

# **TP de Biométrie Semestre 3**

Benoît Simon-Bouhet

2022-09-01T00:00:00+02:00

# Table des matières

<b>Introduction</b>	<b>5</b>
Objectifs . . . . .	5
Organisation . . . . .	6
Volume de travail . . . . .	6
Modalités d’enseignement . . . . .	7
Utilisation de Slack . . . . .	8
Progression conseillée . . . . .	9
Évaluation(s) . . . . .	10
<b>1 R et RStudio : les bases</b>	<b>11</b>
1.1 Que sont R et RStudio ? . . . . .	12
1.1.1 Installation . . . . .	12
1.1.2 Utiliser R depuis RStudio . . . . .	13
1.2 Comment exécuter du code R ? . . . . .	14
1.2.1 La console . . . . .	14
<b>2 Les risques du “copier-coller”</b>	<b>15</b>
2.0.1 Le répertoire de travail CHANGER POUR LES RPROJECTS . . . . .	15
2.0.2 Les scripts . . . . .	16
2.0.3 Concepts de base en programmation et terminologie . . . . .	19
2.1 Les packages additionnels . . . . .	27
2.1.1 Installation d’un package . . . . .	28
2.1.2 Charger un package en mémoire . . . . .	28
2.2 Exercices . . . . .	29
<b>3 Explorez votre premier jeu de données</b>	<b>30</b>
3.1 Le package <code>nycflights13</code> . . . . .	31
3.2 Le data frame <code>flights</code> . . . . .	31
3.3 Explorer un <code>data.frame</code> . . . . .	33
3.3.1 <code>View()</code> . . . . .	33
3.3.2 <code>glimpse()</code> . . . . .	34
3.3.3 L’opérateur <code>\$</code> . . . . .	35
3.3.4 <code>skim()</code> . . . . .	36
3.3.5 Les fichiers d’aide . . . . .	36
3.4 Exercices . . . . .	37

<b>4</b>	<b>Visualiser des données avec ggplot2</b>	<b>38</b>
4.1	Prérequis	39
4.2	La grammaire des graphiques	40
4.2.1	Éléments de la grammaire	40
4.2.2	Gapminder	40
4.2.3	Autres éléments de la grammaire des graphiques	41
4.2.4	Le package <code>ggplot2</code>	42
4.3	Les nuages de points	43
4.3.1	La couche de base : la fonction <code>ggplot()</code>	44
4.3.2	Ajout d'une couche supplémentaire : l'objet géométrique	44
4.3.3	Exercices	46
4.3.4	Over-plotting	46
4.3.5	Couleur, taille et forme	50
4.3.6	Exercices	59
4.4	Les graphiques en lignes	60
4.4.1	Un nouveau jeu de données	60
4.4.2	Exercice	61
4.4.3	La fonction <code>geom_line()</code>	61
4.4.4	À quel endroit placer <code>aes()</code> et les arguments <code>color</code> , <code>size</code> , etc. ?	66
4.5	Les histogrammes	71
4.5.1	L'objet <code>geom_histogram()</code>	71
4.5.2	La taille des classes	73
4.6	Les <code>facets</code>	79
4.6.1	<code>facet_wrap()</code>	79
4.6.2	<code>facet_grid()</code>	81
4.6.3	Exercices	83
4.7	Les boîtes à moustaches ou boxplots	83
4.7.1	Création de boxplots et informations apportées	83
4.7.2	L'intervalle de confiance à 95% de la médiane	86
4.7.3	Une autre façon d'examiner des distributions	87
4.7.4	Pour conclure	88
4.8	Les diagrammes bâtons	88
4.8.1	Représentation graphique avec <code>geom_bar</code> et <code>geom_col</code>	89
4.8.2	Un exemple concret	92
4.8.3	Exercices	96
4.8.4	Éviter à tout prix les diagrammes circulaires	96
4.8.5	Comparer 2 variables catégorielles avec un diagramme bâton	98
4.9	De l'exploration à l'exposition	102
4.9.1	Les labels	102
4.9.2	Les échelles	104
4.9.3	Les thèmes	120
4.10	Exercices	127



# Introduction

## Objectifs

Ce livre contient l'ensemble du matériel (contenus, exemples, exercices...) nécessaire à la réalisation des travaux pratiques de **Biométrie** de l'EC '*Outils pour l'étude et la compréhension du vivant 2*' du semestre 3 de la licence Sciences de la Vie de La Rochelle Université.

Les trois grands chapitres de ce livre correspondent aux 3 objectifs principaux de ces séances de TP et TEA :

1. **Vous faire découvrir les logiciels R et Rstudio** dans lesquels vous allez passer beaucoup de temps en licence 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 des Sciences de la Vie !) 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 apprendre à faire des graphiques de qualités dans RStudio et vous faire prendre conscience de l'importance des visualisations graphiques :**
  - d'une part, pour explorer des données inconnues et vous faire une première idée des informations qu'elles contiennent,
  - 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).
3. **Vous apprendre à manipuler efficacement des tableaux de données de grande taille.** Cela signifie que vous devriez être mesure de sélectionner des variables (colonnes) d'un tableau, d'en créer de nouvelles en modifiant et/ou combinant des variables existantes, de filtrer des lignes spécifiques, etc.

À l'issue de ces TP et TEA, vous devriez donc être suffisamment à l'aise avec le logiciel **RStudio** pour y importer des données issues de tableurs, les manipuler pour les mettre dans un format permettant les représentations graphiques, et pour produire des graphiques pertinents, adaptés aux données dont vous disposez, et d'une qualité vous permettant de les intégrer sans honte à vos compte-rendus de TP et rapports de stages.

D'ailleurs, 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 manipulation de données dans **RStudio** et réalisation de visualisations graphiques parlantes.

Dernière choses : à partir de maintenant, tous les compte-rendus de TP que vous aurez à produire dans le cadre de la licence SV devront respecter les bonnes pratiques décrites dans ce document. En particulier, les collègues de l'équipe pédagogique attendent en effet que les graphiques que vous intégrerez à vos compte-rendus de TP soient systématiquement produits dans **RStudio**.

---

## Organisation

### Volume de travail

Les travaux pratiques et TEA de biométrie auront lieu entre le 12 septembre et le 07 octobre 2022 :

- Semaine 37 (du 12 au 16 septembre) : 1 séance de TP d'1h30 et 1 séance de TEA d'1h30
- Semaine 38 (du 19 au 23 septembre) : 1 séance de TP d'1h30 et 1 séance de TEA d'1h30
- Semaine 39 (du 26 au 30 septembre) : 1 séance de TP d'1h30
- Semaine 40 (du 03 au 07 octobre) : 1 séance de TP d'1h30 et 1 séance de TEA d'1h30

**Tous les TP ont lieu en salle MSI 217. Tous les TEA sont à distance.**

Au total, chaque groupe aura donc 4 séances de TP et 2 séances de TEA. C'est très peu pour atteindre les objectifs fixés et il y aura donc évidemment du travail personnel à fournir en dehors des heures prévues à l'emploi du temps. Pour chaque séance de TP ou TEA prévue à l'emploi du temps, j'estime qu'une à deux heures de travail personnel est nécessaire (soit 9 à 15 heures au total, selon votre degré d'aisance, à répartir sur 4 semaines). Attention donc : pensez bien à prévoir du temps dans vos plannings car le travail personnel est essentiel pour progresser dans cette matière. J'insiste sur l'importance de faire l'effort dès maintenant : vous aller en effet avoir des enseignements qui reposent sur l'utilisation de ces logiciels à chaque semestre de la licence du S3 au S6. C'est maintenant qu'il faut acquérir des automatismes, cela vous fera gagner énormément de temps ensuite.

## Modalités d'enseignement

Pour suivre cet enseignement vous pourrez utiliser les ordinateurs de l'université, mais je ne peux que vous encourager à utiliser vos propres ordinateurs, sous Windows, Linux ou MacOS. Lors de vos futurs stages et pour rédiger vos comptes-rendus de TP, vous utiliserez le plus souvent vos propres ordinateurs, autant prendre dès maintenant de bonnes habitudes en installant les logiciels dont vous aurez besoin tout au long de votre licence. Si vous ne possédez pas d'ordinateur, manifestez vous rapidement auprès de moi car des solutions existent (prêt par l'université, travail sur tablette via [RStudio cloud...](#)).

L'essentiel du contenu de cet enseignement peut être abordé en autonomie, à distance, grâce à ce livre en ligne, aux ressources mises à disposition sur Moodle et à votre ordinateur personnel. Cela signifie que **la présence physique lors de ces séances de TP n'est pas obligatoire**.

Plus que des séances de TP classiques, considérez plutôt qu'il s'agit de **permanences non-obligatoires** : si vous pensez avoir besoin d'aide, si vous avez des points de blocage ou des questions sur le contenu de ce document, sur les exercices demandés ou sur les quizz Moodle, alors venez poser vos questions lors des séances de TP. Vous ne serez d'ailleurs pas tenus de rester pendant 1h30 : si vous obtenez une réponse en 10 minutes et que vous préférez travailler ailleurs, vous serez libres de repartir !

De même, si vous n'avez pas de difficulté de compréhension, que vous n'avez pas de problème avec les exercices ou les quizz Moodle, votre présence n'est pas requise. Si vous souhaitez malgré tout venir en salle de TP, pas de problème, vous y serez toujours les bienvenus.

Ce fonctionnement très souple a de nombreux avantages :

- vous vous organisez comme vous le souhaitez
- vous ne venez que lorsque vous en avez vraiment besoin
- celles et ceux qui se déplacent reçoivent une aide personnalisée
- vous travaillez sur vos ordinateurs
- les effectifs étant réduits, c'est aussi plus confortable pour moi !

Toutefois, pour que cette organisation fonctionne, cela demande de la rigueur de votre part, en particulier sur la régularité du travail que vous devez fournir. Si la présence en salle de TP n'est pas requise, le travail demandé est bel et bien obligatoire ! Si vous venez en salle de TP sans avoir travaillé en amont, votre venue sera totalement inutile puisque vous n'aurez pas de question à poser et que vous passerez votre séance à lire ce livre en ligne. Vous perdrez donc votre temps, celui de vos collègues, et le mien. De même, si vous attendez la 4e semaine pour vous y mettre, vous irez droit dans le mur. Je le répète, outre les 9h de TP/TEA prévus dans vos emplois du temps, vous devez prévoir entre 9 et 15 heures de travail personnel supplémentaire.

Je vous laisse donc une grande liberté d'organisation. À vous d'en tirer le maximum et de faire preuve du sérieux nécessaire. Le rythme auquel vous devriez avancer est présenté dans la partie suivante intitulée "Progression conseillée".

## Utilisation de Slack

Outre les séances de permanence non-obligatoires, nous échangerons aussi sur [l'application Slack](#), qui fonctionne un peu comme un "twitter privé". Slack facilite la communication des équipes et permet de travailler ensemble. Créez-vous un compte en ligne et installez le logiciel sur votre ordinateur (il existe aussi des versions pour tablettes et smartphones). Lorsque vous aurez installé le logiciel, [cliquez sur ce lien](#) pour vous connecter à notre espace de travail commun intitulé **L2 SV 22-23 / EC outils** (ce lien expire régulièrement : faites moi signe s'il n'est plus valide).

Vous verrez que 2 "chaînes" sont disponibles :

- **#général** : c'est là que les questions liées à l'organisation générale du cours, des TP et TEA doivent être posées. Si vous ne savez pas si une séance de permanence a lieu, posez la question ici.
- **#questions-rstudio** : c'est ici que toutes les questions pratiques liées à l'utilisation de R et RStudio devront être posées. Problèmes de syntaxe, problèmes liés à l'interface, à l'installation des packages ou à l'utilisation des fonctions, à la création des graphiques, à la manipulation des tableaux... Tout ce qui concerne directement les logiciels sera traité ici. Vous êtes libres de poser des questions, de poster des captures d'écran, des morceaux de code, des messages d'erreur. Et **vous êtes bien entendus vivement encouragés à vous entraider et à répondre aux questions de vos collègues**. Je n'interviendrai ici que pour répondre aux questions laissées sans réponse ou si les réponses apportées sont inexactes. Le fonctionnement est celui d'un forum de discussion instantané. Vous en tirerez le plus grand bénéfice en participant et en n'ayant pas peur de poser des questions, même si elles vous paraissent idiotes. Rappelez-vous toujours que si vous vous posez une question, d'autres se la posent aussi probablement.

Ainsi, quand vous travaillerez à vos TP ou TEA, que vous soyez installés chez vous ou en salle de TP, prenez l'habitude de garder Slack ouvert sur votre ordinateur. Même si vous n'avez pas de question à poser, votre participation active pour répondre à vos collègues est souhaitable et souhaitée. Je vous donc fortement à vous **entraider** : c'est très formateur pour celui qui explique, et celui qui rencontre une difficulté a plus de chances de comprendre si c'est quelqu'un d'autre qui lui explique plutôt que la personne qui a rédigé les instructions mal comprises.

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 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. J'essaie néanmoins de rester disponible sur



Slack pendant les séances de TP et de TEA de tous les groupes. Cela veut donc dire que même si votre groupe n'est pas en TP, vos questions ont des chances d'être lues et de recevoir des réponses dès que d'autres groupes sont en TP ou TEA. Vous êtes d'ailleurs encouragés à échanger sur Slack aussi pendant vos phases de travail personnels.

## Progression conseillée

Pour apprendre à utiliser un logiciel comme R, il faut faire les choses soi-même, 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. J'utilise ce logiciel presque quotidiennement depuis plus de 15 ans et à chaque session de travail, je rencontre des messages d'erreur. Avec suffisamment d'habitude, on apprend à les déchiffrer, et on corrige les problèmes en quelques secondes. Ce livre est conçu pour vous faciliter la tâche, mais ne vous y trompez pas, vous rencontrerez des difficultés, et c'est normal. C'est le prix à payer pour profiter de la puissance du meilleur logiciel permettant d'analyser des données, de produire des graphiques de qualité et de réaliser toutes les statistiques dont vous aurez besoin d'ici la fin de vos études et au-delà.

Pour que cet apprentissage soit le moins problématique possible, il convient de prendre les choses dans l'ordre. C'est la raison pour laquelle les chapitres de ce livre doivent être lus dans l'ordre, et les exercices d'application faits au fur et à mesure de la lecture.

Idéalement, voilà les étapes que vous devriez avoir franchi chaque semaine :

1. La première semaine (37) est consacrée à l'installation des logiciels, à la découverte de l'environnement de travail, des RProjects, des packages et des scripts. Avant votre deuxième séance, vous devrez être capables de créer un Rproject et un script, de télécharger et d'installer des packages, d'importer des données dans le logiciel, et vous devrez commencer à connaître les types d'objets principaux (vecteurs, facteurs, data.frames et tibbles). Vous devrez effectuer le premier quizz Moodle (attention, ce quizz est à faire seul !).
2. La deuxième semaine (38) est consacrée à la découverte du package `ggplot2`. Avant votre troisième séance, vous devrez avoir compris la syntaxe de base permettant de faire toutes sortes de graphiques avec `ggplot2`. Vous devrez être capables de choisir des graphiques appropriés selon le nombre et la nature des variables dont vous disposez. À ce stade, on ne demande rien de complexe, mais vous devrez, à minima, être capable de faire des barplots, des histogrammes et des nuages de points. Vous devrez effectuer le second quizz Moodle (attention, ce quizz est à faire seul !).
3. La troisième semaine (39) est consacrée à l'amélioration de la qualité de vos graphiques. Avant votre dernière séance de TP, vous devrez être capable de faire, outre les graphiques de la semaine précédente, des stripcharts, des boîtes à moustaches (boxplots) et des

graphiques en ligne. Vous devrez également être capable de faire des sous-graphiques par catégories (`facet()`), de choisir un thème et des palettes de couleurs appropriées, et de légender/annoter correctement vos graphiques. Vous devrez effectuer le troisième quizz Moodle (attention, ce quizz est à faire seul !).

4. La quatrième semaine (40) est consacrée à la manipulation des tableaux de données. Avant la fin de cette semaine, vous devrez être capable de sélectionner des colonnes dans un tableau, de filtrer des lignes, et de créer de nouvelles variables. Vous devrez être capables d'enchaîner correctement les étapes suivantes : ouvrir le logiciel > créer un projet > créer un script > mettre en mémoire les packages utiles > importer des données > mettre en forme ces données > faire un ou des graphiques informatifs et correctement mis en forme. Vous devrez effectuer le quatrième et dernier quizz Moodle (attention, ce quizz est à faire seul !).

## Évaluation(s)

Vous serez évalués à 3 niveaux :

1. Votre participation sur Slack
2. Les quizz Moodle
3. Une évaluation plus classique

Les exercices demandés dans ce document en ligne ne seront ni ramassés, ni notés : ils sont proposés pour que vous puissiez mettre en application les notions récemment apprises et afin d'évaluer votre propre progression et vos apprentissages. Mais tout ce que nous voyons en TP et TEA devra être acquis le jour de la dernière évaluation.

En particulier, je vérifierai que les étapes décrites précédemment sont acquises (ouvrir le logiciel > créer un projet > créer un script > mettre en mémoire les packages utiles > importer des données > mettre en forme ces données > faire un ou des graphiques informatifs et correctement mis en forme) en vous fournissant un jeu de données inconnu que vous devrez importer et mettre en forme dans `RStudio` afin de produire quelques graphiques informatifs. Cette évaluation aura lieu soit en salle informatique, soit dans le cadre d'un compte-rendu de TP que vous devrez rendre pour un collègue. Si cette modalité est mise en place, je vous en dirai plus en temps utile.

# 1 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** ?

Une bonne maîtrise des éléments présentés dans ce chapitre est indispensable pour aborder sereinement les chapitres suivants, à commencer par le chapitre Chapitre 3, qui présente quelques jeux de données que nous explorerons en détail un peu plus tard. Lisez donc attentivement ce chapitre et faites bien tous les exercices demandés.

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 :

1. 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.

## ! Important

Avant d'aller plus loin, rendez-vous sur [le site de DataCamp](#) et créez-vous un compte gratuit.

## 1.1 Que sont R et RStudio ?

Pour l'ensemble de ces TP, j'attends de vous que vous utilisiez **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 ou "Integrated Development Environment". On peut utiliser **R** sans **RStudio**, mais c'est nettement plus compliqué, nettement moins pratique.

### 1.1.1 Installation

#### Avertissement

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

Avant tout, vous devez télécharger et installer **R**, **puis** **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" (si vous êtes sous Windows) ou sur "R-4.2.1.pkg" (si vous êtes sous Mac), et suivez les instructions.

#### 2. [Téléchargez et installez RStudio](#)

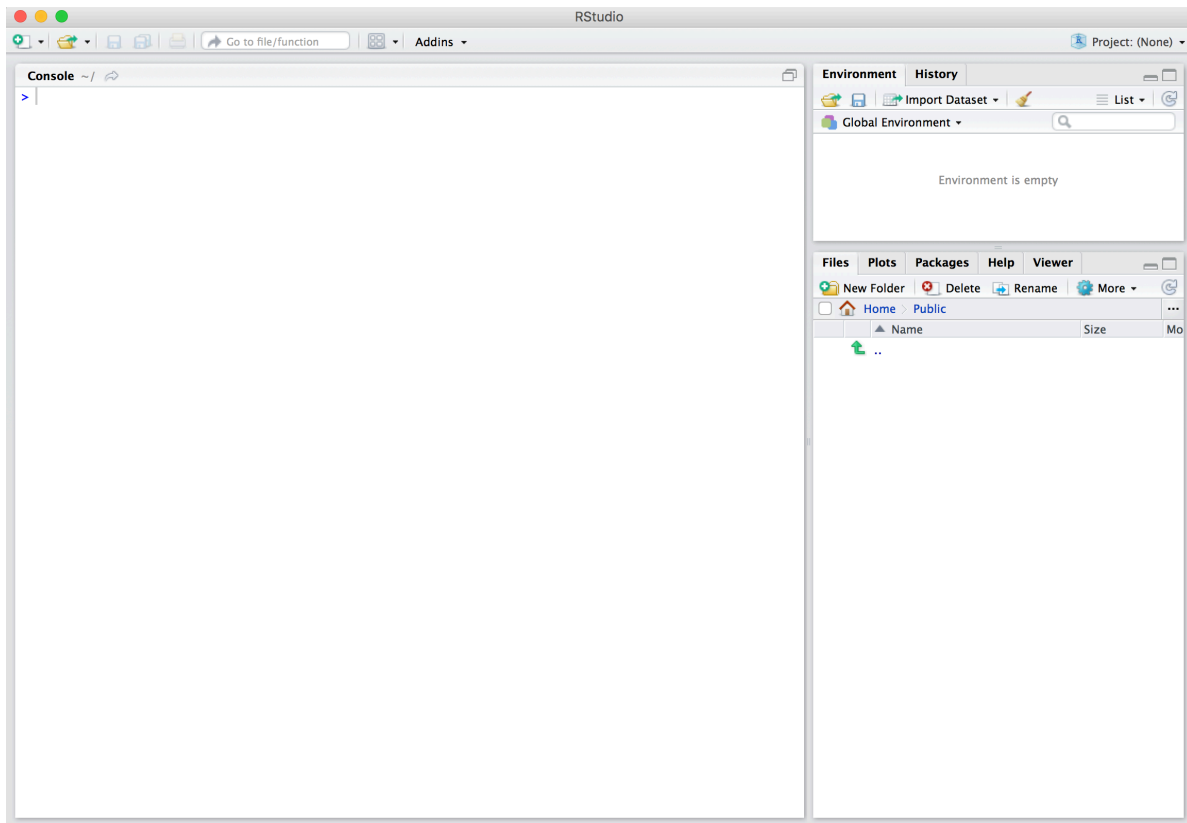
- Cliquez sur "RStudio Desktop", puis sur "Download RStudio Desktop".
- Choisissez la version gratuite et cliquez sur le lien de téléchargement qui correspond à votre système d'exploitation.

### 1.1.2 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 les installations réalisées à la Section 1.1.1, 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.

## 1.2 Comment exécuter du code R ?

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, 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.

### 1.2.1 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 !

Dans la version en ligne de ce livre (en html), à chaque fois que du code R sera fourni, il apparaîtra dans un cadre grisé avec une ligne bleue à gauche, comme ci-dessus. Vous pourrez toujours taper dans RStudio, les commandes qui figurent dans ce livre, afin d'obtenir vous même les résultats souhaités. Dans ce livre, lorsque les commandes R produisent des résultats, ils sont affichés juste en dessous des blocs de code. Enfin, en passant la souris sur les blocs de code, vous verrez apparaître une icône de presse-papier qui vous permettra de copier-coller les commandes du livre dans la console de RStudio ou, très bientôt, dans vos scripts.

## 2 Les risques du “copier-coller”

Attention : il est fortement conseillé de réserver les copier-coller aux blocs de commandes de (très) grande taille, ou en cas d’erreur de syntaxe inexplicable. L’expérience a en effet montré qu’**on apprend beaucoup mieux en tapant soi-même les commandes**. Ça n’est que comme cela que l’on peut prendre conscience de toutes les subtilités du langage (par exemple, faut-il mettre une virgule ou un point, une parenthèse ou un crochet, le symbole moins ou un tilde, etc.). Je vous conseille donc de taper vous-même les commandes autant que possible.

### 2.0.1 Le répertoire de travail CHANGER POUR LES RPROJECTS

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 souhaitez 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** !

## 2.0.2 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'aurez 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
```

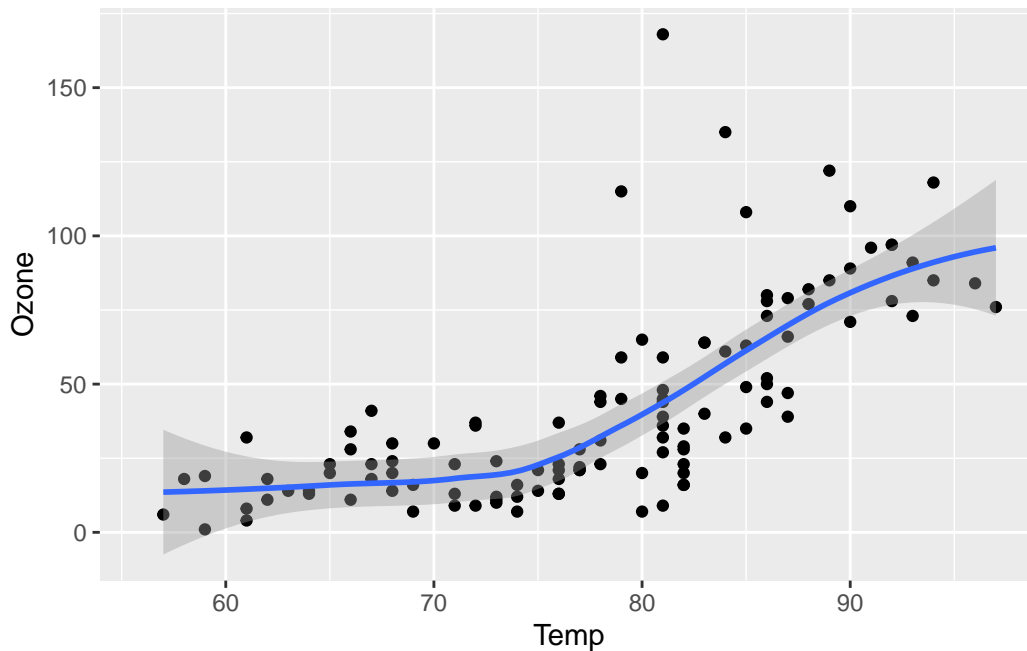
Même si vous ne comprenez pas encore les commandes qui figurent dans ce script (ça viendra !), voici ce que vous devez en retenir :

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 ...
```



### 2.0.3 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.

#### 2.0.3.1 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 [@ref\(sec-dataset\)](#).

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.

### 2.0.3.2 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
- `!=` : différent de
- `>` : supérieur à
- `<` : inférieur à
- `>=` : supérieur ou égal à
- `<=` : 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 certaine 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`: la taille d'un objet plus long n'est pas multiple de la taille d'un objet plus court

```
[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.

### 2.0.3.3 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 ?
<code>c()</code>	Créer des vecteurs
<code>class()</code>	Afficher ou modifier la classe d'un objet
<code>factor()</code>	Créer des facteurs
<code>getwd()</code>	Afficher le chemin du répertoire de travail
<code>head()</code>	Afficher les premiers éléments d'un objet
<code>is.na()</code>	Tester si un objet contient des valeurs manquantes
<code>mean()</code>	Calculer une moyenne
<code>names()</code>	Afficher ou modifier le nom des éléments d'un vecteur
<code>order()</code>	Ordonner les éléments d'un objet
<code>setwd()</code>	Modifier le chemin du répertoire de travail
<code>subset()</code>	Extraire une partie des éléments d'un objet
<code>sum()</code>	Calculer une somme
<code>tail()</code>	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
```

---

## 2.1 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 utiliserons fréquemment :

- Le package **ggplot2** pour la visualisation des données dans le chapitre @ref(sec-viz)
- Le package **dplyr** pour manipuler des tableaux de données dans le chapitre @ref(sec-wrangling)

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.

