce site](#) permet de le faire très simplement :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports\nnde New York") +
  scale_fill_manual(values = c("#6f71f2", "#ff299", "#f2b86f"))
```

Dernière chose concernant les couleurs : un choix de fonction `scale_XXX_XXX()` inapproprié est la cause d'erreur la plus fréquente ! Par exemple, si on reprend le code des figures [10](#) et [11](#) et que l'on modifie les palettes de couleurs, notez que les fonctions utilisées ne sont pas les mêmes :

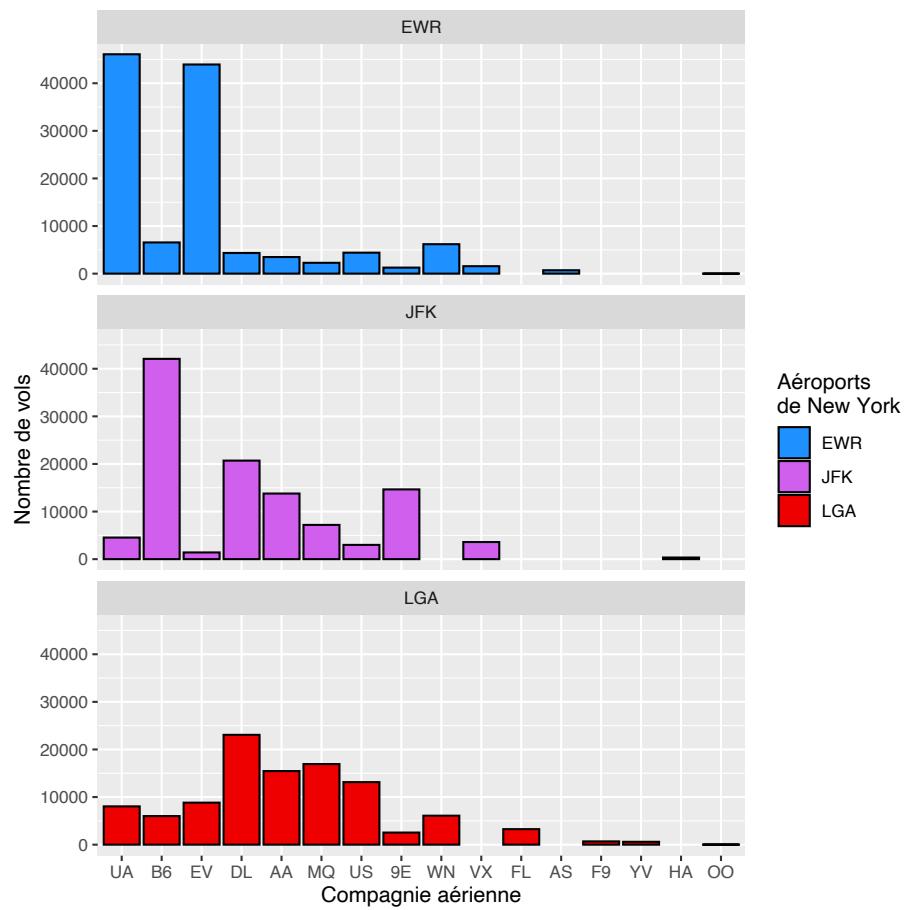


Figure 60 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

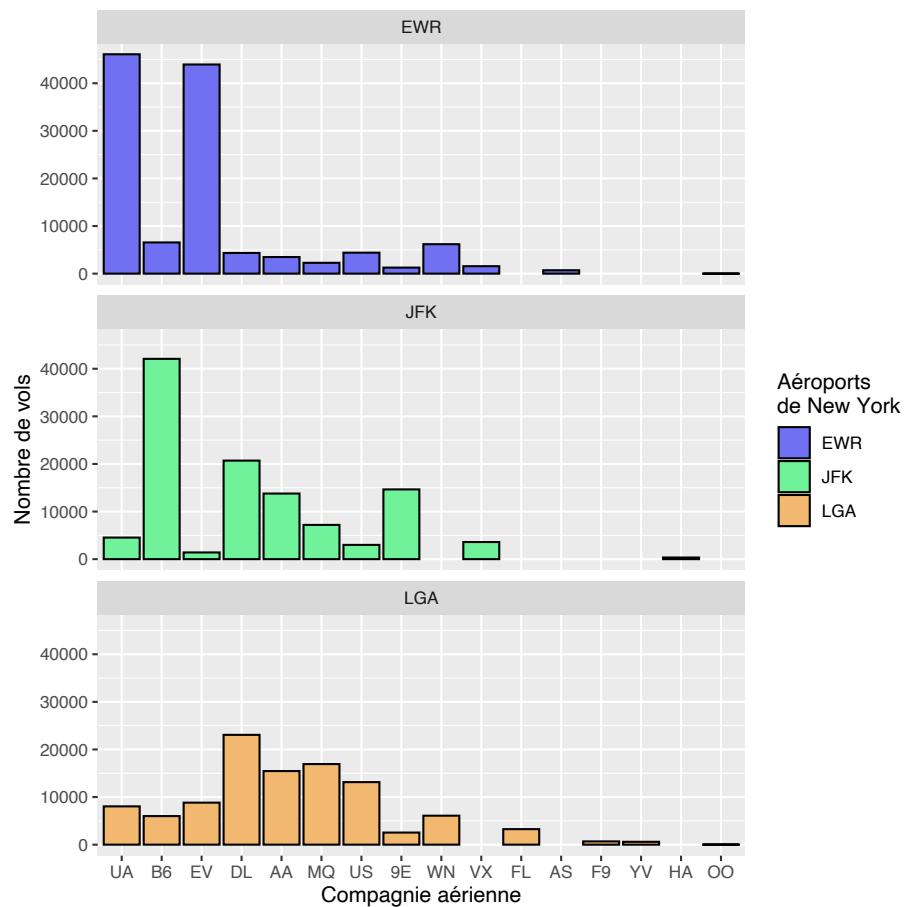


Figure 61 – Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay,
                                              color = factor(month))) +
  geom_point() +
  scale_color_viridis_d()
```

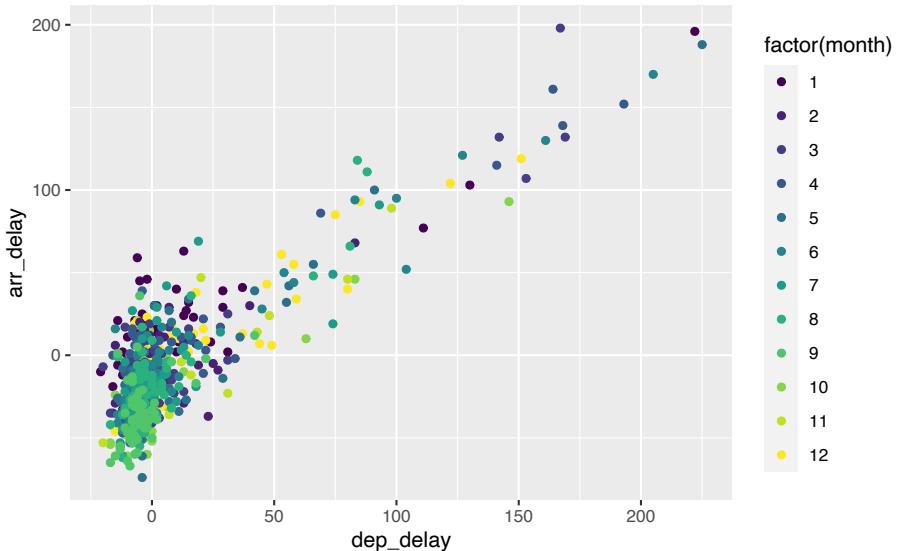


Figure 62 – Association de ‘color’ à une variable catégorielle.

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay,
                                              color = arr_time)) +
  geom_point() +
  scale_color_viridis_c()
```

Pour les 2 figures 62 et 63, j'utilise la palette de couleur viridis. Pour ces 2 graphiques, c'est la couleur des points qui change. Puisque cette couleur est spécifiée avec l'esthétique color et non plus fill, la fonction utilisée est `scale_color_XXX()` et non plus `scale_fill_XXX()`.

Enfin, pour la figure 62, c'est une variable catégorielle qui est associée à l'esthétique de couleur (`factor(month)`). La fonction utilisée pour modifier les couleurs doit donc en tenir compte : le \_d à la fin de `scale_color_viridis_d()` signifie “discrete”, c'est-à-dire “discontinu”. À l'inverse, pour le graphique 63, c'est une variable numérique continue qui est associée à l'esthétique de couleur (`arr_time`). La fonction utilisée pour modifier les couleurs en est le reflet : le \_c à la fin de `scale_color_viridis_c()` est l'abréviation de “continuous”, c'est-à-dire “continue”.

Si vous ne voulez pas avoir de message d'erreur, attention donc, à choisir la fonction `scale_XXX_XXX()` appropriée. Pour cela, aidez-vous de l'aide que RStudio vous apporte en tapant les premières lettres de la fonction et en parcourant la liste des fonctions proposées dans le menu déroullant qui apparaît sous votre curseur.

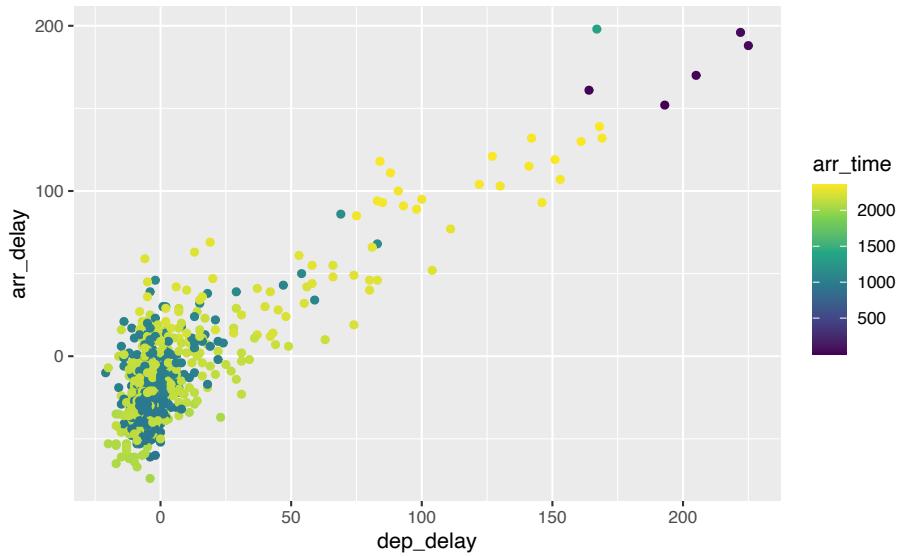


Figure 63 – Association de ‘color’ à une variable numérique.

#### 4.9.3 Les thèmes

L’apparence de tout ce qui ne concerne pas directement les données d’un graphique est sous le contrôle d’un thème. Les thèmes contrôlent l’apparence générale du graphique : quelles polices et tailles de caractères sont utilisées, quel sera l’arrière plan du graphique, faut-il intégrer un quadrillage sous le graphique, et si oui, quelles doivent être ses caractéristiques ?

Il est possible de spécifier chaque élément manuellement. Nous nous contenterons ici de passer en revue quelques thèmes prédéfinis qui devraient couvrir la plupart de vos besoins.

Reprendons par exemple le code de la figure 58 et ajoutons un titre :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports de\nNew York",
       title = "Couverture inégale des aéroports de New York") +
  scale_fill_brewer(palette = "Accent")
```

Le thème utilisé par défaut est `theme_gray()`. Il est notamment responsable de l’arrière plan gris et du quadrillage blanc. Pour changer de thème, il suffit d’ajouter une couche au graphique en donnant le nom du nouveau thème :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
```

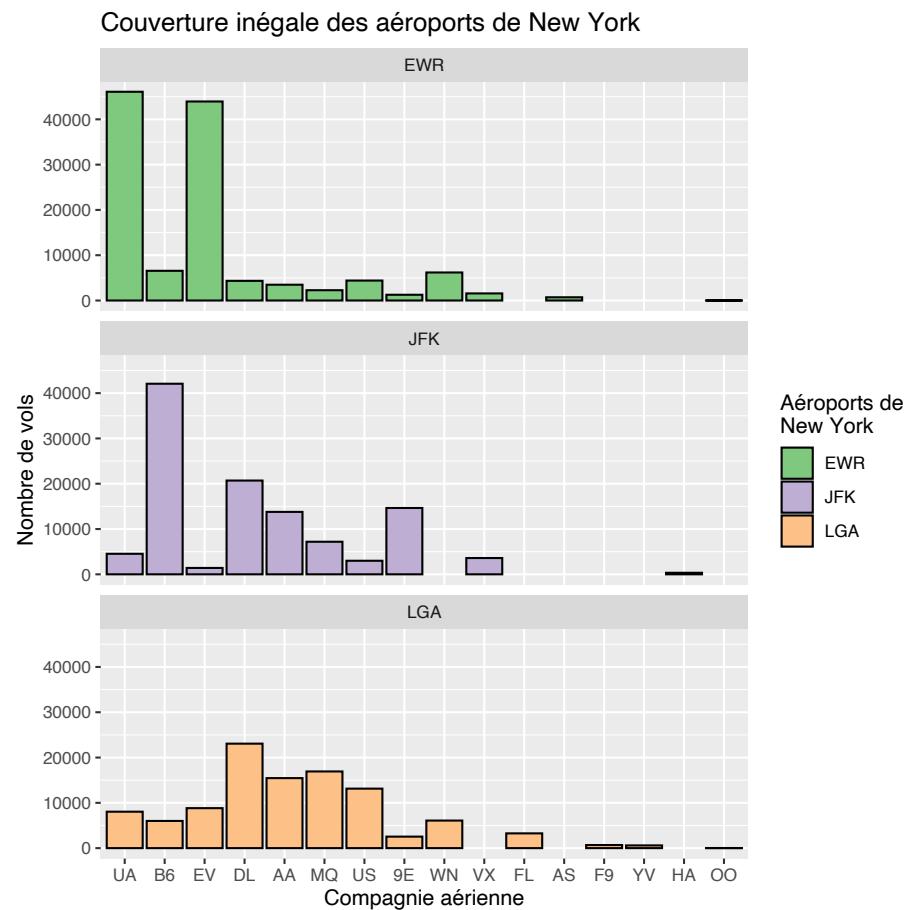


Figure 64 – Utilisation du thème par défaut : theme\_gray().

```

y = "Nombre de vols",
fill = "Aéroports de\nNew York",
title = "Couverture inégale des aéroports de New York") +
scale_fill_brewer(palette = "Accent") +
theme_bw()

```

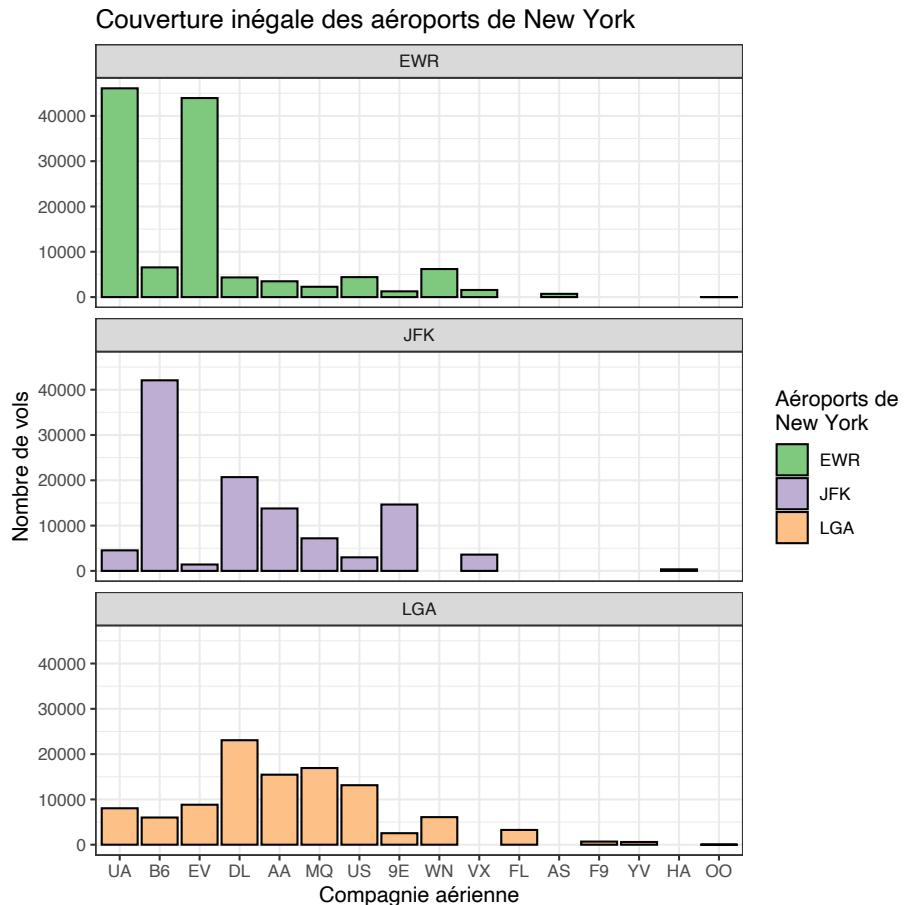


Figure 65 – Utilisation du thème theme\_bw( ).

Les thèmes complets que vous pouvez utiliser sont les suivants :

- theme\_bw() : fond blanc et quadrillage.
- theme\_classic() : thème classique, avec des axes mais pas de quadrillage.
- theme\_dark() : fond sombre pour augmenter le contraste.
- theme\_gray() : thème par défaut : fond gris et quadrillage blanc.
- theme\_light() : axes et quadrillages discrets.
- theme\_linedraw() : uniquement des lignes noires.
- theme\_minimal() : pas d'arrière plan, pas d'axes, quadrillage discret.
- theme\_void() : thème vide, seuls les objets géométriques restent visibles.

```

ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports de\nNew York",
       title = "Couverture inégale des aéroports de New York") +
  scale_fill_brewer(palette = "Accent") +
  theme_minimal()

```

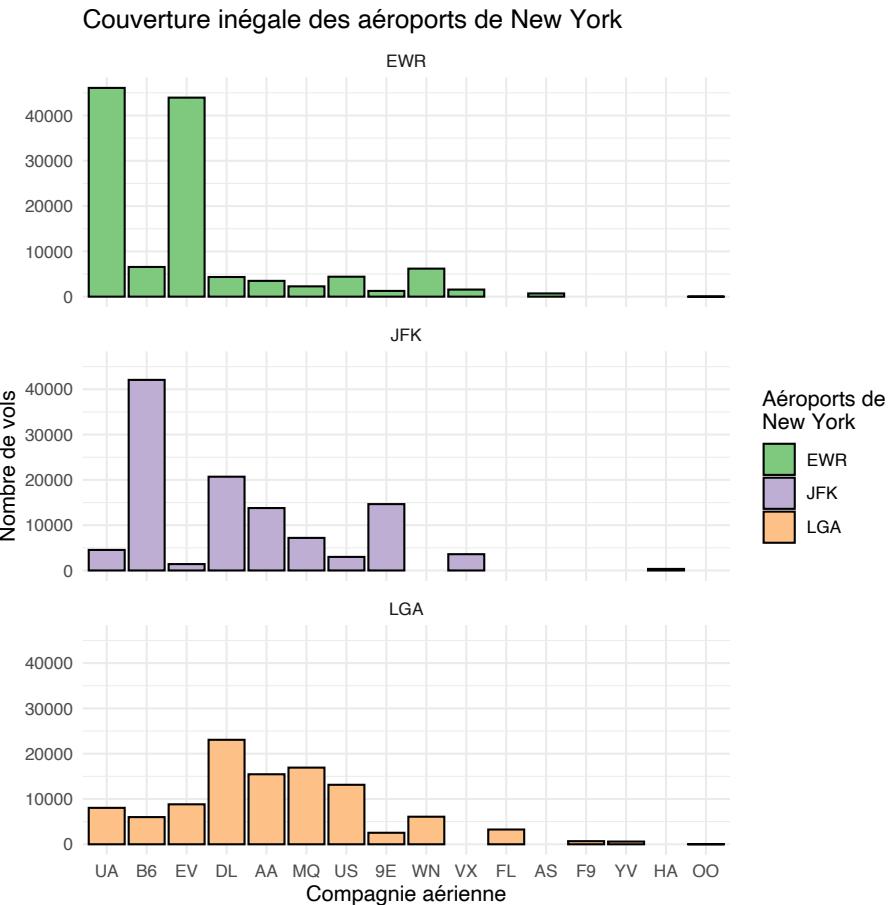


Figure 66 – Utilisation du thème minimalist.

L'argument `base_family` de chaque thème permet de spécifier une police de caractères différente de celle utilisée par défaut. Évidemment, vous ne pourrez utiliser que des polices qui sont disponibles sur l'ordinateur que vous utilisez. Dans l'exemple de la figure 67 ci-dessous, j'utilise la police “Gill Sans”. Si cette police n'est pas disponible sur votre ordinateur, ce code produira une erreur. Si c'est le cas, remplacez-la par une police de votre ordinateur. Attention, son nom exact doit être utilisé. Cela signifie bien sûr le respect des espaces, majuscules, etc.

