Pedro J. Aphalo

Learn R: As a Language

Supplementary Chapters and Appendixes

Contents

Foreword to supplementary chapters

This file contains additional on-line only open-access chapters for the book *Learn R: As a Language* (Aphalo 2020) with specialized and/or advanced content. It also includes information on non-code breaking and code-breaking changes to some of the packages used in the book, when those changes or new features are important. Chapters numbered 9 and higher are in this file, while chapters 1 to 8 are in the book published in *The R Series*. Chapter 9 starts on page 351. This document includes cross references by chapter, section and page numbers to chapters and pages both in this PDF as in the published book.

In the book as a whole I have emphasized the `R` language and its syntax as this is at the core of learning to use a language. Other aspects of `R`, which can be also important in its everyday use were left out to keep the length of the printed book from growing excessively. In addition I avoided including material that I expected to become outdated fast or that are well covered elsewhere, such as how to adjust the visual design of ggplots (see the on-line tutorial *A ggplot2 Tutorial for Beautiful Plotting in R* (Scherer 2019)) or data analysis and statistics (such as the books *Modern Statistics for Modern Biology* (Holmes and Huber 2019), *Generalized Additive Models* (Wood 2017), *The R Book* (Crawley 2012), *An R Companion to Applied Regression* (Fox and Weisberg 2010), *A Handbook of Statistical Analyses Using R* (Everitt and Hothorn 2009), *Introductory Statistics with R* (Dalgaard 2008) and others).

This file will be updated more frequently than the book and will also include errata to the book when needed. This updates and other information related to the book are available at `https://www.learnr-book.info`.

Preface (Reproduced from book)

> "Suppose that you want to teach the 'cat' concept to a very young child. Do you explain that a cat is a relatively small, primarily carnivorous mammal with retractible claws, a distinctive sonic output, etc.? I'll bet not. You probably show the kid a lot of different cats, saying 'kitty' each time, until it gets the idea. To put it more generally, generalizations are best made by abstraction from experience."
>
> R. P. Boas
> *Can we make mathematics intelligible?*, 1981

This book covers different aspects of the use of the R language. Chapters 1 to 5 describe the R language itself. Later chapters describe extensions to the R language available through contributed *packages*, the *grammar of data* and the *grammar of graphics*. In this book, explanations are concise but contain pointers to additional sources of information, so as to encourage the development of a routine of independent exploration. This is not an arbitrary decision, this is the normal *modus operandi* of most of us who use R regularly for a variety of different problems. Some have called approaches like the one used here "learning the hard way," but I would call it "learning to be independent."

I do not discuss statistics or data analysis methods in this book; I describe R as a language for data manipulation and display. The idea is for you to learn the R language in a way comparable to how children learn a language: they work out what the rules are, simply by listening to people speak and trying to utter what they want to tell their parents. Of course, small children receive some guidance, but they are not taught a prescriptive set of rules like when learning a second language at school. Instead of listening, you will read code, and instead of speaking, you will try to execute R code statements on a computer—i.e., you will try your hand at using R to tell a computer what you want it to compute. I do provide explanations and guidance, but the idea of this book is for you to use the numerous examples to find out by yourself the overall patterns and coding philosophy behind the R language. Instead of parents being the sound board for your first utterances in R, the computer will play this role. You will *play* by modifying the examples, see how the computer responds: does R understand you or not? Using a language actively is the most efficient way of learning it. By using it, I mean actually reading, writing, and running scripts or programs (copying and pasting, or typing ready-made examples from books or the internet, does not qualify as using a language).

I have been using R since around 1998 or 1999, but I am still constantly learning new things about R itself and R packages. With time, it has replaced in my work as a researcher and teacher several other pieces of software: SPSS, Systat, Origin, MS-Excel, and it has become a central piece of the tool set I use for producing lecture slides, notes, books, and even web pages. This is to say that it is the most useful piece of software and programming language I have ever learned to use. Of course, in time it will be replaced by something better, but at the moment it is a key language to learn for anybody with a need to analyze and display data.

What is a language? A language is a system of communication. R as a language allows us to communicate with other members of the R community, and with computers. As with all languages in active use, R evolves. New "words" and new "constructs" are incorporated into the language, and some earlier frequently used ones are relegated to the fringes of the corpus. I describe current usage and "modisms" of the R language in a way accessible to a readership unfamiliar with computer science but with some background in data analysis as used in biology, engineering, or the humanities.

When teaching, I tend to lean toward challenging students, rather than telling an over-simplified story. There are two reasons for this. First, I prefer as a student, and I learn best myself, if the going is not too easy. Second, if I would hide the tricky bits of the R language, it would make the reader's life much more difficult later on. You will not remember all the details; nobody could. However, you most likely will remember or develop a sense of when you need to be careful or should check the details. So, I will expose you not only to the usual cases, but also to several exceptions and counterintuitive features of the language, which I have highlighted with icons. Reading this book will be about exploring a new world; this book aims to be a travel guide, but neither a traveler's account, nor a cookbook of R recipes.

Keep in mind that it is impossible to remember everything about R! The R language, in a broad sense, is vast because its capabilities can be expanded with independently developed packages. Learning to use R consists of learning the basics plus developing the skill of finding your way in R and its documentation. In early 2020, the number of packages available in the Comprehensive R Archive Network (CRAN) broke the 15,000 barrier. CRAN is the most important, but not only, public repository for R packages. How good a command of the R language and packages a user needs depends on the type of activities to be carried out. This book attempts to train you in the use of the R language itself, and of popular R language extensions for data manipulation and graphical display. Given the availability of numerous books on statistical analysis with R, in the present book I will cover only the bare minimum of this subject. The same is true for package development in R. This book is somewhere in-between, aiming at teaching programming in the small: the use of R to automate the drudgery of data manipulation, including the different steps spanning from data input and exploration to the production of publication-quality illustrations.