### 2.1.1 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.

### 2.1.2 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()`.

## 2.2 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 @ref(sec-objects) : 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 @ref(sec-packages) 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)
```

Attachement du package : 'dplyr'

Les objets suivants sont masqués depuis 'package:stats':

```
filter, lag
```

Les objets suivants sont masqués depuis 'package:base':

```
intersect, setdiff, setequal, union
```

```
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’atterrissage 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_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	1	1	517	515	2	830	819	11	UA
2	2013	1	1	533	529	4	850	830	20	UA
3	2013	1	1	542	540	2	923	850	33	AA
4	2013	1	1	544	545	-1	1004	1022	-18	B6
5	2013	1	1	554	600	-6	812	837	-25	DL

```

6 2013      1      1      554      558      -4      740      728      12 UA
7 2013      1      1      555      600      -5      913      854      19 B6
8 2013      1      1      557      600      -3      709      723     -14 EV
9 2013      1      1      557      600      -3      838      846      -8 B6
10 2013     1      1      558      600      -2      753      745       8 AA
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay

```

Essayons de décrypter 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
- `<dtm>` : 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 tableau 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, 2013, 2~
$ month     <int> 1, 1, 1, 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, 1, 1, 1~
$ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
$ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
$ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
$ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
$ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
$ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
$ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
$ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
$ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

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 impressionnants 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 @ref(sec-bases) et @ref(sec-dataset), 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 @ref(sec-code) 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 @ref(sec-tidyr) et @ref(sec-wrangling), 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)
```

Attachement du package : 'dplyr'

Les objets suivants sont masqués depuis 'package:stats':

`filter`, `lag`

Les objets suivants sont masqués depuis 'package:base':

`intersect`, `setdiff`, `setequal`, `union`

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'équilibriste : 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 “**g**rammar of **g**raphics” : 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 4.1: 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 @ref(sec-fig:gapmind) 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 :

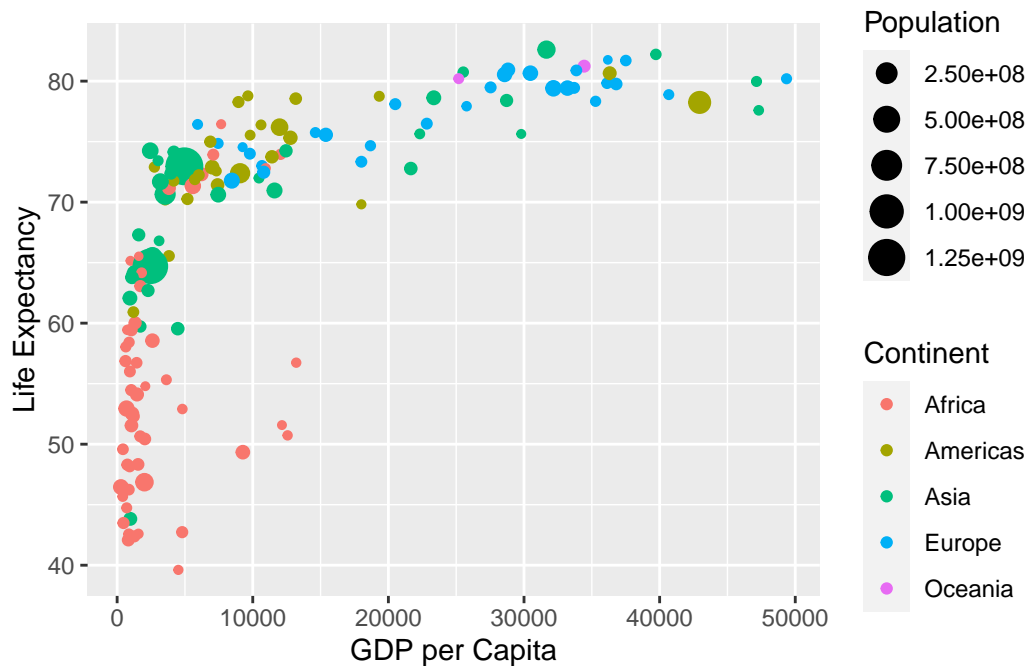


Figure 4.1: Espérance de vie en fonction du PIB par habitant en 2007.

- **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. 2022) 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 @ref(sec-gggraph), 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 @ref(sec-fig:gapmind) 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'atterrissage, 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 @ref(sec-wrangling). 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))
```

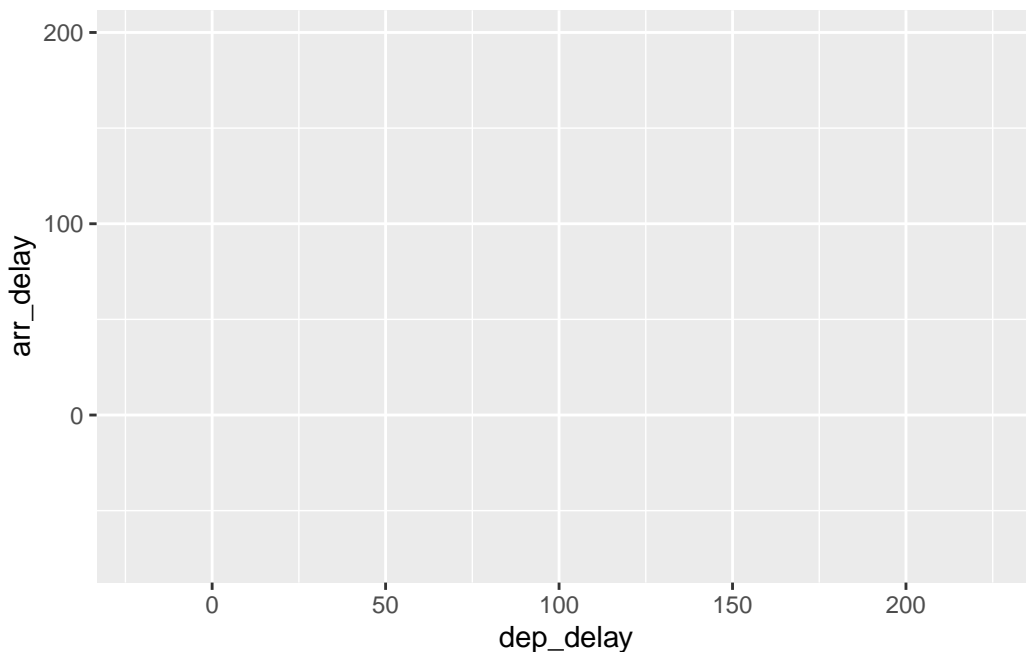


Figure 4.2: Un graphique sans `geom`.

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.

### 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).

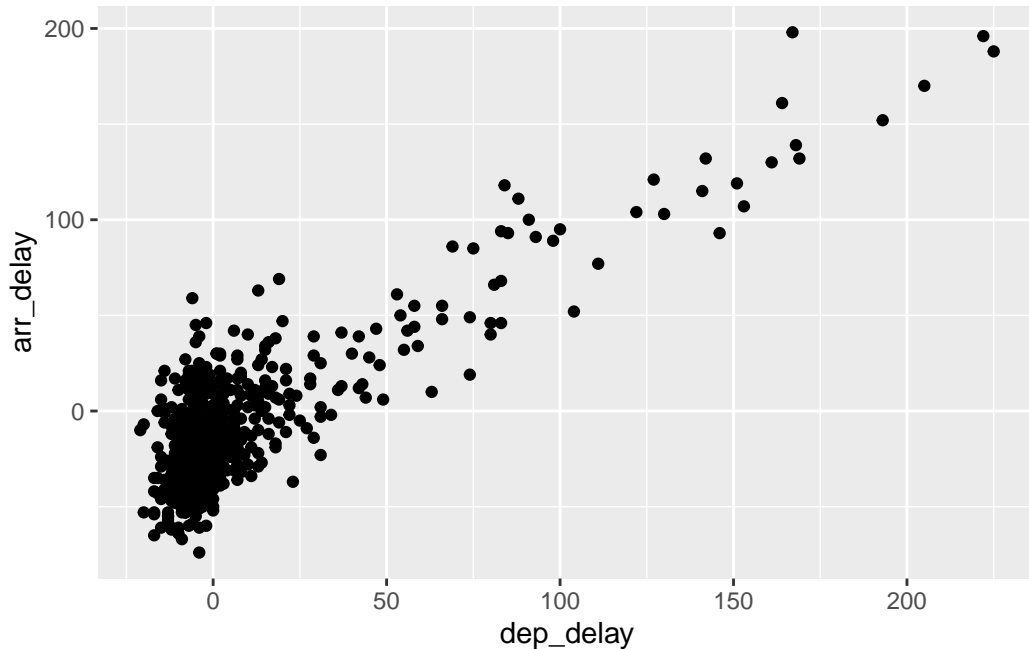


Figure 4.3: Retards à l'arrivée en fonction des retards au décollage pour les vols d'Alaska Airlines au départ de New York City en 2013.

Plusieurs choses importantes sont à remarquer sur la figure @ref(sec-fig:points) :

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 @ref(sec-fonctions), 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 @ref(sec-fig:transparent) :

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

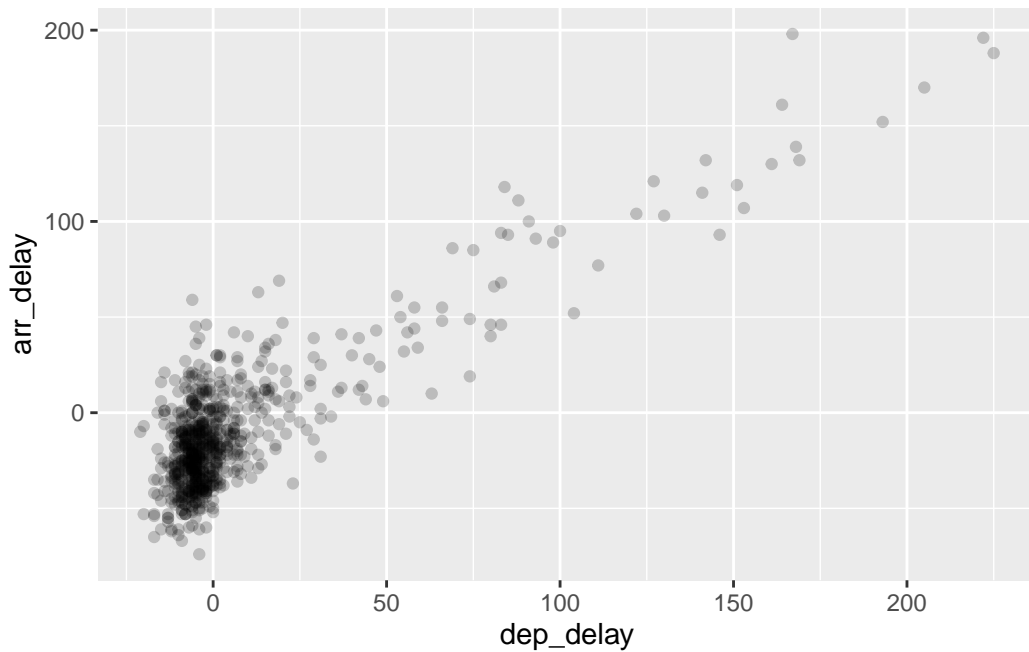


Figure 4.4: La même figure, avec des points semi-transparentes.

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'atterrissage et le mois de l'année :

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

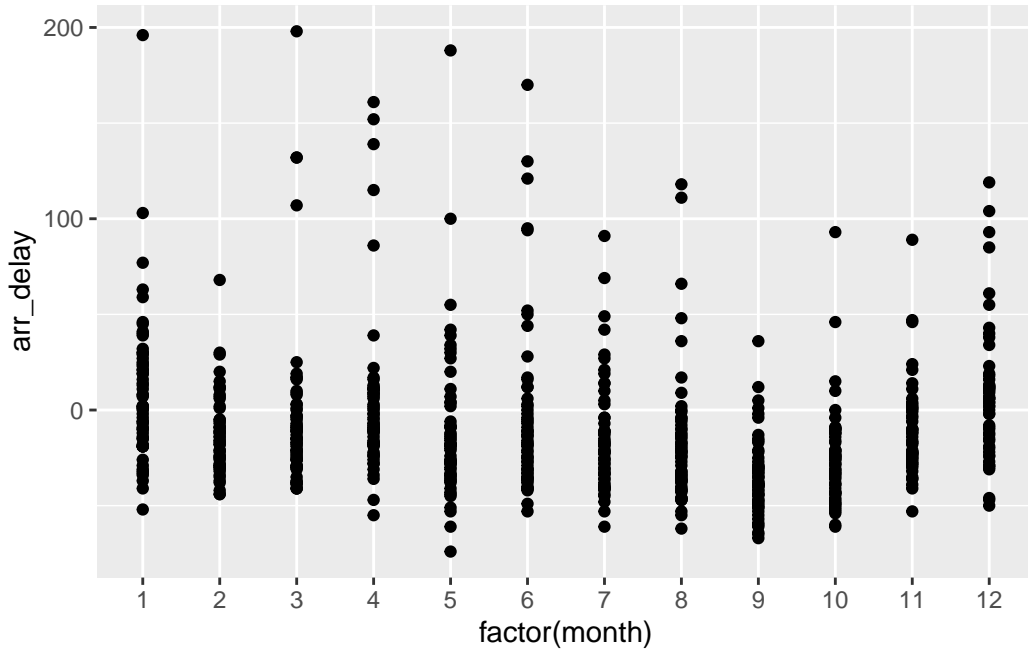


Figure 4.5: Retards à l'arrivée pour les 12 mois de l'année 2013.

Ici (figure @ref(sec-fig:transpfactor)), 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)
```



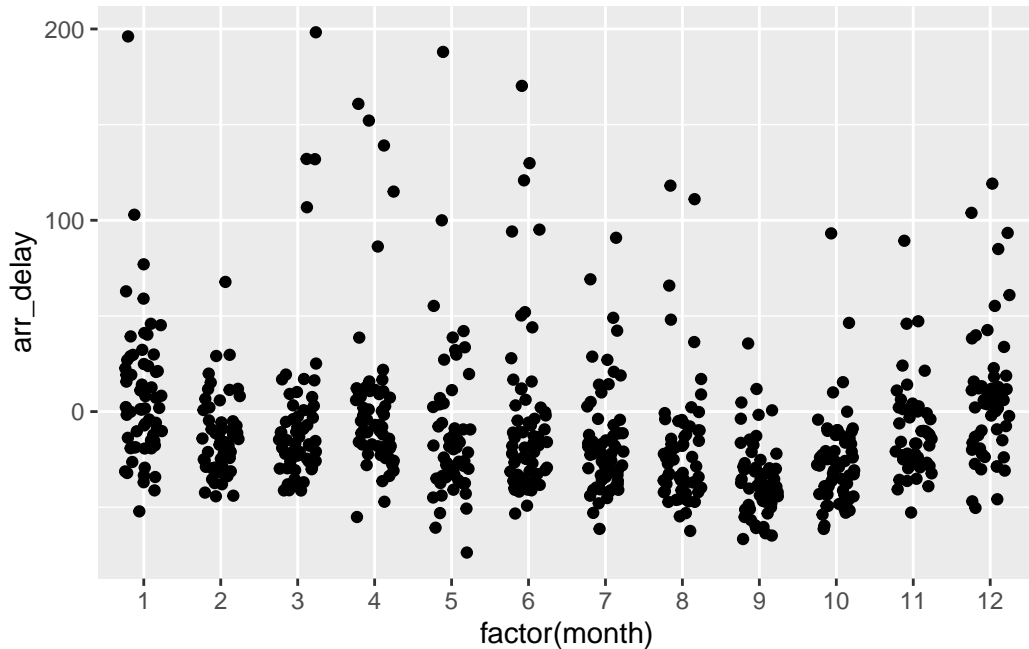


Figure 4.6: Retards à l'arrivée pour les 12 mois de l'année 2013.

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 @ref(sec-fig:jittering) est parfois appelé un “strip-chart”. 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 :

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

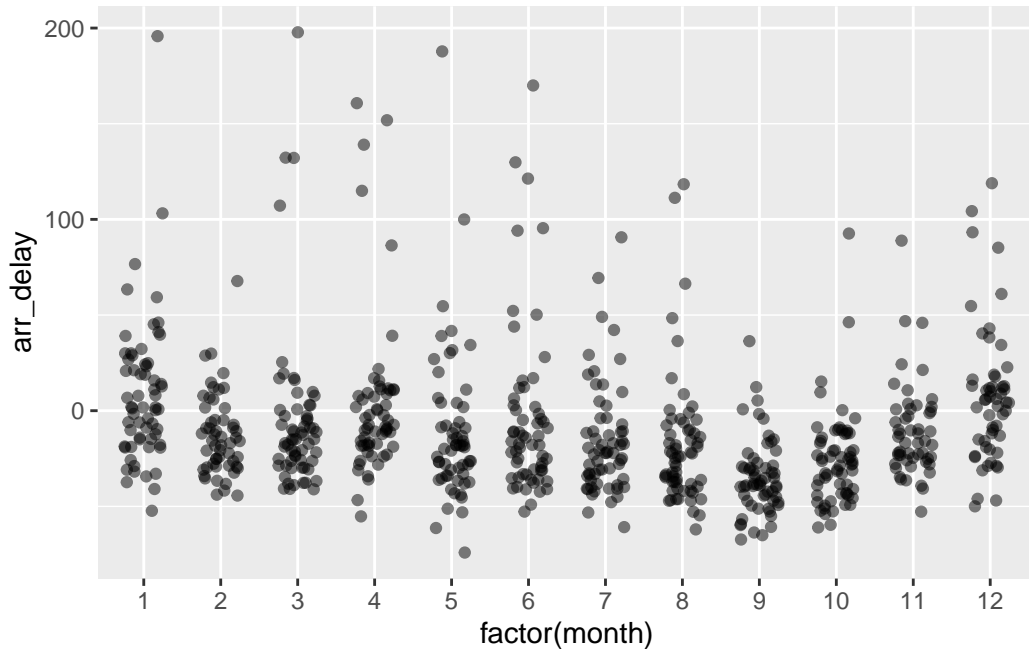


Figure 4.7: Retards à l'arrivée pour les 12 mois de l'année 2013.

### 4.3.5 Couleur, taille et forme

L'argument `color` (ou `colour`, les deux orthographes 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 [@ref\(sec-fig:rightcolor\)](#) et [@ref\(sec-fig:wrongcolor\)](#)) :

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

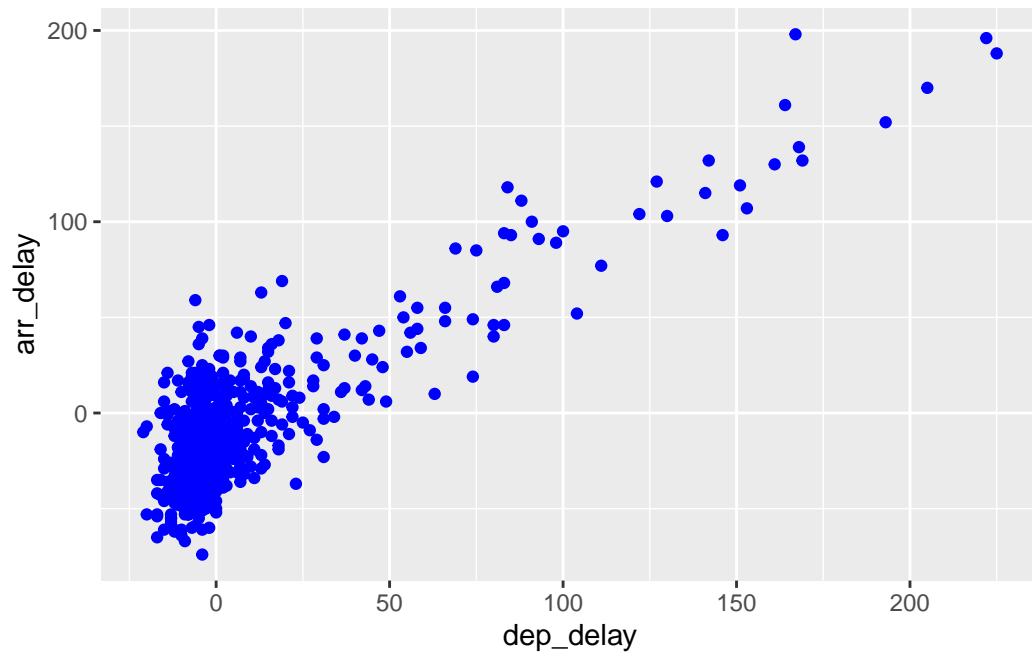


Figure 4.8: Utilisation correcte de color.

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

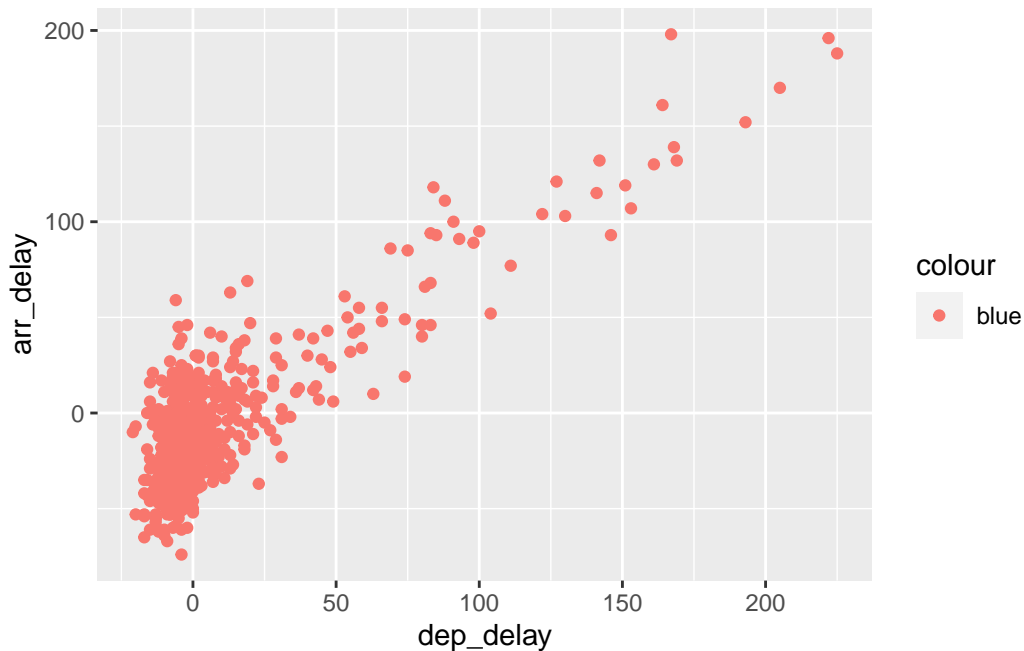


Figure 4.9: Utilisation incorrecte de `color`.

Le code qui permet de produire la figure @ref(sec-fig:rightcolor) fait un usage correct de l'argument `color`. On demande des points de couleur bleue, les points apparaissent bleus. La figure @ref(sec-fig:wrongcolor) 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)))
```

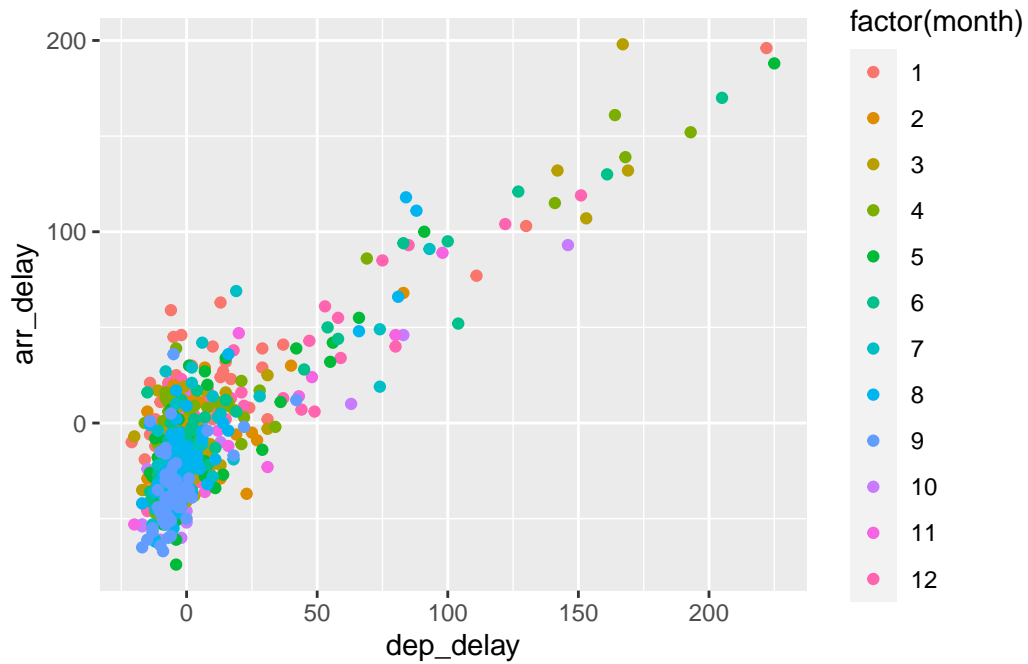


Figure 4.10: Association de color à une variable catégorielle.

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))
```

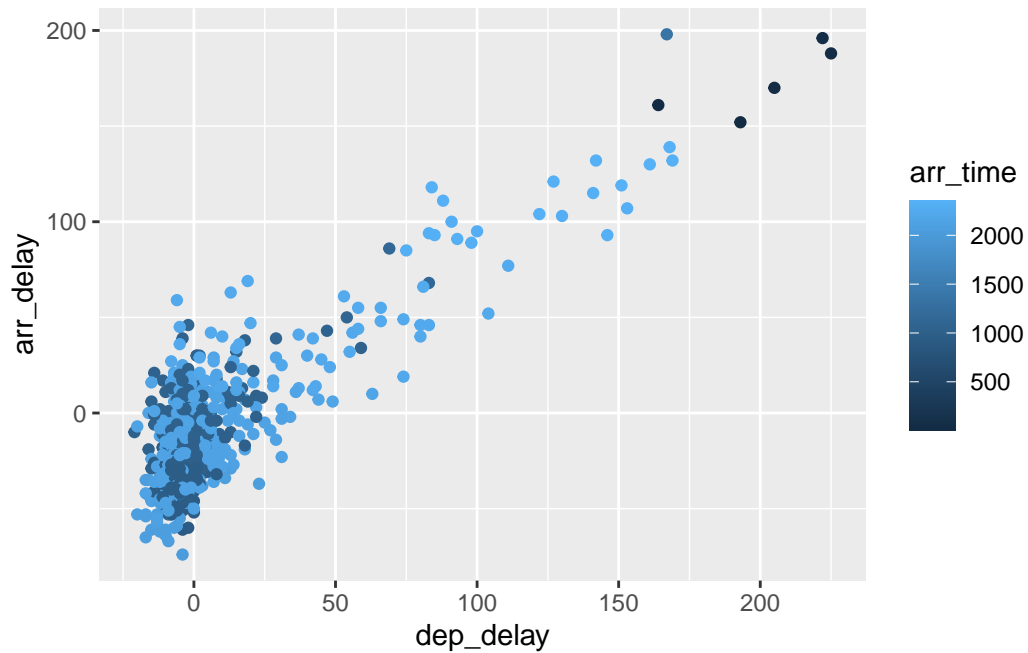


Figure 4.11: Association de `color` à une variable numérique.

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.

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)))
```

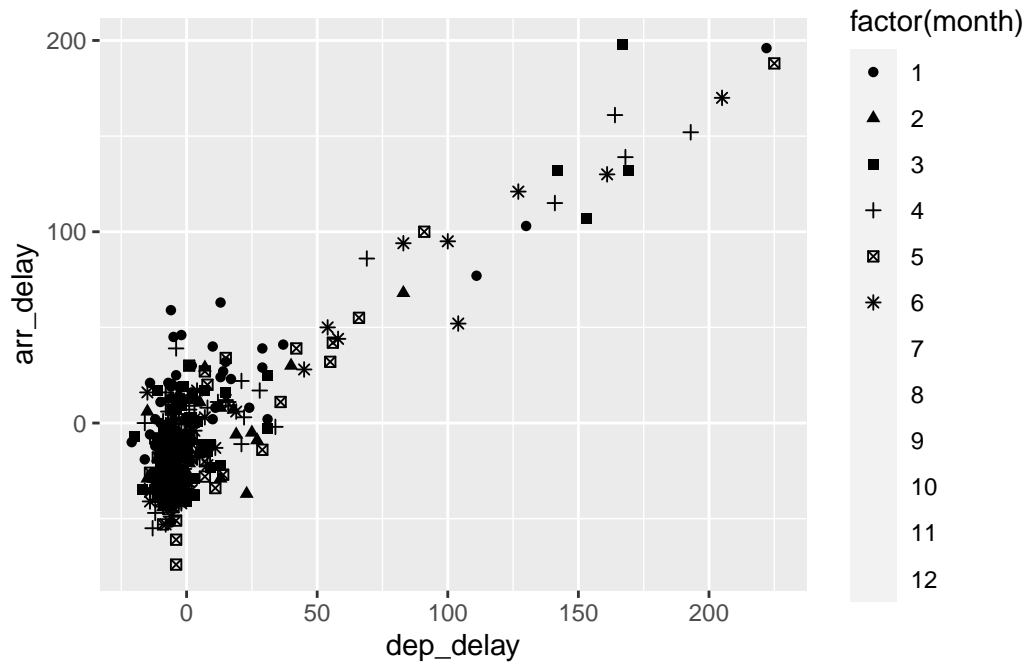


Figure 4.12: Association de `shape` à un facteur.