```

ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +
  geom_bar(color = "black") +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Compagnie aérienne",
       y = "Nombre de vols",
       fill = "Aéroports de New York",
       title = "Couverture inégale des aéroports de New York") +
  scale_fill_brewer(palette = "Accent") +
  theme_minimal(base_family = "Gill Sans")

```

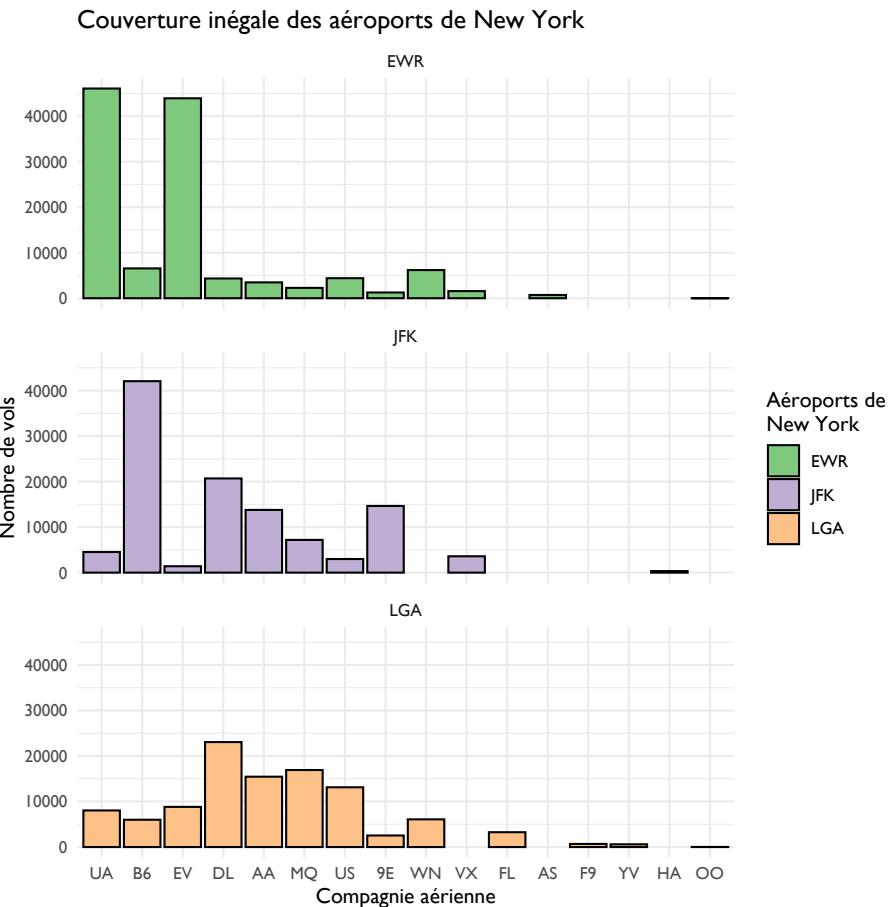


Figure 67 – Modification de la police de caractères.

Le choix d'un thème et d'une police adaptés doivent vous permettre de faire des graphiques originaux et clairs. Rappelez-vous toujours que vos choix en matière de graphiques doivent avoir pour objectif principal de rendre les tendances plus faciles à décrypter pour un lecteur non familier de vos données. C'est un outil de communication au même titre que n'importe quel paragraphe d'un rapport ou compte-rendu. Et comme pour un paragraphe, la première version d'un graphique est rarement la bonne.

Vous devriez donc maintenant être bien armés pour produire 95% des graphiques dont vous aurez besoin tout au long de votre cursus universitaire. Toutefois, un point important a pour

l'instant été omis : l'ajout de barres d'erreurs sur vos graphiques. Nous verrons comment faire cela un peu plus tard, après avoir appris à manipuler efficacement des tableaux de données avec les packages `tidyverse` et `dplyr`.

---

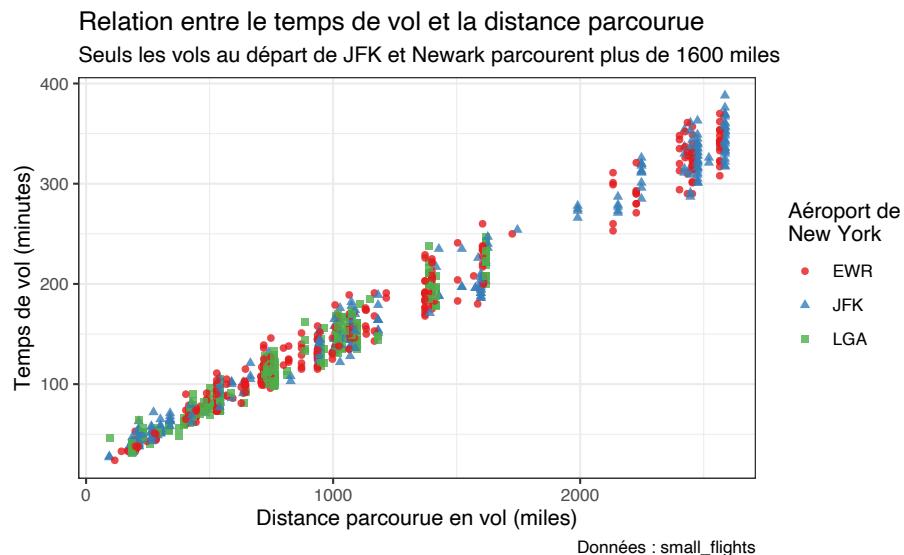
## 4.10 Exercices

Commencez par créer un nouveau jeu de données en exécutant ces commandes :

```
set.seed(1234)
small_flights <- flights %>%
  sample_n(1000) %>%
  filter(!is.na(arr_delay),
        distance < 3000)
```

Ce nouveau jeu de données de petite taille (972 lignes) est nommé `small_flights`. Il contient les mêmes variables que le tableau `flights` mais ne contient qu'une petite fraction de ses lignes. Les lignes retenues ont été choisies au hasard. Vous pouvez visualiser son contenu en tapant son nom dans la console ou en utilisant la fonction `View()`.

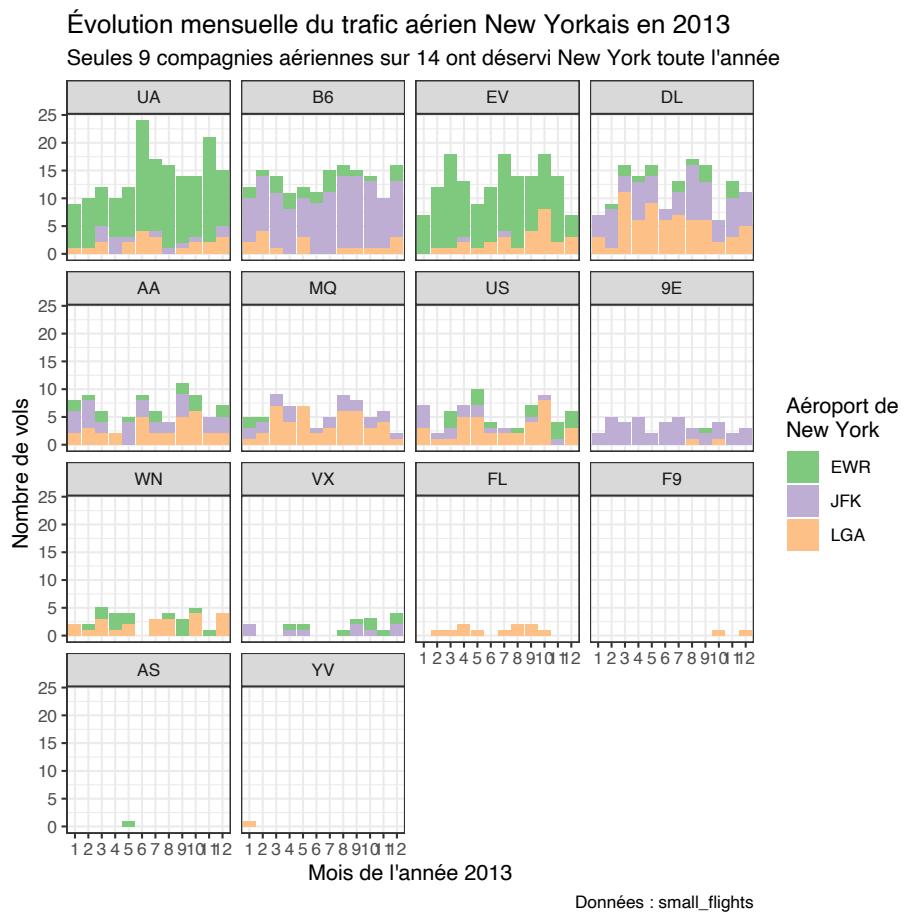
En vous appuyant sur les fonctions et les principes de la grammaire des graphiques que vous avez découverts dans ce chapitre 4, et en vous servant de ce nouveau jeu de données, tapez les commandes qui permettent de produire le graphique ci-dessous :



Quelques indices :

- Les couleurs utilisées sont celles de la palette Set1 du package RColorBrewer.
- Les variables utilisées sont `origin`, `air_time` et `distance`.
- La transparence des symboles est fixée à 0.8.

Toujours avec ce jeu de données small-flights, tapez les commandes permettant de produire le graphique ci-dessous :



Quelques indices :

- Les couleurs utilisées sont celles de la palettes Accent du package RColorBrewer.
- Les variables utilisées sont month, carrier et origin.

## 5 (Ar)ranger des données avec tidyverse

Dans la section 2.2.4.1, nous avons introduit le concept de tableaux de données ou `data.frame` dans R. Il s'agit d'une représentation rectangulaire des données, à la manière d'un tableau, dans laquelle les lignes correspondent aux observations et les colonnes correspondent à des variables décrivant chaque observation.

Dans ce chapitre, nous allons aller plus loin en présentant le concept de “tidy data”, ou “données nettes/rangées/soignées/ordonnées”. Vous verrez que l'idée d'avoir des données stockées dans un format “net” va plus loin que la simple définition usuelle que le terme “rangé” peut avoir lorsque les données sont simplement bien organisées dans un tableau. Nous définirons le terme “tidy data” de manière plus rigoureuse, en établissant un ensemble de règles permettant de

stocker les données correctement afin de rendre plus aisées les analyses statistiques et les représentations graphiques.

Jusqu'à maintenant, vous avez utilisé des données qui étaient déjà dans ce format (c'est le cas des données contenues dans `flights` ou dans `diamonds` par exemple). Pourtant, la plupart du temps, les données que vous manipulerez dans R seront importées depuis un tableur dans lequel vous ou vos collaborateurs en aurez fait la saisie. S'assurer que les données importées manuellement dans R sont correctement "nettoyées" et mises en forme de "tidy data" est indispensable pour éviter les problèmes lors de la réalisation de graphiques (voir chapitre 4) comme lors de la manipulation des données pour en tirer de l'information statistique pertinente (ce que nous verrons au chapitre 6).

---

## 5.1 Prérequis

Dans ce chapitre, nous aurons besoin des packages suivants :

```
library(tidyr)
library(dplyr)
library(nycflights13)
library(ggplot2)
library(readxl) # la dernière lettre est un "L" minuscule, pas le chiffre 1 ...
library(readr)
```

Comme d'habitude, si vous recevez des messages d'erreur, c'est probablement parce que le package que vous essayez de charger en mémoire n'a pas été installé au préalable. Consultez la section 2.3 si vous ne savez plus comment procéder.

Outre ces packages classiques, nous aurons aussi besoin du package EDAWR qui n'est pas disponible sur les serveurs habituels de R. Pour l'installer, on procède de la façon suivante :

1. Installez et chargez en mémoire le package `remotes` :

```
install.packages("remotes")
library(remotes)
```

2. Installez le package EDAWR grâce à la fonction `install_github()` du package `remotes` :

```
install_github("rstudio/EDAWR")
```

Attention, sur les ordinateurs de l'université cette procédure ne fonctionne pas toujours. Si vous rencontrez des difficultés, suivez les instructions décrites à la fin de cette section 5.1

3. Chargez le package EDAWR de la façon habituelle :

```
library(EDAWR)
```

Le package EDAWR contient plusieurs jeux de données dont nous allons nous servir pour illustrer les questions liées au format des tableaux de données. Pour en avoir la liste, vous pouvez taper :

```
data(package = "EDAWR")
```

### **En cas de problème pour installer le package EDAWR sur les ordinateurs de l'université.**

Vous pouvez télécharger manuellement les 4 jeux de données dont nous aurons besoin grâce à ces 4 liens :

- [cases](#)
- [population](#)
- [rates](#)
- [storms](#)

Une fois téléchargés, les données contenues dans ces 4 fichiers peuvent être importées dans RStudio en cliquant sur File > Open File ..., puis en sélectionnant un à un chacun des fichiers. Pour chaque fichier un nouvel objet doit apparaître dans votre environnement de travail (onglet Environnement, dans le panneau en haut à droite de RStudio). L'inconvénient de cette méthode est que les fichiers d'aide de ces jeux de données ne seront pas disponibles dans RStudio. Vous pouvez toutefois en consulter une version brute (non mise en forme) [en cliquant ici](#).

---

## **5.2 C'est quoi des “tidy data” ?**

Les “tidy data” (nous les appellerons “données rangées” dans la suite de ce livre), sont des données qui respectent un format standardisé. En particulier :

- chaque variable est dans une colonne unique
- chaque colonne contient une unique variable
- chaque ligne correspond à une observation pour chaque variable
- les cellules du tableau représentent les valeurs de chaque observation pour chaque variable.

Malheureusement, les données peuvent être présentées sous de nombreux formats qui ne respectent pas ces règles de base. La modification des tableaux est donc souvent un préambule nécessaire à toute analyse statistique ou représentation graphique.

Par exemple, examinez le tableau cases du package EDAWR, qui présente le nombre de cas de tuberculose dans 3 pays en 2011, 2012 et 2013.

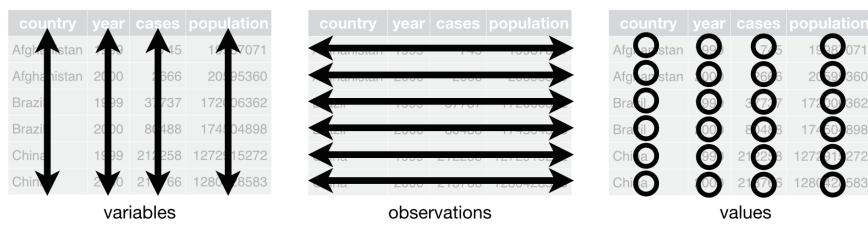


Figure 68 – La définition des 'données rangées', d'après <http://r4ds.had.co.nz/tidy-data.html>

cases

	country	2011	2012	2013
1	FR	7000	6900	7000
2	DE	5800	6000	6200
3	US	15000	14000	13000

Dans ce tableau, essayez d'identifier quelles sont les variables en présence. Indice, vous devriez en trouver 3.

Essayez d'identifier également où se trouvent ces variables.

Pour ma part, je compte les 3 variables suivantes :

1. country : qui indique les pays dans lesquels les cas de tuberculose ont été dénombrés.  
Cette variable occupe la première colonne du tableau.
2. la seconde variable est l'année, qui peut prendre les valeurs 2011, 2012 ou 2013. Cette variable occupe la ligne des titres des 3 colonnes de droite du tableau.
3. et enfin, la troisième variable est le nombre de cas de tuberculose observés dans chaque pays et chaque année. Cette troisième variable occupe 3 lignes et 3 colonnes du tableau.

Autrement dit, les variables peuvent être visualisées de la façon suivante :

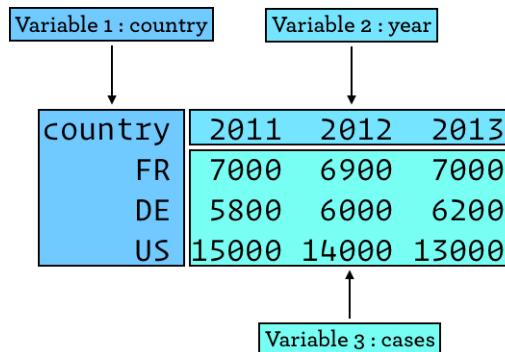


Figure 69 – Position des variables dans le tableau ‘cases’ du package ‘EDAWR’

Donc même si nous disposons ici d'un tableau rectangulaire classique, nous sommes bien loin du format des données rangées.

### 5.2.1 La fonction pivot\_longer()

Afin de transformer les données non rangées du tableau cases en données rangées, nous allons utiliser la fonction `pivot_longer()` du package `tidyverse`. Avant d'aller plus loin, essayez d'imaginer à quoi le tableau rangé devrait ressembler.

La fonction `pivot_longer()` prend 4 arguments :

1. `data` : le nom du tableau de données que l'on souhaite “ranger”.
2. `cols` : La liste des colonnes du tableau initial que l'on souhaite rassembler en 2 nouvelles variables. Ici, les colonnes 2, 3 et 4 (on pourra les noter `2:4` ou, en utilisant leur nom, `“2011”：“2013”`).
3. `names_to` : le nom d'une nouvelle variable qui contiendra les en-têtes des colonnes qui constituent la seconde variable. Ici, nous nommerons cette seconde variable `year` car elle devra contenir les années 2011, 2012 et 2013.
4. `values_to` : le nom d'une nouvelle variable qui contiendra les informations correspondant à la troisième variable identifiée plus haut. Nous appelerons cette variable `n_cases` car elle contiendra les nombres de cas de tuberculose (7000, 5800, 15000, etc).

```
pivot_longer(data = cases,
             cols = `2011`:`2013`,
             names_to = "year",
             values_to = "n_cases")
```

```
# A tibble: 9 x 3
  country year  n_cases
  <chr>   <chr>  <dbl>
1 FR      2011    7000
2 FR      2012    6900
3 FR      2013    7000
4 DE      2011    5800
5 DE      2012    6000
6 DE      2013    6200
7 US      2011   15000
8 US      2012   14000
9 US      2013   13000
```

Nous avons bien transformé le tableau de départ en un “tableau rangé” : chacune de nos 3 variables se trouve dans une unique colonne, et chaque ligne correspond à une observation pour chacune de ces 3 variables. Comme d'habitude, si nous souhaitons pouvoir utiliser ce nouveau tableau, il faut lui donner un nom :

```
cases_tidy <- pivot_longer(data = cases,
                            cols = `2011`:`2013`,
                            names_to = "year",
                            values_to = "n_cases")
```

Il nous est maintenant plus facile de manipuler ces données pour en tirer de l'information, grâce à des analyses statistiques ou des représentations graphiques :

```
ggplot(cases_tidy, aes(x = country, y = n_cases, fill = year)) +
  geom_col(position = "dodge", color = "black") +
  scale_fill_brewer(palette = "Accent") +
  theme_minimal() +
  labs(x = "Pays",
       y = "Nombre de cas",
       fill = "Année",
       title = "Évolution du nombre de cas de tuberculose entre 2011 et 2013",
       subtitle = "DE : Allemagne, FR : France, US : États-Unis")
```

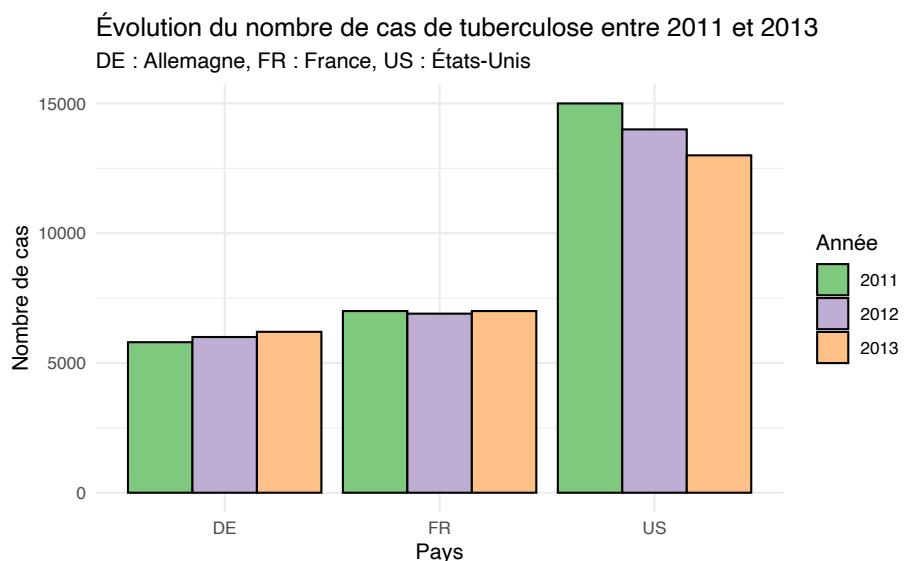


Figure 70 – Évolution du nombre de cas de tuberculose dans 3 pays, de 2011 à 2013.

On constate ici qu'entre 2011 et 2013, le nombre de cas de tuberculose a légèrement augmenté en Allemagne, est resté stable en France, et a diminué aux États-Unis.

Notez ici que la variable `year` de notre nouveau tableau est considérée comme une variable de type “chaîne de caractères” et non comme une variable numérique. On peut le voir en affichant notre tableau en tapant son nom, ou en utilisant la fonction `str()` déjà décrite plus tôt :

```
str(cases_tidy)
```

```
tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
$ country: chr [1:9] "FR" "FR" "FR" "DE" ...
$ year   : chr [1:9] "2011" "2012" "2013" "2011" ...
$ n_cases: num [1:9] 7000 6900 7000 5800 6000 6200 15000 14000 13000
```

C'est le comportement par défaut de la fonction `pivot_longer()` : les anciens titres de

colonnes sont convertis en chaînes de caractères. Si ce comportement n'est pas souhaitable, il y a 2 alternatives possibles :

1. utiliser les arguments `names_transform` et/ou `values_transform` de la fonction `pivot_longer()`. Cela permet de spécifier comment transformer les variables nouvellement créées au moment de leur création.
2. utiliser les fonctions `mutate()` et `as.numeric()` ou `as.integer()` après avoir modifié le tableau de départ avec `pivot_longer()`. Cette façon de faire sera décrite dans la partie [6.7](#).