As with all "rich" languages, there are many different ways of doing things in R. In almost all cases there is no one-size-fits-all solution to a problem. There is always a compromise involved, usually between time spent by the user and processing time required in the computer. Many of the packages that are most popular nowadays did not exist when I started using R, and many of these packages make

new approaches available. One could write many different R books with a given aim using substantially different ways of achieving the same results. In this book, I limit myself to packages that are currently popular and/or that I consider elegantly designed. I have in particular tried to limit myself to packages with similar design philosophies, especially in relation to their interfaces. What is elegant design, and in particular what is a friendly user interface, depends strongly on each user's preferences and previous experience. Consequently, the contents of the book are strongly biased by my own preferences. I have tried to write examples in ways that execute fast without compromising readability. I encourage readers to take this book as a starting point for exploring the very many packages, styles, and approaches which I have not described.

I will appreciate suggestions for further examples, and notification of errors and unclear sections. Because the examples here have been collected from diverse sources over many years, not all sources are acknowledged. If you recognize any example as yours or someone else's, please let me know so that I can add a proper acknowledgement. I warmly thank the students who have asked the questions and posed the problems that have helped me write this text and correct the mistakes and voids of previous versions. I have also received help on online forums and in person from numerous people, learned from archived e-mail list messages, blog posts, books, articles, tutorials, webinars, and by struggling to solve some new problems on my own. In many ways this text owes much more to people who are not authors than to myself. However, as I am the one who has written this version and decided what to include and exclude, as author, I take full responsibility for any errors and inaccuracies.

Why have I chosen the title "*Learn R: As a Language*"? This book is based on exploration and practice that aims at teaching to express various generic operations on data using the R language. It focuses on the language, rather than on specific types of data analysis, and exposes the reader to current usage and does not spare the quirks of the language. When we use our native language in everyday life, we do not think about grammar rules or sentence structure, except for the trickier or unfamiliar situations. My aim is for this book to help you grow to use R in this same way, to become fluent in R. The book is structured around the elements of languages with chapter titles that highlight the parallels between natural languages like English and the R language.

I encourage you to approach R like a child approaches his or her mother tongue when first learning to speak: do not struggle, just play, and fool around with R! If the going gets difficult and frustrating, take a break! If you get a new insight, take a break to enjoy the victory!

Acknowledgements

First I thank Jaakko Heinonen for introducing me to the then new R. Along the way many well known and not so famous experts have answered my questions in usenet and more recently in Stackoverflow. As time went by, answering other people's questions, both in the internet and in person, became the driving force

for me to delve into the depths of the R language. Of course, I still get stuck from time to time and ask for help. I wish to warmly thank all the people I have interacted with in relation to R, including members of my own research group, students participating in the courses I have taught, colleagues I have collaborated with, authors of the books I have read and people I have only met online or at conferences. All of them have made it possible for me to write this book. This has been a time consuming endeavour which has kept me too many hours away from my family, so I specially thank Tarja, Rosa and Tomás for their understanding. I am indebted to Tarja Lehto, Titta Kotilainen, Tautvydas Zalnierius, Fang Wang, Yan Yan, Neha Rai, Markus Laurel, other colleagues, students and anonymous reviewers for many very helpful comments on different versions of the book manuscript, Rob Calver, as editor, for his encouragement and patience during the whole duration of this book writing project, Lara Spieker, Vaishali Singh, and Paul Boyd for their help with different aspects of this project.

Icons used to mark different content

Text boxes are used throughout the book to highlight content that plays specific roles in the learning process or that require special attention from the reader. Each box contains one of five different icons that indicate the type of its contents as described below.

 Signals *playground* boxes which contain open-ended exercises—ideas and pieces of R code to play with at the R console.

 Signals *advanced playground* boxes which will require more time to play with before grasping concepts than regular *playground* boxes.

 Signals important bits of information that must be remembered when using R—i.e., explain some unusual feature of the language.

 Signals in-depth explanations of specific points that may require you to spend time thinking, which in general can be skipped on first reading, but to which you should return at a later peaceful time, preferably with a cup of coffee or tea.

 Signals text boxes providing general information not directly related to the R language.

Part I

More on Grammar of Graphics

9

Grammar of Graphics: What's New

xxx

NN

9.1 Aims of this chapter

Version 3.3.0 of package 'ggplot2' was released after the book went to press. Some of the new features and changes are important because they remove some limitations inherent to earlier versions. My package 'ggpmisc' has tracked also some of these changes and these new features will be used also in later chapters in this same document.

I will discuss in this chapter two enhancements: finer control of the stage at which mapping of variables to aesthetics takes place, and the flipping of stats, i.e., swapping of `x` and `y` aesthetics coordinately before and after computations.

A brief discussion of the new binned scales as also included. Meanwhile, do check the documentation as they can be very useful and make some types of plots easier to read.

9.2 Packages used in this chapter

If the packages used in this chapter are not yet installed in your computer, you can install them with the code shown below, as long as package 'learnrbook' (>= 1.0.3) is already installed.

```
packages_at_cran <-
  setdiff(learnrbook::pkgs_ch_ggplot_extra, learnrbook::pkgs_at_github)
install.packages(packages_at_cran)
for (p in learnrbook::pkgs_at_github) {
  devtools::install_github(p)
}
```

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(ggplot2)
library(ggpmisc)
```

We set a font of larger size than the default