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)
```

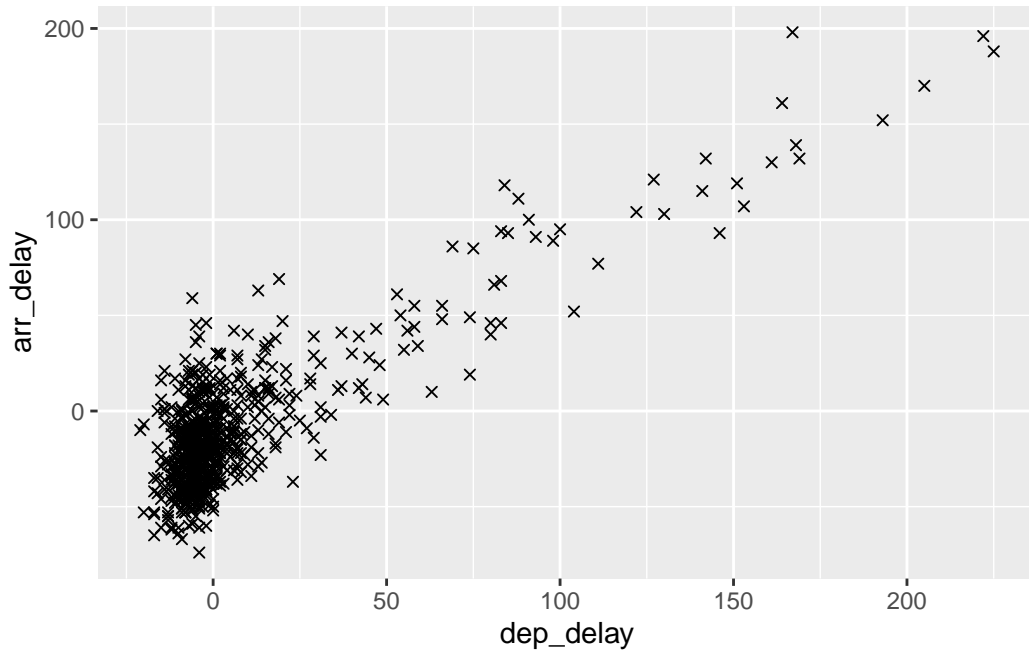


Figure 4.13: Utilisation de `shape` en tant que paramètre.

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)
```



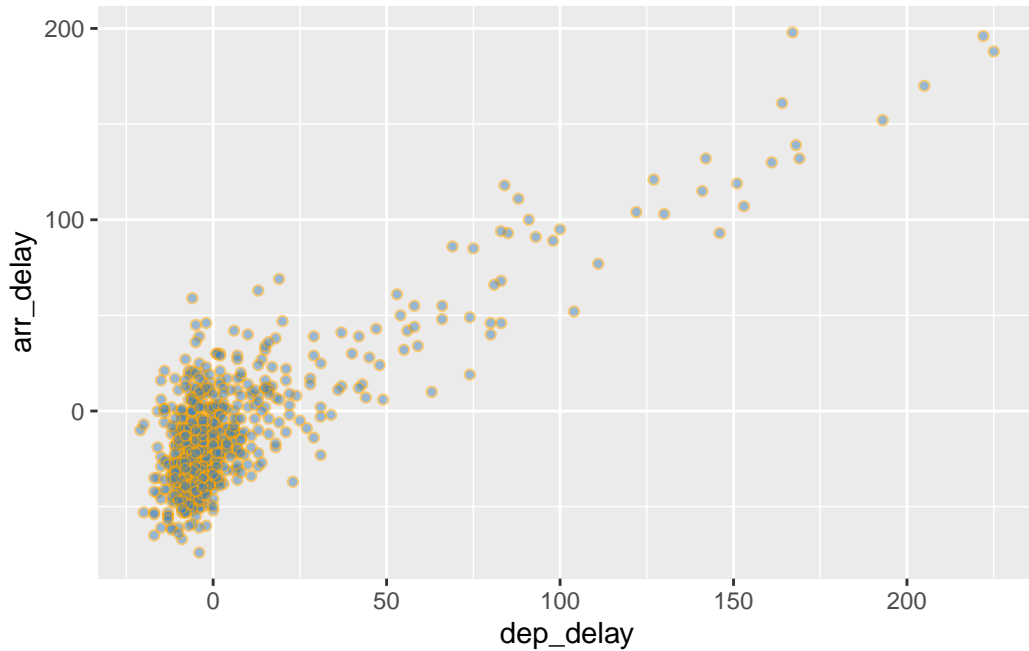


Figure 4.14: Utilisation de `shape`, `color` et `fill`.

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)
```

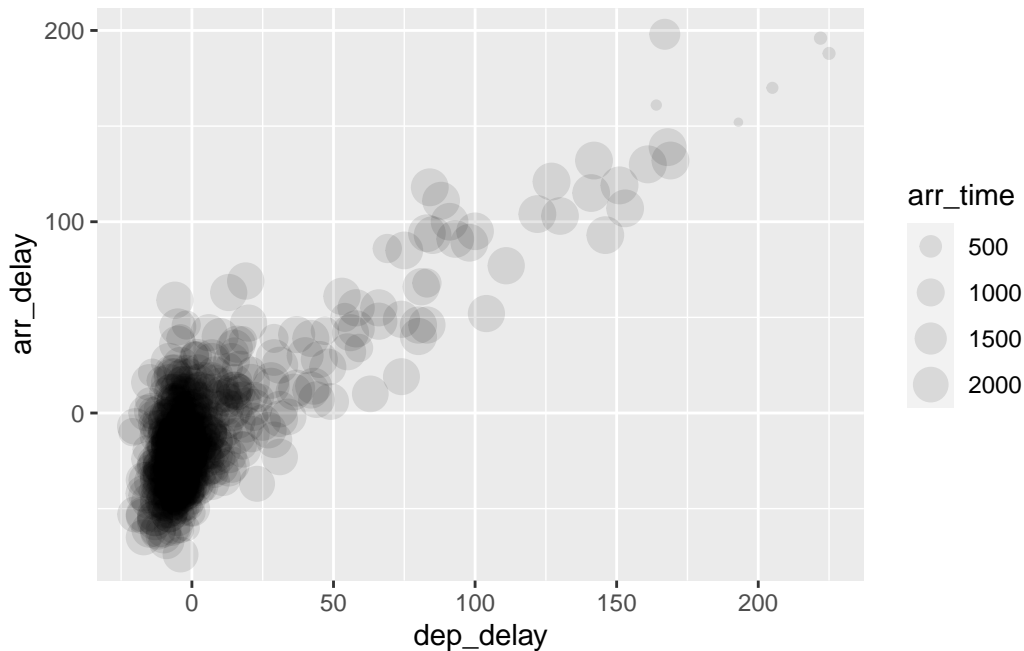


Figure 4.15: Association d’une variable continue à la taille des symboles avec l’argument `size`.

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)
```

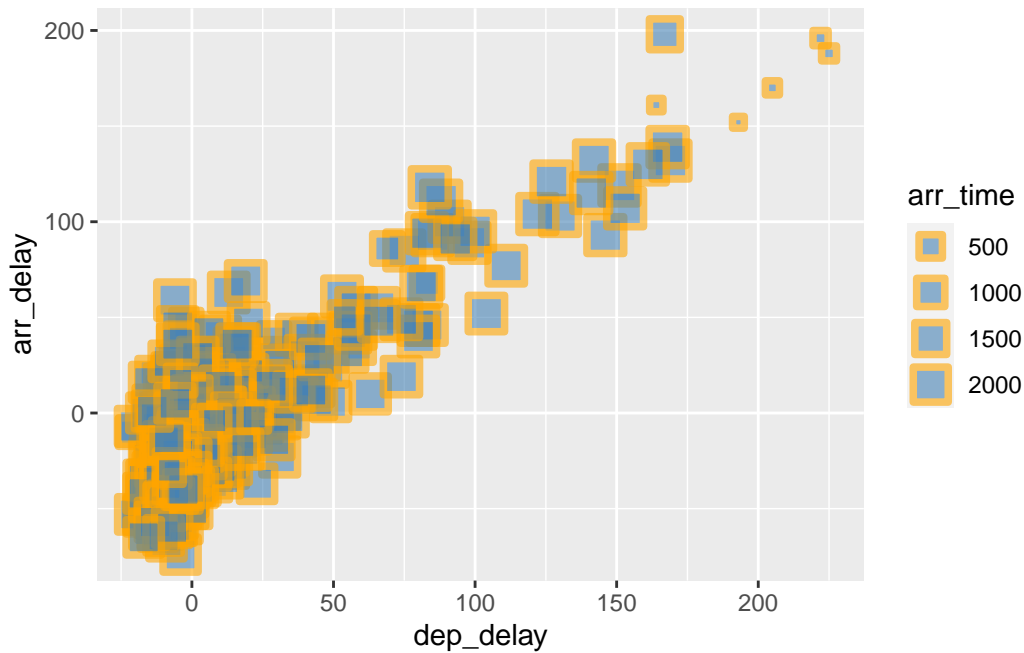


Figure 4.16: Sometimes, less is more!

### 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 @ref(sec-fig:exodiamonds) (Indice : affichez le tableau `diams` dans la console afin de voir quelles sont les variables disponibles).
4. Selon vous, à quoi sont dues les bandes verticales que l'on observe sur ce graphique ?

---

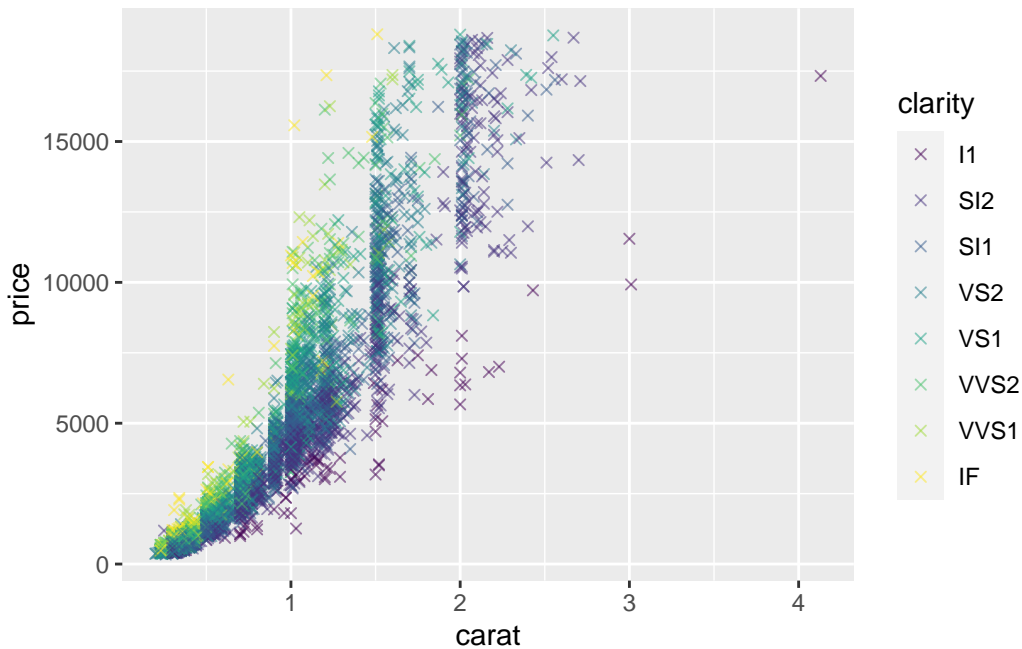


Figure 4.17: Prix de 5000 diamants en fonction de leur taille en carats et de leur clarté.

## 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 @ref(sec-dataset). 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	wind_speed	wind_g~1
	<chr>	<int>	<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	EWR	2013	1	1	1	39.0	26.1	59.4	270	10.4	NA
2	EWR	2013	1	1	2	39.0	27.0	61.6	250	8.06	NA

3	EWR	2013	1	1	3	39.0	28.0	64.4	240	11.5	NA
4	EWR	2013	1	1	4	39.9	28.0	62.2	250	12.7	NA
5	EWR	2013	1	1	5	39.0	28.0	64.4	260	12.7	NA
6	EWR	2013	1	1	6	37.9	28.0	67.2	240	11.5	NA
7	EWR	2013	1	1	7	39.0	28.0	64.4	240	15.0	NA
8	EWR	2013	1	1	8	39.9	28.0	62.2	250	10.4	NA
9	EWR	2013	1	1	9	39.9	28.0	62.2	260	15.0	NA
10	EWR	2013	1	1	10	41	28.0	59.6	260	13.8	NA

```
# ... with 26,105 more rows, 4 more variables: precip <dbl>, pressure <dbl>,
#   visib <dbl>, time_hour <dtm>, and abbreviated variable name 1: wind_gust
```

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()
```

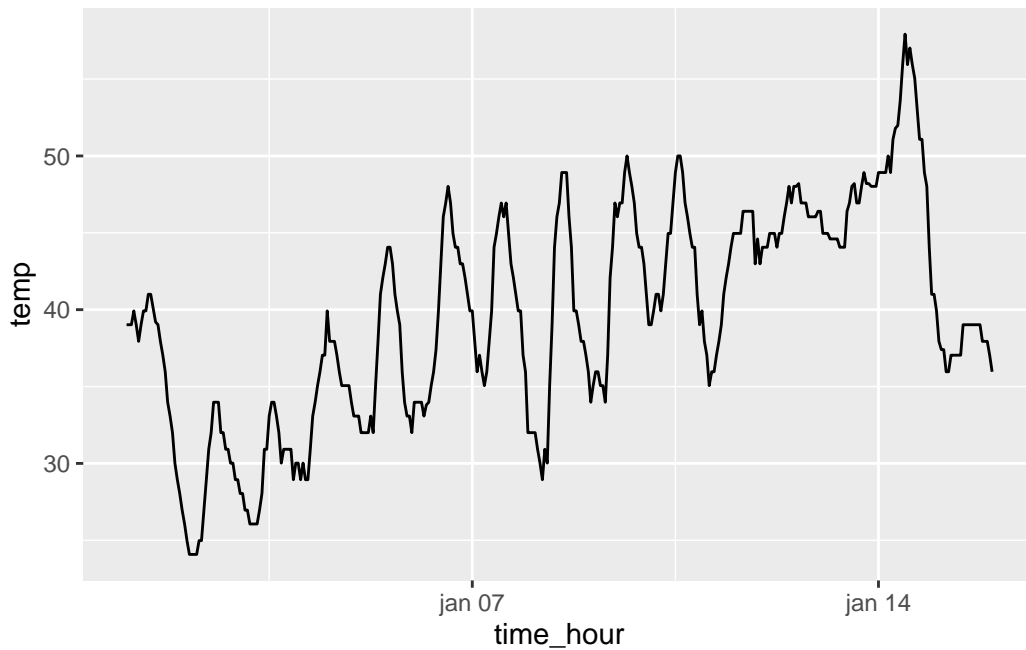


Figure 4.18: Températures horaires à l’aéroport de Newark entre le 1er et le 15 janvier 2013.

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 Farenheit.

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 @ref(sec-fig:lineplotgraph)) :

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

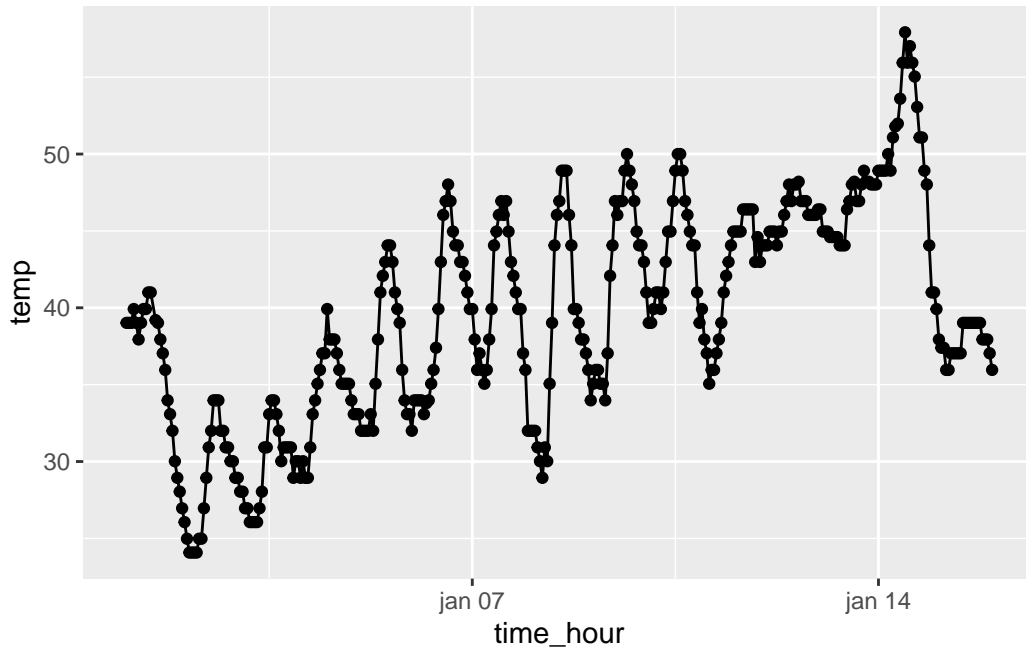


Figure 4.19: Températures horaires à l’aéroport de Newark entre le 1er et le 15 janvier 2013.

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.

Reprenons 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)
```

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 @ref(sec-fig:linecolor)) :

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

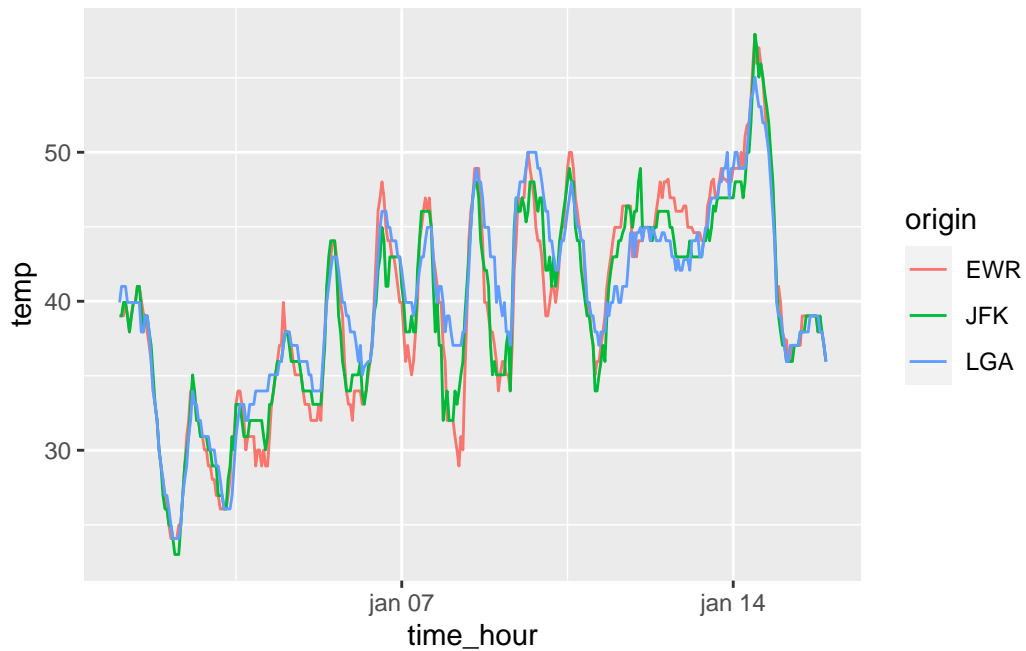


Figure 4.20: Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

Ou bien (figure @ref(sec-fig:linetype)) :

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



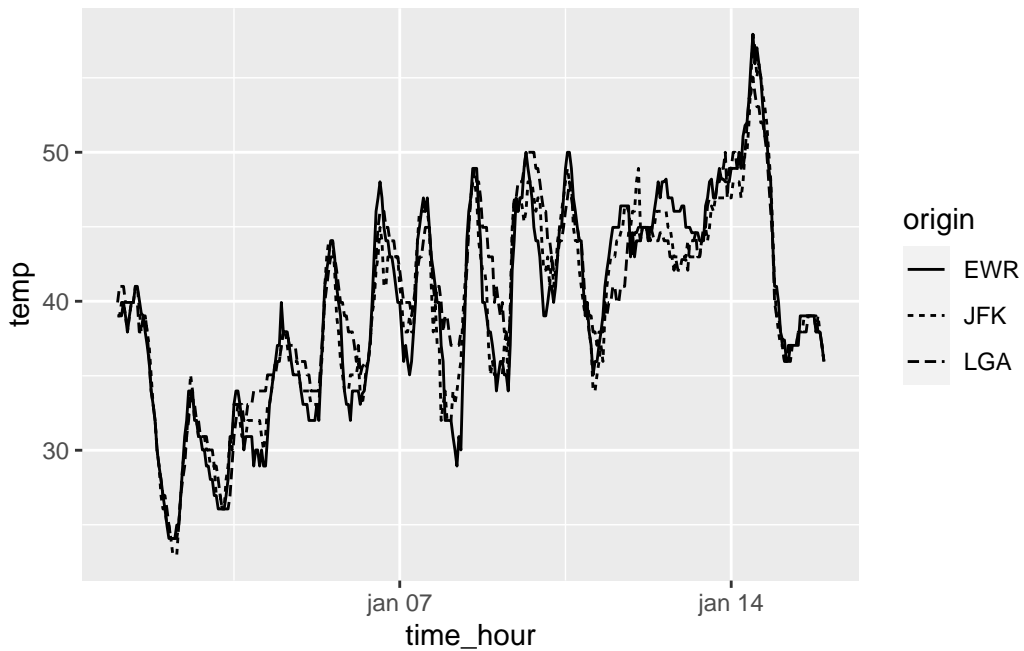


Figure 4.21: Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

Ou encore (figure @ref(sec-fig:linetypecolor)) :

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

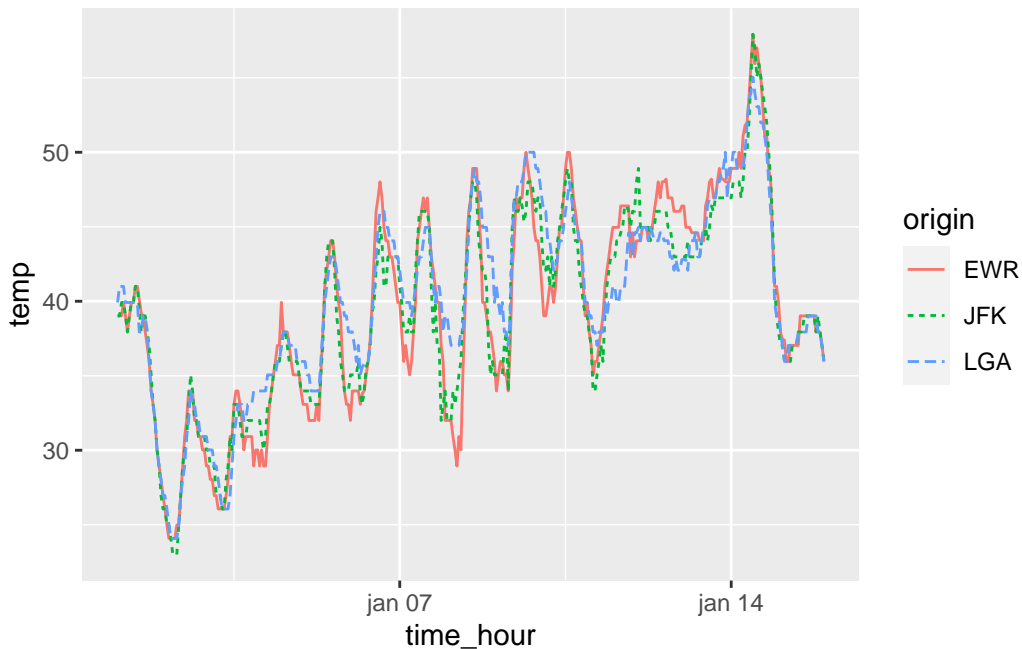


Figure 4.22: Températures horaires des 3 aéroports de New York entre le 1er et le 15 janvier 2013.

#### 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 @ref(sec-fig:lineplotgraph) sur lequel nous avons 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()`) :

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

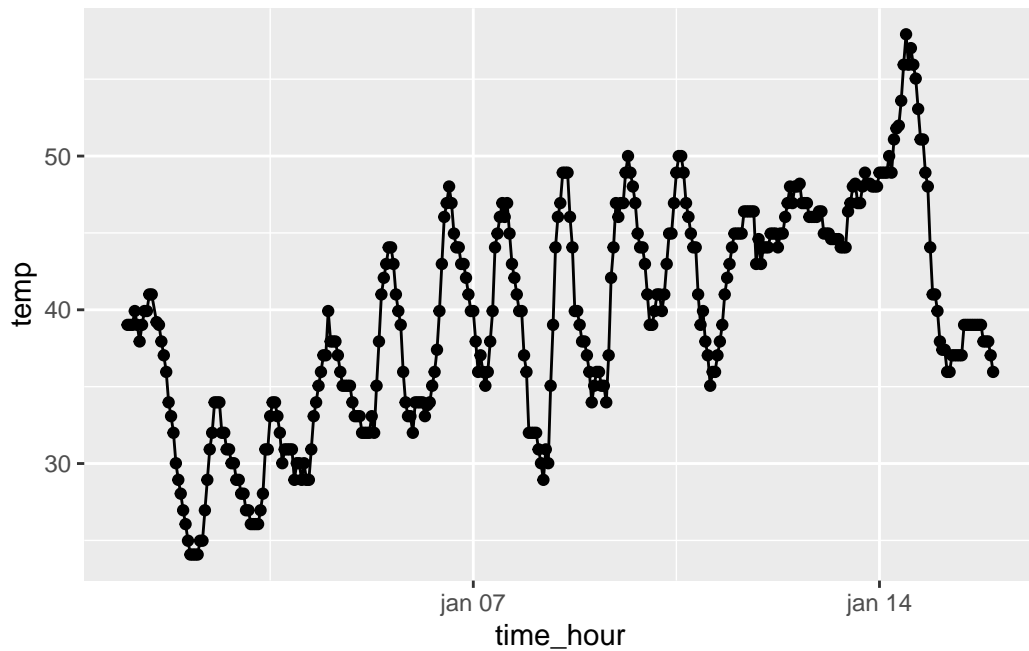


Figure 4.23: Températures horaires à l'aéroport de Newark entre le 1er et le 15 janvier 2013.