```
# On commence par afficher `cases`  
cases
```

```
country 2011 2012 2013  
1      FR  7000  6900  7000  
2      DE  5800  6000  6200  
3      US 15000 14000 13000
```

```
# On utilise ensuite pivot_longer avec l'argument  
# names_transform pour transformer year en facteur  
pivot_longer(data = cases,  
             cols = `2011`:`2013`,  
             names_to = "year",  
             values_to = "n_cases",  
             names_transform = list(year = as.integer))
```

```
# A tibble: 9 x 3  
  country   year n_cases  
  <chr>     <int>   <dbl>  
1 FR         2011    7000  
2 FR         2012    6900  
3 FR         2013    7000  
4 DE         2011    5800  
5 DE         2012    6000  
6 DE         2013    6200  
7 US         2011   15000  
8 US         2012   14000  
9 US         2013   13000
```

On voit ici que la variable `year` est maintenant une colonne numérique (`<int>` : nombres entiers), et non plus une variable de type “character”. En utilisant `as.numeric()` au lieu de `as.integer()`, on aurait transformé la variable `year` en `<dbl>` (nombre réel au lieu de nombre entier), ce qui ici, reviendrait exactement au même.

De la même façon, on peut avoir besoin de présenter la colonne `year` sous la forme d'un facteur :

```

pivot_longer(data = cases,
             cols = `2011`:`2013`,
             names_to = "year",
             values_to = "n_cases",
             names_transform = list(year = as.factor))

```

```

# A tibble: 9 x 3
  country year  n_cases
  <chr>   <fct>   <dbl>
1 FR      2011     7000
2 FR      2012     6900
3 FR      2013     7000
4 DE      2011     5800
5 DE      2012     6000
6 DE      2013     6200
7 US      2011    15000
8 US      2012    14000
9 US      2013    13000

```

### 5.2.2 La fonction pivot\_wider()

La fonction `pivot_wider()` permet de réaliser l'opération inverse de `pivot_longer()`. Elle “disperse” une unique colonne catégorielle en plusieurs colonnes, le tableau obtenue est donc plus large (“wider”) que le tableau de départ.

Reprendons par exemple notre tableau `cases_tidy` :

```
cases_tidy
```

```

# A tibble: 9 x 3
  country year  n_cases
  <chr>   <chr>   <dbl>
1 FR      2011     7000
2 FR      2012     6900
3 FR      2013     7000
4 DE      2011     5800
5 DE      2012     6000
6 DE      2013     6200
7 US      2011    15000
8 US      2012    14000
9 US      2013    13000

```

La fonction `pivot_wider()` prend 3 arguments :

1. Le nom du tableau contenant les données (ici, `cases_tidy`).

2. `names_from`: le nom de la variable contenant les catégories qui devront être transformées en colonnes (ici, `year`).
3. `values_from` : le nom de la variable contenant les valeurs qui devront remplir les nouvelles colonnes (ici, `n_cases`).

```
pivot_wider(data = cases_tidy,
             names_from = year,
             values_from = n_cases)
```

```
# A tibble: 3 x 4
  country `2011` `2012` `2013`
  <chr>    <dbl>   <dbl>   <dbl>
1 FR        7000    6900    7000
2 DE        5800    6000    6200
3 US       15000   14000   13000
```

Cette fonction sera donc rarement utilisée puisqu'elle ne permet pas d'obtenir des "tableaux rangés". Toutefois, elle pourra vous être utile pour présenter des résultats sous forme synthétique. Prenons un exemple avec le jeu de données flights. Imaginons que vous deviez créer un tableau `n_vols` présentant, pour chacun des 3 aéroports de New York, le nombre de vols affrétés par chaque compagnie aérienne en 2013. Une possibilité serait de taper ceci :

```
n_vols <- flights %>%
  group_by(origin, carrier) %>%
  count()
n_vols
```

```
# A tibble: 35 x 3
# Groups:   origin, carrier [35]
  origin carrier     n
  <chr>  <chr>   <int>
1 EWR    9E        1268
2 EWR    AA        3487
3 EWR    AS         714
4 EWR    B6        6557
5 EWR    DL        4342
6 EWR    EV        43939
7 EWR    MQ        2276
8 EWR    OO          6
9 EWR    UA        46087
10 EWR   US        4405
# ... with 25 more rows
```

Les commandes permettant de produire ce tableau seront expliquées dans le chapitre 6. On peut cependant constater ici que ce tableau contient 35 lignes et 3 colonnes. Il s'agit bien d'un

“tableau rangé” parfaitement adapté pour faire des statistiques et des visualisations graphiques, mais son format n'est pas terrible si notre objectif est de le faire figurer dans un rapport. La solution : utiliser `pivot_wider()` :

```
pivot_wider(n_vols,  
            names_from = origin,  
            values_from = n)
```

```
# A tibble: 16 x 4  
# Groups:   carrier [16]  
  carrier   EWR    JFK    LGA  
  <chr>     <int> <int> <int>  
1 9E        1268  14651  2541  
2 AA        3487  13783  15459  
3 AS         714    NA     NA  
4 B6        6557  42076  6002  
5 DL        4342  20701  23067  
6 EV        43939 1408   8826  
7 MQ        2276  7193   16928  
8 OO         6     NA     26  
9 UA        46087 4534   8044  
10 US       4405  2995   13136  
11 VX       1566  3596   NA  
12 WN       6188  NA     6087  
13 HA       NA     342    NA  
14 F9       NA     NA     685  
15 FL       NA     NA     3260  
16 YV       NA     NA     601
```

Ce nouveau tableau contient maintenant 16 lignes (une par compagnie aérienne), et 4 colonnes : une pour la variable `carrier`, et 3 pour la variable `origin`, soit une colonne pour chacun des 3 aéroports de New York. On parle de tableau au format large (par opposition au “tableau rangé”, dit “format long”). Cela rend la présentation dans un rapport plus aisée.

Notez également que certaines compagnies aériennes ne desservent pas tous les aéroports. Par exemple, la compagnie Alaska Airlines (AS) ne dessert ni JFK, ni La Guardia. Pour ces catégories, notre nouveau tableau au format large indique NA. Or, NA signifie “Not Available”, autrement dit : données manquantes. Ici, il ne s'agit pas du tout de données manquantes. Cela signifie simplement qu'aucun vol d'Alaska Airline n'a décollé de ces 2 aéroports. Nous pouvons donc indiquer à R quelle valeur utiliser pour les catégories qui ne sont pas représentées dans le tableau de départ grâce à l'argument `values_fill` :

```
pivot_wider(n_vols,  
            names_from = origin,  
            values_from = n,  
            values_fill = 0)
```

```

# A tibble: 16 x 4
# Groups:   carrier [16]
  carrier    EWR    JFK    LGA
  <chr>     <int> <int> <int>
1 9E        1268  14651  2541
2 AA        3487  13783  15459
3 AS         714     0     0
4 B6        6557  42076  6002
5 DL        4342  20701  23067
6 EV        43939  1408   8826
7 MQ        2276  7193   16928
8 OO          6     0    26
9 UA        46087  4534   8044
10 US       4405  2995  13136
11 VX        1566  3596     0
12 WN       6188     0   6087
13 HA          0    342     0
14 F9          0    0    685
15 FL          0    0   3260
16 YV          0    0    601

```

D'autres arguments existent. **Je vous encourage vivement** à consulter l'aide des fonctions `pivot_longer()` et `pivot_wider()` et à faire des essais.

### 5.2.3 Les fonctions `separate()` et `unite()`

Ces fonctions sont complémentaires : tout comme `pivot_longer()` et `pivot_wider()`, elles effectuent 2 opérations opposées. Reprenons le jeu de données `cases_tidy` (que nous transformons au préalable en tibble pour mieux voir ce qui se passe) :

```

cases_tidy <- as_tibble(cases_tidy)
cases_tidy

```

```

# A tibble: 9 x 3
  country year  n_cases
  <chr>   <chr>   <dbl>
1 FR      2011    7000
2 FR      2012    6900
3 FR      2013    7000
4 DE      2011    5800
5 DE      2012    6000
6 DE      2013    6200
7 US      2011   15000
8 US      2012   14000
9 US      2013   13000

```

Imaginons que nous ayons besoin de séparer les données de la colonne year en 2 variables : le siècle d'une part, et l'année d'autre part. La fonction separate() permet de faire exactement cela :

```
separate(cases_tidy, year, into = c("century", "year"), sep = 2)
```

```
# A tibble: 9 x 4
  country century year  n_cases
  <chr>    <dbl> <dbl>    <dbl>
1 FR        20     11     7000
2 FR        20     12     6900
3 FR        20     13     7000
4 DE        20     11     5800
5 DE        20     12     6000
6 DE        20     13     6200
7 US        20     11    15000
8 US        20     12    14000
9 US        20     13    13000
```

1. Le premier argument est le nom du tableau de données
2. le second argument est la variable que l'on souhaite scinder en plusieurs morceaux
3. into est un vecteur qui contient le nom des nouvelles colonnes à créer
4. sep peut prendre plusieurs formes. Lorsqu'on utilise un nombre, ce nombre correspond à la position de la coupure dans la variable d'origine. Ici, la variable d'origine a été coupée après le second caractère. Il est aussi possible d'utiliser un symbole. Par exemple, certaines variables contiennent des tirets - ou des slash \. Utiliser ces caractères en guise de séparateur permet de couper les variables à ce niveau là. Nous en verrons un exemple plus tard.

Notez ici que les 2 nouvelles variables sont de type <chr>. Si nous souhaitons que ces variables soient considérées comme numériques, nous devons ajouter un argument lorsque nous utilisons separate() :

```
cases_split <- separate(cases_tidy,
                        year,
                        into = c("century", "year"),
                        sep = 2,
                        convert = TRUE)

cases_split
```

```
# A tibble: 9 x 4
  country century year  n_cases
  <chr>    <dbl> <dbl>    <dbl>
1 FR        20     11     7000
2 FR        20     12     6900
```

```

3 FR          20    13    7000
4 DE          20    11    5800
5 DE          20    12    6000
6 DE          20    13    6200
7 US          20    11   15000
8 US          20    12   14000
9 US          20    13   13000

```

Notre nouvel objet `cases_split` contient maintenant 2 nouvelles colonnes de nombres entiers, l'une contenant le siècle, l'autre contenant l'année

La fonction `unite()` fait exactement le contraire : elle fusionne 2 colonnes existantes en accolant leurs contenus (et en ajoutant un séparateur) :

```
unite(cases_split, new, century, year)
```

```

# A tibble: 9 x 3
  country new     n_cases
  <chr>   <chr>   <dbl>
1 FR      20_11    7000
2 FR      20_12    6900
3 FR      20_13    7000
4 DE      20_11    5800
5 DE      20_12    6000
6 DE      20_13    6200
7 US      20_11   15000
8 US      20_12   14000
9 US      20_13   13000

```

La colonne `new` a été créée par la fusion des colonnes `century` et `year` du tableau `cases_split`. Si l'on souhaite supprimer le tiret, il nous faut le spécifier explicitement :

```
unite(cases_split, new, century, year, sep = "")
```

```

# A tibble: 9 x 3
  country new     n_cases
  <chr>   <chr>   <dbl>
1 FR      2011    7000
2 FR      2012    6900
3 FR      2013    7000
4 DE      2011    5800
5 DE      2012    6000
6 DE      2013    6200
7 US      2011   15000
8 US      2012   14000
9 US      2013   13000

```

#### **5.2.4 Exercices**

Examinez les tableaux rates, storms et population du package EDAWR.

1. Ces tableaux sont-ils des “tableaux rangés” (tidy data) ?
  2. Si oui, quelles sont les variables représentées ?
  3. Si non, transformez-les en “tableaux rangés”.
- 

### **5.3 Importer des données depuis un tableur**

#### **5.3.1 Les règles de base**

Jusqu'à maintenant, nous avons travaillé exclusivement avec des jeux de données déjà disponibles dans R. La plupart du temps, les données sur lesquelles vous devrez travailler devront au préalable être importées dans R, à partir de fichiers issus de tableurs. De tels fichiers se présentent généralement sous l'un des 2 formats suivants :

1. fichiers au format “.csv” : il s'agit d'un format de fichier dit “texte brut”, c'est à dire qu'il peut être ouvert avec n'importe quel éditeur de texte, y compris le bloc notes de Windows. L'extension “.csv” est l'abréviation de Comma Separated Values, autrement dit, dans ce type de fichiers, les colonnes sont séparées par des virgules. Cela peut poser problème en France puisque le symbole des décimales est souvent aussi la virgule (et non le point comme dans les pays anglo-saxons). Le séparateur de colonnes utilisé en France dans les fichiers .csv est alors souvent le point-virgule. Il est possible de créer des fichiers .csv à partir de n'importe quel tableur en choisissant Fichier > Exporter ... ou Fichier > Enregistrer sous ... puis en sélectionnant le format approprié (les dénominations sont variables selon les logiciels : format texte brut, format csv, plain text, etc...).
2. fichiers au format tableur : .xls ou .xlsx pour Excel, .calc pour Open Office.

Dans les 2 cas, pour que R puisse importer les données contenues dans ces fichiers, un certain nombre de règles doivent être respectées :

1. La première chose à laquelle il faut veiller est la présentation des données. Les variables doivent être en colonnes et les observations en lignes. Dans l'idéal, les données doivent donc être “rangées”.
2. Les cases vides qui correspondent à des données manquantes doivent contenir les lettres NA en majuscule. Il est important de bien faire la distinction entre les vrais zéros (i.e. les grandeurs mesurées pour lesquelles un zéro a été obtenu), et les valeurs manquantes, c'est à dire pour lesquelles aucune valeur n'a pu être obtenue (e.g. variable non mesurée pour un individu donné ou à une station donnée).
3. Il est généralement conseillé d'utiliser la première ligne du tableau Excel pour stocker le nom des variables et la première colonne pour stocker le nom des observations (identifiant des individus, des échantillons ou des stations par exemple).

4. Ne jamais utiliser de caractères spéciaux tels que #, \$, %, ^, &, \*, (,), {}, [, ], des accents, des cédilles des guillemets ou des apostrophes... Cela pourrait causer des erreurs dans R. Si votre fichier en contient, faites une recherche (*via* le menu Edition > Rechercher et remplacer ...) pour remplacer chaque instance par un caractère qui ne posera pas de problème.
5. Évitez les espaces dans vos noms de variables, d'observations ou de catégories et remplacez-les par des points ou des \_.
6. Si des noms de lignes sont présents dans votre tableau, chaque ligne doit avoir un nom unique (il ne faut pas que plusieurs lignes portent le même nom).
7. Des noms courts pour les variables sont généralement plus faciles à manipuler par la suite.
8. La première valeur de votre tableau devrait toujours se trouver dans la cellule A1 du tableur. Autrement dit, il ne devrait jamais y avoir de lignes incomplètes ou de lignes de commentaires au-dessus des données, ou de colonne vide à gauche de votre tableau. D'ailleurs, il ne devrait jamais y avoir de commentaires à droite ou en-dessous de vos données non plus.

### 5.3.2 Fichiers au format tableur (.xls ou .xlsx)

À titre d'exemple, téléchargez le fichier [dauphin.xls](#) et placez-le dans votre répertoire de travail. Ce jeu de données contient des résultats de dosage de différents métaux lourds (cadmium, cuivre et mercure) dans différents organes (foie et rein) de plusieurs dauphins communs *Delphinus delphis*. Les informations de taille, d'âge et de statut reproducteur sont également précisées. Ouvrez ce fichier dans un tableur. Vous constaterez que son format ne permet pas de l'importer tel quel dans R :

- Il contient des lignes vides inutiles au-dessus des données
- Il contient des commentaires inutiles au-dessus des données
- Les titres de colonnes sont complexes et contiennent des caractères spéciaux
- Dans le tableau, les données manquantes sont représentées soit par des “\*”, soit par des cellules vides

Importer un tel jeu de données dans R par les méthodes classiques (c'est-à-dire sans utiliser RStudio et uniquement grâce aux fonctions de base de R) demanderait donc un gros travail de mise en forme préalable. Heureusement, RStudio et le package `readxl` facilitent grandement le processus.

Dans RStudio, localisez l'onglet Files situé dans le panneau en bas à droite de l'interface du logiciel. Dans ce panneau, naviguez jusqu'à votre répertoire de travail, qui doit maintenant contenir le fichier `dauphin.xls` que vous avez téléchargé. Cliquez sur son nom, puis, dans le menu qui s'affiche, choisissez Import Dataset ... :

La nouvelle fenêtre qui s'ouvre est un “assistant d’importation” :

Cette fenêtre contient plusieurs zones importantes :

- I. File/URL (en haut) : lien vers le fichier contenant les données, sur votre ordinateur ou en ligne

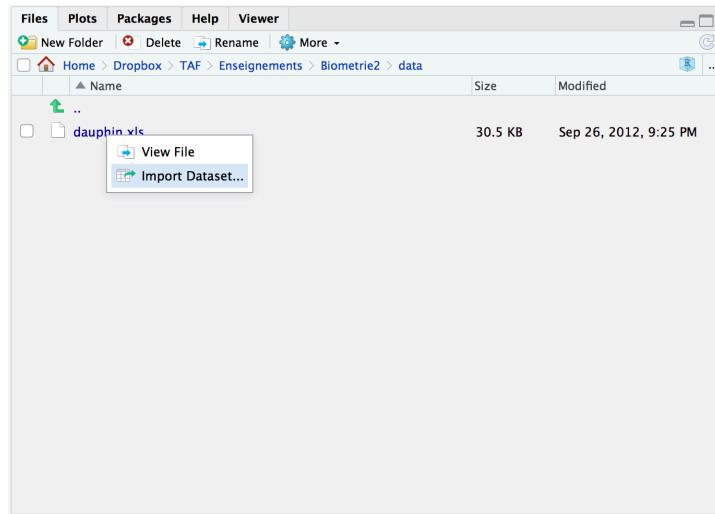


Figure 71 – L'option ‘Import Dataset...’ dans la fenêtre ‘Files’ de RStudio

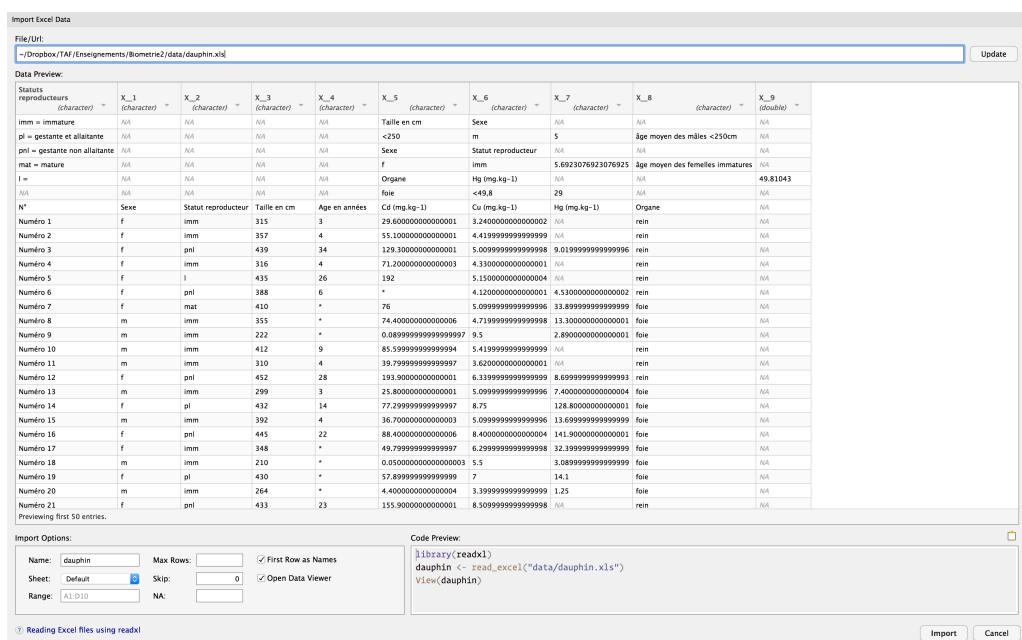


Figure 72 – L'assistant d'importation de RStudio

2. Data Preview : zone principale affichant les 50 premières lignes du fichier que l'on souhaite importer.
3. Import Options (en bas à gauche) : zone dans laquelle des options permettant d'importer les données correctement peuvent être spécifiées
4. Code Preview (en bas à droite) : les lignes de codes que vous pourrez copier-coller dans votre script une fois les réglages corrects effectués.

Ici, nous constatons que les données ne sont pas au bon format. La première chose que nous pouvons faire est d'indiquer à R que nous souhaitons ignorer les 9 premières lignes du fichier. Ensuite, nous précisons à RStudio que l'étoile "\*" a été utilisée pour indiquer des données manquantes :

The screenshot shows the 'Import Excel Data' dialog in RStudio. The 'Data Preview' section displays the first 50 entries of the 'dauphin.xls' file. The 'Import Options' section contains the following settings:

Name: dauphin	Max Rows:	<input checked="" type="checkbox"/> First Row as Names
Sheet: Default	Skip: 9	<input checked="" type="checkbox"/> Open Data Viewer
Range: A1:D10		

The 'Code Preview' section shows the R code used to import the data:

```
library(readxl)
dauphin <- read_excel("data/dauphin.xls",
na = "*", skip = 9)
View(dauphin)
```

Figure 73 – Les bons réglages pour ce fichier

Notez qu'à chaque fois que vous modifiez une valeur dans la zone Import Options, 2 choses se produisent simultanément :

1. La zone Data Preview est mise à jour. Cela permet de s'assurer que les changements effectués ont bien les effets escomptés
2. La zone Code Preview est mise à jour. Cela permet de copier-coller dans un script les commandes permettant d'importer correctement les données. Ici, voilà le code que nous devons ajouter à notre script :

```
dauphin <- read_excel("data/dauphin.xls", na = "*", skip = 9)
```

La commande `library(readxl)` est inutile puisque nous l'avons déjà saisie au début de ce chapitre. Nous disposons maintenant d'un nouvel objet nommé `dauphin`. Il est stocké sous la forme d'un tibble :

```
dauphin
```

```
# A tibble: 93 x 9
`N°`      Sexe `Statut reproducteu~ `Taille en cm` `Age en années`
<chr>     <chr> <chr>                      <dbl>          <dbl>
1 Numéro 1 f    imm                         315            3
2 Numéro 2 f    imm                         357            4
3 Numéro 3 f    pnl                         439            34
4 Numéro 4 f    imm                         316            4
5 Numéro 5 f    l                           435            26
6 Numéro 6 f    pnl                         388            6
7 Numéro 7 f    mat                         410            NA
8 Numéro 8 m    imm                         355            NA
9 Numéro 9 m    imm                         222            NA
10 Numéro 10 m   imm                        412            9
# ... with 83 more rows, and 4 more variables: Cd (mg.kg-1) <dbl>,
#   Cu (mg.kg-1) <dbl>, Hg (mg.kg-1) <dbl>, Organe <chr>
```

Notez toutefois que les noms de colonnes complexes sont toujours présents. Avec de tels noms, les variables ne seront pas faciles à manipuler et les risques d'erreurs de frappes seront nombreux. Nous avons tout intérêt à les modifier à l'aide de la fonction `names()` :

```
names(dauphin) <- c("ID", "Sexe", "Statut", "Taille", "Age", "Cd", "Cu", "Hg", "Organe")
dauphin
```

```
# A tibble: 93 x 9
ID      Sexe Statut Taille Age      Cd      Cu      Hg Organe
<chr>    <chr> <chr>   <dbl> <dbl>    <dbl>    <dbl>    <dbl> <chr>
1 Numéro 1 f    imm      315     3  29.6   3.24  NA     rein
2 Numéro 2 f    imm      357     4  55.1   4.42  NA     rein
3 Numéro 3 f    pnl      439     34 129.   5.01  9.02  rein
4 Numéro 4 f    imm      316     4  71.2   4.33  NA     rein
5 Numéro 5 f    l       435     26 192    5.15  NA     rein
6 Numéro 6 f    pnl      388     6  NA     4.12  4.53  rein
7 Numéro 7 f    mat      410     NA  76     5.1   33.9   foie
8 Numéro 8 m    imm      355     NA  74.4   4.72  13.3  foie
9 Numéro 9 m    imm      222     NA  0.09   9.5   2.89  foie
10 Numéro 10 m   imm     412     9  85.6   5.42  NA     rein
# ... with 83 more rows
```

Enfin, vous pouvez également noter que certaines variables devraient être modifiées :

- les variables `Sexe`, `Statut` (qui contient l'information de statut reproducteur des dauphins) et `Organe` (qui indique dans quel organe les métaux ont été dosés) sont de type `<chr>`. L'idéal serait de disposer de facteurs puisqu'ils s'agit de variables catégorielles.

- la variable ID est totalement inutile puisqu'elle est parfaitement redondante avec le numéro de ligne. Nous pourrions donc la supprimer.
- certaines catégories (ou niveaux) de la variable Statut devraient être ordonnées puisqu'elles reflètent une progression logique : imm (immature), mat (mature), pnl (pregnant non lactating), pl (pregnant lactating), l (lactating), repos (repos somatique)

Nous verrons dans la partie 6 comment effectuer simplement ces différentes opérations

### 5.3.3 Fichiers au format texte brut (.csv)

Nous allons utiliser les mêmes données que précédemment, mais cette fois-ci, elles sont contenues dans un fichier au format .csv. Téléchargez le fichier [dauphin.csv](#) (pour cela, faites un clic droit sur le lien et choisissez Enregistrez la cible du lien sous ... ou une mention équivalente), placez-le dans votre répertoire de travail, et ouvrez-le avec le bloc notes Windows ou tout autre éditeur de texte brut disponible sur votre ordinateur. **Attention :** Microsoft Word n'est pas un éditeur de texte brut. Un fichier au format .doc ou .docx est illisible dans un éditeur de texte brut car outre le texte, ces formats de documents contiennent toutes les informations concernant la mise en forme du texte (polices de caractères, tailles, couleurs et autres attributs, présence de figures, de tableaux dans le document, etc.).

À l'inverse, les fichiers au format .txt, .csv et même .R (vos scripts !) sont des fichiers au format texte brut. Vous pouvez d'ailleurs essayer d'ouvrir dauphin.csv depuis RStudio, en allant dans la fenêtre Files puis en cliquant sur le nom du fichier et en choisissant View File. RStudio ouvre un nouvel onglet à côté de votre script vous permettant d'inspecter le contenu de ce fichier. Par rapport au fichier excel, vous pouvez noter un certain nombre de différences :

1. les colonnes sont séparées par des tabulations
2. les nombres décimaux utilisent la virgule (et non le point comme dans les pays anglo-saxons)
3. les noms de colonne ont déjà été corrigés/simplifiés par rapport au tableau d'origine
4. les valeurs manquantes sont toutes codées par des NAs

Attention, à ce stade, vous avez ouvert un fichier au format texte brut dans RStudio, mais les données contenues dans ce fichier n'ont pas été importées dans R pour autant. Pour les importer, on procède comme pour les fichiers au format tableur (voir section 5.3.2 ci-dessus).

On commence par cliquer sur dauphin.csv dans l'onglet Files de RStudio. On sélectionne ensuite Import Dataset ... :

La fenêtre qui s'ouvre est en tous points identique à celle obtenue pour l'importation de fichiers tableurs :

Nous voyons ici que par défaut, RStudio considère qu'une unique colonne est présente. En effet, les fichiers .csv utilisent généralement la virgule pour séparer les colonnes. Ce n'est pas le cas ici. Il nous faut donc sélectionner, dans le champ Delimiter, l'option Tab (tabulation) et non Comma (virgule).

À ce stade, chaque variable est maintenant reconnue comme telle, chaque variable occupe donc une colonne distincte. Mais les colonnes Cd, Cu et Hg ne contiennent pas les bonnes valeurs (vous

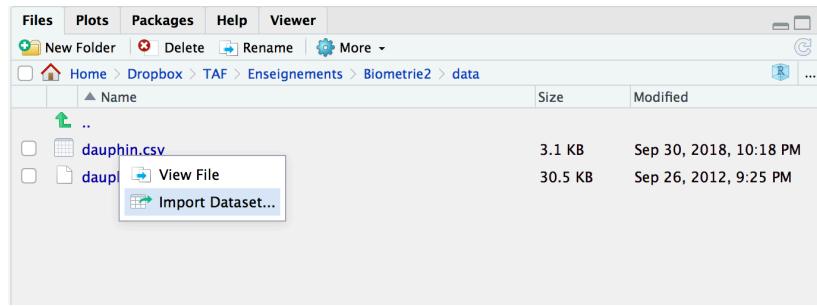


Figure 74 – Importer un fichier ‘.csv’ depuis l’onglet ‘Files’ de RStudio

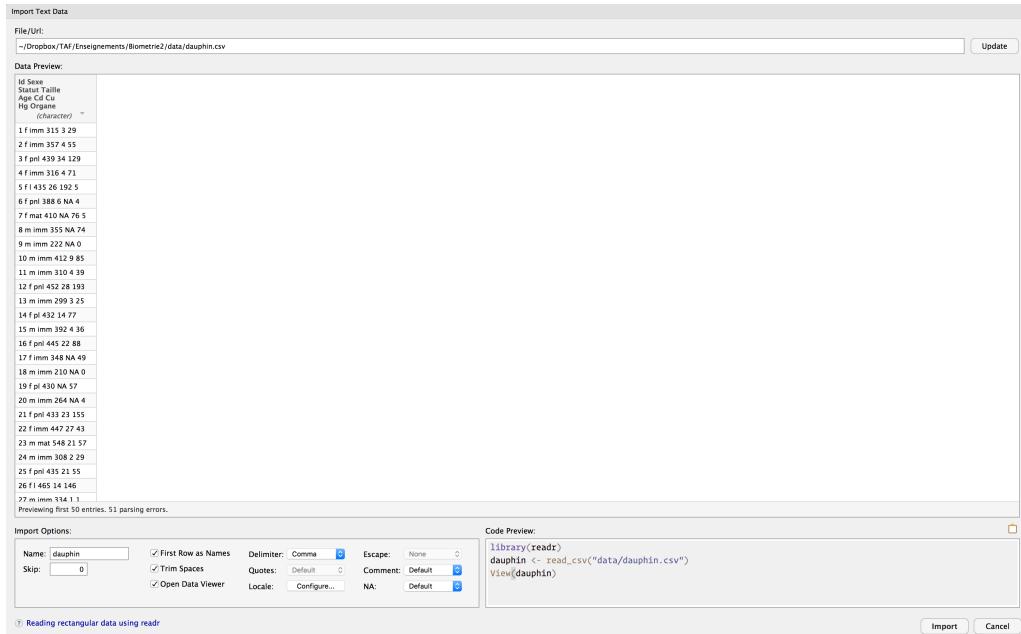


Figure 75 – Importer un fichier ‘.csv’ depuis l’onglet ‘Files’ de RStudio

pouvez le vérifier en consultant l'onglet dauphin.csv que vous avez ouvert un peu plus tôt à côté de votre script). La cause est simple : R s'attend à ce que les nombres décimaux utilisent le point en guise de symbole des décimales. Or, notre fichier .csv utilise la virgule. C'est une convention qui dépend du pays dans lequel vous vous trouvez, et de la langue de votre système d'exploitation (en langage technique, on parle de Locale). Le fichier dauphin.csv ayant été créé sur un ordinateur français, la virgule a été utilisée en guise de symbole des décimales. Pour l'indiquer à R, cliquez sur Locale > Configure ..., changez le . en , dans le champ Decimal Mark et validez en cliquant sur Configure :

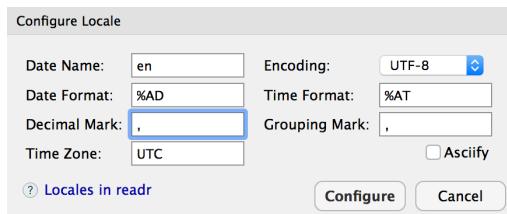


Figure 76 – Changement du symbole utilisé pour les décimales

Les données sont maintenant au bon format, prêtes à être importées dans RStudio. Afin de ne pas écraser l'objet dauphin que nous avons créé à partir du fichier tableau un peu plus tôt, nous stockerons ces nouvelles données dans un objet nommé dauphin2. Pour cela, ajoutez un 2 au nom dauphin dans le champ Name en bas à gauche :

Figure 77 – Les données, dans un format correct, permettant l'importation

Nous n'avons plus qu'à copier-coller dans notre script le code généré automatiquement en bas à droite de la fenêtre (comme précédemment, la ligne library(readr) est inutile : nous avons déjà chargé ce package en début de chapitre).

```

dauphin2 <- read_delim("data/dauphin.csv",
  "\t", escape_double = FALSE, locale = locale(decimal_mark = ","),
  trim_ws = TRUE)

Rows: 93 Columns: 9

-- Column specification -----
Delimiter: "\t"
chr (3): Sexe, Statut, Organe
dbl (6): Id, Taille, Age, Cd, Cu, Hg

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

```

Notez que :

1. c'est le package `readr` et non plus `readxl` qui est utilisé.
2. la fonction `read_delim()` a remplacé la fonction `read_excel()`. Il existe beaucoup d'autres fonctions selon le format de vos données (par exemple `read_csv()` et `read_csv2()`). Il est inutile de toutes les connaître dans la mesure où généralement, RStudio vous propose automatiquement la plus appropriée.
3. R indique de quelle façon les colonnes ont été "parsées", autrement dit, R indique quelles fonctions ont été utilisées pour reconnaître le type des données présentes dans chaque colonne.

Toutes les fonctions permettant d'importer des données n'ont pas nécessairement le même comportement. Ainsi, si l'on compare les objets importés depuis le fichier tableau (`dauphin`) et depuis le fichier texte brut (`dauphin2`), le type de certaines variables est différent :

```

dauphin

# A tibble: 93 x 9
  ID      Sexe Statut Taille   Age     Cd     Cu     Hg Organe
  <chr>    <chr> <chr>   <dbl> <dbl>   <dbl> <dbl> <dbl> <chr>
1 Numéro 1 f     imm     315     3 29.6   3.24 NA    rein
2 Numéro 2 f     imm     357     4 55.1   4.42 NA    rein
3 Numéro 3 f     pnl     439     34 129.   5.01  9.02 rein
4 Numéro 4 f     imm     316     4 71.2   4.33 NA    rein
5 Numéro 5 f     l      435     26 192    5.15 NA    rein
6 Numéro 6 f     pnl     388     6 NA     4.12  4.53 rein
7 Numéro 7 f     mat     410     NA 76     5.1   33.9  foie
8 Numéro 8 m     imm     355     NA 74.4   4.72  13.3 foie
9 Numéro 9 m     imm     222     NA 0.09   9.5   2.89 foie
10 Numéro 10 m    imm    412     9 85.6   5.42 NA    rein
# ... with 83 more rows

```

```
dauphin2
```

```
# A tibble: 93 x 9
  Id Sexe Statut Taille Age Cd Cu Hg Organe
  <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
1 1 f imm     315    3 29.6 3.24 NA   rein
2 2 f imm     357    4 55.1 4.42 NA   rein
3 3 f pnl     439    34 129. 5.01 9.02 rein
4 4 f imm     316    4 71.2 4.33 NA   rein
5 5 f l       435    26 192  5.15 NA   rein
6 6 f pnl     388    6 NA   4.12 4.53 rein
7 7 f mat     410    NA 76   5.1  33.9 foie
8 8 m imm     355    NA 74.4 4.72 13.3 foie
9 9 m imm     222    NA 0.09 9.5  2.89 foie
10 10 m imm    412    9 85.6 5.42 NA   rein
# ... with 83 more rows
```

En particulier les variables Taille et Age sont considérées comme réelles dans dauphin mais comme entières (ce qui semble plus logique) dans dauphin2. Afin d'éviter les confusions dans la suite du document, nous allons supprimer dauphin2 en tapant :

```
rm(dauphin2)
```

Taper dauphin2 dans la console devrait maintenant produire une erreur :

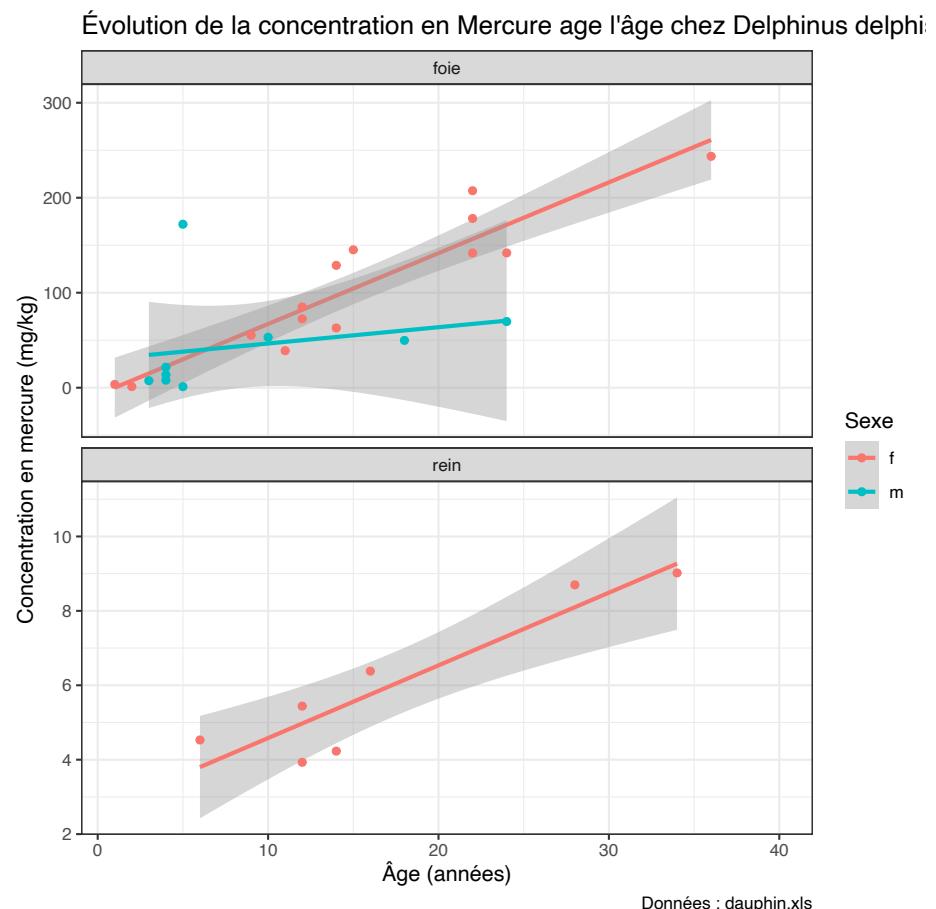
```
dauphin2
```