```
theme_set(theme_grey(14))
```

9.3 Delayed mapping

Delayed mapping of variables to aesthetics has been possible in 'ggplot2' for a long time using as notation to enclose the name of a variable returned by a statistic in `...`, but this notation has been deprecated some time ago and replaced by `stat()`. In both cases, this imposed a limitation: it was impossible to map a computed variable to the same aesthetic as input to the statistic and to the geometry in the same layer. There were also some other quirks that prevented passing some arguments to the geometry through the dots `...` parameter of a statistic.

In version 3.3.0 of 'ggplot2' the syntax used for mapping variables to aesthetics was changed adding functions `stage()`, `after_stat()` and `after_scale()`. Function `after_stat()` is essentially a new name for `stat()`.

As described in section **??** on page **??** there are three steps through which data to be plotted go. Variables in the data frame passed as argument to `data` are mapped to aesthetics before they are received as input by a statistic (possibly `stat_identity()`). The mappings of variables in the data frame returned statistics are the input to the geometry. This mapping is usually handled through defaults coded in the layer function definitions. However, the user can also control this mapping explicitly using `after_stat()`, which lets us differentiate between the data frame supplied by the user and that returned by the statistic. The third stage was not accessible, but lack of access was usually not insurmountable. Now this third stage can be accessed with `after_scale()` and this can make coding simpler.

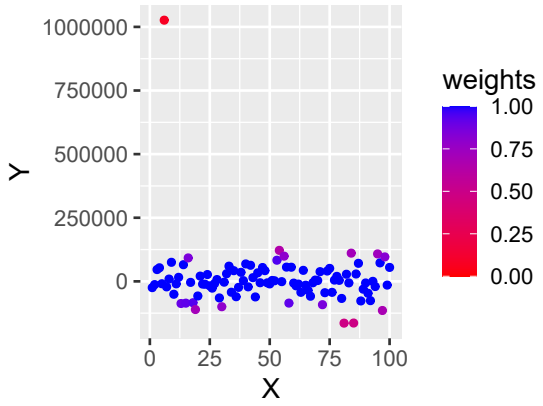
The 'ggplot2' documentation gives several good examples of cases when the new syntax is useful. I give here a different example. User-coded transformations of the data returned by a statistic are best handled using scale transformations, but when the intention is to combine different computed variables returned by a statistic, or the values returned by a statistic with values in the data supplied as argument to `data` we need to control mapping at multiple stages.

```
# we use capital letters X and Y as variable names to distinguish
# them from the x and y aesthetics
set.seed(4321)
X <- 1:100
Y <- (X + X^2 + X^3) + rnorm(length(X), mean = 0, sd = mean(X^3) / 4)
my.data <- data.frame(X, Y)
my.data.outlier <- my.data
my.data.outlier[6, "Y"] <- my.data.outlier[6, "Y"] * 10
```

```
my.formula <- y ~ poly(x, 3, raw = TRUE)
```

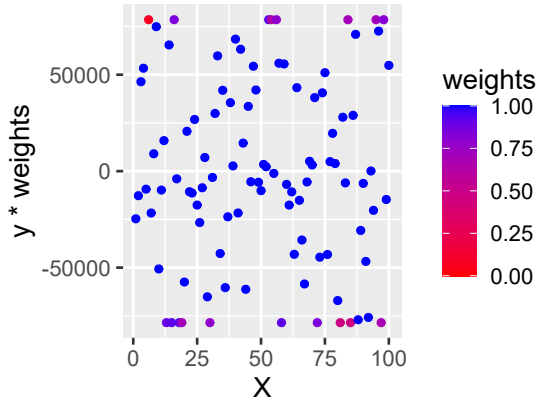
For the first plot it is enough to use `after_stat()`, as the `colour` aesthetic is only to `geom_point()` which is used by default.

```
ggplot(my.data.outlier, aes(x = X, y = Y)) +
  stat_fit_residuals(formula = my.formula, method = "rlm",
                     mapping = aes(colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_gradient(low = "red", high = "blue", limits = c(0, 1),
                       guide = "colourbar")
```



For the second plot we need to use `stage()` to be able to distinguish the mapping ahead of the statistic (`start`) from that after the statistic, i.e., ahead of the geometry.

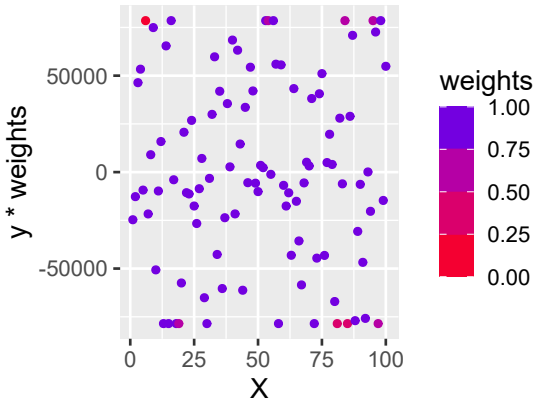
```
ggplot(my.data.outlier) +
  stat_fit_residuals(formula = my.formula,
                     method = "rlm",
                     mapping = aes(x = X,
                                   y = stage(start = Y,
                                             after_stat = y * weights),
                                   colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_gradient(low = "red", high = "blue", limits = c(0, 1),
                       guide = "colourbar")
```



9.4 Binned scales

Before version 3.3.0 of 'ggplot2' only two types of scales were available, continuous and discrete. A third type of scales (implemented for all the aesthetics where relevant) was added in version 3.3.0 called *binned*. They are to be used with continuous variables, but they discretize the continuous values into bins or classes, each for a range of values, and then represent them in the plot using a discrete set of values. We exemplify this by replotting the last figure above replacing `scale_color_gradient()` by `scale_color_binned()`.