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()
```

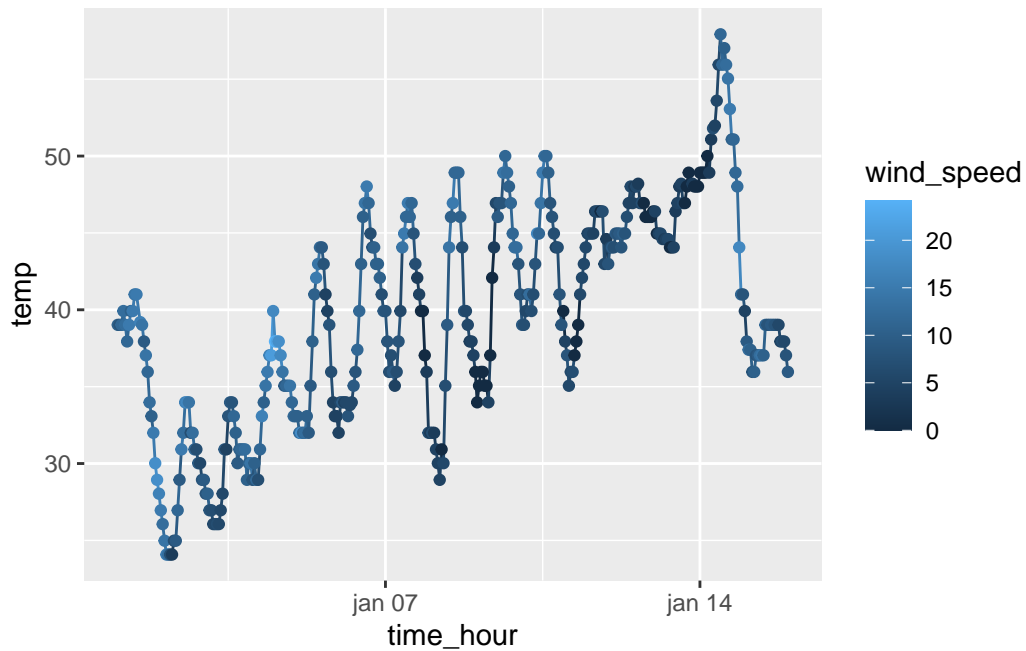


Figure 4.24: Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur de la ligne et des points renseigne sur la vitesse du vent.

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

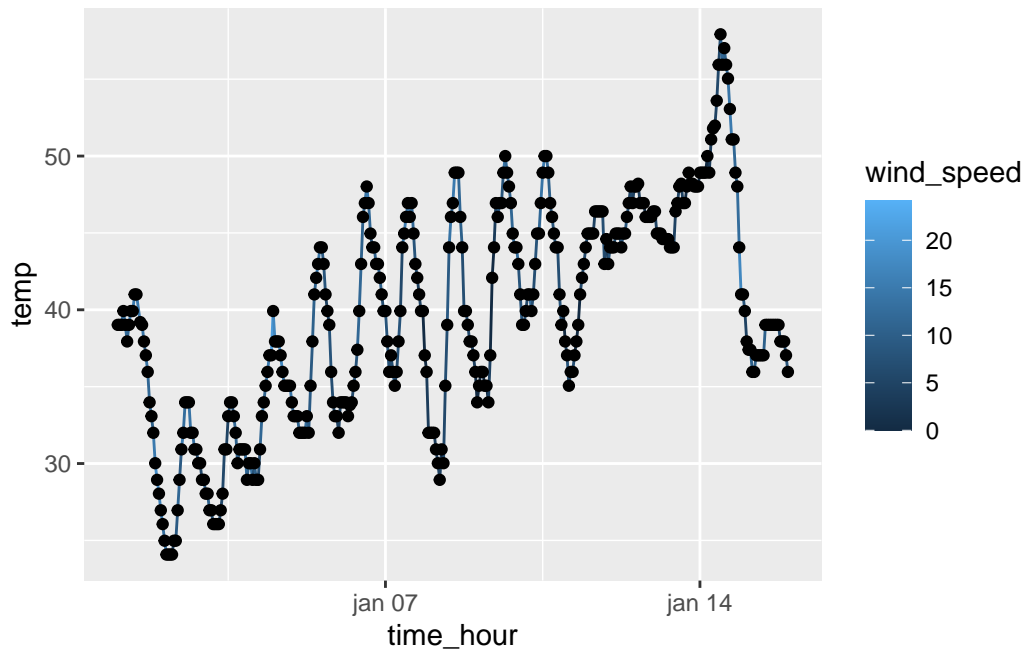


Figure 4.25: Températures horaires et vitesse du vent à l'aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur de la ligne renseigne sur la vitesse du vent.

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

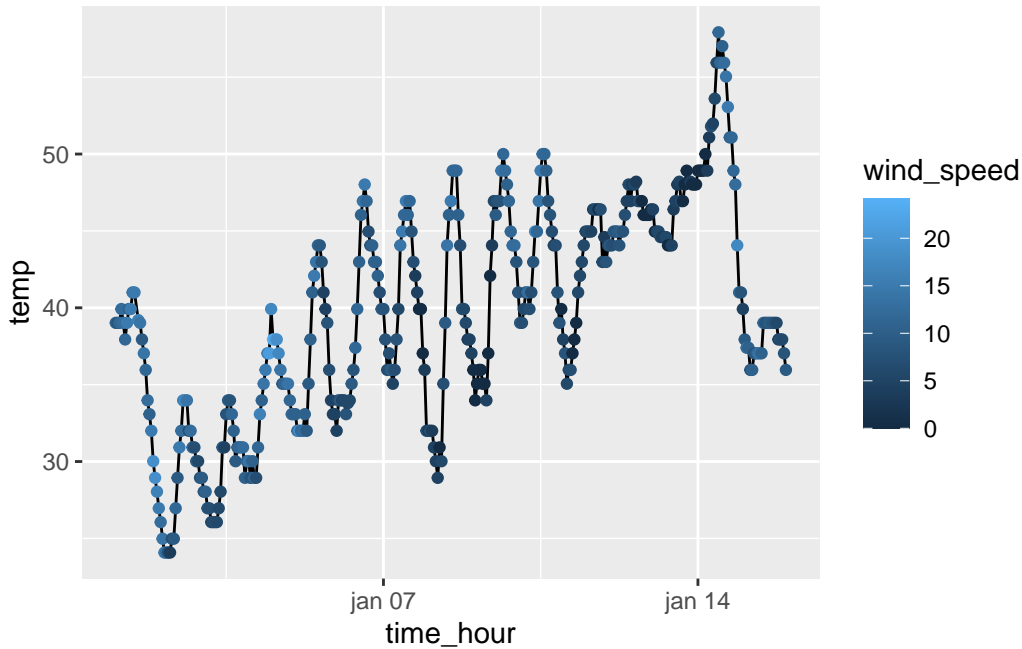


Figure 4.26: Températures horaires et vitesse du vent à l’aéroport de Newark entre le 1er et le 15 janvier 2013. La couleur des points renseigne sur la vitesse du vent.

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()
```

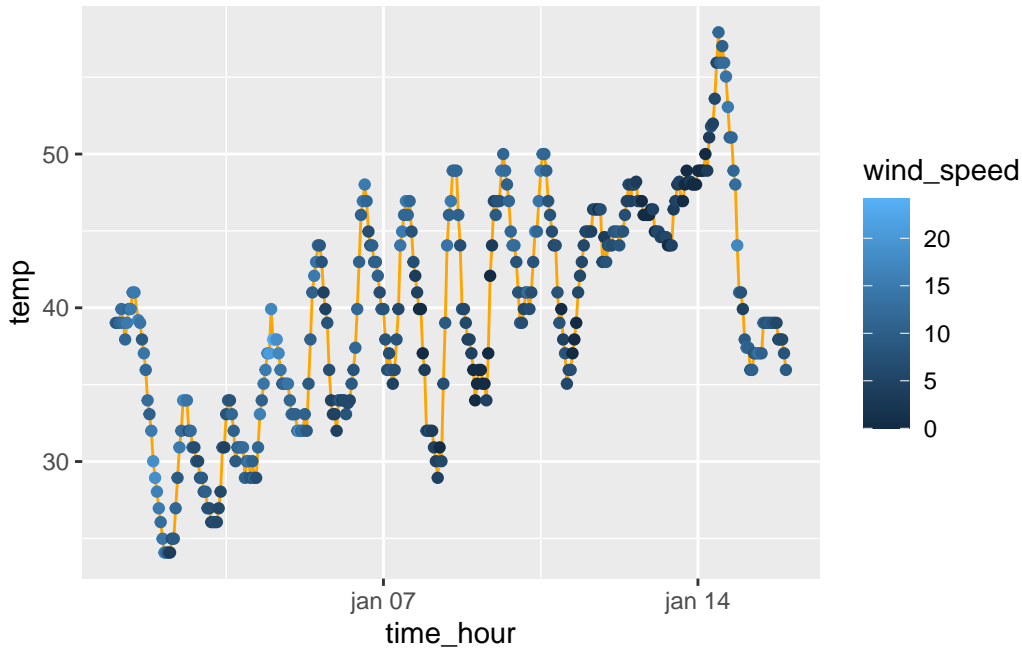


Figure 4.27: Températures horaires et vitesse du vent à l’aéroport de Newark entre le 1er et le 15 janvier 2013.

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.

## 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``.

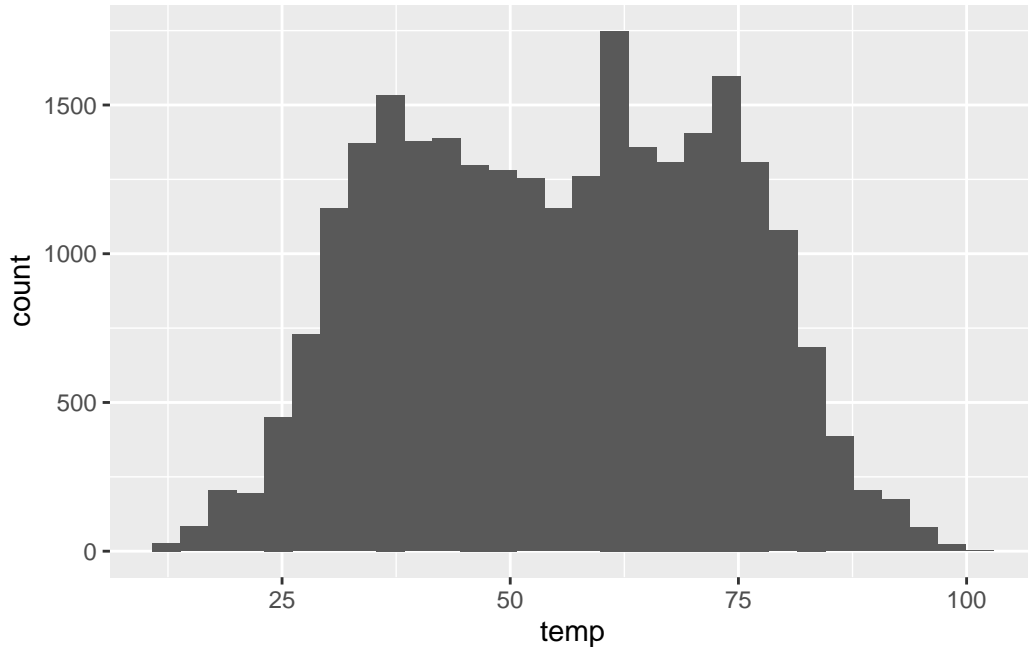


Figure 4.28: Histogramme des températures enregistrées en 2013 dans les 3 aéroports de New York.

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).
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")
```

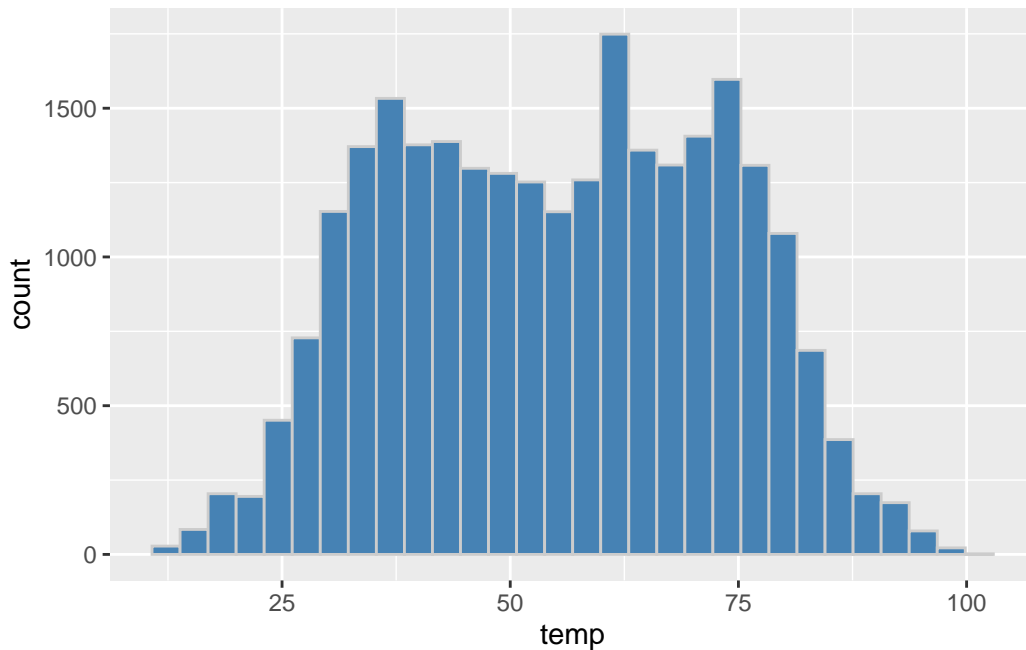


Figure 4.29: Utilisation des arguments `fill` et `color` pour modifier l'aspect de l'histogramme.

### 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")
```

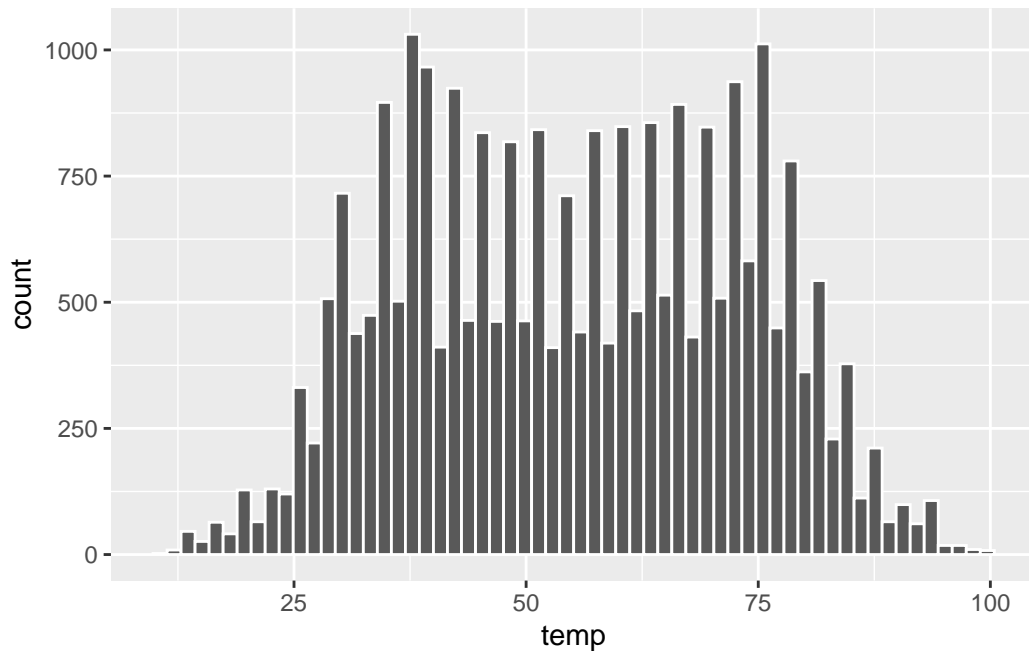


Figure 4.30: Modification du nombre de classes.

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 @ref(sec-fig:rughist)) :

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

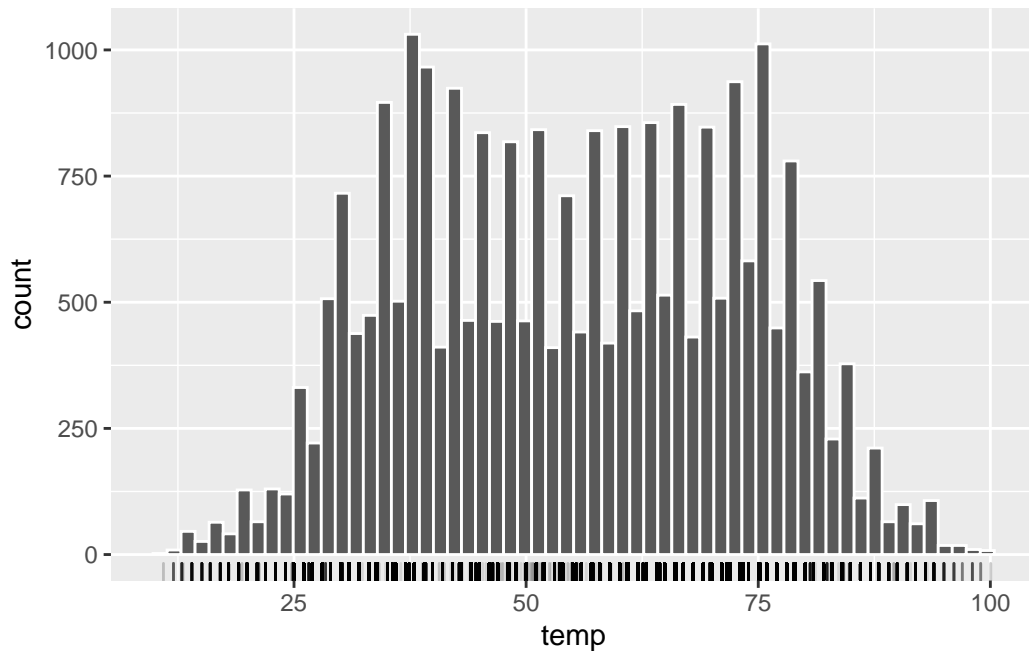


Figure 4.31: Ajout des données brutes sous forme de ‘tapis’ (rug) sous l’histogramme.

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")
```

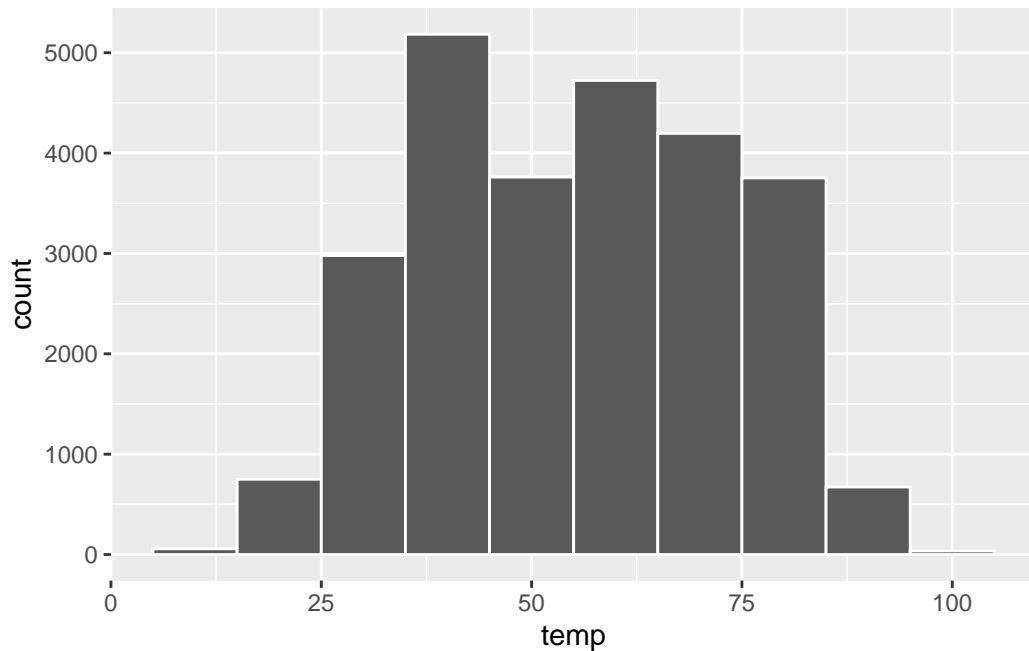


Figure 4.32: Modification de la largeur des classes avec `binwidth`.

Ici, chaque catégorie recouvre 10 degrés Farenheit, 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 @ref(sec-fig:irregclasses)) :

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

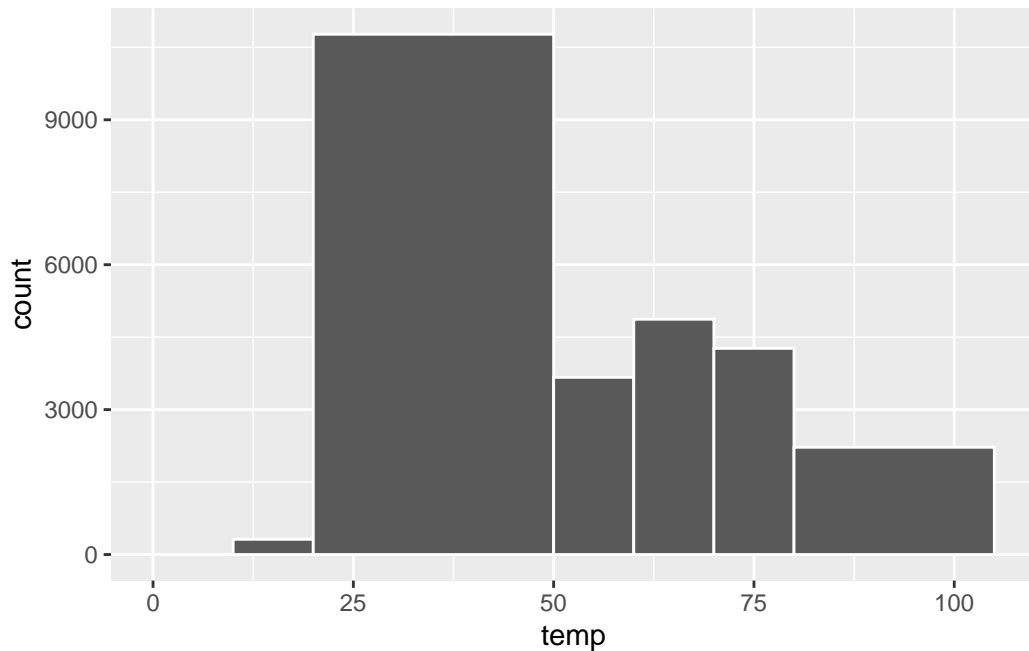


Figure 4.33: Spécification manuelle des limites de classes de tailles (classes irrégulières).

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) :

```
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 90 95 100
[20] 105
```

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

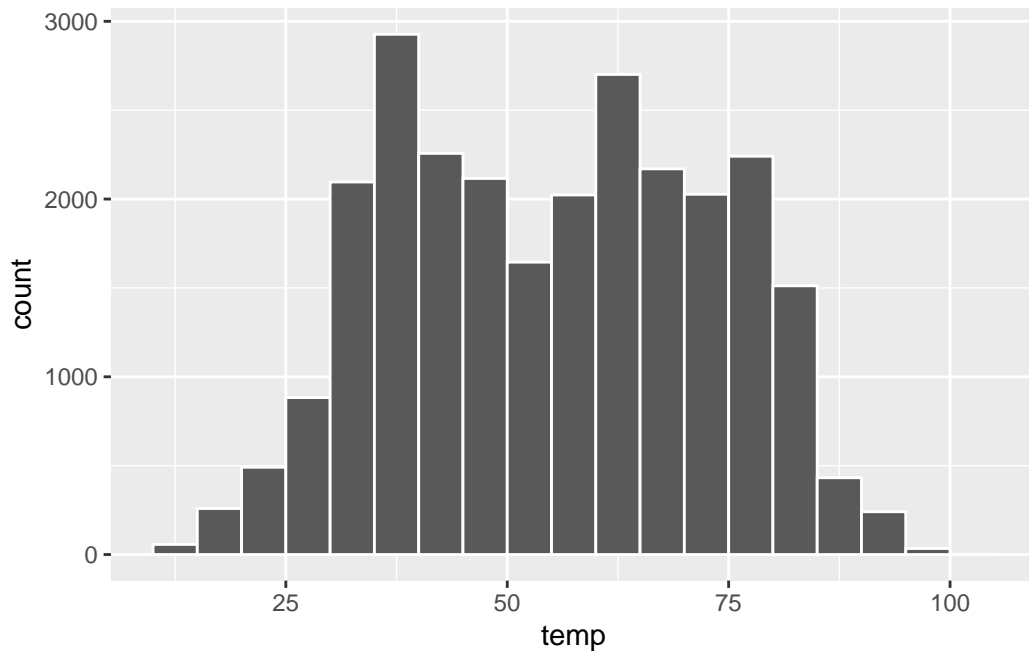


Figure 4.34: Un exemple d'utilisation de l'argument `breaks`.

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 Fahrenheit. 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")
```

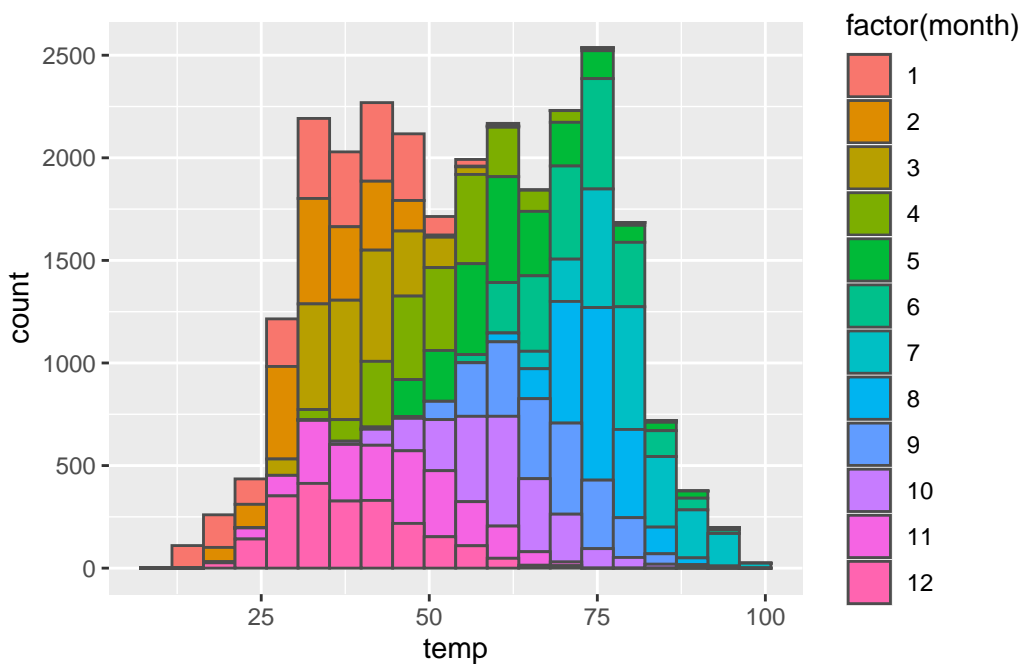


Figure 4.35: Distribution des températures avec visualisation des données mensuelles.