```
Error in eval(expr, envir, enclos): object 'dauphin2' not found
```

### 5.3.4 Exercices

1. L'objet dauphin est-il “tidy” (autrement dit, s'agit-il de “données rangées”)? Justifiez.
2. Produisez le graphique ci-dessous :

```
`geom_smooth()` using formula 'y ~ x'
```



Indice : les droites de régression avec les intervalles de confiance sont ajoutés grâce à la fonction `geom_smooth(method = "lm")`

- Importez dans R le jeu de données [whoTB.csv](#). Ce jeu de données contient les cas de tuberculose (TB) rapportés par l'Organisation Mondiale de la Santé (OMS, ou WHO en anglais : World Health Organization). Les cas sont répertoriés par année, pays, âge, sexe, type de tuberculose et méthode de diagnostique. Selon vous, ce jeu de données est-il “rangé”? Pourquoi?
- Si ce jeu de données n'est pas rangé, rangez-le en utilisant les fonctions du package `tidyverse` que nous avons découvertes dans ce chapitre : `pivot_longer()`, `pivot_wider()`, `separate()` et `unite()` (vous n'aurez pas nécessairement besoin d'utiliser ces 4 fonctions, et à l'inverse, certaines devront peut-être être utilisées plusieurs fois).

Pour vous aider, l'OMS donne la signification des codes utilisés en guise de noms pour la plupart des colonnes. Ainsi :

- new indique des nouveaux cas, old des anciens (ici, seuls les nouveaux cas sont rapportés)
- le type de cas est précisé ensuite :
  - sp signifie “Smear Positive” (tuberculose pulmonaire à frottis positif)
  - sn signifie “Smear Negative” (tuberculose pulmonaire à frottis négatif)
  - rel signifie “relapse” (rechute)

- ep signifie “Extra Pulmonary” (tuberculose extra-pulmonaire)
- le sexe est codé par m (male) ou f (female)
- enfin, les chiffres correspondent à des tranches d’âges : 014 signifie “de 0 à 14 ans”, “1524” signifie “de 15 à 24 ans”, etc.

Dans ces colonnes aux noms composés, les nombres de cas de tuberculose sont rapportés.

## 6 Tripatouiller les données avec dplyr

### 6.1 Pré-requis

Nous abordons ici une étape essentielle de toute analyse de données : la manipulation de tableaux, la sélection de lignes, de colonnes, la création de nouvelles variables, etc. Bien souvent, les données brutes que nous importons dans R ne sont pas utiles en l’état. Il nous faut parfois sélectionner seulement certaines lignes pour travailler sur une petite partie du jeu de données. Il nous faut parfois modifier des variables existantes (pour modifier les unités par exemple) ou en créer de nouvelles à partir des variables existantes. Nous avons aussi très souvent besoin de constituer des groupes et d’obtenir des statistiques descriptives pour chaque groupe (moyenne, écart-type, erreur type, etc). Nous verrons dans ce chapitre comment faire tout cela grâce au package dplyr qui fournit un cadre cohérent et des fonctions simples permettant d’effectuer tous les tripatouillages de données dont nous pourrons avoir besoin.

Dans ce chapitre, nous aurons besoin des packages suivants :

```
library(dplyr)
library(ggplot2)
library(nycflights13)
```

---

### 6.2 Le pipe %>%

Avant d’entrer dans le vif du sujet, je souhaite introduire ici la notion de “pipe” (prononcer à l’anglo-saxonne). Le pipe est un opérateur que nous avons déjà vu apparaître à plusieurs reprises dans les chapitres précédents sans expliquer son fonctionnement.

Le pipe, noté %>%, peut être obtenu en pressant les touches **ctrl + shift + M** de votre clavier. Il permet d’enchaîner logiquement des actions les unes à la suite des autres. Globalement, le pipe prend l’objet situé à sa gauche, et le transmet à la fonction situé à sa droite. En d’autres termes, les 2 expressions suivantes sont strictement équivalentes :

```
# Ici, "f" est une fonction quelconque, "x" et "y" sont 2 objets dont la fonction a besoin.

# Il s'agit d'un exemple fictif : ne tapez pas ceci dans votre script !
f(x, y)
x %>% f(y)
```

Travailler avec le pipe est très intéressant car toutes les fonctions de `dplyr` que nous allons décrire ensuite sont construites autour de la même syntaxe : on leur fournit un `data.frame` (ou encore mieux, un `tibble`), elles effectuent une opération et renvoient un nouveau `data.frame` (ou un nouveau `tibble`). Il est ainsi possible de créer des groupes de commandes cohérentes qui permettent, grâce à l'enchaînement d'étapes simples, d'aboutir à des résultats complexes.

De la même façon que le `+` permet d'ajouter une couche supplémentaire à un graphique `ggplot2`, le pipe `%>%` permet d'ajouter une opération supplémentaire dans un groupe de commandes.

Pour reprendre un exemple de la section 4.3 sur les nuages de points, nous avions commencé par créer un objet nommé `alaska_flights` à partir de l'objet `flights` :

```
alaska_flights <- flights %>%  
  filter(carrier = "AS")
```

Nous avions ensuite créé notre premier nuage de points avec ce code :

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```

Nous savons maintenant qu'il n'est pas indispensable de faire figurer le nom des arguments `data` = et `mapping` =. Mais nous pouvons aller plus loin. En fait, il n'était même pas nécessaire de créer l'objet `alaska_flights` : nous aurions pu utiliser le pipe pour enchaîner les étapes suivantes :

1. On prend le tableau `flights`, *puis*
2. On filtre les données pour ne retenir que la compagnie aérienne AS, *puis*
3. On réalise le graphique

Voilà comment traduire cela avec le pipe :

```
flights %>%  
  filter(carrier = "AS") %>%  
  ggplot(aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```

Notez bien qu'ici, aucun objet intermédiaire n'a été créé. Notez également que le premier argument de la fonction `ggplot()` a disparu : le pipe a fourni automatiquement à `ggplot()` les données générées au préalable (les données `flights` filtrées grâce à la fonction `filter()`).

Comme pour le `+` de `ggplot2`, il est conseillé de placer un seul pipe par ligne, de le placer en fin de ligne et de revenir à la ligne pour préciser l'étape suivante.

Toutes les commandes que nous utiliserons à partir de maintenant reposeront sur le pipe puisqu'il permet de rendre le code plus lisible.

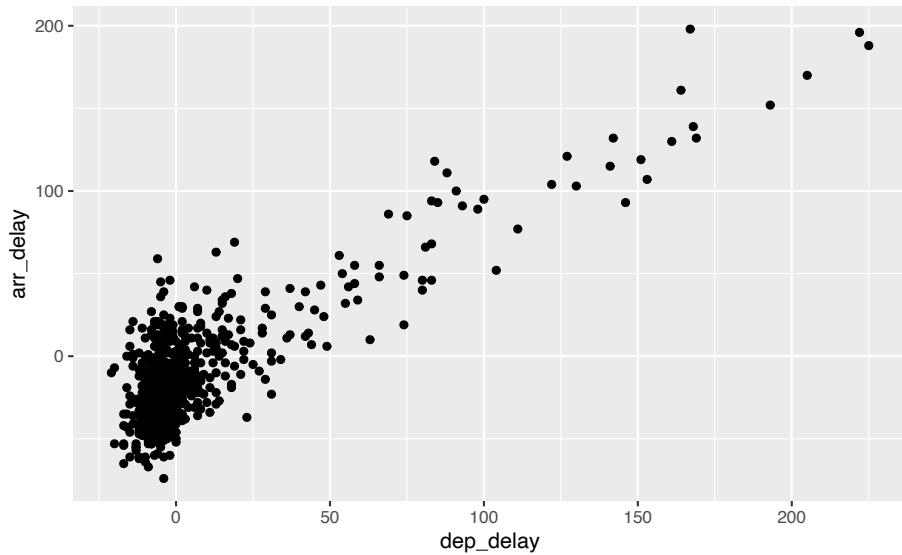


Figure 78 – Notre premier graphique, produit grâce au pipe

### 6.3 Les verbes du tripatouillage de données

Nous allons ici nous concentrer sur les fonctions les plus couramment utilisées pour manipuler et résumer des données. Nous verrons 6 verbes principaux, chacun correspondant à une fonction précise de `dplyr`. Chaque section de ce chapitre sera consacrée à la présentation d'un exemple utilisant un ou plusieurs de ces verbes.

Les 6 verbes sont :

1. `filter()` : choisir des lignes dans un tableau à partir de conditions spécifiques (filtrer).
2. `arrange()` : trier les lignes d'un tableau selon un ou plusieurs critères (arranger).
3. `select()` : sélectionner des colonnes d'un tableau.
4. `mutate()` : créer de nouvelles variables en transformant et combinant des variables existantes (muter).
5. `summarise()` : calculer des résumés statistiques des données (résumer). Souvent utilisé en combinaison avec `group_by()`, qui permet de constituer des groupes au sein des données.
6. `join()` : associer, fusionner 2 `data.frames` en faisant correspondre les éléments d'une colonne commune entre les 2 tableaux (joindre). Il y a de nombreuses façons de joindre des tableaux. Nous nous contenterons d'examiner les fonctions `left_join()` et `inner_join()`.

Toutes ces fonctions, tous ces verbes, sont utilisés de la même façon : on prend un `data.frame`, grâce au pipe, on le transmet à l'une de ces fonctions dont on précise les arguments entre parenthèses, la fonction nous renvoie un nouveau tableau modifié. Évidemment, on peut enchaîner les actions pour modifier plusieurs fois le même tableau, c'est tout l'intérêt du pipe.

Enfin, gardez en tête qu'il existe beaucoup plus de fonctions dans `dplyr` que les 6 que nous allons détailler ici. Nous verrons parfois quelques variantes, mais globalement, maîtriser ces 6

fonctions simples devrait vous permettre de conduire une très large gamme de manipulations de données, et ainsi vous faciliter la vie pour la production de graphiques et l'analyse statistique de vos données.

---

## 6.4 Filtrer des lignes avec filter()

### 6.4.1 Principe

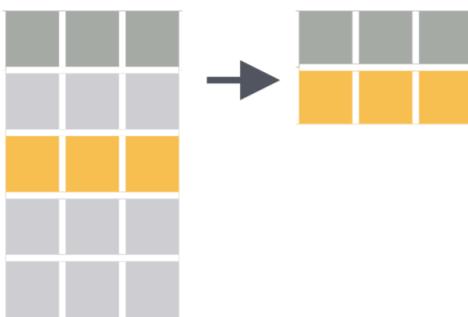


Figure 79 – Schéma de la fonction ‘filter()’ tiré de la ’cheatsheet’ de ‘dplyr’ et ‘tidy’

Comme son nom l'indique, `filter()` permet de filtrer des lignes en spécifiant un ou des critères de tri portant sur une ou plusieurs variables. Nous avons déjà utilisé cette fonction à plusieurs reprises pour créer les jeux de données `alaska_flights` et `small_weather` :

```
alaska_flights <- flights %>%  
  filter(carrier = "AS")  
  
small_weather <- weather %>%  
  filter(origin = "EWR",  
         month = 1,  
         day <= 15)
```

Dans les 2 cas, la première ligne de code nous permet :

1. d'indiquer le nom du nouvel objet dans lequel les données modifiées seront stockées (`alaska_flights` et `small_weather`)
2. d'indiquer de quel objet les données doivent être extraites (`flights` et `weather`)
3. de passer cet objet à la fonction suivante avec un pipe `%>%`

Le premier argument de la fonction `filter()` doit être le nom d'un `data.frame` ou d'un `tibble`. Ici, puisque nous utilisons le pipe, il est inutile de spécifier cet argument : c'est ce qui est placé à gauche du pipe qui est utilisé comme premier argument de la fonction `filter()`. Les arguments suivants constituent la ou les conditions qui doivent être respectées par les lignes du tableau de départ afin d'être intégrées au nouveau tableau de données.

### 6.4.2 Exercice

Dans la section 3.3.1, nous avons utilisé la fonction View et l'application manuelle de filtres pour déterminer combien de vols avaient quitté l'aéroport JFK le 12 février 2013. En utilisant la fonction filter( ), créez un objet nommé JFK\_12fev qui contiendra les données de ces vols.

Vérifiez que cet objet contient bien 282 lignes.

### 6.4.3 Les conditions logiques

Dans la section 2.2.4.2, nous avons présenté en détail le fonctionnement des opérateurs de comparaison dans R. Relisez cette section si vous ne savez plus de quoi il s'agit. Les opérateurs de comparaison permettent de vérifier l'égalité ou l'inégalité entre des éléments. Ils renvoient TRUE ou FALSE et seront particulièrement utiles pour filtrer des lignes dans un tableau. Comme indiqué dans la section 2.2.4.2, voici la liste des opérateurs de comparaison usuels :

- = : égal à
- ≠ : différent de
- > : supérieur à
- < : inférieur à
- ≥ : supérieur ou égal à
- ≤ : inférieur ou égal à

À cette liste, nous pouvons ajouter quelques éléments utiles :

- is.na() : renvoie TRUE en cas de données manquantes.
- ! : permet de tester le contraire d'une expression logique. Par exemple !is.na() renvoie TRUE s'il n'y a pas de données manquantes.
- %in% : permet de tester si l'élément de gauche est contenu dans la série d'éléments fournie à droite. Par exemple 2 %in% 1:5 renvoie TRUE, mais 2 %in% 5:10 renvoie FALSE.
- | : opérateur logique OU. Permet de tester qu'une condition OU une autre est remplie.
- & : opérateur logique ET. Permet de tester qu'une condition ET une autre sont remplies.

Voyons comment utiliser ces opérateurs avec la fonction filter().

Dans le tableau flights, tous les vols prévus ont-ils effectivement décollé ? Une bonne façon de le savoir est de regarder si, pour la variable dep\_time (heure de décollage), des données manquantes sont présentes :

```
flights %>%  
  filter(is.na(dep_time))
```

```
# A tibble: 8,255 x 19  
  year month   day dep_time sched_dep_time dep_delay arr_time  
  <int> <int> <int>    <int>          <int>     <dbl>    <int>  
1 2013     1     1        NA            1630       NA      NA  
2 2013     1     1        NA            1935       NA      NA
```

```

3 2013    1    1    NA      1500    NA    NA
4 2013    1    1    NA       600    NA    NA
5 2013    1    2    NA      1540    NA    NA
6 2013    1    2    NA      1620    NA    NA
7 2013    1    2    NA      1355    NA    NA
8 2013    1    2    NA      1420    NA    NA
9 2013    1    2    NA      1321    NA    NA
10 2013   1    2    NA      1545    NA    NA
# ... with 8,245 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

Seules les lignes contenant NA dans la colonne dep\_time sont retenues. Il y a donc 8255 vols qui n'ont finalement pas décollé.

Dans le même ordre d'idée, y a t-il des vols qui ont décollé mais qui ne sont pas arrivés à destination ? Là encore, une façon d'obtenir cette information est de sélectionner les vols qui ont décollé (donc pour lesquels l'heure de décollage n'est pas manquante), mais pour lesquels l'heure d'atterrissement est manquante :

```

flights %>%
  filter(!is.na(dep_time),
        is.na(arr_time))

# A tibble: 458 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013    1     1      2016        1930      46    NA
2 2013    1     2      2041        2045     -4    NA
3 2013    1     2      2145        2129      16    NA
4 2013    1     9       615        615       0    NA
5 2013    1     9      2042        2040      2    NA
6 2013    1    11      1344        1350     -6    NA
7 2013    1    13      1907        1634     153   NA
8 2013    1    13      2239        2159      40    NA
9 2013    1    16       837        840      -3    NA
10 2013   1    25      1452        1500     -8    NA
# ... with 448 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

Notez l'utilisation du ! pour la première condition. Nous récupérons ici les lignes pour lesquelles

`dep_time` n'est pas NA et pour lesquelles `arr_time` est NA. Seules les lignes qui respectent cette double condition sont retenues. Cette syntaxe est équivalente à :

```
flights %>%
  filter(!is.na(dep_time) & is.na(arr_time))

# A tibble: 458 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     1     1      2016          1930       46     NA
2 2013     1     2      2041          2045      -4     NA
3 2013     1     2      2145          2129       16     NA
4 2013     1     9       615          615        0     NA
5 2013     1     9      2042          2040       2     NA
6 2013     1    11      1344          1350      -6     NA
7 2013     1    13      1907          1634      153     NA
8 2013     1    13      2239          2159       40     NA
9 2013     1    16       837          840       -3     NA
10 2013    1    25      1452          1500      -8     NA
# ... with 448 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

Dans la fonction `filter()`, séparer plusieurs conditions par des virgules signifie que seules les lignes qui remplissent toutes les conditions seront retenues. C'est donc l'équivalent du ET logique.

Il y a donc 458 vols qui ne sont pas arrivés à destination (soit moins de 0,2% des vols au départ de New York en 2013). Selon vous, quelles raisons peuvent expliquer qu'un vol qui a décollé n'ait pas d'heure d'atterrissement ?

Enfin, pour illustrer l'utilisation de `|` (le OU logique) et de `%in%`, imaginons que nous souhaitions extraire les informations des vols ayant quitté l'aéroport JFK à destination d'Atlanta, Géorgie (ATL) et de Seattle, Washington (SEA), aux mois d'octobre, novembre et décembre :

```
atl_sea_fall <- flights %>%
  filter(origin == "JFK",
         dest == "ATL" | dest == "SEA",
         month > 10)
atl_sea_fall

# A tibble: 962 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
```

```

1 2013 10 1 638 640 -2 839
2 2013 10 1 729 735 -6 1049
3 2013 10 1 824 830 -6 1030
4 2013 10 1 853 900 -7 1217
5 2013 10 1 1328 1330 -2 1543
6 2013 10 1 1459 1500 -1 1817
7 2013 10 1 1544 1545 -1 1815
8 2013 10 1 1754 1800 -6 2102
9 2013 10 1 1825 1830 -5 2159
10 2013 10 1 1841 1840 1 2058

# ... with 952 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

Examinez ce tableau avec `View()` pour vérifier que la variable `dest` contient bien uniquement les codes ATL et SEA correspondant aux 2 aéroports qui nous intéressent. Nous avons extrait ici les vols à destination d'Atlanta **et** Seattle, pourtant, il nous a fallu utiliser le OU logique. Car chaque vol n'a qu'une unique destination, or nous souhaitons récupérer toutes les lignes pour lesquelles la destination est soit ATL, soit SEA (l'une **ou** l'autre).

Une autre solution pour obtenir le même tableau est de remplacer l'expression contenant `|` par une expression contenant `%in%`:

```

atl_sea_fall2 <- flights %>%
  filter(origin == "JFK",
        dest %in% c("ATL", "SEA"),
        month >= 10)
atl_sea_fall2

```

```

# A tibble: 962 x 19
  year month day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>      <dbl>    <int>
1 2013    10     1       638           640        -2     839
2 2013    10     1       729           735        -6    1049
3 2013    10     1       824           830        -6    1030
4 2013    10     1       853           900        -7    1217
5 2013    10     1      1328          1330        -2    1543
6 2013    10     1      1459          1500        -1    1817
7 2013    10     1      1544          1545        -1    1815
8 2013    10     1      1754          1800        -6    2102
9 2013    10     1      1825          1830        -5    2159
10 2013   10     1      1841          1840        1    2058

# ... with 952 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,

```

```
# flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
# air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
# time_hour <dttm>
```

Ici, toutes les lignes du tableau dont la variable dest est égale à un élément du vecteur `c("ATL", "SEA")` sont retenues. L'utilisation du OU logique eput être source d'erreur. Je préfère donc utiliser `%in%` qui me semble plus parlant. La fonction `identical()` nous confirme que les deux façons de faire produisent exactement le même résultat, libre à vous de rpivilégier la méthode qui vous convient le mieux :

```
identical(atl_sea_fall, atl_sea_fall2)
```

```
[1] TRUE
```

---

## 6.5 Créer des résumés avec `summarise()` et `group_by()`

### 6.5.1 Principe de la fonction `summarise()`

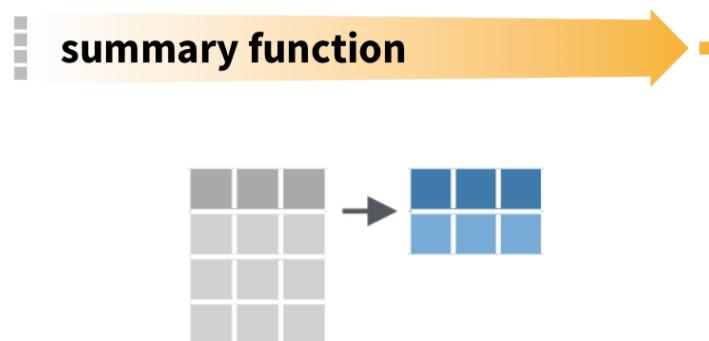


Figure 80 – Schéma de la fonction ‘summarise()’ tiré de la ‘cheatsheet’ de ‘dplyr’ et ‘tidyverse’

La figure 80 ci-dessus indique comment fonctionne la fonction `summarise()` : elle prend plusieurs valeurs (potentiellement, un très grand nombre) et les réduit à une unique valeur qui les résume. Lorsque l'on applique cette démarche à plusieurs colonnes d'un tableau, on obtient un tableau qui ne contient plus qu'une unique ligne de résumé.