```
ggplot(my.data.outlier) +
  stat_fit_residuals(formula = my.formula,
                     method = "rlm",
                     mapping = aes(x = X,
                                   y = stage(start = Y,
                                             after_stat = y * weights),
                                   colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_binned(low = "red", high = "blue", limits = c(0, 1),
                     guide = "colourbar", n.breaks = 5)
```



9.5 Flipped plot layers

Although it is the norm to design plots so that the independent variable is on the x axis, i.e., mapped to the `x` aesthetic, there are situations where swapping the roles of x and y is useful. In 'ggplot2' this is described as *flipping the orientation* of a plot. In the present section I exemplify both cases where the flipping is automatic and where it requires user intervention.

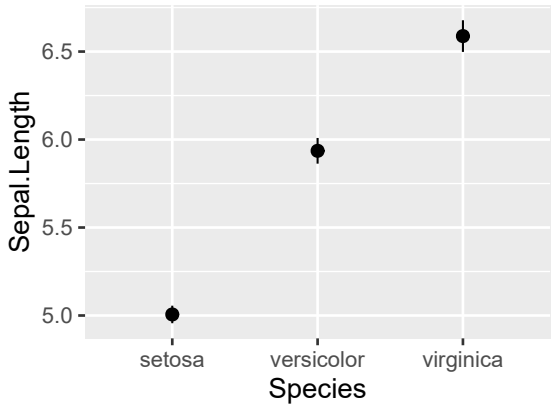
In the case of ordinary least squares (OLS), regressions of y on x and of x on y in most cases yield different fitted lines, even if R^2 is consistent.

This is due to the assumption that x values are known, either set or measured without error, i.e., not subject to random variation. All unexplained variation in the data is assumed to be in y. See Chapter ?? on page ?? or consult a Statistics book such as *Modern Statistics for Modern Biology* (Holmes and Huber 2019, pp. 168–170) for additional information.

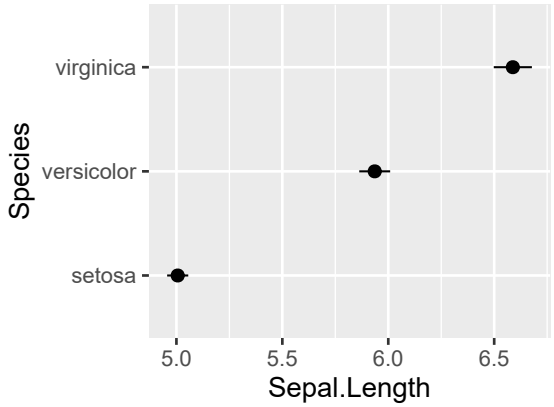
Package 'ggstance' defines horizontal versions of common ggplot *geometries*, *statistics* and *positions*. Although 'ggplot2' has had `coord_flip()` for a long time, 'ggstance' provided a more intuitive user interface and more consistent plot formatting. As the time of writing, 'ggplot2' version 3.3.5, supports flipping in most geometries and statistics where it is meaningful using a new syntax. This new approach is different to the flip of the coordinate system, and similar to that implemented by package 'ggstance'. However, instead of defining new horizontal layer functions as in 'ggstance', now the orientation of many layer functions from 'ggplot2' can change. This has made 'ggstance' nearly redundant and the coding of flipped plots easier and more intuitive.

When a factor is mapped to x or y flipping is automatic. However, there are cases that can require user intervention. For example, manual flipping is required if both x and y are mapped to continuous variables and we want `stat_smooth()` to make a fit of x on y. The first two examples exemplify automatic flipping using `stat_summary()`. Other statistics like `stat_boxplot()`, `stat_histogram()` and `stat_density()` behave similarly. Dodging and jitter do not need any special syntax as it was the case with package 'ggstance'.

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) +
    stat_summary(fun.data = "mean_se")
```

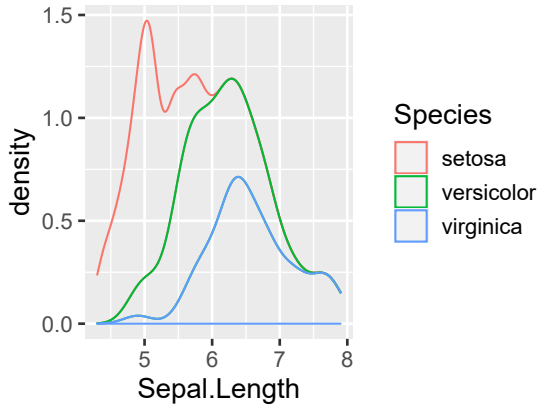



```
ggplot(iris, aes(x = Sepal.Length, y = Species)) +
    stat_summary(fun.data = "mean_se")
```

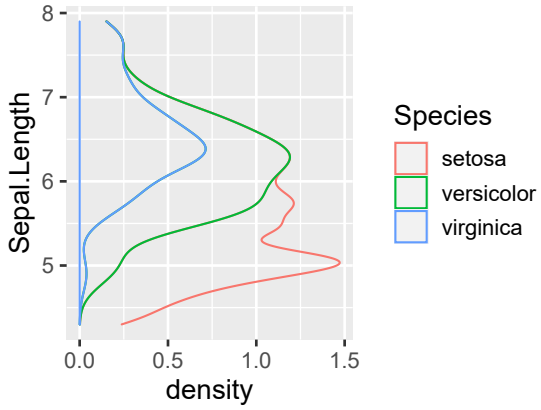


When we map a variable only to x or y the flip is also automatic.

```
ggplot(iris, aes(x = Sepal.Length, color = Species)) +
    stat_density(fill = NA)
```



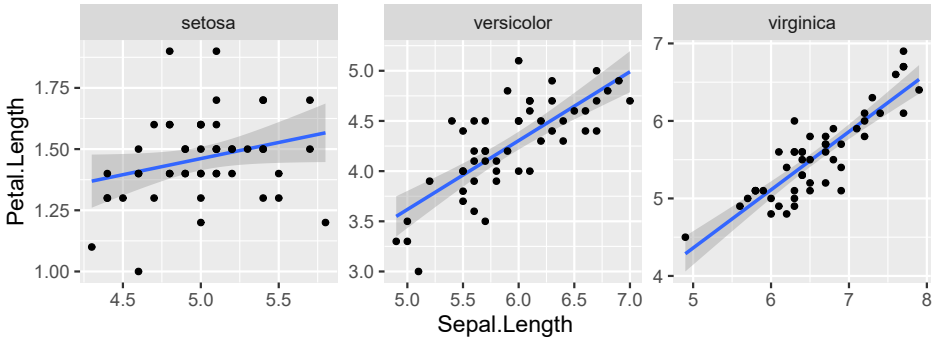

```
ggplot(iris, aes(y = Sepal.Length, color = Species)) +
    stat_density(fill = NA)
```



In the case of two continuous variables the default is to take x as independent and y as dependent. This matters, of course, when computations as in model fit-