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)
```

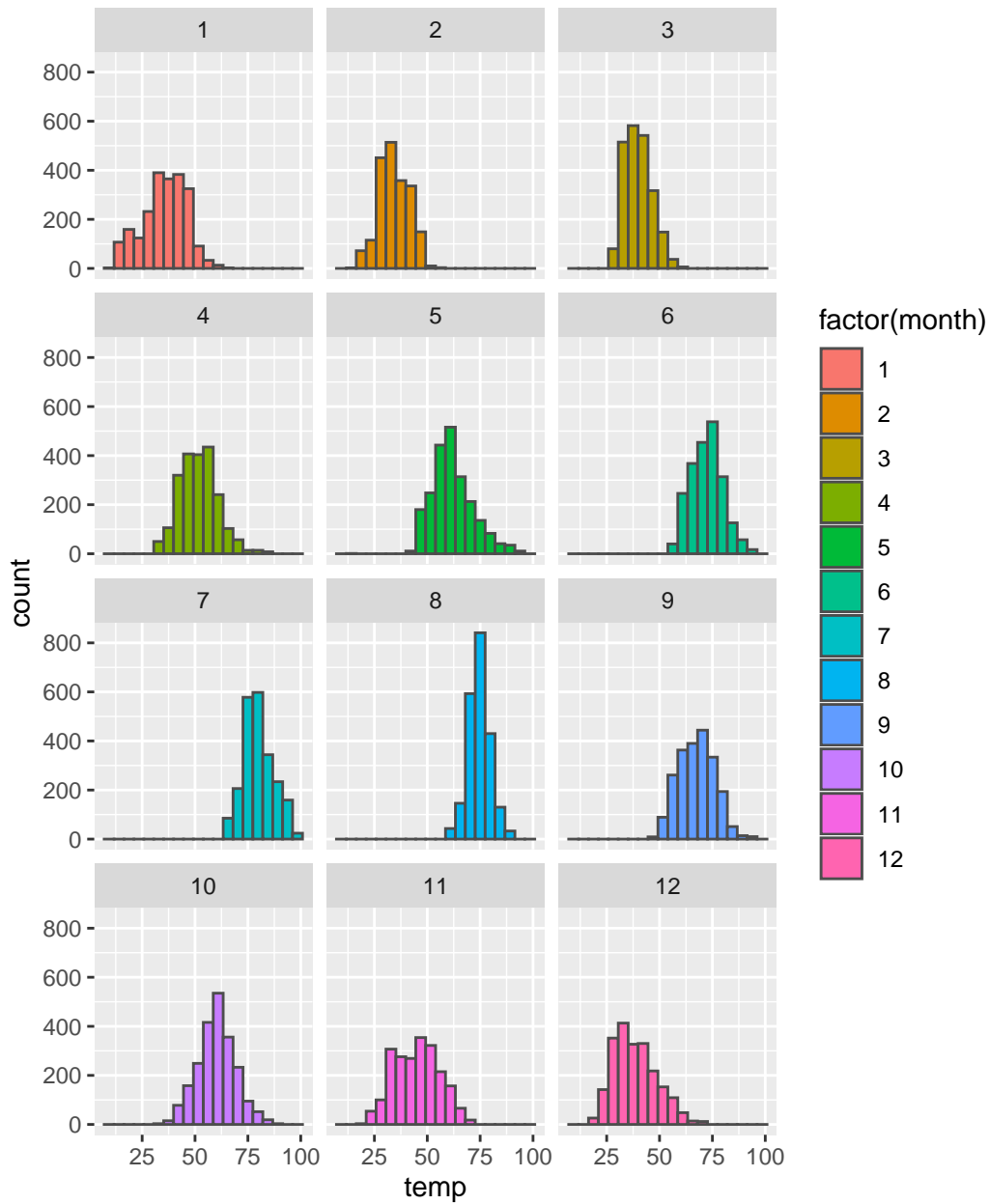


Figure 4.36: (ref:wrap)

(ref:wrap) Un exemple d'utilisation de `facet_wrap()`.

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`.

#### 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)
```

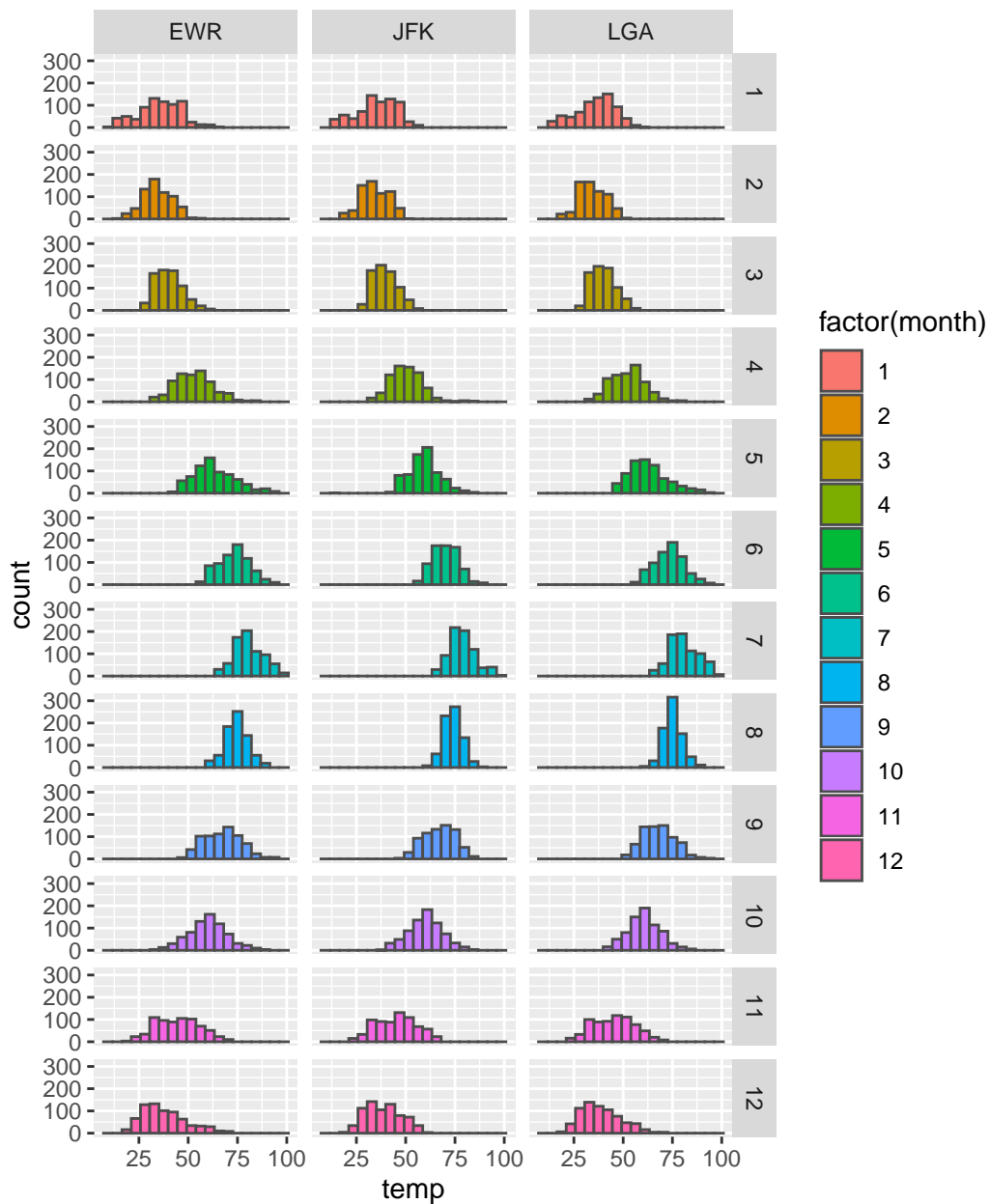


Figure 4.37: (ref:grid)

(ref:grid) 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 @ref(sec-fig:grid).

1. Quels éléments nouveaux ce graphique nous apprend-il par rapport au graphique @ref(sec-fig:break) 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 Création 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).

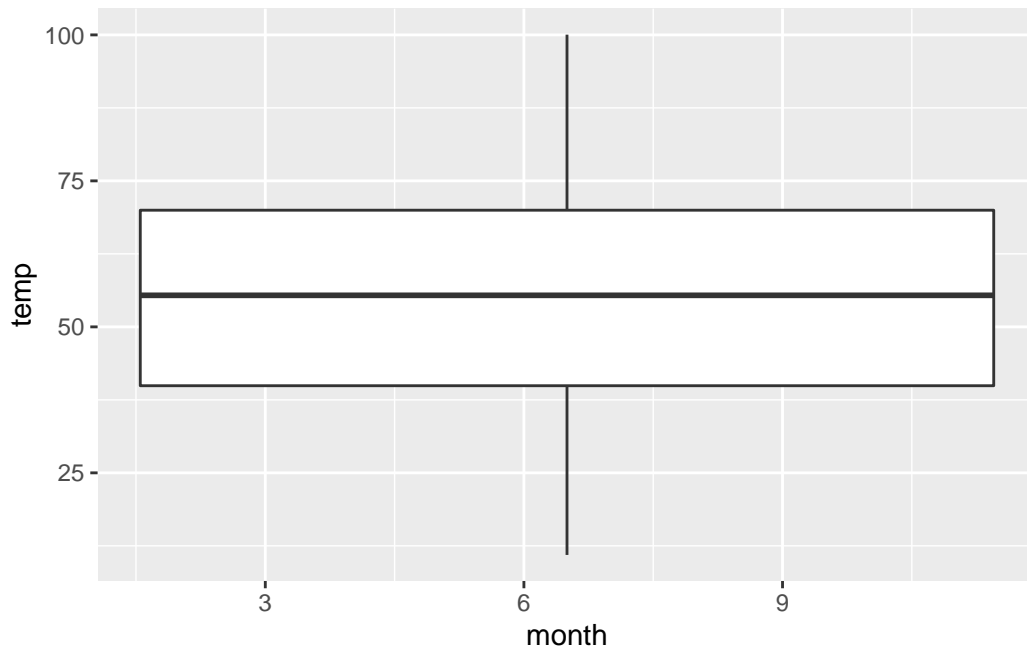


Figure 4.38: Un boxplot fort peu utile...

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.

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()
```

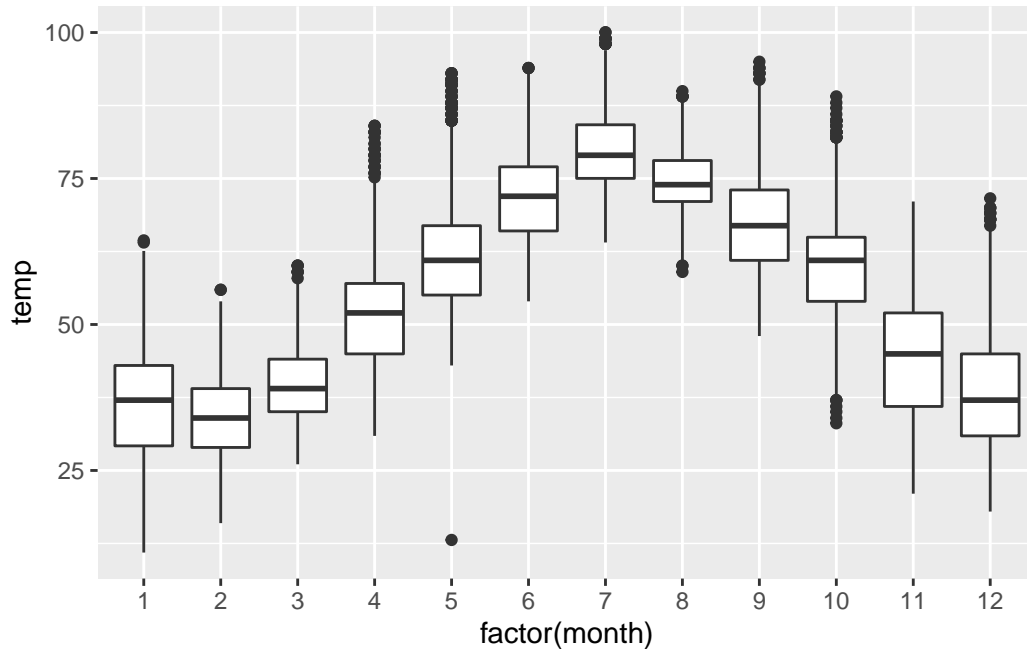


Figure 4.39: Boxplot des températures mensuelles.

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 façon 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)
```

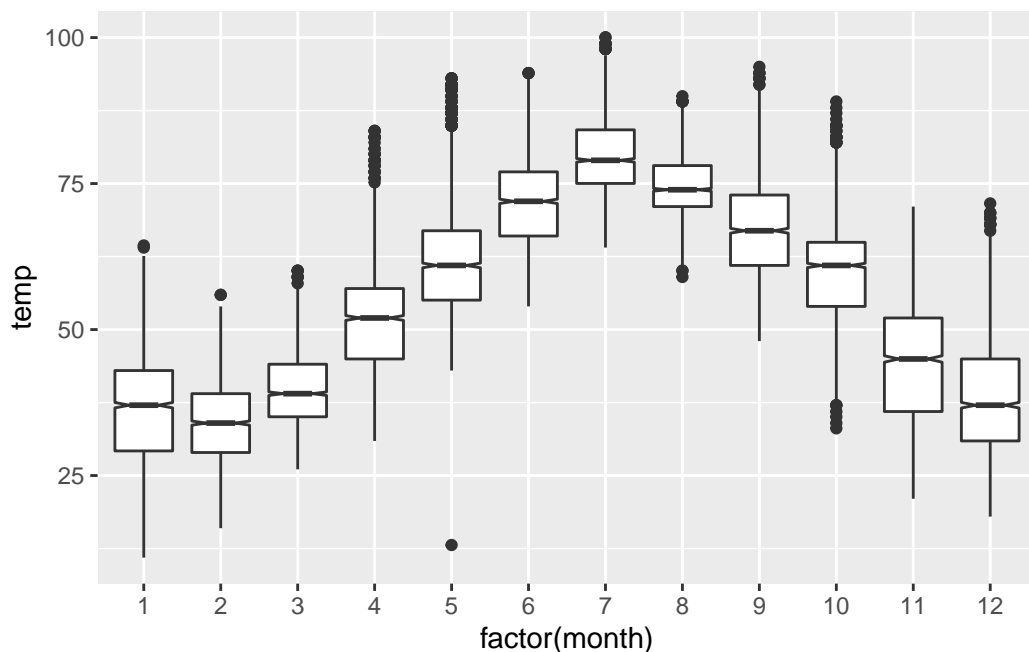


Figure 4.40: (ref:boxplot)

(ref:boxplot) Boxplot des températures mensuelles. Les intervalles de confiance à 95% de la médiane sont affichés.

Comme l'indique la légende de la figure @ref(sec-fig:notchedboxplot), 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. 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 @ref(sec-fig:compboxplot) 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.

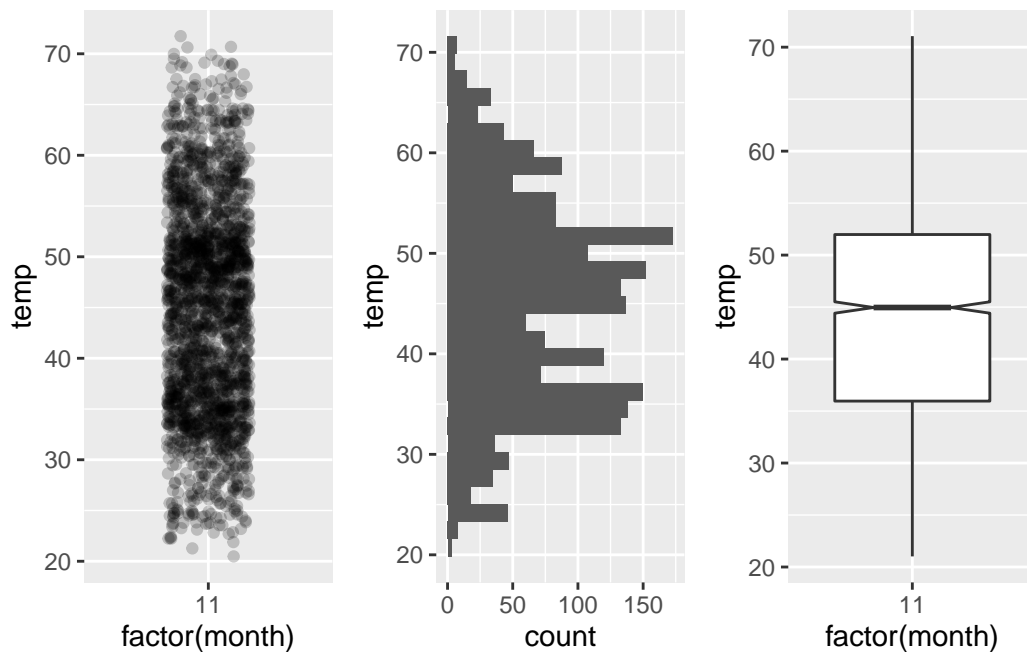


Figure 4.41: Distribution des températures de Novembre 2013.

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 permuté 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 Fahrenheit. 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 côte à côte (voir la figure @ref(sec-fig:notchedboxplot) 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 @ref(sec-fig:notchedboxplot), 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()
```

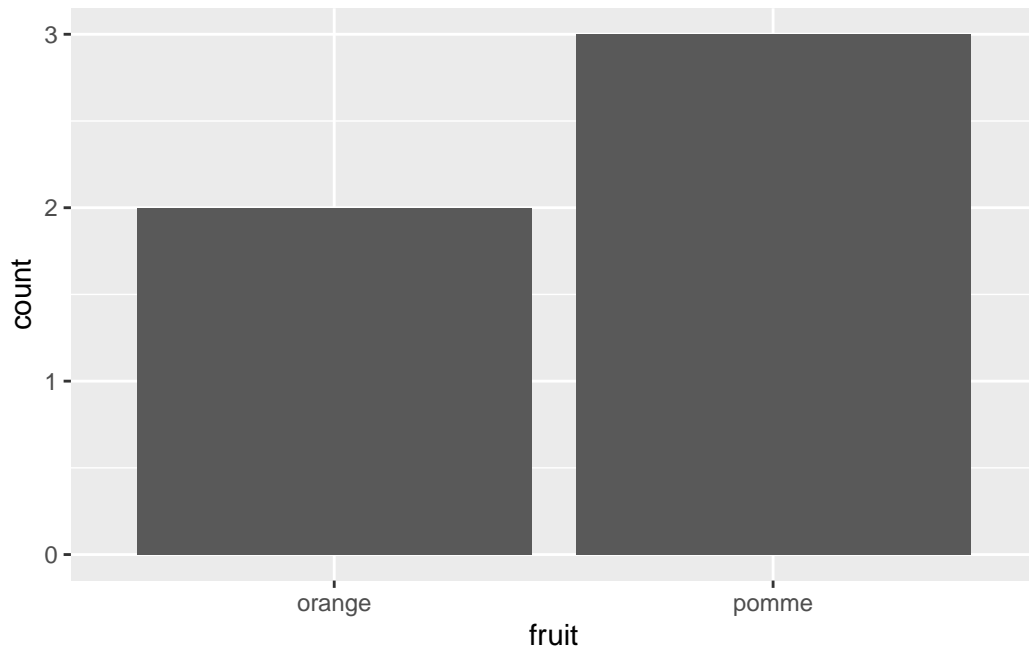


Figure 4.42: Barplot pour des données non pré-comptées.

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

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

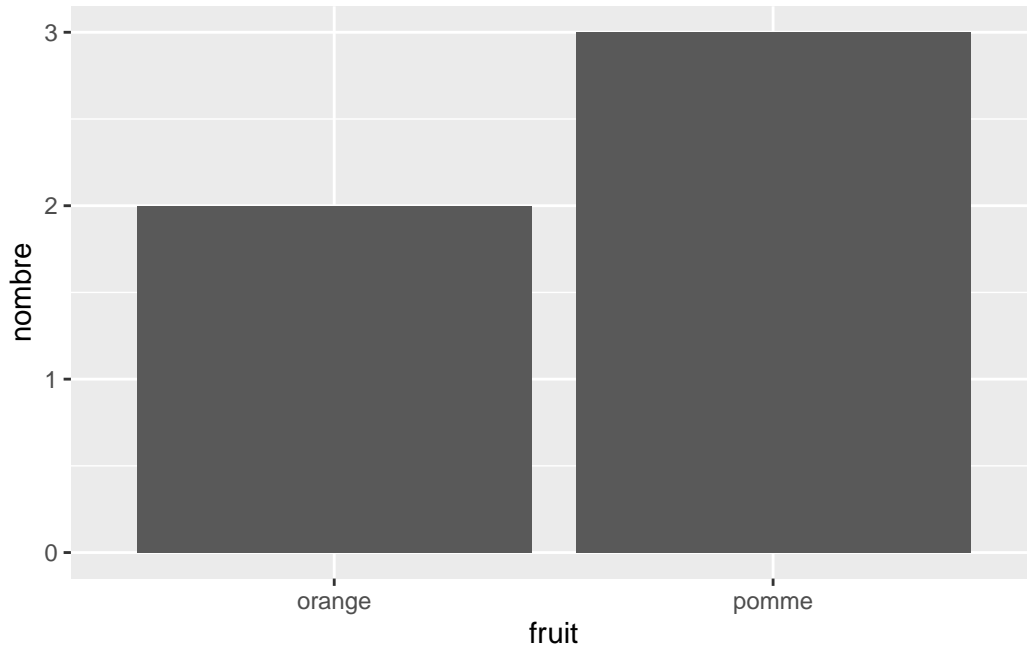


Figure 4.43: Barplot pour des données pré-comptées.

Notez que les figures @ref(sec-fig:barplot) et @ref(sec-fig:barplotcol) 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 :

- Le code qui génère la figure @ref(sec-fig:barplot) 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 @ref(sec-fig:barplotcol) 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 souhaitez 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 appartient 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()
```

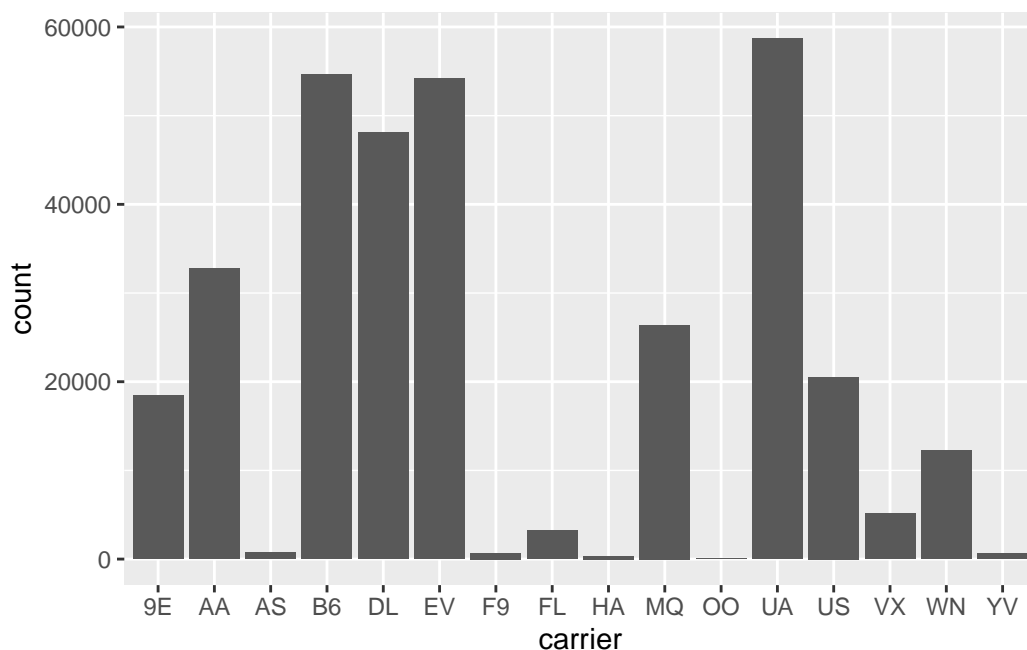


Figure 4.44: Nombre de vols par compagnie aérienne au départ de New York en 2013.

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 package `forcats`. Si vous avez installé le `tidyverse`, le package `forcats` 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()
```

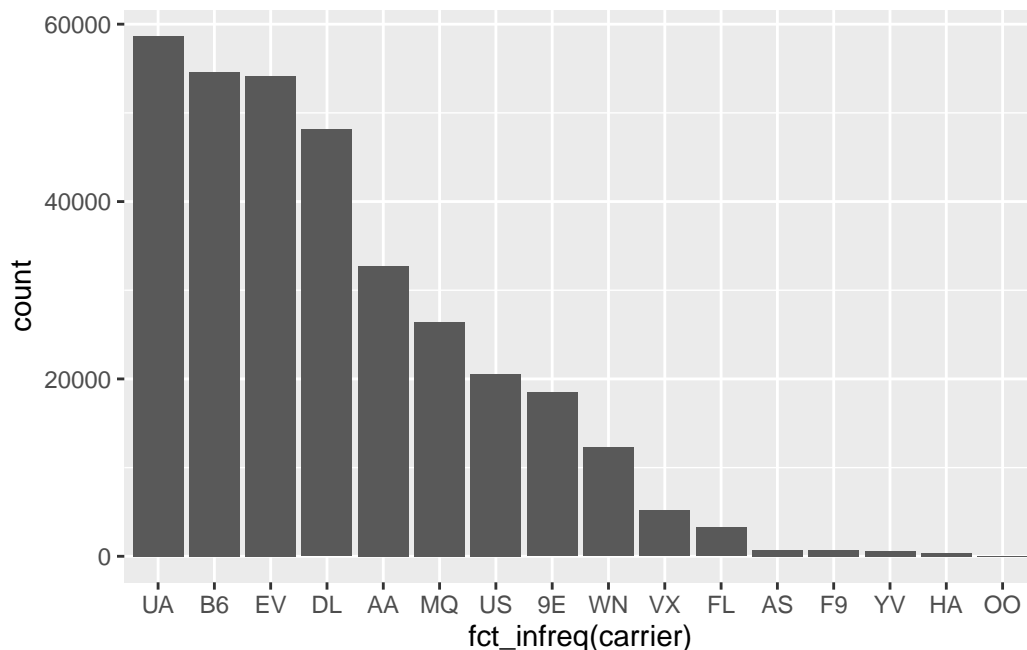


Figure 4.45: Nombre de vols par compagnie aérienne au départ de New York en 2013.

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 [@ref\(sec-wrangling\)](#). 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()
```

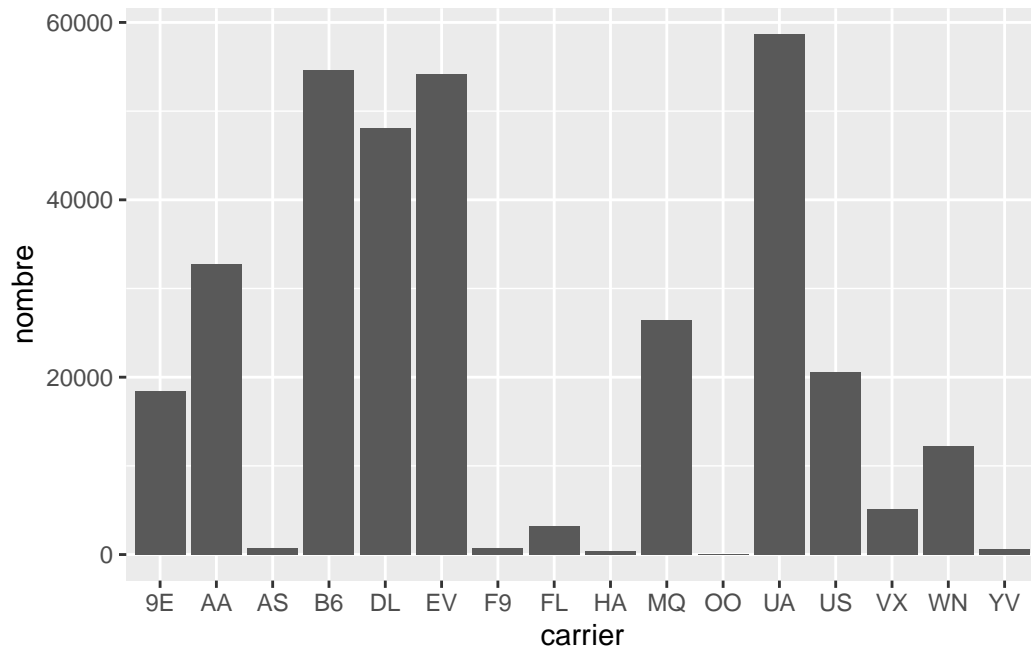


Figure 4.46: Nombre de vols par compagnie aérienne au départ de New York en 2013.