La valeur qui résume les données est choisie par l'utilisateur. Il peut s'agir par exemple d'un calcul moyenne ou de variance, il peut s'agir de calculer une somme, ou d'extraire la valeur maximale ou minimale, ou encore, il peut tout simplement s'agir de déterminer un nombre d'observations.

Ainsi, pour connaître la température moyenne et l'écart-type des températures dans les aéroports de New York, il suffit d'utiliser le tableau `weather` et sa variable `temp` que nous avons déjà utilisés dans les chapitres précédents :

```
weather %>%  
  summarise(moyenne = mean(temp),  
            ecart_type = sd(temp))
```

```
# A tibble: 1 x 2  
moyenne ecart_type  
<dbl>      <dbl>  
1       NA        NA
```

Les fonctions `mean()` et `sd()` permettent de calculer une moyenne et un écart-type respectivement. Ici, les valeurs rentrées sont NA car une valeur de température est manquante :

```
weather %>%  
  filter(is.na(temp))
```

```
# A tibble: 1 x 15  
origin year month day hour temp dewp humid wind_dir wind_speed  
<chr>  <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>  
1 EWR     2013     8    22     9     NA     NA     NA      320      12.7  
# ... with 5 more variables: wind_gust <dbl>, precip <dbl>,  
#   pressure <dbl>, visib <dbl>, time_hour <dttm>
```

Pour obtenir les valeurs souhaitées, il faut indiquer à R d'exclure les valeurs manquantes lors des calculs de moyenne et écart-types :

```
weather %>%  
  summarise(moyenne = mean(temp, na.rm = TRUE),  
            ecart_type = sd(temp, na.rm = TRUE))
```

```
# A tibble: 1 x 2  
moyenne ecart_type  
<dbl>      <dbl>  
1      55.3      17.8
```

La température moyenne est donc de 55.3 degrés Farenheit et l'écart-type vaut 17.8 degrés Farenheit.

### 6.5.2 Intérêt de la fonction `group_by()`

La fonction devient particulièrement puissante lorsqu'elle est combinée avec la fonction `group_by()` :

Comme son nom l'indique, la fonction `group_by()` permet de créer des sous-groupes dans un tableau, afin que le résumé des données soit calculé pour chacun des sous-groupes plutôt que



Figure 81 – Fonctionnement de `group_by()` travaillant de concert avec `summarise()`, tiré de la ‘cheatsheet’ de `dplyr` et `tidyR`.

sur l'ensemble du tableau. En ce sens, son fonctionnement est analogue à celui des facets de `ggplot2` qui permettent de scinder les données d'un graphique en plusieurs sous-groupes.

Pour revenir à l'exemple des températures, imaginons que nous souhaitons calculer les températures moyennes et les écarts-types pour chaque mois de l'année. Voilà comment procéder :

```
weather %>%
  group_by(month) %>%
  summarise(moyenne = mean(temp, na.rm = TRUE),
            ecart_type = sd(temp, na.rm = TRUE))
```

```
# A tibble: 12 x 3
  month moyenne ecart_type
  <int>   <dbl>     <dbl>
1     1     35.6      10.2
2     2     34.3      6.98
3     3     39.9      6.25
4     4     51.7      8.79
5     5     61.8      9.68
6     6     72.2      7.55
7     7     80.1      7.12
8     8     74.5      5.19
9     9     67.4      8.47
10    10    60.1      8.85
11    11    45.0      10.4
12    12    38.4      9.98
```

Ici, les étapes sont les suivantes :

1. On prend le tableau `weather`, puis
2. On groupe les données selon la variable `month`, puis

### 3. On résume les données groupées sous la forme de moyennes et d'écart-types

Nous pouvons aller plus loin. Ajoutons à ce résumé 2 variables supplémentaires : le nombre de mesures et l'**erreur standard** (notée  $se$ ), qui peut être calculée de la façon suivante :

$$se \approx \frac{s}{\sqrt{n}}$$

avec  $s$ , l'écart-type de l'échantillon et  $n$ , la taille de l'échantillon. Cette grandeur est très importante en statistique puisqu'elle nous permet de quantifier l'**imprécision** de la moyenne. Elle intervient d'ailleurs dans le calcul de l'intervalle de confiance de la moyenne d'un échantillon. Nous allons donc calculer ici ces résumés, et nous donnerons un nom au tableau créé pour pouvoir ré-utiliser ces statistiques descriptives :

```
monthly_temp <- weather %>%
  group_by(month) %>%
  summarise(moyenne = mean(temp, na.rm = TRUE),
            ecart_type = sd(temp, na.rm = TRUE),
            nb_obs = n(),
            erreur_std = ecart_type / sqrt(nb_obs))
monthly_temp
```

```
# A tibble: 12 x 5
  month moyenne ecart_type nb_obs erreur_std
  <int>   <dbl>     <dbl>   <int>      <dbl>
1     1    35.6     10.2    2226      0.217
2     2    34.3      6.98   2010      0.156
3     3    39.9      6.25   2227      0.132
4     4    51.7      8.79   2159      0.189
5     5    61.8      9.68   2232      0.205
6     6    72.2      7.55   2160      0.162
7     7    80.1      7.12   2228      0.151
8     8    74.5      5.19   2217      0.110
9     9    67.4      8.47   2159      0.182
10   10    60.1      8.85   2212      0.188
11   11    45.0      10.4   2141      0.226
12   12    38.4      9.98   2144      0.216
```

Vous constatez ici que nous avons 4 statistiques descriptives pour chaque mois de l'année. Deux choses sont importantes à retenir ici :

1. on peut obtenir le nombre d'observations dans chaque sous-groupe d'un tableau groupé en utilisant la fonction `n()`. Cette fonction n'a besoin d'aucun argument : elle détermine automatiquement la taille des groupes créés par `group_by()`.
2. on peut créer de nouvelles variables en utilisant le nom de variables créées auparavant. Ainsi, nous avons créé la variable `erreur_std` en utilisant deux variables créées au préalable : `ecart-type` et `nb_obs`

### 6.5.3 Ajouter des barres d'erreurs sur un graphique

Ce jeu de données contient donc les données nécessaires pour nous permettre de visualiser sur un graphique l'évolution des températures moyennes enregistrées dans les 3 aéroports de New York en 2013. Outre les température moyennes, nous devons faire figurer l'imprécision des estimations de moyenne avec des barres d'erreur (à l'aide de la fonction `geom_linerange()`). Comme expliqué plus haut, l'imprécision des moyennes calculées est estimée grâce à l'erreur standard. Toutefois, ici, les imprécisions sont tellement faibles que les barres d'erreurs resteront invisibles :

```
monthly_temp %>%
  ggplot(aes(x = factor(month), y = moyenne, group = 1)) +
  geom_line() +
  geom_point() +
  geom_linerange(aes(ymin = moyenne - erreur_std,
                      ymax = moyenne + erreur_std),
                 color = "red")
```

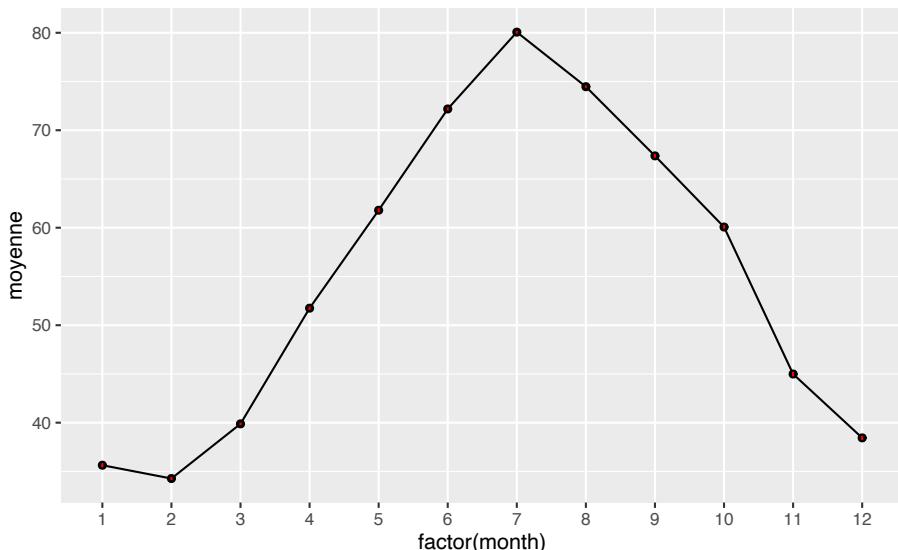


Figure 82 – Évolution des températures moyenne dans 3 aéroports de New York en 2013

Vous remarquerez que :

1. j'associe `factor(month)`, et non simplement `month`, à l'axe des x afin d'avoir, sur l'axe des abscisses, des chiffres cohérents allant de 1 à 12, et non des chiffres à virgule
2. l'argument `group = 1` doit être ajouté pour que la ligne reliant les points apparaisse. En effet, les lignes sont censées relier des points qui appartiennent à une même série temporelle. Or ici, nous avons transformé `month` en facteur. Préciser `group = 1` permet d'indiquer à `geom_line()` que toutes les catégories du facteur `month` appartiennent au même groupe, que ce facteur peut être considéré comme une variable continue, et qu'il est donc correct de relier les points.

3. la fonction `geom_linerange()` contient de nouvelles caractéristiques esthétiques qu'il nous faut obligatoirement renseigner : les extrémités inférieures et supérieures des barres d'erreur. Il nous faut donc associer 2 variables à ces caractéristiques esthétiques. Ici, nous utilisons `moyenne - erreur_std` pour la borne inférieure des barres d'erreur, et `moyenne + erreur_std` pour la borne supérieure. Les variables `moyenne` et `erreur_std` faisant partie du tableau `monthly_temp`, `geom_linerange()` les trouve sans difficulté.
4. les barres d'erreur produites sont minuscules. Je les ai fait apparaître en rouge afin de les rendre visibles, mais même comme cela, il faut zoomer fortement pour les distinguer. Afin de rendre l'utilisation de `geom_linerange()` plus explicite, je produis ci-dessous un autre graphique en remplaçant les erreurs standard par les écarts-types en guise de barres d'erreur. Attention, ce n'est pas correct d'un point de vue statistique ! Les barres d'erreur doivent permettre de visualiser l'imprécision de la moyenne. C'est donc bien les erreurs standard qu'il faut faire figurer en guise de barres d'erreurs et non les écarts-types. Le graphique ci-dessous ne figure donc qu'à titre d'exemple, afin d'illustrer de façon plus parlante le fonctionnement de la fonction `geom_linerange()` :

```
monthly_temp %>%
  ggplot(aes(x = factor(month), y = moyenne, group = 1)) +
  geom_line() +
  geom_point() +
  geom_linerange(aes(ymin = moyenne - ecart_type, ymax = moyenne + ecart_type)) +
  labs(x = "Mois",
       y = "Température (°Farenheit)",
       title = "Évolution des températures
dans 3 aéroports de New York en 2013",
       subtitle = "Attention : les barres d'erreurs sont les écarts-types.\nIl faut normalement faire figurer les erreurs standard.")
```

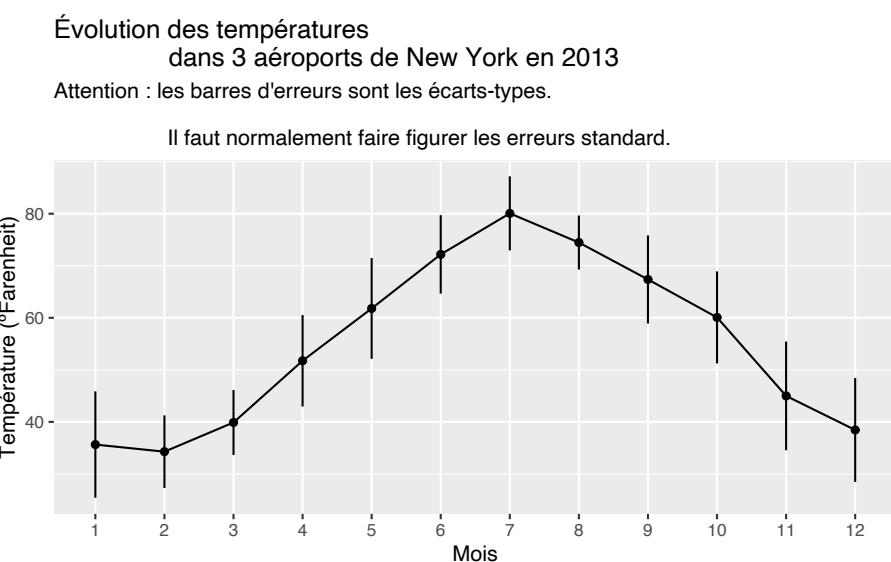


Figure 83 – Évolution des températures moyenne dans 3 aéroports de New York en 2013

#### 6.5.4 Grouper par plus d'une variable

Jusqu'ici, nous avons groupé les données de température par mois. Il est tout à fait possible de grouper les données par plus d'une variable, par exemple, par mois et par aéroport d'origine :

```
monthly_orig_temp <- weather %>%
  group_by(origin, month) %>%
  summarise(moyenne = mean(temp, na.rm = TRUE),
            ecart_type = sd(temp, na.rm = TRUE),
            nb_obs = n(),
            erreur_std = ecart_type / sqrt(nb_obs))
```

`summarise()` has grouped output by 'origin'. You can override using the `groups` argument.

```
monthly_orig_temp
```

```
# A tibble: 36 x 6
# Groups:   origin [3]
  origin month moyenne ecart_type nb_obs erreur_std
  <chr>   <int>   <dbl>      <dbl>   <int>      <dbl>
1 EWR       1     35.6      10.8     742      0.396
2 EWR       2     34.3      7.28     669      0.282
3 EWR       3     40.1      6.72     743      0.247
4 EWR       4     53.0      9.60     720      0.358
5 EWR       5     63.3      10.6     744      0.389
6 EWR       6     73.3      8.05     720      0.300
7 EWR       7     80.7      7.37     741      0.271
8 EWR       8     74.5      5.87     740      0.216
9 EWR       9     67.3      9.32     719      0.348
10 EWR      10    59.8      9.79     736      0.361
# ... with 26 more rows
```

En plus de la variable `month`, la tableau `monthly_orig_temp` contient une variable `origin`. Les statistiques que nous avons calculées plus tôt sont maintenant disponibles pour chaque mois et chacun des 3 aéroports de New York. Nous pouvons utiliser ces données pour comparer les 3 aéroports :

```
monthly_orig_temp %>%
  ggplot(aes(x = factor(month), y = moyenne, group = origin, color = origin)) +
  geom_line() +
  geom_point() +
  geom_linerange(aes(ymin = moyenne - ecart_type, ymax = moyenne + ecart_type)) +
  labs(x = "Mois",
       y = "Température (°Farenheit)",
       title = "Évolution des températures")
```

```

dans 3 aéroports de New York en 2013",
subtitle = "Attention : les barres d'erreurs sont les écarts-types.\n
Il faut normalement faire figurer les erreurs standard.")

```

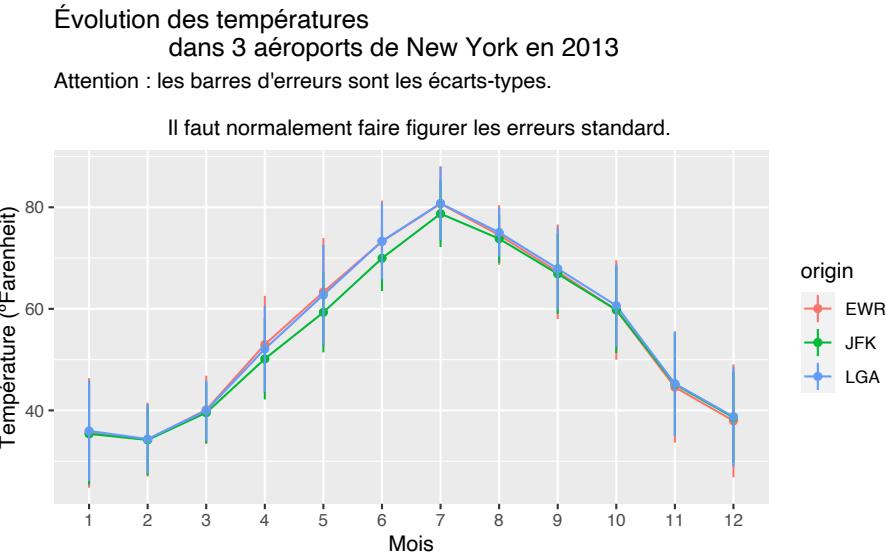


Figure 84 – Évolution des températures moyenne dans 3 aéroports de New York en 2013

Notez que j'utilise maintenant `group = origin` et non plus `group = 1`. Ici, les températures des 3 aéroports sont tellement similaires que les courbes sont difficiles à distinguer. Nous pouvons donc utiliser `facet_wrap()` pour tenter d'améliorer la visualisation :

```

monthly_orig_temp %>%
  ggplot(aes(x = factor(month), y = moyenne, group = origin, color = origin)) +
  geom_line() +
  geom_point() +
  geom_linerange(aes(ymax = moyenne + ecart_type, ymin = moyenne - ecart_type)) +
  facet_wrap(~origin, ncol = 1) +
  labs(x = "Mois",
       y = "Température (°Farenheit)",
       title = "Évolution des températures
                 dans 3 aéroports de New York en 2013",
       subtitle = "Attention : les barres d'erreurs sont les écarts-types.\n
Il faut normalement faire figurer les erreurs standard.")

```

Enfin, lorsque nous groupons par plusieurs variables, il peut être utile de présenter les résultats sous la forme d'un tableau large (grâce à la fonction `pivot_wider()`, voir section 5.2.2) pour l'intégration dans un rapport par exemple :

```

weather %>%
  group_by(origin, month) %>%

```

**Évolution des températures  
dans 3 aéroports de New York en 2013**

Attention : les barres d'erreurs sont les écarts-types.

Il faut normalement faire figurer les erreurs standard.

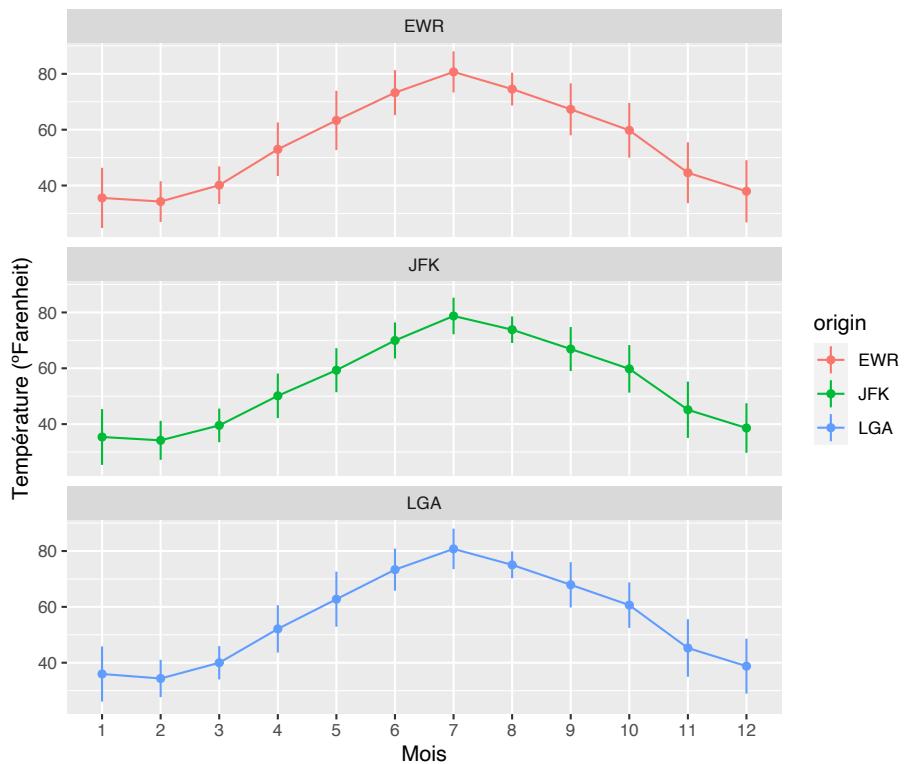


Figure 85 – Évolution des températures moyenne dans 3 aéroports de New York en 2013

```

summarise(moyenne = mean(temp, na.rm = TRUE)) %>%
pivot_wider(names_from = origin,
values_from = moyenne)

`summarise()` has grouped output by 'origin'. You can override using the `groups` argument.

# A tibble: 12 x 4
  month   EWR   JFK   LGA
  <int> <dbl> <dbl> <dbl>
1     1   35.6  35.4  36.0
2     2   34.3  34.2  34.4
3     3   40.1  39.5  40.0
4     4   53.0  50.1  52.1
5     5   63.3  59.3  62.8
6     6   73.3  70.0  73.3
7     7   80.7  78.7  80.8
8     8   74.5  73.8  75.0
9     9   67.3  66.9  67.9
10    10  59.8  59.8  60.6
11    11  44.6  45.1  45.3
12    12  38.0  38.6  38.8

```

Sous cette forme, les données ne sont plus “rangées”, nous n'avons plus des “tidy data”, mais nous avons un tableau plus synthétique, facile à inclure dans un rapport.