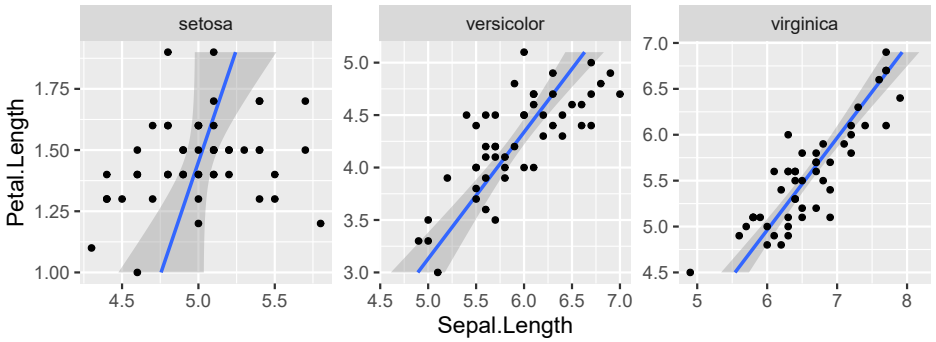
ting treat x and y differently. In this case parameter `orientation` can be used to indicate which of x or y is the independent or explanatory variable.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
  stat_smooth(method = "lm", formula = y ~ x) +
  geom_point() +
  facet_wrap(~Species, scales = "free")
```



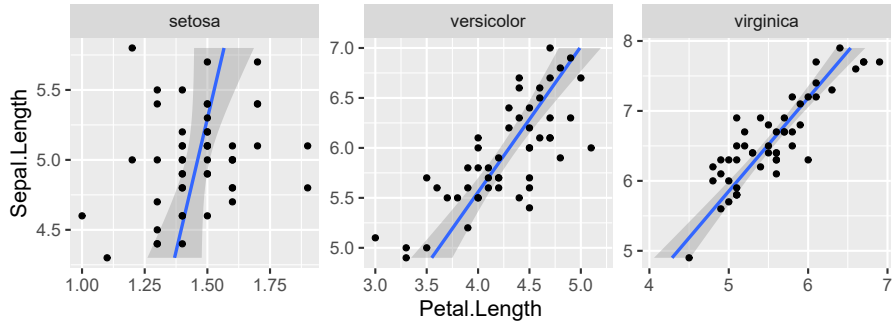
With `orientation = "y"` we tell that y is the independent variable. In the case of `geom_smooth()` this means implicitly swapping x and y in `formula`.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_smooth(method = "lm", formula = y ~ x, orientation = "y") +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```



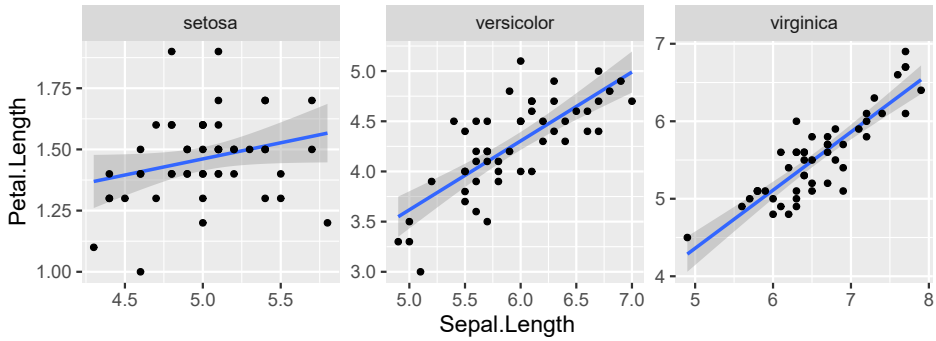
Flipping the orientation of plot layers with `orientation = "y"` is not equivalent to flipping the whole plot with `coord_flip()`. In the first case which axis is considered independent for computation changes but not the positions of the axes in the plot, while in the second case the position of the x and y axes in the plot is swapped. So, when coordinates are flipped the `x` aesthetic is plotted on the vertical axis and the `y` aesthetic on the horizontal axis, but the role of the variable mapped to the `x` aesthetic remains as explanatory variable.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_smooth(method = "lm", formula = y ~ x) +
    geom_point() +
    coord_flip() +
    facet_wrap(~Species, scales = "free")
```

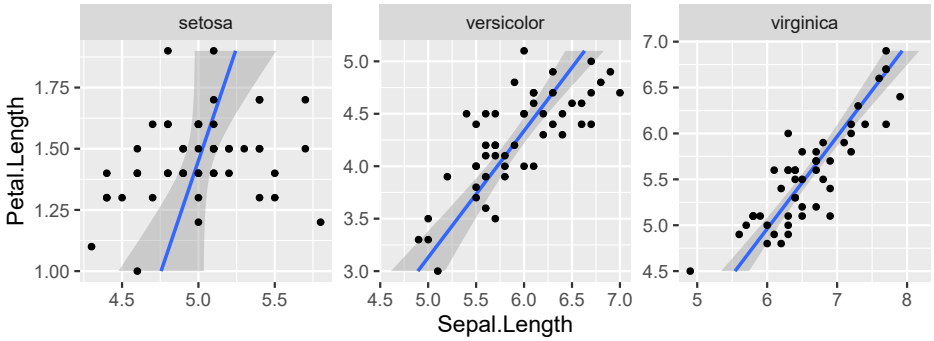


In package 'ggpmisc' as of version 0.4.1 statistics related to model fitting have an `orientation` parameter as the those from package 'ggplot2' do, but in addition directly accept formulas where x is on the lhs and y on the rhs, such as `formula = x ~ y` providing a syntax consistent with R's model fitting functions. In the first example the default `formula = y ~ x` is used, while in the second example we pass explicitly `formula = x ~ y` to force the flipping.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_poly_line() +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```