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

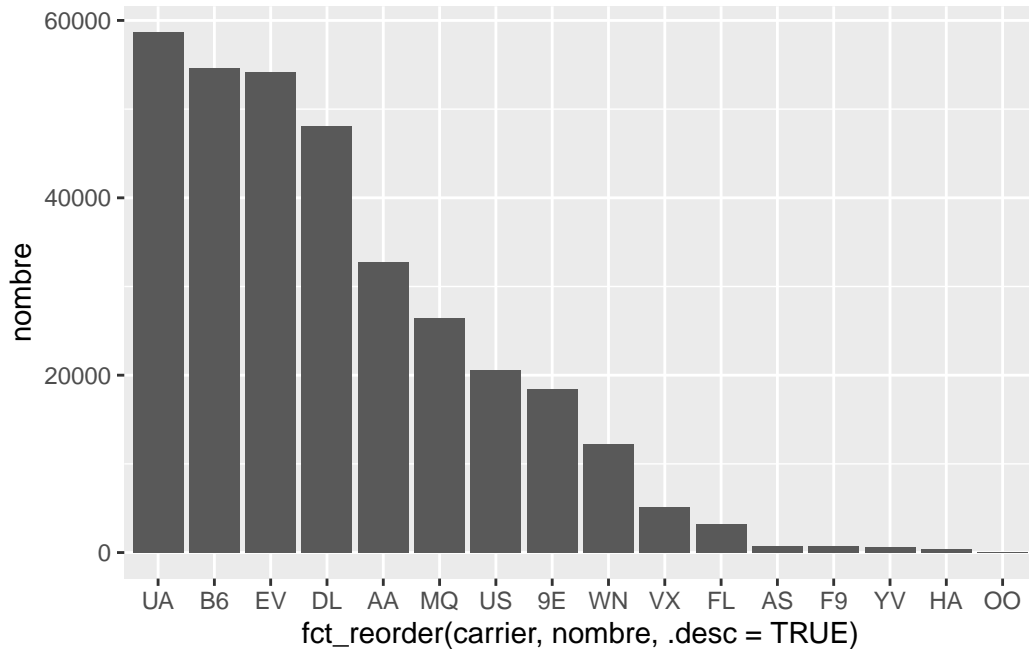


Figure 4.47: Nombre de vols par compagnie aérienne au départ de New York en 2013.

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 ?
4. 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  $90^\circ$ , et à sous-estimer les angles inférieurs à  $90^\circ$ . 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 :

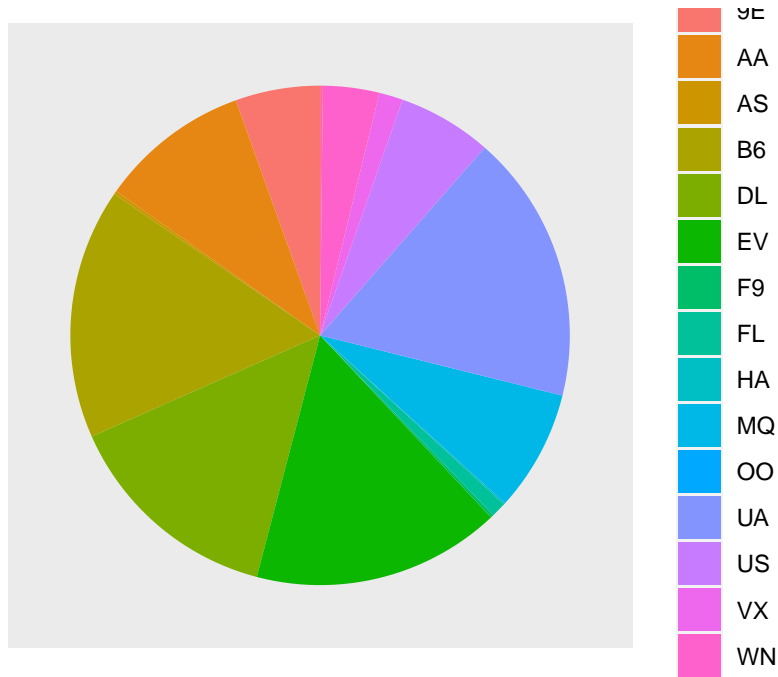


Figure 4.48: Nombre de vols par compagnie aérienne au départ de New York en 2013.

- 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 (voir impossible) de répondre précisément à ces questions avec le diagramme circulaire de la figure @ref(sec-fig:piechart), alors qu'il est très simple d'obtenir des réponses précises avec un diagramme bâtons tel que présenté à la figure @ref(sec-fig:bpcarriersortedcol) (vérifiez-le !).

### 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 devons 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()
```

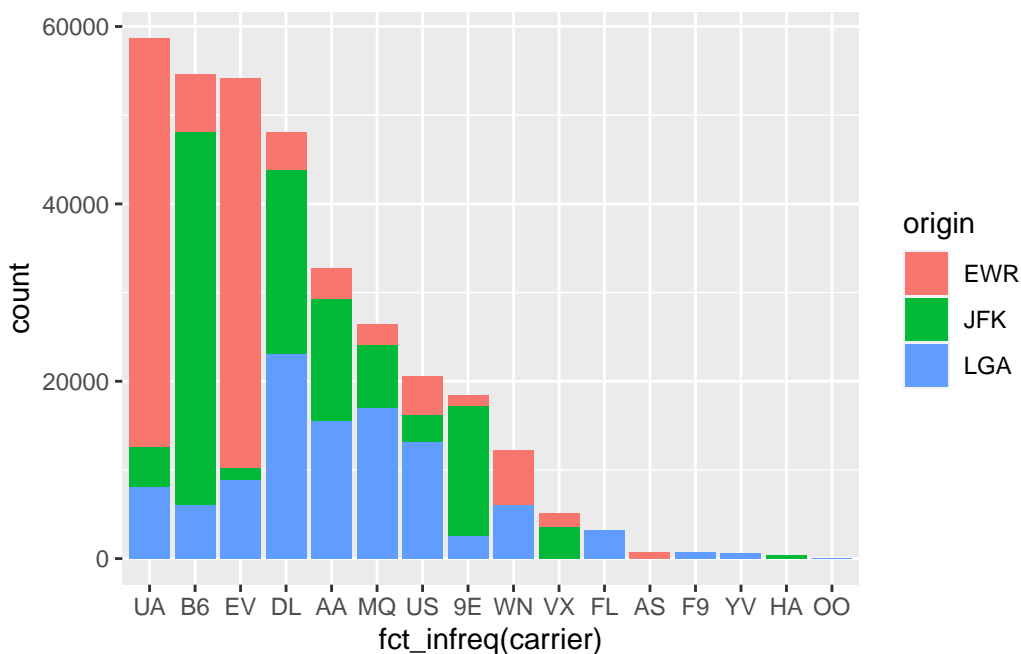


Figure 4.49: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Notez qu'il s'agit du même code que celui utilisé pour la figure @ref(sec-fig:bpcarriersorted), 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 :

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

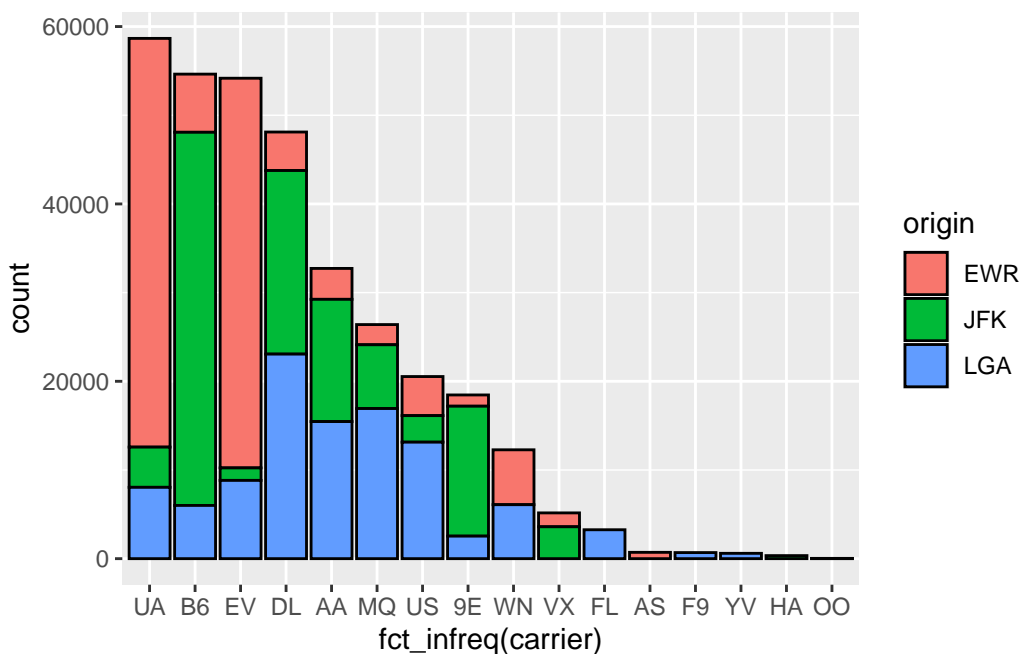


Figure 4.50: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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 aisé 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")
```

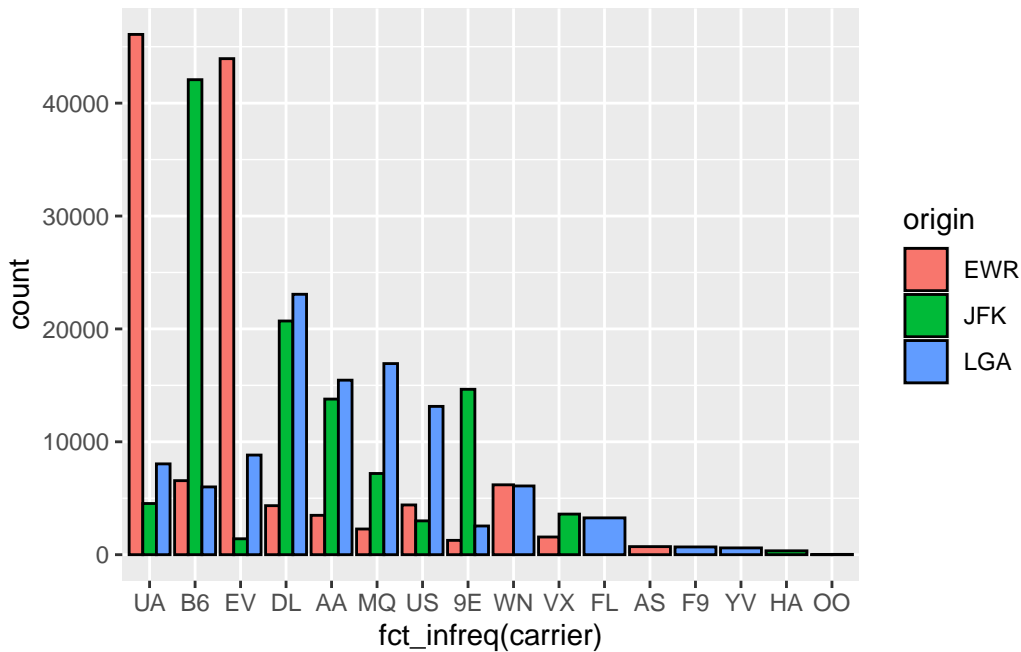


Figure 4.51: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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 @ref(sec-fig:dodge), les 8 premières compagnies aériennes déservent les 3 aéroports de New York, mais les 2 suivantes (WN et VX) n'en déservent 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.

### 4.8.5.3 Utilisation des facets

La meilleure alternative est probablement l'utilisation de **facets** que nous avons déjà décrite à la section @ref(sec-facets) :

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

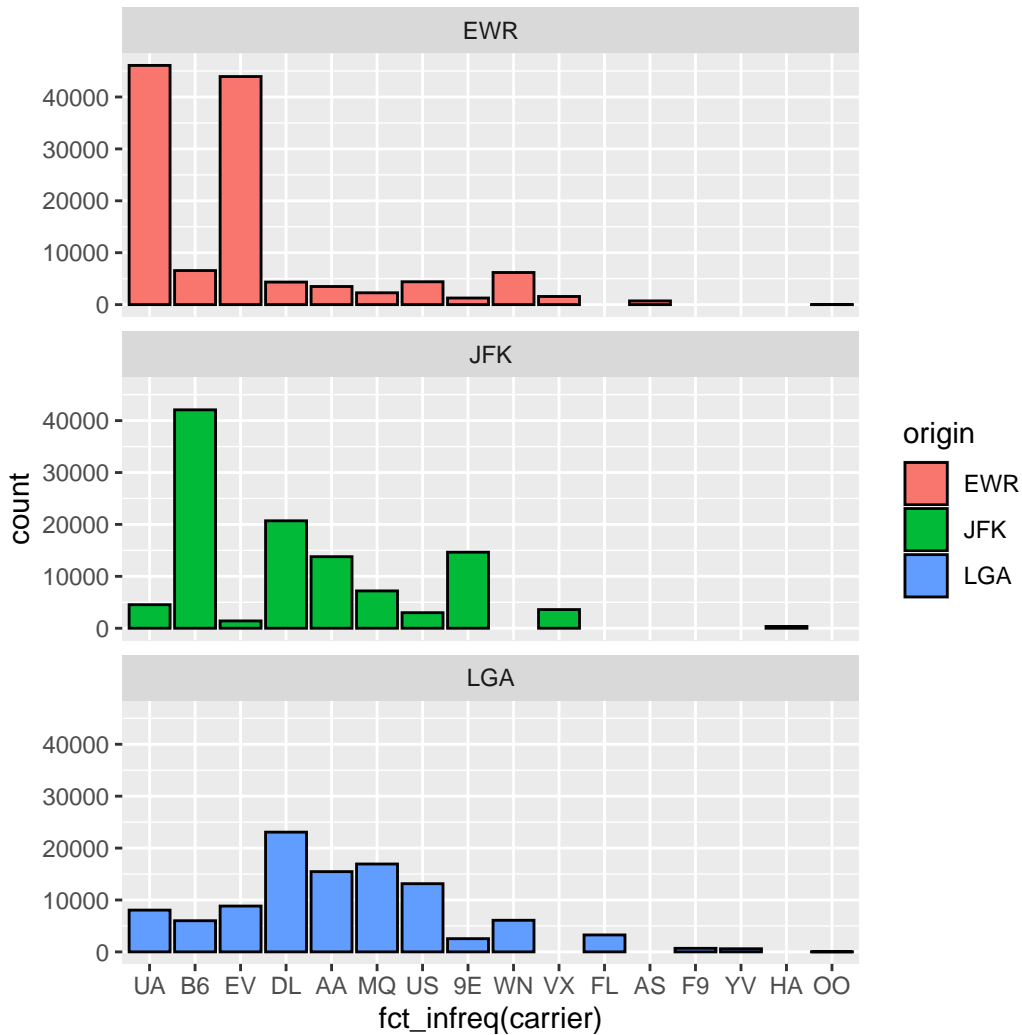


Figure 4.52: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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.

Reprenons par exemple le graphique de la figure @ref(sec-fig:varcolor) :

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

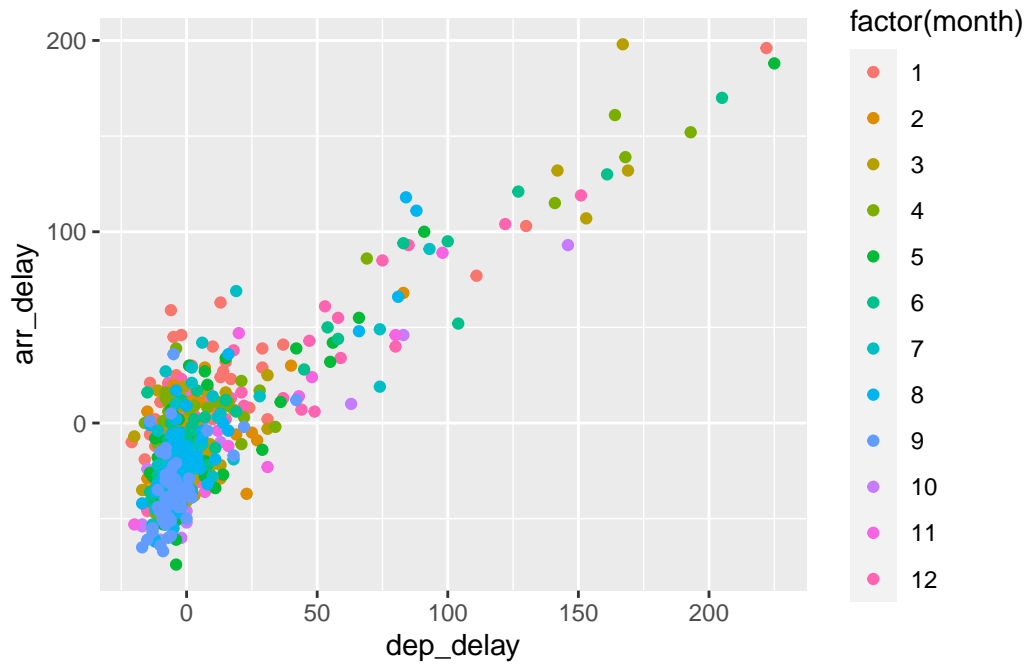


Figure 4.53: Association de color à une variable catégorielle.

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")
```

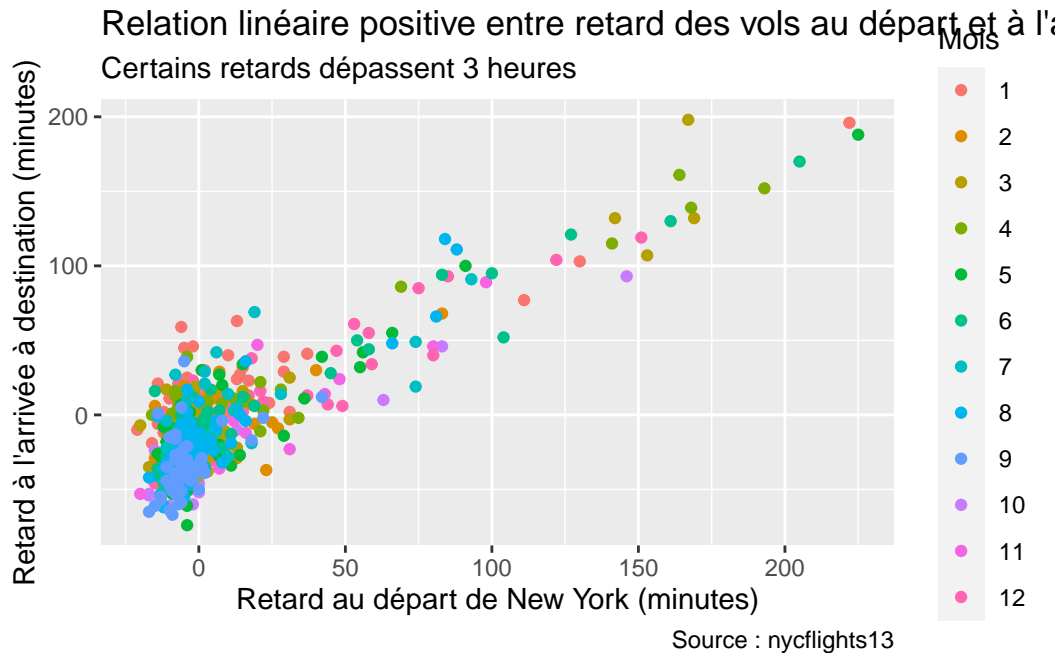


Figure 4.54: Exemple d'utilisation de `labs()`.

À partir de maintenant, vous devriez systématiquement légender 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 @ref(sec-fig:barfacet), 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")
```

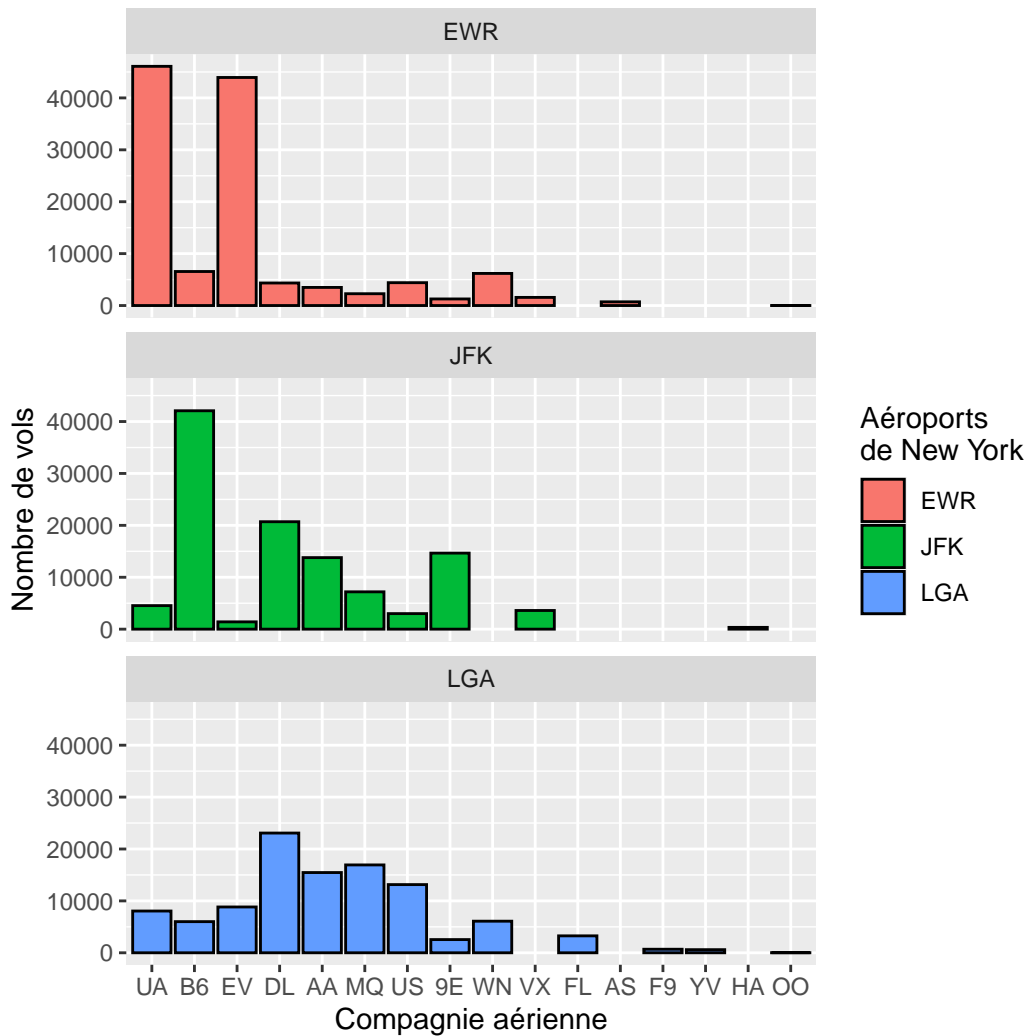


Figure 4.55: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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()
```

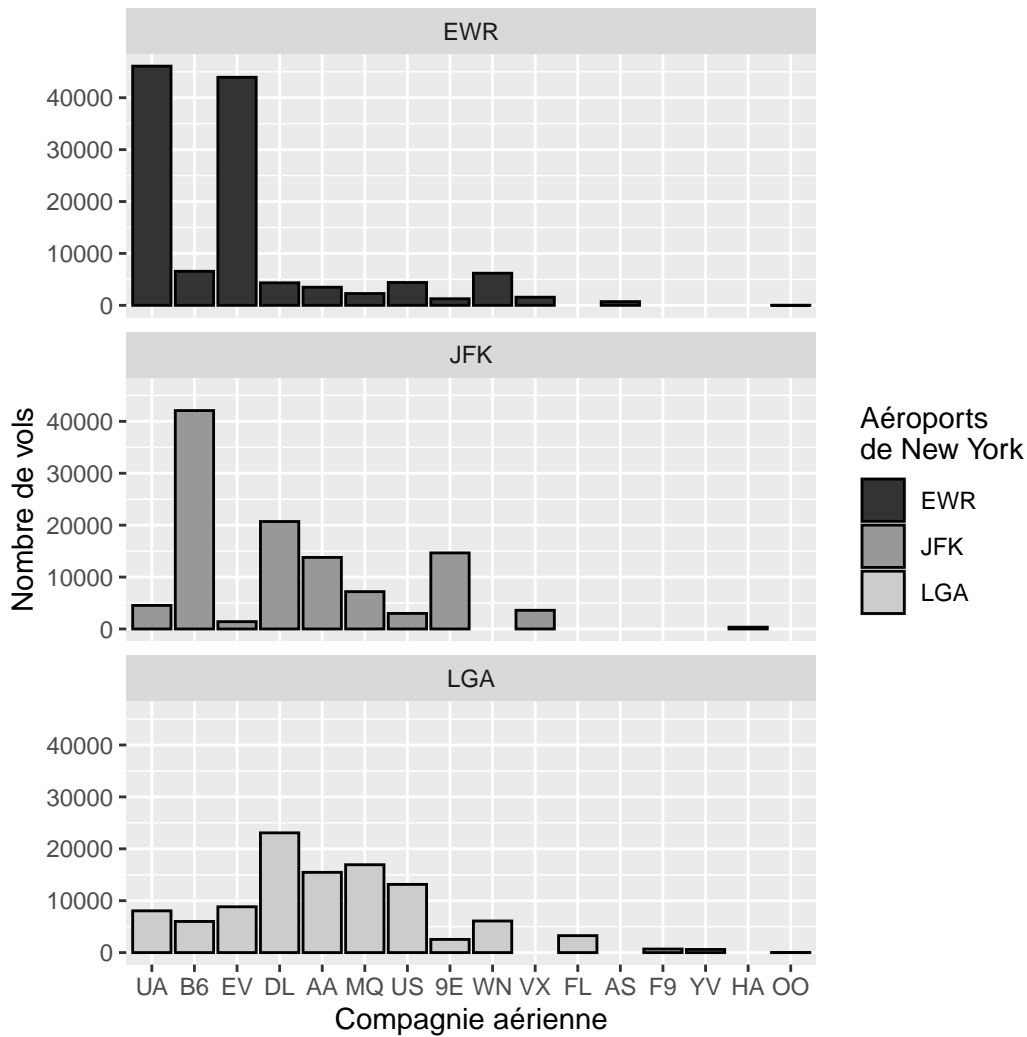


Figure 4.56: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Le package `RColorBrewer` propose une large gamme de palettes de couleurs (figure @ref(sec-fig:brew)) :

`ggplot2` permet d'appliquer ces palettes très simplement :

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

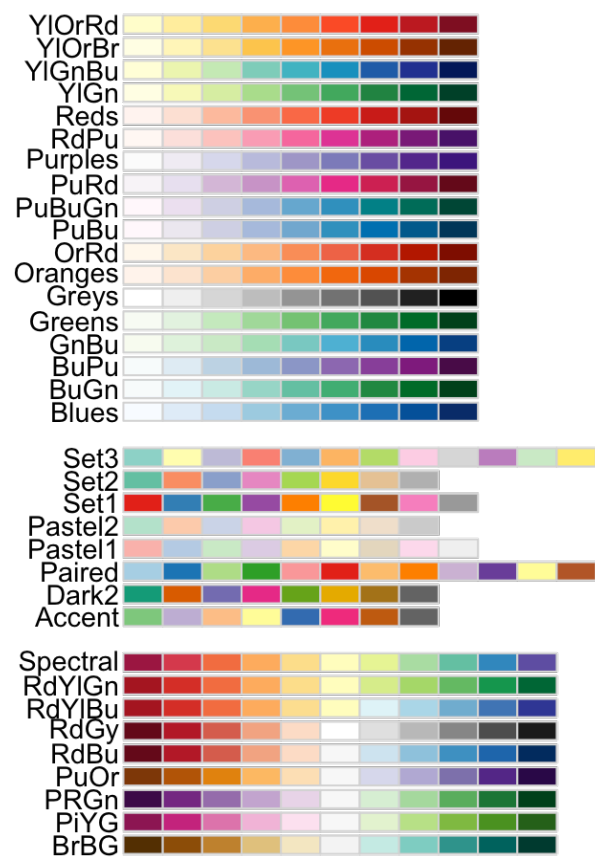


Figure 4.57: Toutes les palettes de couleur du package RColorBrewer.