### 6.5.5 Un raccourci pratique pour compter des effectifs

Il est tellement fréquent d'avoir à grouper des données en fonction d'une variable puis à compter le nombre d'observations dans chaque catégorie avec `n()` que `dplyr` nous fournit un raccourci : la fonction `count()`.

Ce code :

```

weather %>%
  group_by(month) %>%
  summarise(n = n())

```

```

# A tibble: 12 x 2
  month     n
  <int> <int>
1     1    2226
2     2    2010
3     3    2227
4     4    2159
5     5    2232

```

```
6     6 2160
7     7 2228
8     8 2217
9     9 2159
10    10 2212
11    11 2141
12    12 2144
```

est équivalent à celui-ci :

```
weather %>%
  count(month)
```

```
# A tibble: 12 x 2
  month     n
  <int> <int>
1     1 2226
2     2 2010
3     3 2227
4     4 2159
5     5 2232
6     6 2160
7     7 2228
8     8 2217
9     9 2159
10    10 2212
11    11 2141
12    12 2144
```

Comme avec `group_by()`, il est bien sûr possible d'utiliser `count()` avec plusieurs variables :

```
weather %>%
  count(origin, month)
```

```
# A tibble: 36 x 3
  origin month     n
  <chr>   <int> <int>
1 EWR      1    742
2 EWR      2    669
3 EWR      3    743
4 EWR      4    720
5 EWR      5    744
6 EWR      6    720
7 EWR      7    741
8 EWR      8    740
```

```

9 EWR      9    719
10 EWR     10   736
# ... with 26 more rows

```

### 6.5.6 Exercices

- Faites un tableau indiquant combien de vols ont été annulés après le décollage, pour chaque compagnie aérienne. Vous devriez obtenir le tableau suivant :

```

# A tibble: 13 x 2
  carrier cancelled
  <chr>      <int>
1 9E          71
2 AA          34
3 B6          32
4 DL          15
5 EV         105
6 F9           1
7 FL           6
8 MQ          87
9 UA          63
10 US          31
11 VX           4
12 WN           8
13 YV           1

```

- Faites un tableau indiquant les vitesses de vents minimales, maximales et moyennes, enregistrées chaque mois dans chaque aéroport de New York. Votre tableau devrait ressembler à ceci :

``summarise()` has grouped output by 'origin'. You can override using the `groups` argument.`

```

# A tibble: 36 x 5
# Groups:   origin [3]
  origin month max_wind min_wind moy_wind
  <chr>   <int>    <dbl>    <dbl>    <dbl>
1 EWR      1     42.6      0     9.87
2 EWR      2    1048.      0    12.2
3 EWR      3     29.9      0    11.6
4 EWR      4     25.3      0    9.63
5 EWR      5     33.4      0    8.49
6 EWR      6     34.5      0    9.55
7 EWR      7     20.7      0    9.15
8 EWR      8     21.9      0    7.62
9 EWR      9     23.0      0    8.03

```

```

10 EWR      10    26.5      0    8.32
# ... with 26 more rows

```

3. Sachant que les vitesses du vent sont exprimées en miles par heure, certaines valeurs sont-elles surprenantes ? À l'aide de la fonction `filter()`, éliminez la ou les valeurs aberrantes. Vous devriez obtenir ce tableau :

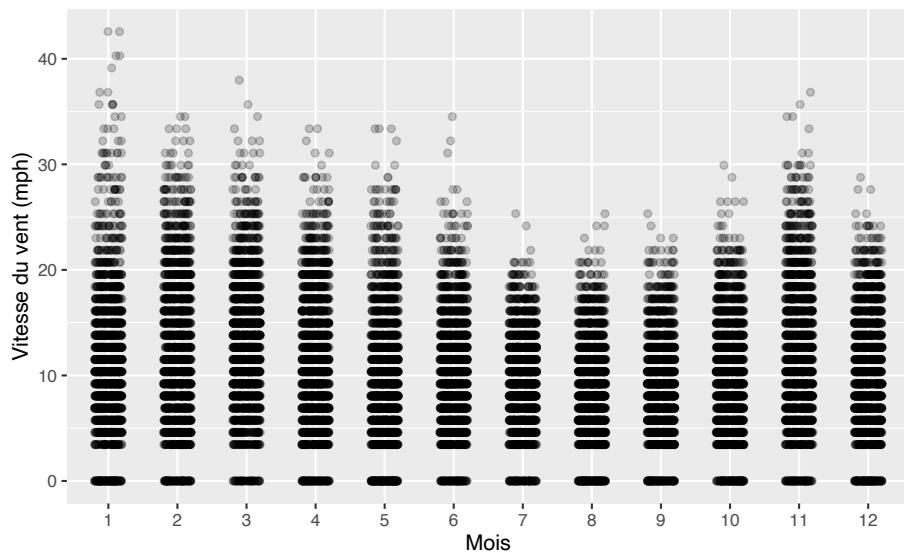
``summarise()` has grouped output by 'origin'. You can override using the `groups` argument.`

```

# A tibble: 36 x 5
# Groups:   origin [3]
  origin month max_wind min_wind moy_wind
  <chr>   <int>    <dbl>     <dbl>     <dbl>
1 EWR       1      42.6      0     9.87
2 EWR       2      31.1      0    10.7
3 EWR       3      29.9      0    11.6
4 EWR       4      25.3      0    9.63
5 EWR       5      33.4      0    8.49
6 EWR       6      34.5      0    9.55
7 EWR       7      20.7      0    9.15
8 EWR       8      21.9      0    7.62
9 EWR       9      23.0      0    8.03
10 EWR      10     26.5      0    8.32
# ... with 26 more rows

```

4. En utilisant les données de vitesse de vent du tableau `weather`, produisez le graphique suivant :



Indications :

- les vitesses de vent aberrantes ont été éliminées grâce à la fonction `filter()`

- la fonction `geom_jitter()` a été utilisée avec l'argument `height = 0`
- la transparence des points est fixée à `0.2`

Selon vous, pourquoi les points sont-ils organisés en bandes horizontales ?

Selon vous, pourquoi n'y a t'il jamais de vent entre 0 et environ 3 miles à l'heure (mph) ?

Sachant qu'en divisant des mph par 1.151 on obtient des vitesses en noeuds, que nous apprend cette commande :

```
sort(unique(weather$wind_speed)) / 1.151
```

```
[1] 0.000000 2.999427 3.999235 4.999044 5.998853 6.998662
[7] 7.998471 8.998280 9.998089 10.997897 11.997706 12.997515
[13] 13.997324 14.997133 15.996942 16.996751 17.996560 18.996368
[19] 19.996177 20.995986 21.995795 22.995604 23.995413 24.995222
[25] 25.995030 26.994839 27.994648 28.994457 29.994266 30.994075
[31] 31.993884 32.993692 33.993501 34.993310 36.992928 910.825873
```

---

## 6.6 Sélectionner des variables avec `select()`



Figure 86 – Schéma de la fonction ‘select()’ tiré de la ‘cheatsheet’ de ‘dplyr’ et ‘tidyverse’

Il n'est pas rare de travailler avec des tableaux contenant des centaines, voir des milliers de colonnes. Dans de tels cas, il peut être utile de réduire le jeu de données aux variables qui vous intéressent. Le rôle de la fonction `select()` est de retenir uniquement les colonnes dont on a spécifié le nom, afin de recentrer l'analyse sur les variables utiles.

`select()` n'est pas particulièrement utile pour le jeu de données `flights` puisqu'il ne contient que 19 variables. Toutefois, on peut malgré tout comprendre le fonctionnement général. Par exemple, pour sélectionner uniquement les colonnes `year`, `month` et `day`, on tape :

```
# Sélection de variables par leur nom
flights %>%
  select(year, month, day)
```

```
# A tibble: 336,776 x 3
```

```
  year month   day
```

```
  <int> <int> <int>
```

```

1 2013    1    1
2 2013    1    1
3 2013    1    1
4 2013    1    1
5 2013    1    1
6 2013    1    1
7 2013    1    1
8 2013    1    1
9 2013    1    1
10 2013   1    1
# ... with 336,766 more rows

```

Puisque ces 3 variables sont placées les unes à côté des autres dans le tableau `flights`, on peut utiliser la notation : pour les sélectionner :

```

# Sélection de toutes les variables entre `year` et `day` (inclus)
flights %>%
  select(year:day)

```

```

# A tibble: 336,776 x 3
  year month   day
  <int> <int> <int>
1 2013    1    1
2 2013    1    1
3 2013    1    1
4 2013    1    1
5 2013    1    1
6 2013    1    1
7 2013    1    1
8 2013    1    1
9 2013    1    1
10 2013   1    1
# ... with 336,766 more rows

```

À l'inverse, si on veut supprimer certaines colonnes, on peut utiliser la notation - :

```

# Sélection de toutes les variables de `flights` à l'exception
# de celles comprises entre `year` et `day` (inclus)
flights %>%
  select(-(year:day))

```

```

# A tibble: 336,776 x 16
  dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
  <int>        <int>     <dbl>    <int>        <int>      <dbl>
1      517          515       2       830         819       11

```

```

2      533        529       4     850       830       20
3      542        540       2     923       850       33
4      544        545      -1    1004      1022      -18
5      554        600      -6     812       837      -25
6      554        558      -4     740       728       12
7      555        600      -5     913       854       19
8      557        600      -3     709       723      -14
9      557        600      -3     838       846       -8
10     558        600      -2     753       745        8
# ... with 336,766 more rows, and 10 more variables: carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

Il y a beaucoup de fonctions permettant de sélectionner des variables dont le noms respectent certains critères. Par exemple :

- starts\_with("abc") : renvoie toutes les variables dont les noms commencent par "abc"
- ends\_with("xyz") : renvoie toutes les variables dont les noms se terminent par "xyz"
- contains("ijk") : renvoie toutes les variables dont les noms contiennent "ijk"

Il en existe beaucoup d'autres. Vous pouvez consulter l'aide de ?select() pour en savoir plus.

Ainsi, il est par exemple possible d'extraire toutes les variables contenant le mot "time" ainsi :

```

flights %>%
  select(contains("time"))

```

```

# A tibble: 336,776 x 6
  dep_time sched_dep_time arr_time sched_arr_time air_time
  <int>        <int>     <int>        <int>      <dbl>
1      517        515     830         819      227
2      533        529     850         830      227
3      542        540     923         850      160
4      544        545    1004        1022      183
5      554        600     812         837      116
6      554        558     740         728      150
7      555        600     913         854      158
8      557        600     709         723      53
9      557        600     838         846      140
10     558        600     753         745      138
# ... with 336,766 more rows, and 1 more variable: time_hour <dttm>

```

Évidemment, le tableau flights n'est pas modifié par cette opération : il contient toujours les 19 variables de départ. Pour travailler avec ces tableaux de données contenant moins de variables, il faut les stocker dans un nouvel objet en leur donnant un nom :

```
flights_time <- flights %>%  
  select(contains("time"))
```

Enfin, on peut utiliser `select()` pour renommer des variables. Mais ce n'est que rarement utile car `select()` élimine toutes les variables qui n'ont pas été explicitement nommées :

```
flights %>%  
  select(year:day,  
         heure_depart = dep_time,  
         retard_depart = dep_delay)
```

```
# A tibble: 336,776 x 5  
  year month day heure_depart retard_depart  
  <int> <int> <int>      <int>       <dbl>  
1 2013     1    1        517        2  
2 2013     1    1        533        4  
3 2013     1    1        542        2  
4 2013     1    1        544       -1  
5 2013     1    1        554       -6  
6 2013     1    1        554       -4  
7 2013     1    1        555       -5  
8 2013     1    1        557       -3  
9 2013     1    1        557       -3  
10 2013    1    1        558       -2  
# ... with 336,766 more rows
```

Il est donc généralement préférable d'utiliser `rename()` pour renommer certaines variables sans en éliminer aucune :

```
flights %>%  
  rename(heure_depart = dep_time,  
         retard_depart = dep_delay)
```

```
# A tibble: 336,776 x 19  
  year month day heure_depart sched_dep_time retard_depart  
  <int> <int> <int>      <int>       <int>       <dbl>  
1 2013     1    1        517        515        2  
2 2013     1    1        533        529        4  
3 2013     1    1        542        540        2  
4 2013     1    1        544        545       -1  
5 2013     1    1        554        600       -6  
6 2013     1    1        554        558       -4  
7 2013     1    1        555        600       -5  
8 2013     1    1        557        600       -3  
9 2013     1    1        557        600       -3
```

```

10 2013     1     1      558       600        -2
# ... with 336,766 more rows, and 13 more variables: arr_time <int>,
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

---

## 6.7 Créer de nouvelles variables avec `mutate()`

### 6.7.1 Principe

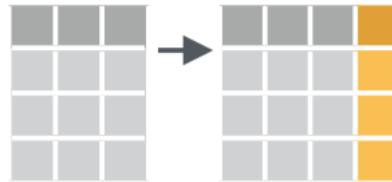


Figure 87 – Schéma de la fonction ‘`mutate()`’ tiré de la ‘cheatsheet’ de ‘dplyr’ et ‘tidyR’

La fonction `mutate()` permet de créer de nouvelles variables à partir des variables existantes, ou de modifier des variables déjà présentes dans un jeu de données. Il est en effet fréquent d'avoir besoin de calculer de nouvelles variables, souvent plus informatives que les variables disponibles.

Voyons un exemple. Nous commençons par créer un nouveau jeu de données `flights_sml` en sélectionnant uniquement les variables qui nous seront utiles :

```

flights_sml <- flights %>%
  select(year:day,
         ends_with("delay"),
         distance,
         air_time)
flights_sml

```

```

# A tibble: 336,776 x 7
  year month   day dep_delay arr_delay distance air_time
  <int> <int> <int>    <dbl>    <dbl>    <dbl>    <dbl>
1 2013     1     1        2        11     1400     227
2 2013     1     1        4        20     1416     227
3 2013     1     1        2        33     1089     160
4 2013     1     1       -1       -18     1576     183
5 2013     1     1       -6       -25      762     116

```

```

6 2013    1    1     -4      12     719     150
7 2013    1    1     -5      19    1065     158
8 2013    1    1     -3     -14     229      53
9 2013    1    1     -3      -8     944     140
10 2013   1    1     -2       8     733     138
# ... with 336,766 more rows

```

À partir de ce nouveau tableau, nous allons calculer 2 nouvelles variables :

1. gain : afin de savoir si les avions peuvent rattraper une partie de leur retard en vol, nous allons calculer la différence entre le retard au départ et à l'arrivée (donc le gain de temps accumlé lors du vol). En effet, si un avion décolle avec 15 minutes de retard, mais qu'il atterrit avec seulement 5 minutes de retard, 10 minutes ont été gagnées en vol, et les passagers sont moins mécontents.
2. speed afin de connaître la vitesse moyenne des avions en vol, nous allons diviser la distance parcourue par les avions par le temps passé en l'air. Il ne faudra pas oublier de multiplier par 60 car les temps sont exprimés en minutes. Nous en profiterons pour transformer les distances exprimées en miles par des distances exprimées en kilomètres en multipliant les distances en miles par 1.60934. Ainsi, speed sera exprimée en kilomètres par heure.

```

flights_sml %>%
  mutate(gain = dep_delay - arr_delay,
        distance = distance * 1.60934,
        speed = (distance / air_time) * 60)

```

```

# A tibble: 336,776 x 9
  year month   day dep_delay arr_delay distance air_time   gain speed
  <int> <int> <int>     <dbl>     <dbl>     <dbl>     <dbl> <dbl> <dbl>
1 2013    1    1        2      11    2253.     227     -9  596.
2 2013    1    1        4      20    2279.     227    -16  602.
3 2013    1    1        2      33    1753.     160    -31  657.
4 2013    1    1       -1     -18    2536.     183     17  832.
5 2013    1    1       -6     -25    1226.     116     19  634.
6 2013    1    1       -4      12    1157.     150    -16  463.
7 2013    1    1       -5      19    1714.     158    -24  651.
8 2013    1    1       -3     -14     369.      53     11  417.
9 2013    1    1       -3      -8    1519.     140      5  651.
10 2013   1    1       -2       8    1180.     138    -10  513.
# ... with 336,766 more rows

```

Si on souhaite conserver uniquement les variables nouvellement créées par `mutate()`, on peut utiliser `transmute()` :

```

flights_sml %>%
  transmute(gain = dep_delay - arr_delay,
            distance = distance * 1.60934,
            speed = (distance / air_time) * 60)

```

```

    distance = distance * 1.60934,
    speed = (distance / air_time) * 60)

# A tibble: 336,776 x 3
  gain distance speed
  <dbl>    <dbl> <dbl>
1     -9    2253.  596.
2    -16    2279.  602.
3    -31    1753.  657.
4     17    2536.  832.
5     19    1226.  634.
6    -16    1157.  463.
7    -24    1714.  651.
8     11     369.  417.
9      5    1519.  651.
10    -10    1180.  513.
# ... with 336,766 more rows

```

Et comme toujours, pour pouvoir réutiliser ces données, on leur donne un nom :

```

gain_speed <- flights_sml %>%
  transmute(gain = dep_delay - arr_delay,
            distance = distance * 1.60934,
            speed = (distance / air_time) * 60)

```

### 6.7.2 Exercices

1. Dans `ggplot2` le jeu de données `mpg` contient des informations sur 234 modèles de voitures. Examinez ce jeu de données avec la fonction `View()` et consultez l'aide de ce jeu de données pour savoir à quoi correspondent les différentes variables. Quelle(s) variable(s) nous renseignent sur la consommation des véhicules ? À quoi correspond la variable `disp` ?
2. La consommation est donnée en miles par gallon. Créez une nouvelle variable `conso` qui contiendra la consommation sur autoroute, exprimée en nombre de litres pour 100 kilomètres.
3. Faîtes un graphique présentant la relation entre la cylindrée en litres et la consommation sur autoroute exprimée en nombre de litres pour 100 kilomètres. Vous exclurez les véhicules dont la classe est `2seater` de ce graphique (il s'agit de voitures de sports très compactes qu'il est difficile de mesurer aux autres). Sur votre graphique, la couleur devrait représenter le type de véhicule. Vous ajouterez une droite de régression en utilisant `geom_smooth(method = "lm")`. Votre graphique devrait ressembler à ceci :