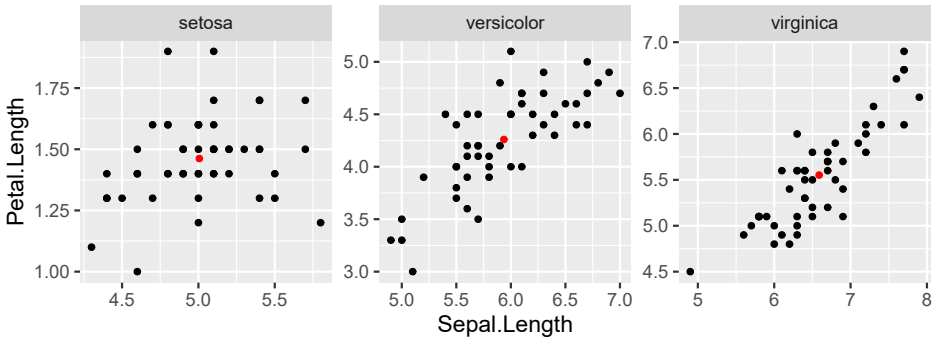



```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_poly_line(formula = x ~ y) +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```

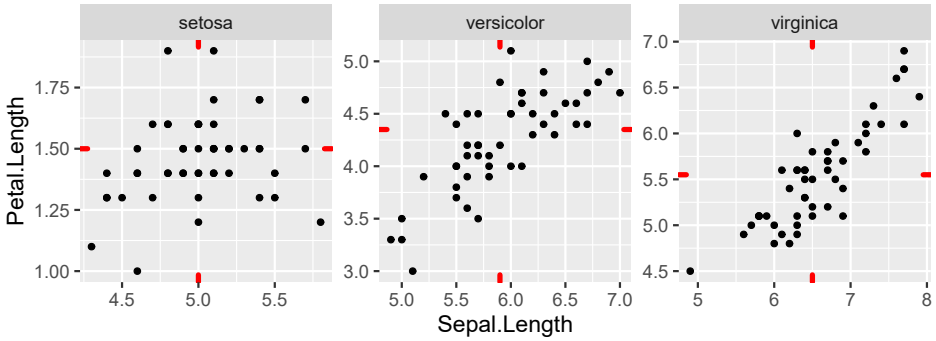


A related problem is when we need to summarize in the same plot layer x and y values. Package 'ggplot2' provides `stat_density2d()` and `stat_summary2d()`. However, `stat_summary2d()` uses bins, and is similar to `stat_density2d()` in how the computed values are returned. Package 'ggpmisc' provides two dimensional equivalents of `stat_summary()`: `stat_centroid()`, which applies the same summary function along x and y, and `stat_summary_xy()`, which accepts one function for x and one for y.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    geom_point() +
    stat_centroid(color = "red") +
    facet_wrap(~Species, scales = "free")
```




```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    geom_point() +
    stat_centroid(geom = "rug", sides = "trbl",
                  .fun = median, color = "red", size = 1.5) +
    facet_wrap(~Species, scales = "free")
```



Can the plots in the last two chunks be created also with `stat_summary()`? Which one(s) and why? When possible, recreate these plots using `stat_summary()`.

```
try(detach(package:ggpmisc))
try(detach(package:ggpp))
try(detach(package:ggplot2))
```

10

Grammar of Graphics: Fonts

> Typographic clarity consists of **legibility** and **readability**. Good legibility means that individual words are easily and quickly recognizable, and that the letters are distinct.
>
> Juuso Koponen and Jonatan Hildén
> *Data visualization handbook*, 2019

10.1 Aims of this chapter

Even though English is widely understood, I expect a considerable number of readers of *Learn R: As a Language* to use R to create plots annotated in languages other than English. In many cases these languages are written using non-Latin alphabets. A simpler but related problem is the use of fonts and font-faces different to the default ones.

When using 'ggplot2' we can rely on two different approaches to the use of other alphabets: making system fonts available within R and relying on R expressions or character strings containing *special characters* using an specific or extended font encoding (such as UTF-8 or UTF-16) or using an "external" text formatting language like LaTeX, Markdown or HTML. We describe both approaches.

10.2 Packages used in this chapter

The list of packages used in the current chapter is very long. However, there are only few incompatibilities among them. Even if one is not attempting to use functions from different packages within the same plot, one may want to use them to create different figures within the same document when using the literate approach to programming and scripting (see 3.2.4 on page 90), with incompatibilities requiring additional coding to work around them.

If the packages used in this chapter are not yet installed in your computer, you

can install them with the code shown below, as long as package 'learnrbook' (>= 1.0.4) is already installed.

```
packages_at_cran <-
  setdiff(learnrbook::pkgs_ch_ggplot_extra, learnrbook::pkgs_at_github)
install.packages(packages_at_cran)
for (p in learnrbook::pkgs_at_github) {
  devtools::install_github(p)
}
```

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(learnrbook)
library(tibble)
library(magrittr)
library(ggplot2)
library(ggpmisc)
library(ggtext)
library(ragg)
library(showtext)
library(tikzDevice)
```

We set a font of larger size than the default

```
theme_set(theme_grey(14))
```

10.3 System fonts

Access to system fonts is not native to the R language, and requires the use of extension packages. Recent versions of R support UTF encoded characters, making it possible to use of non-Latin and large alphabets. See section 7.4.7 for the use of text natively in 'ggplot2' using character strings and *plotmath* expressions.

P

System fonts are those fonts available through the operating system. These fonts are installed so that, at least in principle, they are accessible to any program run locally. There is usually a basic set of fonts distributed with the operating system but additional ones, both free and commercial, can be installed. Installers of software like word processors or graphic editors frequently install additional system fonts. The installed system fonts will also depend on the localization, i.e., on the language in use by the operating system. Operating systems support browsing, installation and uninstallation of system fonts while additional free (e.g., NexusFont, FontBase) and commercial (e.g., FontBase Awesome) programs for these tasks also exist.

In this section we give examples of the use of non-Latin characters from default

fonts and of package 'showtext' to make available locally installed fonts known to the operating system and of Google fonts available through the internet. Enabling such fonts not only makes available glyphs used in other than the default system language but also to collections of icons such as those used in this book.

A font with Chinese characters is included in package 'showtext'. This example is borrowed from the package vignette, but modified to use default fonts, of which `"wqy-microhei"` is a Chinese font.

 When using 'knitr' as for this book, rendering using system fonts made available with 'showtext', the chunk option `fig.showtext=TRUE` needs to be set. Otherwise, instead of the expected characters a faint square will be used as place holder for characters in the output.