```

y = "Nombre de vols",
fill = "Aéroports\nde New York") +
scale_fill_brewer(palette = "Accent")

```

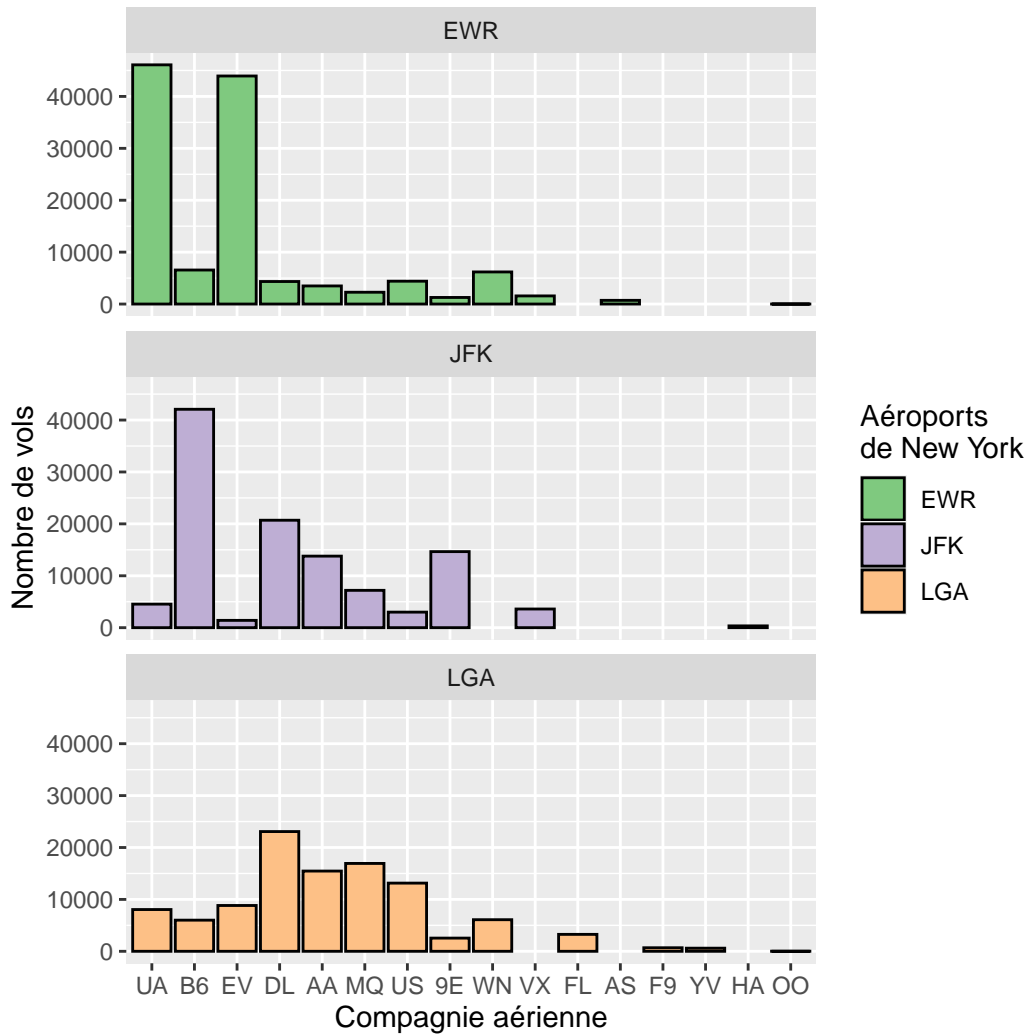


Figure 4.58: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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()
```

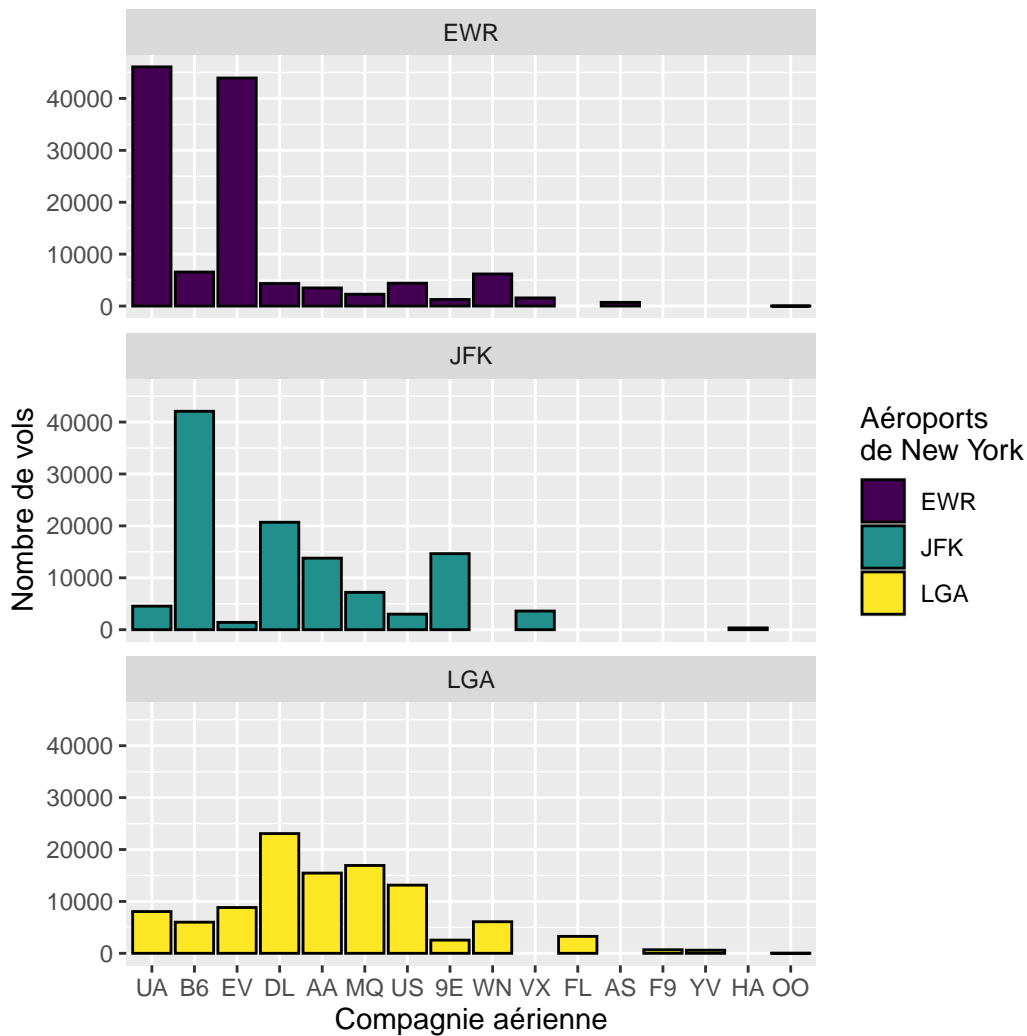


Figure 4.59: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

Enfin, si les palettes de couleurs ne conviennent pas, il est toujours possible de spécifier manuel-

lement les couleurs souhaitées. R propose un accès rapide à 657 noms de couleurs. Pour les afficher, il suffit de taper :

```
colors()
```

[1] "white"	"aliceblue"	"antiquewhite"
[4] "antiquewhite1"	"antiquewhite2"	"antiquewhite3"
[7] "antiquewhite4"	"aquamarine"	"aquamarine1"
[10] "aquamarine2"	"aquamarine3"	"aquamarine4"
[13] "azure"	"azure1"	"azure2"
[16] "azure3"	"azure4"	"beige"
[19] "bisque"	"bisque1"	"bisque2"
[22] "bisque3"	"bisque4"	"black"
[25] "blanchedalmond"	"blue"	"blue1"
[28] "blue2"	"blue3"	"blue4"
[31] "blueviolet"	"brown"	"brown1"
[34] "brown2"	"brown3"	"brown4"
[37] "burlywood"	"burlywood1"	"burlywood2"
[40] "burlywood3"	"burlywood4"	"cadetblue"
[43] "cadetblue1"	"cadetblue2"	"cadetblue3"
[46] "cadetblue4"	"chartreuse"	"chartreuse1"
[49] "chartreuse2"	"chartreuse3"	"chartreuse4"
[52] "chocolate"	"chocolate1"	"chocolate2"
[55] "chocolate3"	"chocolate4"	"coral"
[58] "coral1"	"coral2"	"coral3"
[61] "coral4"	"cornflowerblue"	"cornsilk"
[64] "cornsilk1"	"cornsilk2"	"cornsilk3"
[67] "cornsilk4"	"cyan"	"cyan1"
[70] "cyan2"	"cyan3"	"cyan4"
[73] "darkblue"	"darkcyan"	"darkgoldenrod"
[76] "darkgoldenrod1"	"darkgoldenrod2"	"darkgoldenrod3"
[79] "darkgoldenrod4"	"darkgray"	"darkgreen"
[82] "darkgrey"	"darkkhaki"	"darkmagenta"
[85] "darkolivegreen"	"darkolivegreen1"	"darkolivegreen2"
[88] "darkolivegreen3"	"darkolivegreen4"	"darkorange"
[91] "darkorange1"	"darkorange2"	"darkorange3"
[94] "darkorange4"	"darkorchid"	"darkorchid1"
[97] "darkorchid2"	"darkorchid3"	"darkorchid4"
[100] "darkred"	"darksalmon"	"darkseagreen"
[103] "darkseagreen1"	"darkseagreen2"	"darkseagreen3"
[106] "darkseagreen4"	"darkslateblue"	"darkslategray"
[109] "darkslategray1"	"darkslategray2"	"darkslategray3"

[112]	"darkslategray4"	"darkslategrey"	"darkturquoise"
[115]	"darkviolet"	"deeppink"	"deeppink1"
[118]	"deeppink2"	"deeppink3"	"deeppink4"
[121]	"deepskyblue"	"deepskyblue1"	"deepskyblue2"
[124]	"deepskyblue3"	"deepskyblue4"	"dimgray"
[127]	"dimgrey"	"dodgerblue"	"dodgerblue1"
[130]	"dodgerblue2"	"dodgerblue3"	"dodgerblue4"
[133]	"firebrick"	"firebrick1"	"firebrick2"
[136]	"firebrick3"	"firebrick4"	"floralwhite"
[139]	"forestgreen"	"gainsboro"	"ghostwhite"
[142]	"gold"	"gold1"	"gold2"
[145]	"gold3"	"gold4"	"goldenrod"
[148]	"goldenrod1"	"goldenrod2"	"goldenrod3"
[151]	"goldenrod4"	"gray"	"gray0"
[154]	"gray1"	"gray2"	"gray3"
[157]	"gray4"	"gray5"	"gray6"
[160]	"gray7"	"gray8"	"gray9"
[163]	"gray10"	"gray11"	"gray12"
[166]	"gray13"	"gray14"	"gray15"
[169]	"gray16"	"gray17"	"gray18"
[172]	"gray19"	"gray20"	"gray21"
[175]	"gray22"	"gray23"	"gray24"
[178]	"gray25"	"gray26"	"gray27"
[181]	"gray28"	"gray29"	"gray30"
[184]	"gray31"	"gray32"	"gray33"
[187]	"gray34"	"gray35"	"gray36"
[190]	"gray37"	"gray38"	"gray39"
[193]	"gray40"	"gray41"	"gray42"
[196]	"gray43"	"gray44"	"gray45"
[199]	"gray46"	"gray47"	"gray48"
[202]	"gray49"	"gray50"	"gray51"
[205]	"gray52"	"gray53"	"gray54"
[208]	"gray55"	"gray56"	"gray57"
[211]	"gray58"	"gray59"	"gray60"
[214]	"gray61"	"gray62"	"gray63"
[217]	"gray64"	"gray65"	"gray66"
[220]	"gray67"	"gray68"	"gray69"
[223]	"gray70"	"gray71"	"gray72"
[226]	"gray73"	"gray74"	"gray75"
[229]	"gray76"	"gray77"	"gray78"
[232]	"gray79"	"gray80"	"gray81"
[235]	"gray82"	"gray83"	"gray84"
[238]	"gray85"	"gray86"	"gray87"



[241]	"gray88"	"gray89"	"gray90"
[244]	"gray91"	"gray92"	"gray93"
[247]	"gray94"	"gray95"	"gray96"
[250]	"gray97"	"gray98"	"gray99"
[253]	"gray100"	"green"	"green1"
[256]	"green2"	"green3"	"green4"
[259]	"greenyellow"	"grey"	"grey0"
[262]	"grey1"	"grey2"	"grey3"
[265]	"grey4"	"grey5"	"grey6"
[268]	"grey7"	"grey8"	"grey9"
[271]	"grey10"	"grey11"	"grey12"
[274]	"grey13"	"grey14"	"grey15"
[277]	"grey16"	"grey17"	"grey18"
[280]	"grey19"	"grey20"	"grey21"
[283]	"grey22"	"grey23"	"grey24"
[286]	"grey25"	"grey26"	"grey27"
[289]	"grey28"	"grey29"	"grey30"
[292]	"grey31"	"grey32"	"grey33"
[295]	"grey34"	"grey35"	"grey36"
[298]	"grey37"	"grey38"	"grey39"
[301]	"grey40"	"grey41"	"grey42"
[304]	"grey43"	"grey44"	"grey45"
[307]	"grey46"	"grey47"	"grey48"
[310]	"grey49"	"grey50"	"grey51"
[313]	"grey52"	"grey53"	"grey54"
[316]	"grey55"	"grey56"	"grey57"
[319]	"grey58"	"grey59"	"grey60"
[322]	"grey61"	"grey62"	"grey63"
[325]	"grey64"	"grey65"	"grey66"
[328]	"grey67"	"grey68"	"grey69"
[331]	"grey70"	"grey71"	"grey72"
[334]	"grey73"	"grey74"	"grey75"
[337]	"grey76"	"grey77"	"grey78"
[340]	"grey79"	"grey80"	"grey81"
[343]	"grey82"	"grey83"	"grey84"
[346]	"grey85"	"grey86"	"grey87"
[349]	"grey88"	"grey89"	"grey90"
[352]	"grey91"	"grey92"	"grey93"
[355]	"grey94"	"grey95"	"grey96"
[358]	"grey97"	"grey98"	"grey99"
[361]	"grey100"	"honeydew"	"honeydew1"
[364]	"honeydew2"	"honeydew3"	"honeydew4"
[367]	"hotpink"	"hotpink1"	"hotpink2"

[370]	"hotpink3"	"hotpink4"	"indianred"
[373]	"indianred1"	"indianred2"	"indianred3"
[376]	"indianred4"	"ivory"	"ivory1"
[379]	"ivory2"	"ivory3"	"ivory4"
[382]	"khaki"	"khaki1"	"khaki2"
[385]	"khaki3"	"khaki4"	"lavender"
[388]	"lavenderblush"	"lavenderblush1"	"lavenderblush2"
[391]	"lavenderblush3"	"lavenderblush4"	"lawngreen"
[394]	"lemonchiffon"	"lemonchiffon1"	"lemonchiffon2"
[397]	"lemonchiffon3"	"lemonchiffon4"	"lightblue"
[400]	"lightblue1"	"lightblue2"	"lightblue3"
[403]	"lightblue4"	"lightcoral"	"lightcyan"
[406]	"lightcyan1"	"lightcyan2"	"lightcyan3"
[409]	"lightcyan4"	"lightgoldenrod"	"lightgoldenrod1"
[412]	"lightgoldenrod2"	"lightgoldenrod3"	"lightgoldenrod4"
[415]	"lightgoldenrodyellow"	"lightgray"	"lightgreen"
[418]	"lightgrey"	"lightpink"	"lightpink1"
[421]	"lightpink2"	"lightpink3"	"lightpink4"
[424]	"lightsalmon"	"lightsalmon1"	"lightsalmon2"
[427]	"lightsalmon3"	"lightsalmon4"	"lightseagreen"
[430]	"lightskyblue"	"lightskyblue1"	"lightskyblue2"
[433]	"lightskyblue3"	"lightskyblue4"	"lightslateblue"
[436]	"lightslategray"	"lightslategrey"	"lightsteelblue"
[439]	"lightsteelblue1"	"lightsteelblue2"	"lightsteelblue3"
[442]	"lightsteelblue4"	"lightyellow"	"lightyellow1"
[445]	"lightyellow2"	"lightyellow3"	"lightyellow4"
[448]	"limegreen"	"linen"	"magenta"
[451]	"magenta1"	"magenta2"	"magenta3"
[454]	"magenta4"	"maroon"	"maroon1"
[457]	"maroon2"	"maroon3"	"maroon4"
[460]	"mediumaquamarine"	"mediumblue"	"mediumorchid"
[463]	"mediumorchid1"	"mediumorchid2"	"mediumorchid3"
[466]	"mediumorchid4"	"mediumpurple"	"mediumpurple1"
[469]	"mediumpurple2"	"mediumpurple3"	"mediumpurple4"
[472]	"mediumseagreen"	"mediumslateblue"	"mediumspringgreen"
[475]	"mediumturquoise"	"mediumvioletred"	"midnightblue"
[478]	"mintcream"	"mistyrose"	"mistyrose1"
[481]	"mistyrose2"	"mistyrose3"	"mistyrose4"
[484]	"moccasin"	"navajowhite"	"navajowhite1"
[487]	"navajowhite2"	"navajowhite3"	"navajowhite4"
[490]	"navy"	"navyblue"	"oldlace"
[493]	"olivedrab"	"olivedrab1"	"olivedrab2"
[496]	"olivedrab3"	"olivedrab4"	"orange"

[499]	"orange1"	"orange2"	"orange3"
[502]	"orange4"	"orangered"	"orangered1"
[505]	"orangered2"	"orangered3"	"orangered4"
[508]	"orchid"	"orchid1"	"orchid2"
[511]	"orchid3"	"orchid4"	"palegoldenrod"
[514]	"palegreen"	"palegreen1"	"palegreen2"
[517]	"palegreen3"	"palegreen4"	"paleturquoise"
[520]	"paleturquoise1"	"paleturquoise2"	"paleturquoise3"
[523]	"paleturquoise4"	"palevioletred"	"palevioletred1"
[526]	"palevioletred2"	"palevioletred3"	"palevioletred4"
[529]	"papayawhip"	"peachpuff"	"peachpuff1"
[532]	"peachpuff2"	"peachpuff3"	"peachpuff4"
[535]	"peru"	"pink"	"pink1"
[538]	"pink2"	"pink3"	"pink4"
[541]	"plum"	"plum1"	"plum2"
[544]	"plum3"	"plum4"	"powderblue"
[547]	"purple"	"purple1"	"purple2"
[550]	"purple3"	"purple4"	"red"
[553]	"red1"	"red2"	"red3"
[556]	"red4"	"rosybrown"	"rosybrown1"
[559]	"rosybrown2"	"rosybrown3"	"rosybrown4"
[562]	"royalblue"	"royalblue1"	"royalblue2"
[565]	"royalblue3"	"royalblue4"	"saddlebrown"
[568]	"salmon"	"salmon1"	"salmon2"
[571]	"salmon3"	"salmon4"	"sandybrown"
[574]	"seagreen"	"seagreen1"	"seagreen2"
[577]	"seagreen3"	"seagreen4"	"seashell"
[580]	"seashell1"	"seashell2"	"seashell3"
[583]	"seashell4"	"sienna"	"sienna1"
[586]	"sienna2"	"sienna3"	"sienna4"
[589]	"skyblue"	"skyblue1"	"skyblue2"
[592]	"skyblue3"	"skyblue4"	"slateblue"
[595]	"slateblue1"	"slateblue2"	"slateblue3"
[598]	"slateblue4"	"slategray"	"slategray1"
[601]	"slategray2"	"slategray3"	"slategray4"
[604]	"slategrey"	"snow"	"snow1"
[607]	"snow2"	"snow3"	"snow4"
[610]	"springgreen"	"springgreen1"	"springgreen2"
[613]	"springgreen3"	"springgreen4"	"steelblue"
[616]	"steelblue1"	"steelblue2"	"steelblue3"
[619]	"steelblue4"	"tan"	"tan1"
[622]	"tan2"	"tan3"	"tan4"
[625]	"thistle"	"thistle1"	"thistle2"

[628]	"thistle3"	"thistle4"	"tomato"
[631]	"tomato1"	"tomato2"	"tomato3"
[634]	"tomato4"	"turquoise"	"turquoise1"
[637]	"turquoise2"	"turquoise3"	"turquoise4"
[640]	"violet"	"violetred"	"violetred1"
[643]	"violetred2"	"violetred3"	"violetred4"
[646]	"wheat"	"wheat1"	"wheat2"
[649]	"wheat3"	"wheat4"	"whitesmoke"
[652]	"yellow"	"yellow1"	"yellow2"
[655]	"yellow3"	"yellow4"	"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\nde New York") +
  scale_fill_manual(values = c("dodgerblue1", "mediumorchid2", "red2"))
```

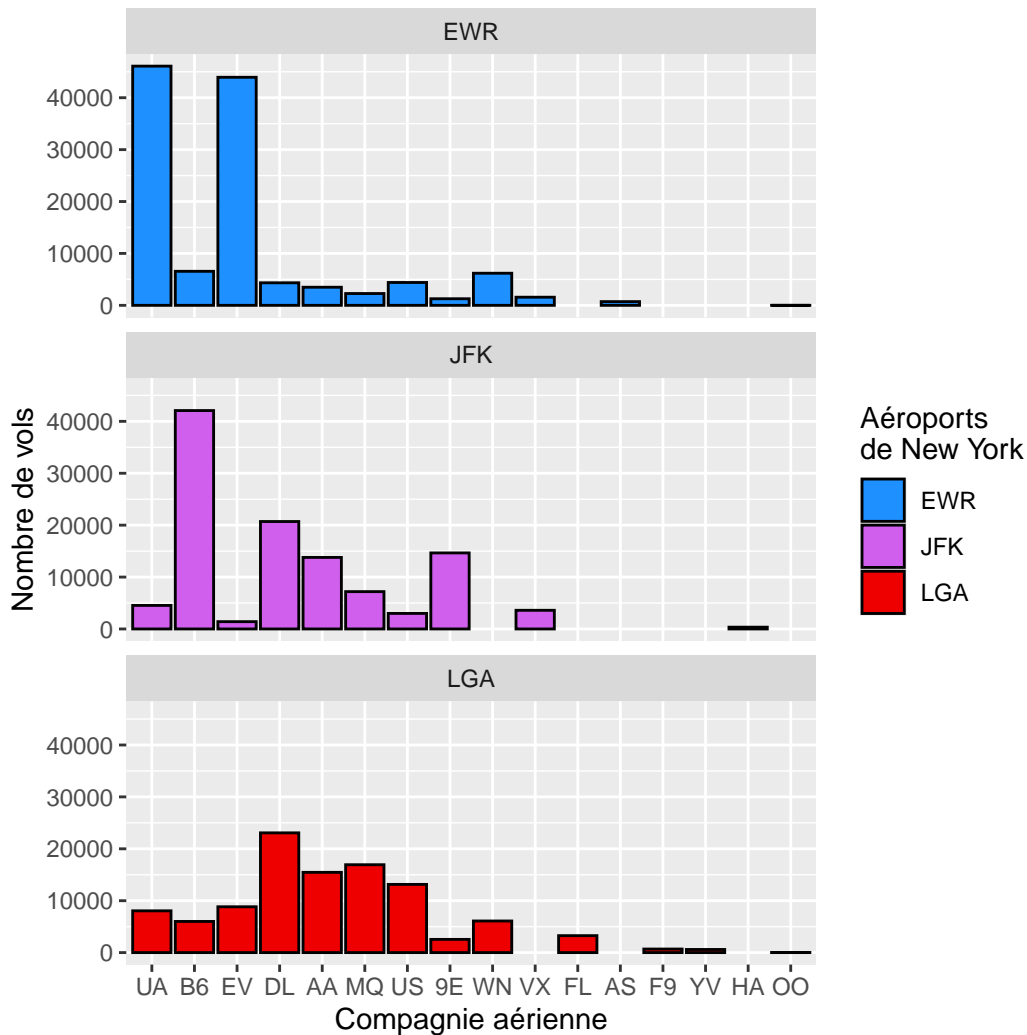


Figure 4.60: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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\nde New York") +
scale_fill_manual(values = c("#6f71f2", "#6ff299", "#f2b86f"))

```

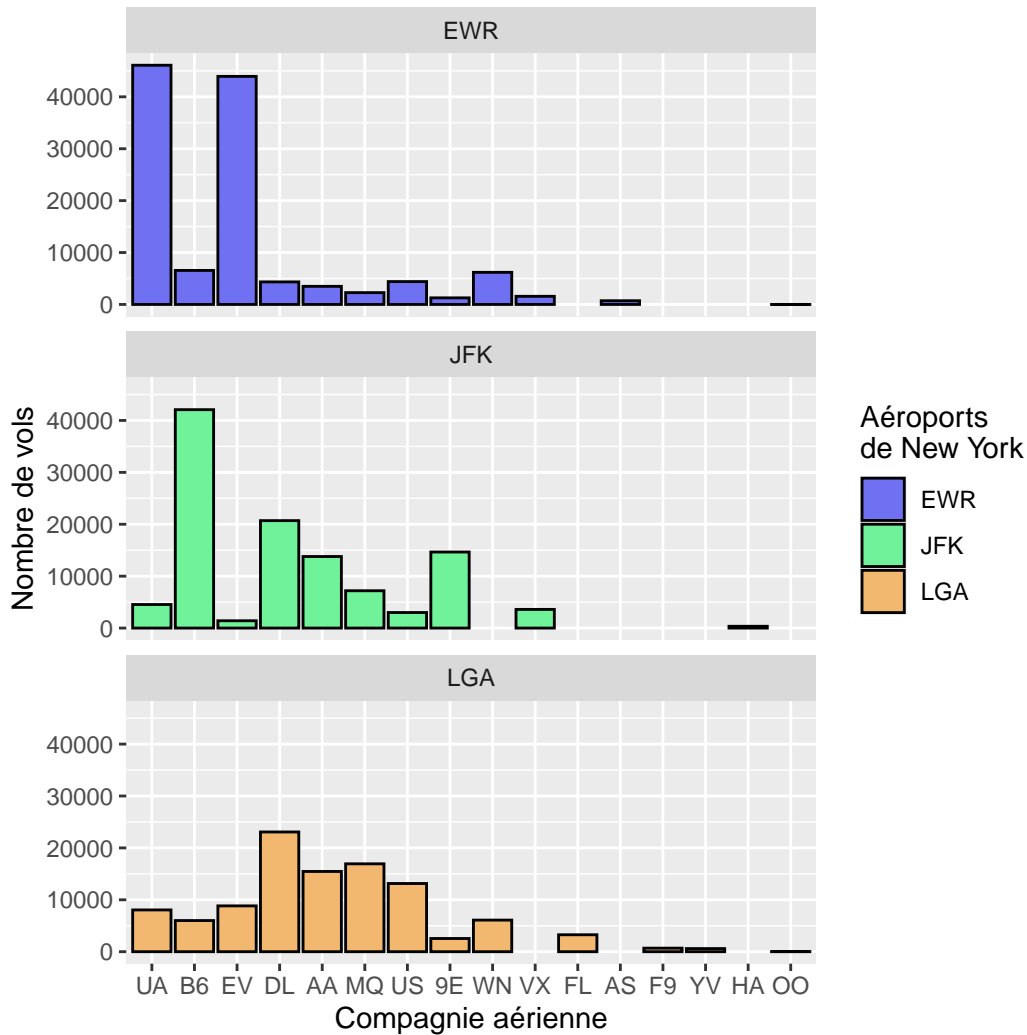


Figure 4.61: Nombre de vols par compagnie aérienne au départ des 3 aéroports de New York en 2013.