``geom_smooth()` using formula 'y ~ x'`

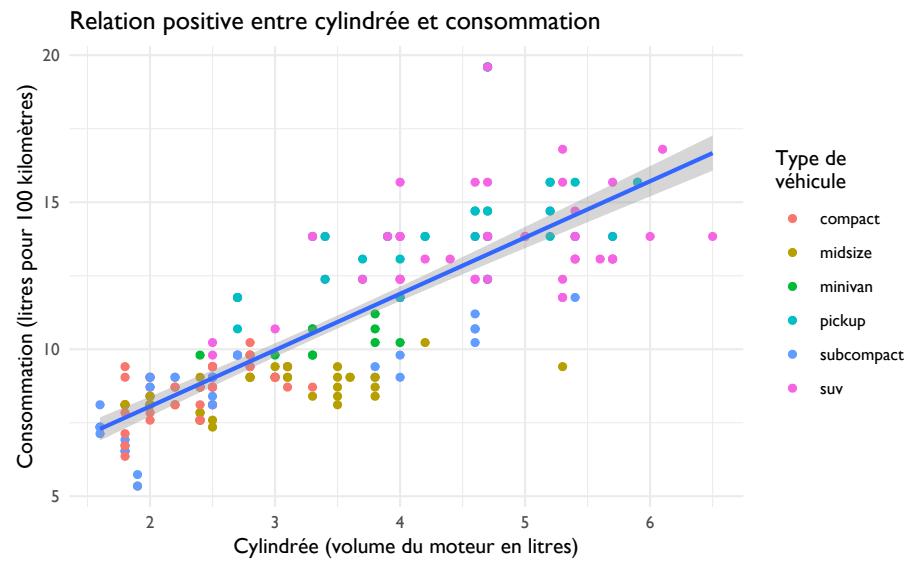


Figure 88 – Consommation en fonction de la cylindrée

4. Ce graphique présente-t'il correctement l'ensemble des données de ces 2 variables ? Pourquoi ? Comparez le graphique 88 de la question 3 ci-dessus et le graphique 89 présenté ci-dessous. Selon vous, quels arguments et/ou fonctions ont été modifiés pour arriver à ce nouveau graphique ? Quels sont les avantages et les inconvénients de ce graphique par rapport au précédent ?

```
`geom_smooth()` using formula 'y ~ x'
```

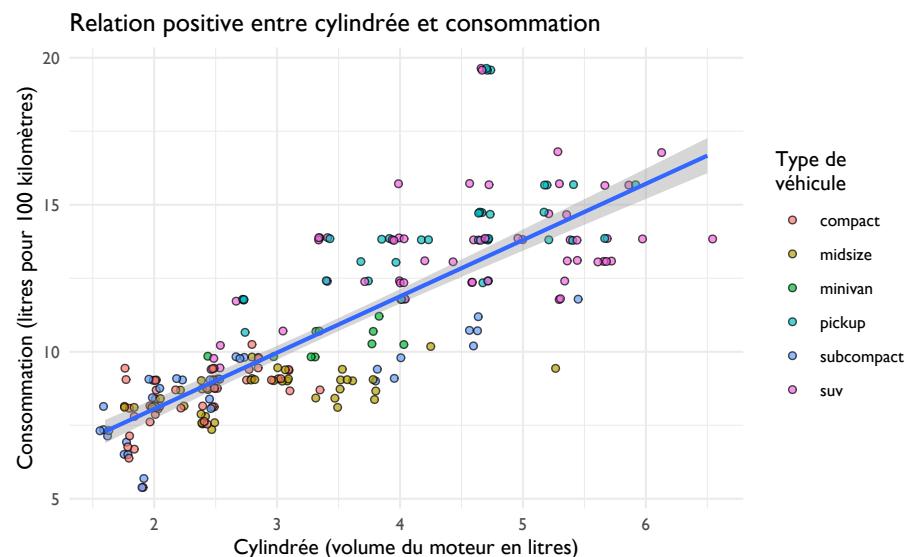


Figure 89 – Consommation en fonction de la cylindrée

## 6.8 Trier des lignes avec arrange()

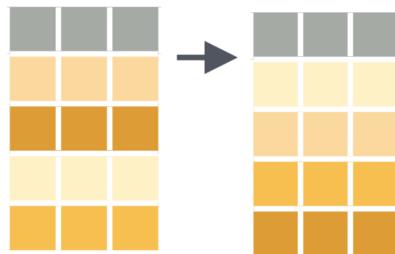


Figure 90 – Schéma de la fonction ‘arrange()’ tiré de la ‘cheatsheet’ de ‘dplyr’ et ‘tidyr’

La fonction `arrange()` permet de trier des tableaux en ordonnant les éléments d'une ou plusieurs colonnes. Les tris peuvent être en ordre croissants (c'est le cas par défaut) ou décroissants (grâce à la fonction `desc()`, abréviation de “descending”).

`arrange()` fonctionne donc comme `filter()`, mais au lieu de sélectionner des lignes, cette fonction change leur ordre. Il faut lui fournir le nom d'un tableau et au minimum le nom d'une variable selon laquelle le tri doit être réalisé. Si plusieurs variables sont fournies, chaque variable supplémentaire permet de résoudre les égalités. Ainsi, pour ordonner le tableau `flights` par ordre croissant de retard au départ (`dep_delay`), on tape :

```
flights %>%  
  arrange(dep_delay)
```

```
# A tibble: 336,776 x 19  
  year month   day dep_time sched_dep_time dep_delay arr_time  
  <int> <int> <int>    <int>          <int>     <dbl>    <int>  
1 2013     12     7      2040          2123     -43       40  
2 2013      2     3      2022          2055     -33      2240  
3 2013     11    10      1408          1440     -32      1549  
4 2013      1    11      1900          1930     -30      2233  
5 2013      1    29      1703          1730     -27      1947  
6 2013      8     9       729           755     -26      1002  
7 2013     10    23      1907          1932     -25      2143  
8 2013      3    30      2030          2055     -25      2213  
9 2013      3     2      1431          1455     -24      1601  
10 2013     5     5      934           958     -24      1225  
# ... with 336,766 more rows, and 12 more variables:  
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,  
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,  
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,  
#   time_hour <dttm>
```

Notez que la variable `dep-delay` est maintenant triée en ordre croissant. Notez également que 2 vols ont eu exactement 25 minutes d'avance. Comparez le tableau précédent avec celui-ci :

```

flights %>%
  arrange(dep_delay, month, day)

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>    <dbl>    <int>
1 2013     12     7      2040          2123     -43      40
2 2013      2     3      2022          2055     -33     2240
3 2013     11    10      1408          1440     -32     1549
4 2013      1    11      1900          1930     -30     2233
5 2013      1    29      1703          1730     -27     1947
6 2013      8     9       729           755     -26     1002
7 2013      3    30      2030          2055     -25     2213
8 2013     10    23      1907          1932     -25     2143
9 2013      3     2      1431          1455     -24     1601
10 2013     5     5      934           958     -24     1225
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>

```

Les lignes des 2 vols ayant 25 minutes d'avance au décollage ont été inversées : le vol du mois de mars apparait maintenant avant le vol du mois d'octobre. La variable `month` a été utilisée pour ordonner les lignes en cas d'égalité de la variable `dep_delay`.

Comme indiqué plus haut, il est possible de trier les données par ordre décroissant :

```

flights %>%
  arrange(desc(dep_delay))

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>    <dbl>    <int>
1 2013     1     9      641           900     1301     1242
2 2013      6    15     1432          1935     1137     1607
3 2013      1    10     1121          1635     1126     1239
4 2013      9    20     1139          1845     1014     1457
5 2013      7    22      845          1600     1005     1044
6 2013      4    10     1100          1900      960     1342
7 2013      3    17     2321          810      911      135
8 2013      6    27      959          1900      899     1236
9 2013      7    22     2257          759      898      121
10 2013     12     5      756          1700      896     1058
# ... with 336,766 more rows, and 12 more variables:

```

```
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

Cela est particulièrement utile après l'obtention de résumés groupés pour connaître la catégorie la plus représentée. Par exemple, si nous souhaitons connaître les destinations la plus fréquentes au départ de New York, on peut procéder ainsi :

1. prendre le tableau `flights`, *puis*,
2. grouper les données par destination (variable `dest`) et compter le nombre de vols.  
Ces deux opérations peuvent être effectuées avec `group_by()` puis `summarise()`, ou directement avec `count()`. *Puis*,
3. trier les données par effectif décroissant.

```
flights %>%
  count(dest) %>%
  arrange(desc(n))
```

```
# A tibble: 105 x 2
  dest      n
  <chr> <int>
1 ORD     17283
2 ATL     17215
3 LAX     16174
4 BOS     15508
5 MCO     14082
6 CLT     14064
7 SFO     13331
8 FLL     12055
9 MIA     11728
10 DCA    9705
# ... with 95 more rows
```

Nous voyons ici que les aéroports ORD et ATL sont les 2 destinations les plus fréquentes, avec plus de 17000 vols par an.

## 6.9 Associer plusieurs tableaux avec `left_join()` et `inner_join()`

### 6.9.1 Principe

Une autre règle que nous n'avons pas encore évoquée au sujet des “tidy data” ou “données rangées” est la suivante :

Chaque tableau contient des données appartenant à une unité d'observation cohérente et unique.

Autrement dit, les informations concernant les vols vont dans un tableau, les informations concernant les aéroports dans un autre tableau, et celles concernant les avions dans un troisième tableau. Cela semble évident, pourtant, on constate souvent qu'un même tableau contient des variables qui concernent des unités d'observations différentes.

Lorsque nous avons plusieurs tableaux à disposition, il peut alors être nécessaire de récupérer des informations dans plusieurs d'entre eux afin, notamment de produire des tableaux de synthèse. Par exemple, dans la section 6.8 ci-dessus, nous avons affiché les destinations les plus fréquentes au départ des aéroports de New York. Donnons un nom à ce tableau pour pouvoir le ré-utiliser :

```
popular_dest <- flights %>%  
  count(dest) %>%  
  arrange(desc(n))  
popular_dest
```

```
# A tibble: 105 x 2  
  dest      n  
  <chr> <int>  
1 ORD     17283  
2 ATL     17215  
3 LAX     16174  
4 BOS     15508  
5 MCO     14082  
6 CLT     14064  
7 SFO     13331  
8 FLL     12055  
9 MIA     11728  
10 DCA    9705  
# ... with 95 more rows
```

À quel aéroport correspondent les codes ORD et ATL ? S'agit-il d'Orlando et Atlanta ? Pour le savoir, il faut aller chercher l'information qui se trouve dans le tableau airports : il contient, parmi d'autres variables, les codes et les noms de 1458 aéroports aux États-Unis.

dplyr fournit toute une gamme de fonctions permettant d'effectuer des associations de tableaux en fonction de critères spécifiés par l'utilisateur.

### 6.9.2 inner\_join()

La fonction inner\_join() permet de relier des tableaux en ne conservant que les lignes qui sont présentes à la fois dans l'un et dans l'autre. Il faut identifier dans chacun des tableaux une colonne contenant des données en commun, qui servira de guide pour mettre les lignes

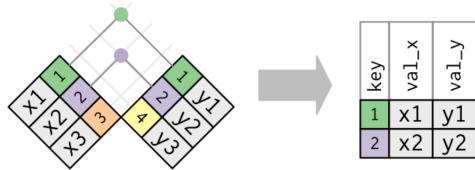


Figure 91 – Schéma de la fonction `inner_join()` tiré de la ‘cheatsheet’ de `dplyr` et `tidyR`.

correctes les unes en face des autres. Ici, nous partons de notre tableau `popular_dest`, qui contient les codes des aéroports dans sa colonne `dest`. Et nous faisons une “jointure interne” avec le tableau `airports` qui contient lui aussi une colonne contenant les codes des aéroports : la variable `faa`.

```
inner_popular <- popular_dest %>%
  inner_join(airports, by = c("dest" = "faa"))
inner_popular
```

```
# A tibble: 101 x 9
  dest      n name          lat   lon   alt   tz dst tzone
  <chr> <int> <chr>       <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 ORD     17283 Chicago Ohare~ 42.0 -87.9  668    -6 A   America/~
2 ATL     17215 Hartsfield Ja~ 33.6 -84.4  1026   -5 A   America/~
3 LAX     16174 Los Angeles I~ 33.9 -118.   126    -8 A   America/~
4 BOS     15508 General Edwar~ 42.4 -71.0   19    -5 A   America/~
5 MCO     14082 Orlando Intl  28.4 -81.3   96    -5 A   America/~
6 CLT     14064 Charlotte Dou~ 35.2 -80.9  748    -5 A   America/~
7 SFO     13331 San Francisco~ 37.6 -122.   13    -8 A   America/~
8 FLL     12055 Fort Lauderda~ 26.1 -80.2   9    -5 A   America/~
9 MIA     11728 Miami Intl    25.8 -80.3   8    -5 A   America/~
10 DCA    9705 Ronald Reagan~ 38.9 -77.0  15    -5 A   America/~
# ... with 91 more rows
```

Le nouvel objet `inner_popular` contient donc les données du tableau `popular_dest` auxquelles ont été ajoutées les colonnes correspondantes du tableau `airports`. Si tout ce qui nous intéresse, c'est de connaître le nom complet des aéroports les plus populaires, on peut utiliser `select()` pour ne garder que les variables intéressantes :

```
inner_popular <- popular_dest %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  select(dest, name, n)
inner_popular
```

```
# A tibble: 101 x 3
  dest   name          n
  <chr> <chr>     <int>
1 ORD   Chicago Ohare~ 17283
2 ATL   Hartsfield Ja~ 17215
3 LAX   Los Angeles I~ 16174
4 BOS   General Edwar~ 15508
5 MCO   Orlando Intl  14082
6 CLT   Charlotte Dou~ 14064
7 SFO   San Francisco~ 13331
8 FLL   Fort Lauderda~ 12055
9 MIA   Miami Intl    11728
10 DCA  Ronald Reagan~ 9705
```

```

1 ORD Chicago Ohare Intl           17283
2 ATL Hartsfield Jackson Atlanta Intl 17215
3 LAX Los Angeles Intl            16174
4 BOS General Edward Lawrence Logan Intl 15508
5 MCO Orlando Intl                14082
6 CLT Charlotte Douglas Intl      14064
7 SFO San Francisco Intl          13331
8 FLL Fort Lauderdale Hollywood Intl 12055
9 MIA Miami Intl                  11728
10 DCA Ronald Reagan Washington Natl 9705
# ... with 91 more rows

```

On peut noter plusieurs choses dans ce nouveau tableau :

- ORD n'est pas l'aéroport d'Orlando mais l'aéroport international de Chicago Ohare. C'est donc la destination la plus fréquente au départ de New York.
- ATL est bien l'aéroport d'Atlanta.
- inner\_popular contient 101 lignes alors que notre tableau de départ en contenait 105.

```
nrow(popular_dest)
```

```
[1] 105
```

```
nrow(inner_popular)
```

```
[1] 101
```

Certaines lignes ont donc été supprimées car le code aéroport dans popular\_dest (notre tableau de départ) n'a pas été retrouvé dans la colonne faa du tableau airports. C'est le principe même de la jointure interne (voir figure 91) : seules les lignes communes trouvées dans les 2 tableaux sont conservées. Si l'on souhaite absolument conserver toutes les lignes du tableau de départ, il faut faire une jointure gauche, ou "left join".

### 6.9.3 left\_join()

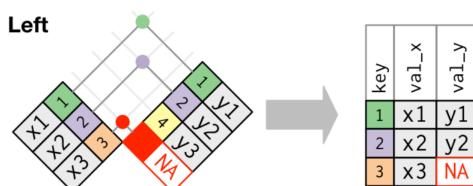


Figure 92 – Schéma de la fonction `left_join()` tiré de la ‘cheatsheet’ de `dplyr` et `tidyR`.

Comme indiqué par la figure 92 ci-dessus, une jointure gauche permet de conserver toutes les lignes du tableau de gauche, et de leur faire correspondre des lignes du second tableau. Si aucune

correspondance n'est trouvée dans le second tableau, des données manquantes sont ajoutées sous forme de NAs. Voyons ce que cela donne avec le même exemple que précédemment :

```
left_popular <- popular_dest %>%
  left_join(airports, by = c("dest" = "faa")) %>%
  select(dest, name, n)
left_popular
```

```
# A tibble: 105 x 3
  dest   name                 n
  <chr> <chr>                <int>
1 ORD    Chicago Ohare Intl  17283
2 ATL    Hartsfield Jackson Atlanta Intl 17215
3 LAX    Los Angeles Intl   16174
4 BOS    General Edward Lawrence Logan Intl 15508
5 MCO    Orlando Intl      14082
6 CLT    Charlotte Douglas Intl 14064
7 SFO    San Francisco Intl 13331
8 FLL    Fort Lauderdale Hollywood Intl 12055
9 MIA    Miami Intl        11728
10 DCA   Ronald Reagan Washington Natl  9705
# ... with 95 more rows
```

En apparence, le tableau `left_popular`, créé avec `left_join()` semble identique au tableau `inner_popular` créé avec `inner_join()`. Pourtant, ce n'est pas le cas :

```
identical(inner_popular, left_popular)
```

```
[1] FALSE
```

En l'occurrence, nous avons vu que `inner_popular` ne contenait pas autant de ligne que le tableau de départ `popular_dest`. Avec une jointure gauche, les lignes du tableau de départ sont toutes conservées. `popular_dest` et `left_popular` ont donc le même nombre de lignes.

```
nrow(inner_popular)
```

```
[1] 101
```

```
nrow(left_popular)
```

```
[1] 105
```

```
nrow(popular_dest)
```

```
[1] 105
```

Pour savoir quelles lignes de `popular_dest` manquent dans `inner_dest` (il devrait y en avoir 4), il suffit de filtrer les lignes de `left_dest` qui contiennent des données manquantes dans la colonne `name` :

```
left_popular %>%
  filter(is.na(name))
```

```
# A tibble: 4 x 3
  dest   name      n
  <chr> <chr> <int>
1 SJU    <NA>    5819
2 BQN    <NA>     896
3 STT    <NA>     522
4 PSE    <NA>     365
```

Une rapide recherche dans un moteur de recherche vous apprendra que ces aéroports ne sont pas situés sur le sol américain. Trois d'entre eux sont situés à Puerto Rico (SJU, BQN et PSE) et le dernier (STT) est situé aux îles Vierges.

Il y aurait bien plus à dire sur les jointures :

- Quelles sont les autres possibilités de jointures (`right_join()`, `outer_join()`, `full_join()`, etc...)?
- Que se passe-t'il si les colonnes communes des 2 tableaux contiennent des éléments dupliqués?
- Est-il toujours nécessaire d'utiliser l'argument `by`?
- Est-il possible de joindre des tableaux en associant plus d'une colonne de chaque tableau d'origine?

Pour avoir la réponse à toutes ces questions, je vous conseille de lire [ce chapitre](#) de cet ouvrage très complet sur “la science des données” avec R et le tidyverse : [R for Data Science](#).

Néanmoins, avec les deux fonctions `inner_join()` et `left_join()` décrite ici devraient vous permettre de couvrir l'essentiel de vos besoins.

---

## 6.10 Exercices

- I. Créez un tableau `delayed` indiquant, pour chaque compagnie aérienne et chaque mois de l'année, le nombre de vols ayant eu un retard supérieur à 30 minutes à l'arrivée à destination. Ce tableau devrait contenir uniquement 3 colonnes :

- carrier : la compagnie aérienne
  - month : le mois de l'année 2013
  - n\_delayed : le nombre de vols ayant plus de 30 minutes de retard
2. Créez un tableau total indiquant le nombre total de vols affrétés (et non annulés) par chaque compagnie aérienne et chaque mois de l'année. Ce tableau devrait contenir seulement 3 colonnes :
    - carrier : la compagnie aérienne
    - month : le mois de l'année 2013
    - n\_total : le nombre total de vols arrivés à destination  3. Fusionnez ces 2 tableaux en réalisant la jointure appropriée. Le tableau final, que vous nommerez `carrier_stats` devrait contenir 185 lignes. Si certaines colonnes contiennent des données manquantes, remplacez-les par des 0 à l'aide des fonctions `mutate()` et `replace_na()`.
  4. Ajoutez à votre tableau `carrier_stats` une variable `rate` qui contient la proportion de vols arrivés à destination avec plus de 30 minutes de retard, pour chaque compagnie aérienne et chaque mois de l'année.
  5. Ajoutez à votre tableau `carrier_stats` le nom complet des compagnies aériennes en réalisant la jointure appropriée avec le tableau `airlines`.
  6. Faites un graphique synthétique présentant ces résultats de la façon la plus claire possible
  7. Quelle compagnie aérienne semble se comporter très différemment des autres ? À quoi pouvez-vous attribuer ce comportement atypique ?
  8. Pour les compagnies affrétant un grand nombre de vols chaque année (e.g. UA, B6 et EV), quelles sont les périodes où les plus fortes proportions de vols en retard sont observées ? Et les plus faibles ? Quelle(s) hypothèse(s) pouvez-vous formuler pour expliquer ces observations ?
  9. Faites un tableau synthétique présentant ces résultats de la façon la plus compacte et claire que possible, afin par exemple de les intégrer à un rapport.

## Références

- Wickham H, Chang W, Henry L, Pedersen TL, Takahashi K, Wilke C, ... Dunnington D. (2021). *Ggplot2 : Create elegant data visualisations using the grammar of graphics*. Retrieved from <https://CRAN.R-project.org/package=ggplot2>
- Wilkinson L. (2005). *The grammar of graphics* (2nd ed.). New-York : Springer-Verlag. Retrieved from <https://www.springer.com/us/book/9780387245447>