```
ggplot(NULL, aes(x = 1, y = 1)) + ylim(0.8, 1.2) +
    theme(axis.title = element_blank(), axis.ticks = element_blank(),
          axis.text = element_blank()) +
    annotate("text", 1, 1.1, family = "wqy-microhei", size = 8,
             label = "\u4F60\u597D\uFF0C\u4E16\u754C") +
    annotate("text", 1, 0.9, label = 'Chinese for "Hello, world!"',
             family = "sans", fontface = "italic", size = 6)
```

Next we load some system fonts, from the same family used for the text of this book. Within code chunks when using 'knitr' we can enable `showtext` with chunk option `fig.showtext = TRUE` as done here (but not visible). In a script or at the console we can use `showtext.auto()`, or `showtext.begin()` and `showtext.end()`. As explained in the package vignette, using `showtext` can increase the size of the PDF files created, but on the other hand, it ensures by embedding the fonts that the PDF is self-contained and portable.

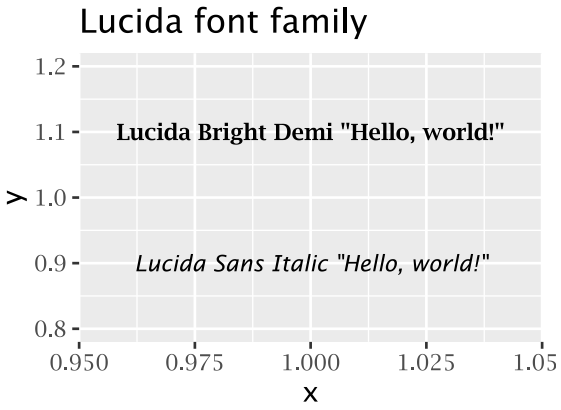
Function `font.families()` lists the fonts known to R, and function `font.add()` can be used to make *system fonts* visible to R. We set `family`, and indicate the font names for each `face`.


The fonts shown in this example are likely to be unavailable in your own computer, so you may need to replace the font names by those of available fonts. Each combination of `family` and `fontface` corresponds to a font file (in this case Open Type fonts, hence the `.otf` ending of the files names).

```
font_families()
## [1] "sans"         "serif"        "mono"         "wqy-microhei"

font_add(family = "Lucida.Sans",
         regular = "LucidaSansOT.otf",
         italic = "LucidaSansOT-Italic.otf",
         bold = "LucidaSansOT-Demi.otf",
         bolditalic = "LucidaSansOT-DemiItalic.otf")

font_add(family = "Lucida.Bright",
         regular = "LucidaBrightOT.otf",
         italic = "LucidaBrightOT-Italic.otf",
         bold = "LucidaBrightOT-Demi.otf",
         bolditalic = "LucidaBrightOT-DemiItalic.otf")

font_families()
## [1] "sans"          "serif"         "mono"          "wqy-microhei"
## [5] "Lucida.Sans"   "Lucida.Bright"
```

We can then select these fonts similarly as we would for R's `"serif"`, `"sans"`, `"mono"` and `"symbol"`. Please, see section 7.10 on page 275 for examples of how to set the `family` for individual elements of the plot. We use the same `family` names as used above to add the fonts. We show the effect passing family and fontface constants to `geom_text()` and by overriding the default for the theme.

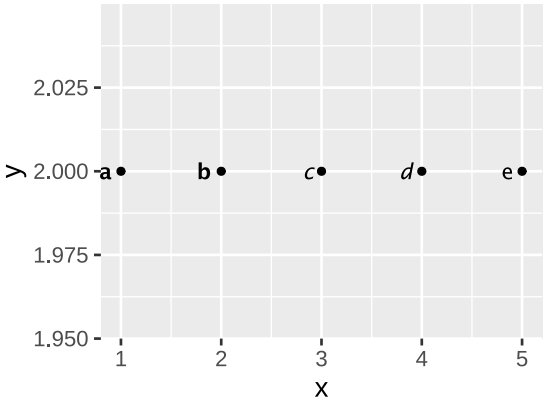
```
ggplot(NULL, aes(x = 1, y = 1)) +
    ylim(0.8, 1.2) +
    annotate("text", 1, 1.1, label = 'Lucida Bright Demi "Hello, world!"',
             family = "Lucida.Bright", fontface = "bold", size = 4) +
    annotate("text", 1, 0.9, label = 'Lucida Sans Italic "Hello, world!"',
             family = "Lucida.Sans", fontface = "italic", size = 4) +
    labs(title = "Lucida font family") +
    theme(text = element_text(family = "Lucida.Bright"),
          title = element_text(family = "Lucida.Sans"))
```



Be aware that in *geometries* the equivalent of `face` in theme's `element_text()` is called `fontface`, while the character string values they accept are the same.

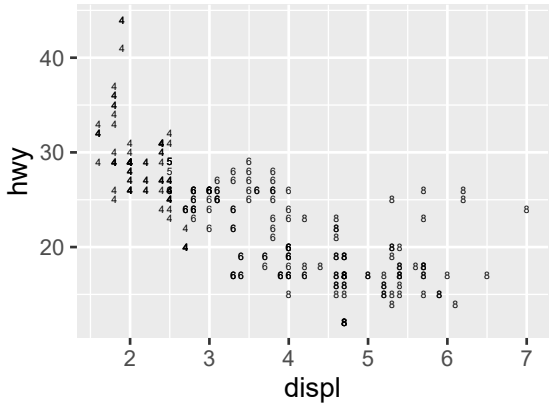
In geometries `family` and `fontface` can be passed constants as arguments or variables mapped to the `family` and `fontface` aesthetics.

```
my.data <-
  data.frame(x = 1:5, y = rep(2, 5),
             label = c("a", "b", "c", "d", "e"),
             face = c("bold", "bold", "italic", "italic", "plain"))

ggplot(my.data, aes(x, y, label = label, fontface = face)) +
  geom_text(hjust = 1.7, family = "Lucida.Sans") +
  geom_point()
```



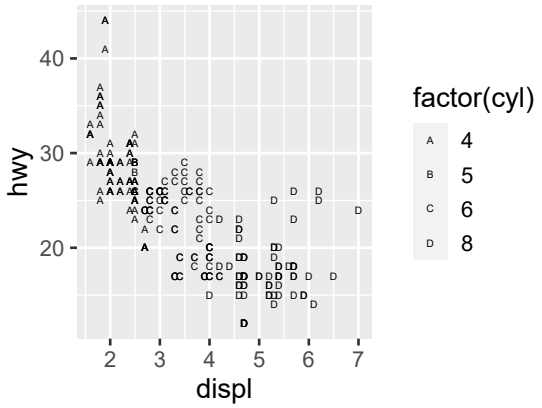
R uses by default special characters from the Symbol font as *points* in plots. However, any character such as letters or numbers are also accepted. A vector containing character strings, each a single character in length, can be mapped to the 'shape' aesthetic can be mapped. These characters do need to be alphanumeric.

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



The intuitive approach of using `scale_shape_manual()` does not allow shapes from system fonts, as 'Grid' always uses the Symbol font for point shapes. Code as shown here let us map characters, but not to switch fonts or access all characters in this font.