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 @ref(sec-fig:varcolor) et @ref(sec-fig:varcolor2) et que l'on modifie les palettes de couleurs, notez que les fonctions utilisées ne sont pas les mêmes :

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

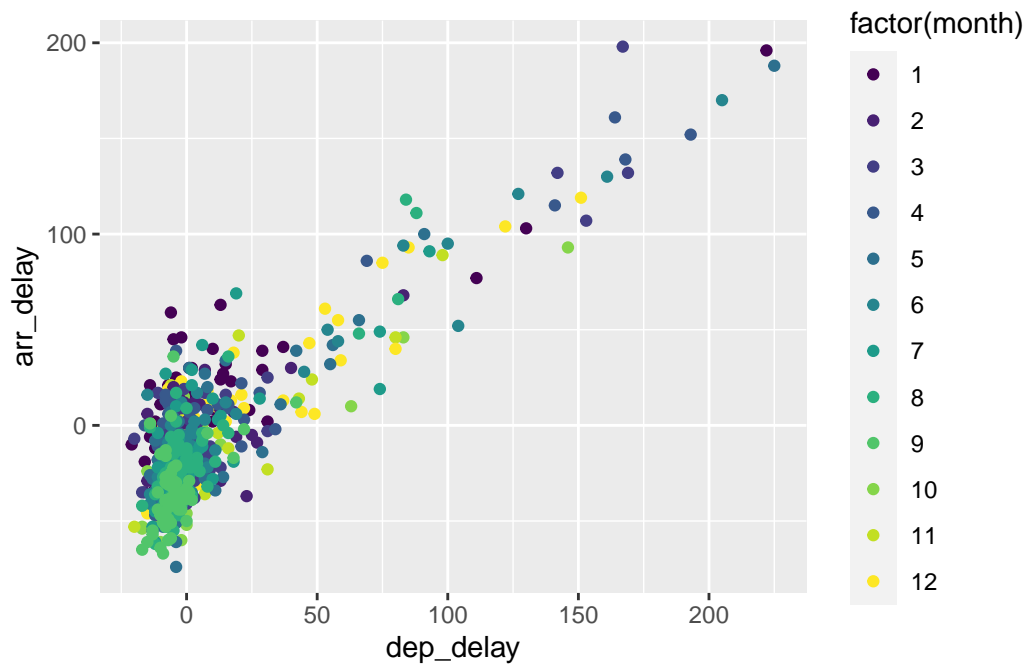


Figure 4.62: Association de color à une variable catégorielle.

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

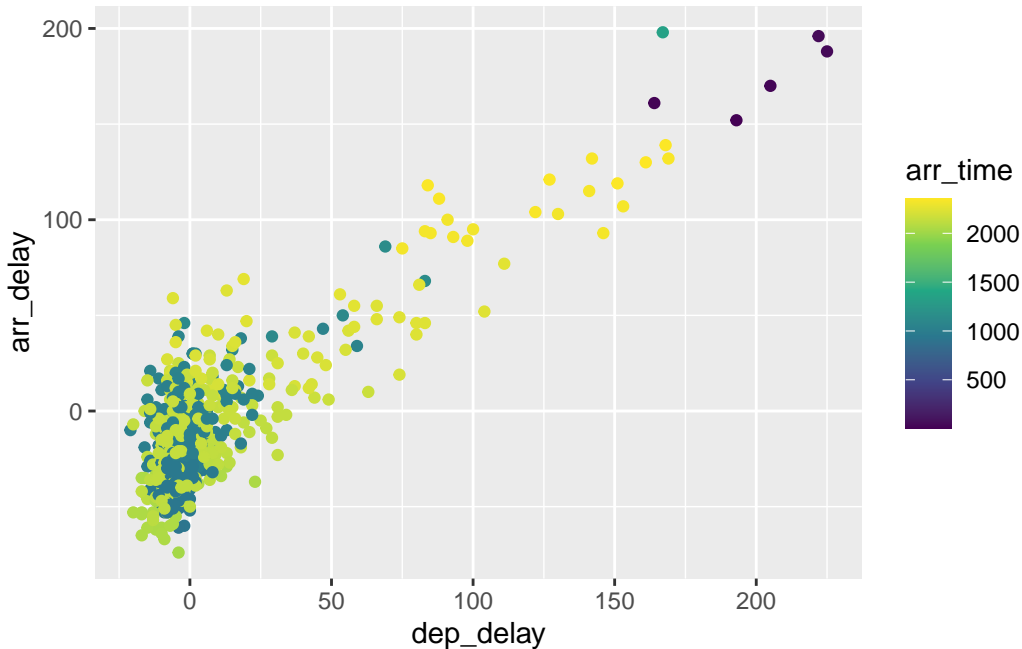


Figure 4.63: Association de `color` à une variable numérique.

Pour les 2 figures @ref(sec-fig:varcolorviridis) et @ref(sec-fig:varcolorviridis2), 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 @ref(sec-fig:varcolorviridis), 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 "discontinue". À l'inverse, pour le graphique @ref(sec-fig:varcolorviridis2), 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 déroulant qui apparaît sous votre curseur.

### 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.

Reprenons par exemple le code de la figure @ref(sec-fig:barfacetbrewer) 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")
```

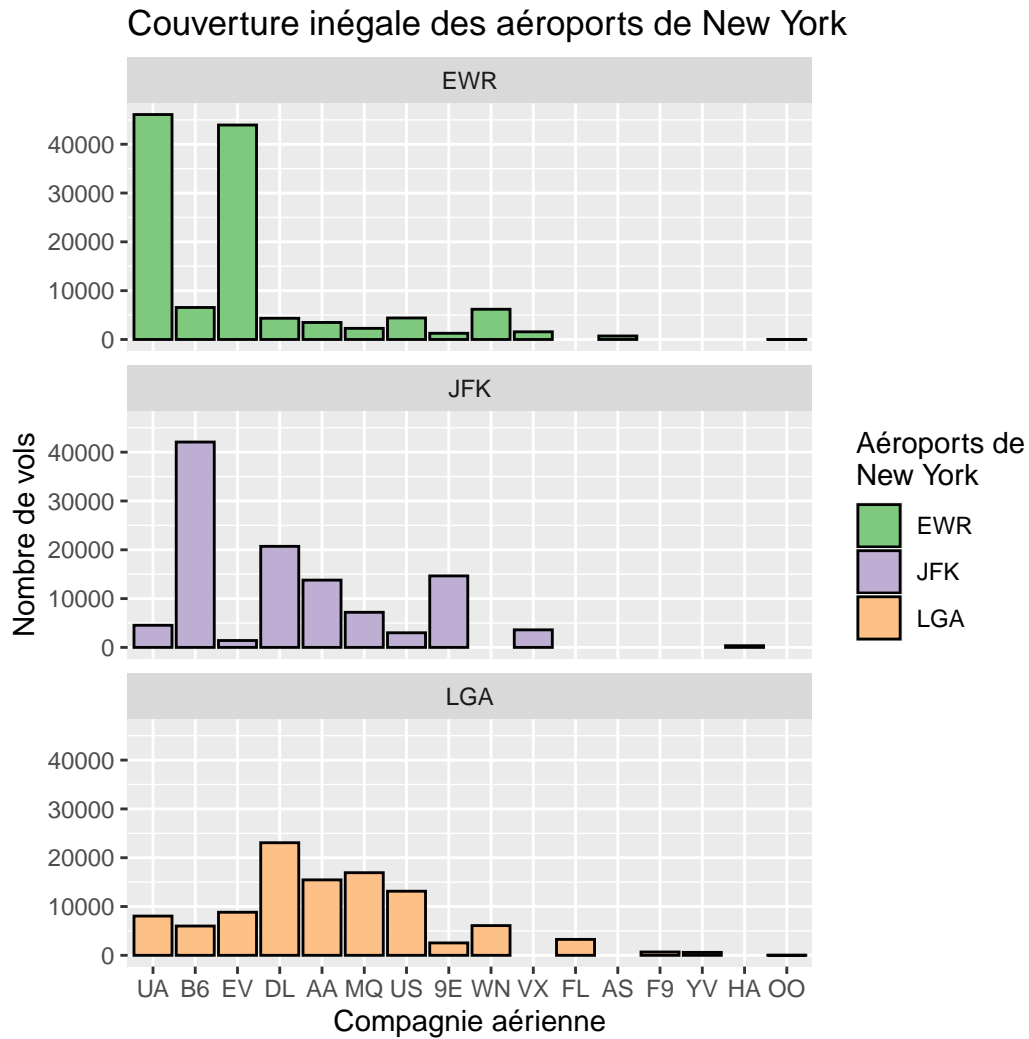


Figure 4.64: (ref:gray)

(ref:gray) Utilisation du thème par défaut : `theme_gray()`.

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",
```

```

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

```

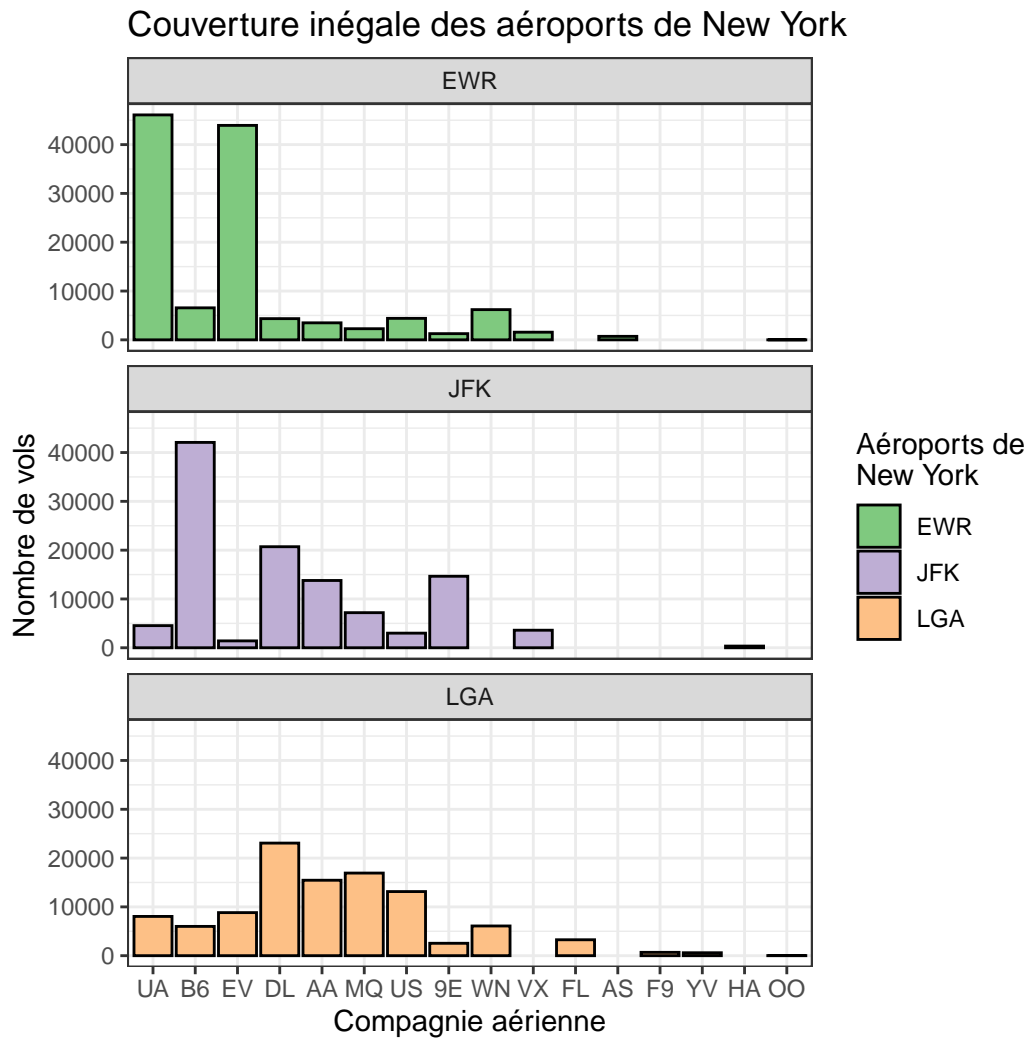


Figure 4.65: (ref:bw)

(ref:bw) 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()` : theme 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()
```

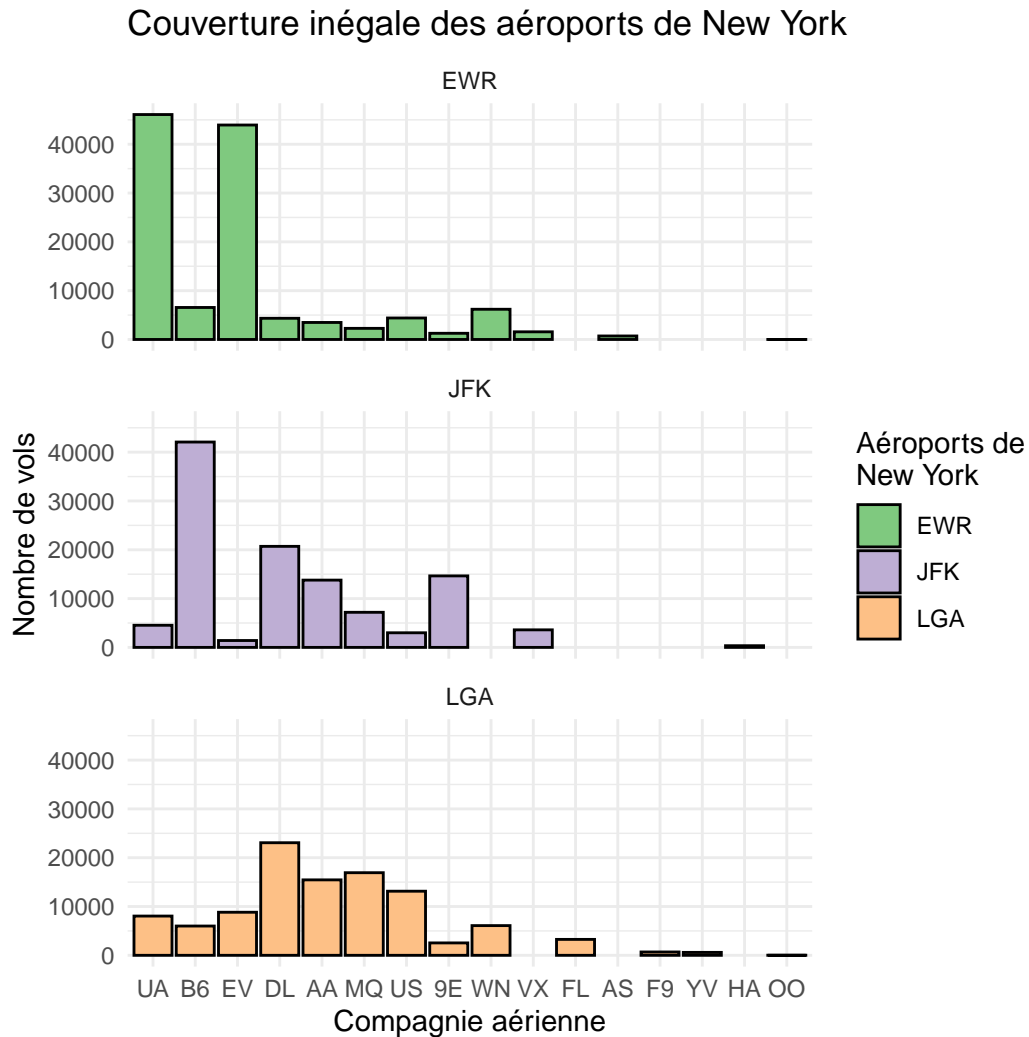


Figure 4.66: Utilisation du thème minimaliste.

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 @ref(sec-fig:themefont) 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\nNew York",
     title = "Couverture inégale des aéroports de New York") +
scale_fill_brewer(palette = "Accent") +
theme_minimal(base_family = "FuturaLT")

```

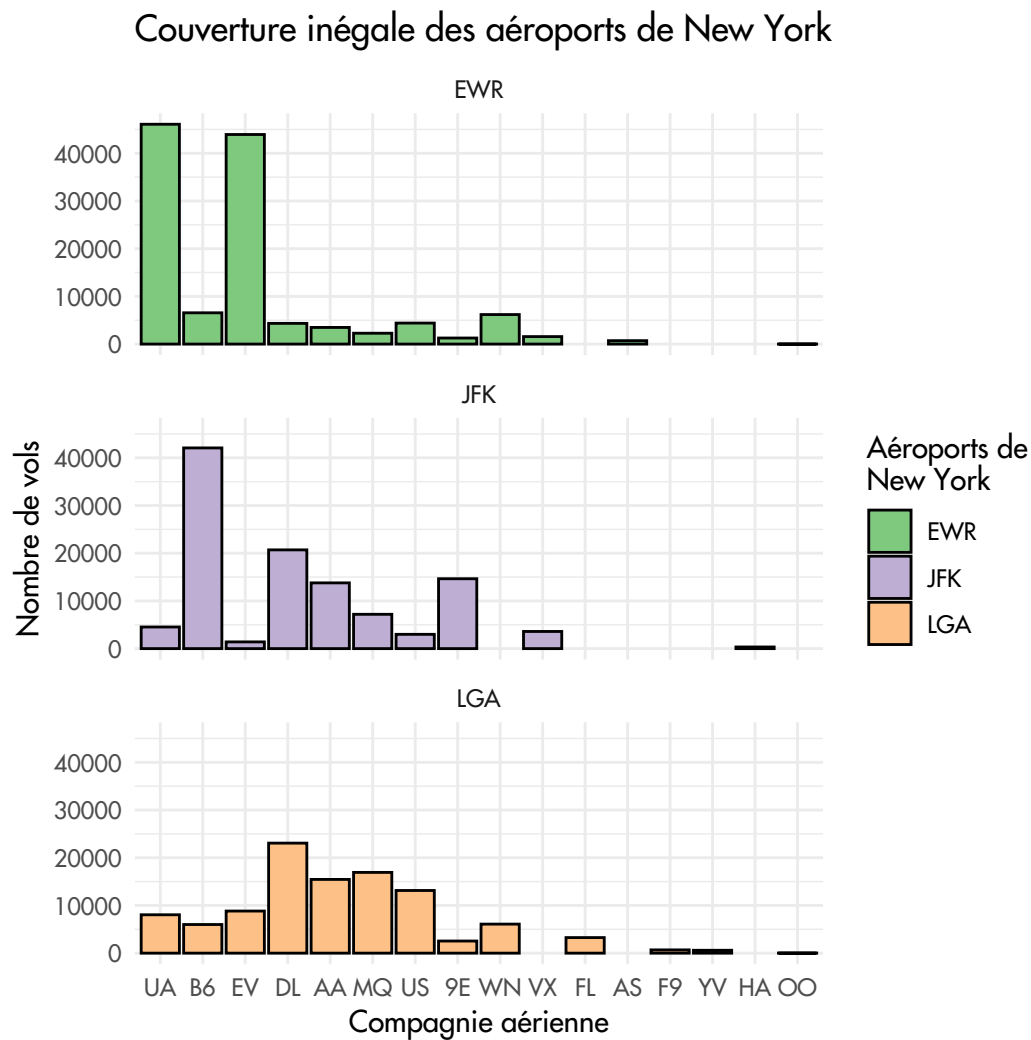


Figure 4.67: Modification de la police de caractères.

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 `tidyr` et `dplyr`.

---

## 4.10 Exercices

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

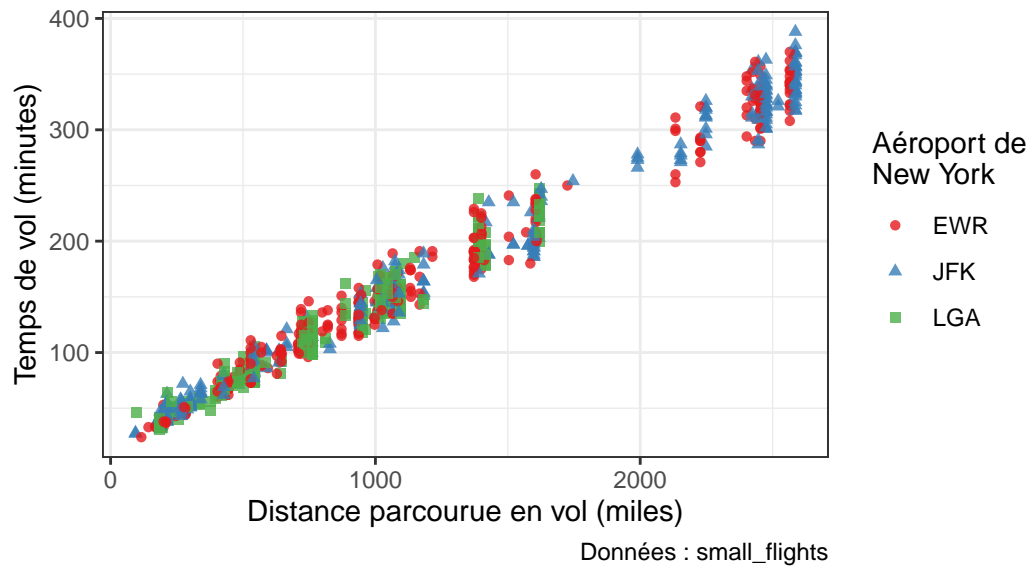
```
set.seed(1234)
small_flights <- flights %>%
  sample_n(1000) %>%
  filter(!is.na(arr_delay),
         distance < 3000)
```

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 @ref(sec-viz), et en vous servant de ce nouveau jeu de données, tapez les commandes qui permettent de produire le graphique ci-dessous :

## Relation entre le temps de vol et la distance parcourue

Seuls les vols au départ de JFK et Newark parcourent plus de 1600 miles



Quelques indices :

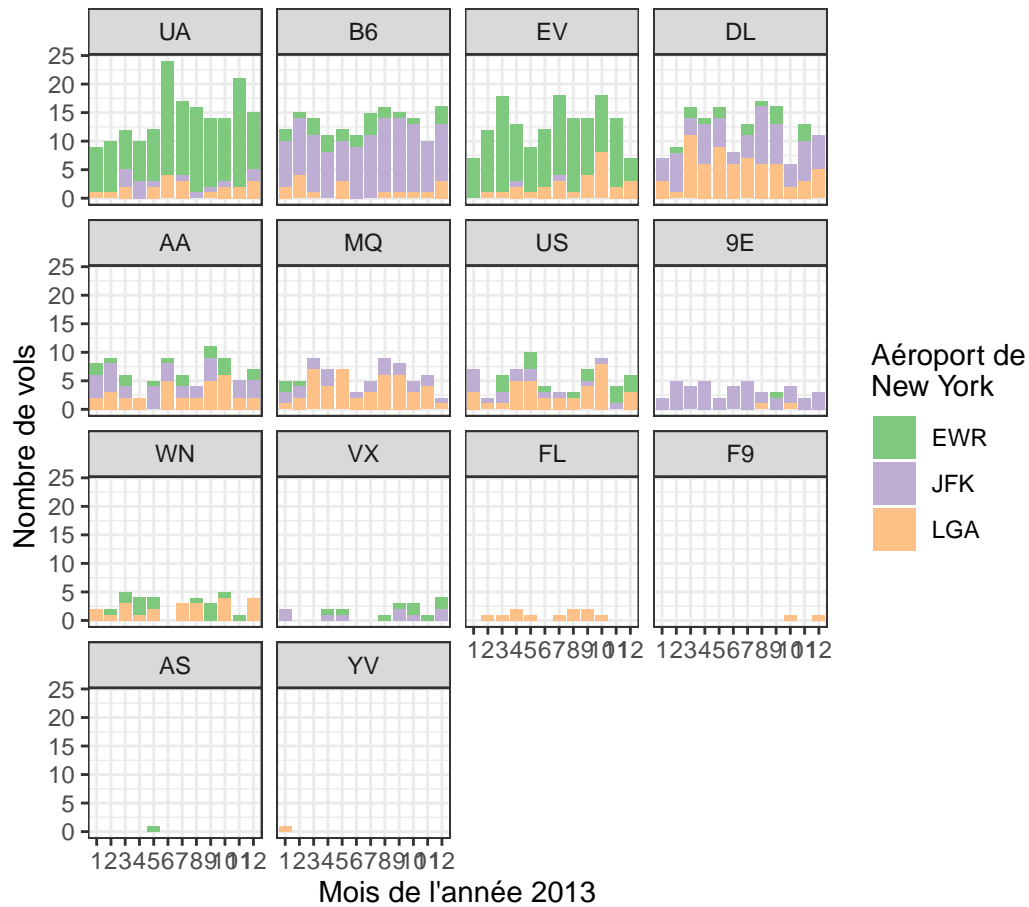
- Les couleurs utilisées sont celles de la palette `Set1` du package `RColorBrewer`.
- Les variables utilisées sont `origin`, `air_time` et `distance`.
- La transparence des symboles est fixée à 0.8.

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



## Évolution mensuelle du trafic aérien New Yorkais en 2013

Seules 9 compagnies aériennes sur 14 ont déservi New York toute l'année



Données : small\_flights

Quelques indices :

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

## Références

- Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, et Dewey Dunnington. 2022. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. <https://CRAN.R-project.org/package=ggplot2>.
- Wilkinson, Leland. 2005. *The grammar of graphics*. 2nd éd. New-York: Springer-Verlag. <https://www.springer.com/us/book/9780387245447>.