```
ggplot(mpg, aes(displ, hwy, shape = factor(cyl))) +
  geom_point() +
  scale_shape_manual(values = LETTERS[1:length(unique(mpg$cyl))])
```

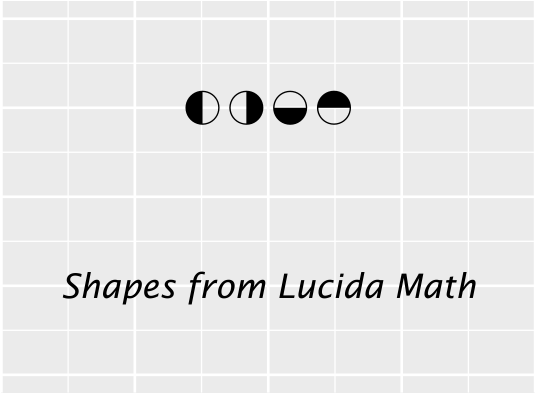


Only the code in one the last two chunks above would have worked if the data set had included cars with 12 cylinders engines. Which one and why?

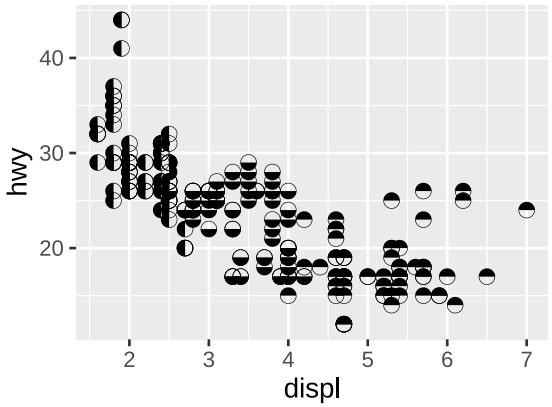
To use shapes from other fonts we need to use single-character text labels. We make available the Math font corresponding to Lucida Bright. This font is very comprehensive and uses a mapping compatible with LaTeX.

```
font_add(family = "Lucida.Bright.Math",
         regular = "LucidaBrightMathOT.otf",
         italic = "LucidaBrightMathOT.otf",
         bold = "LucidaBrightMathOT-Demi.otf",
         bolditalic = "LucidaBrightMathOT-Demi.otf")

font_families()
## [1] "sans"                "serif"               "mono"
## [4] "wqy-microhei"        "Lucida.Sans"         "Lucida.Bright"
## [7] "Lucida.Bright.Math"
```

```
ggplot(NULL, aes(x = 1, y = 1)) +
    ylim(0.8, 1.2) +
    theme(axis.title = element_blank(), axis.ticks = element_blank(),
          axis.text = element_blank()) +
    annotate("text", 1, 1.1, family = "Lucida.Bright.Math", size = 8,
             label = "\u25d0\u25d1\u25d2\u25d3") +
    annotate("text", 1, 0.9, label = 'Shapes from Lucida Math',
             family = "Lucida.Sans", fontface = "italic", size = 6)
```




```
my.mpg <- mpg
# we manually map factor lebels to labels stored in variable shapes
my.mpg$cyl.f <- factor(mpg$cyl)
shapes.map <- c("\u25d0", "\u25d1", "\u25d2", "\u25d3")
names(shapes.map) <- levels(my.mpg$cyl.f)
my.mpg$shapes <- unname(shapes.map[my.mpg$cyl.f])
head(my.mpg$shapes)
## [1] "<U+25D0>" "<U+25D0>" "<U+25D0>" "<U+25D0>" "<U+25D2>" "<U+25D2>"

ggplot(my.mpg, aes(displ, hwy, label = shapes)) +
    geom_text(family = "Lucida.Bright.Math")
```



The examples above will work only if the fonts are locally available in the computer. This compromises portability but such flexibility is important from a graphics design perspective.

New R (virtual/software) graphic devices based on the AGG library support system fonts natively. These graphic devices are implemented in R package 'ragg' and support rendering to different *bitmap* formats: PNG, JPEG, TIFF and PPM. We can use device `agg_png()` in place of R's own `png()` device, or when using 'knitr' add the chunk option `dev = "ragg_png"` as we do here. AGG in addition to supporting system fonts provides better output faster. When using system fonts, one needs to implement a fall back mechanism as there is no guarantee of availability of the requested fonts. This is in contrast to the use of R's generic names like `"sans"` and `"serif"` which are always available.

10.4 Google fonts

Next we use function `font.add.google()`, also from package 'showtext', to make Google fonts available. As long as internet access is available, Google fonts can be downloaded if not available locally. You can browse the available fonts at `https://fonts.google.com/` and access them through the names listed in this page.

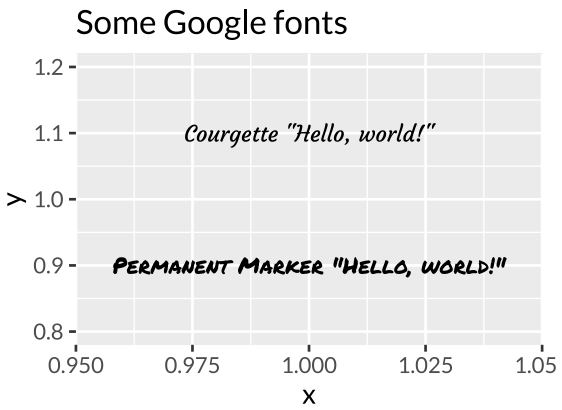
The concept of a font `family` in R is not exactly consistent with the graphic designers' use of the concept of a *font family* as being a set of fonts designed to be used together. In R, with 'showtext' we can create a `family` with an arbitrary choice of fonts for the different `faces`, however, it is in general best to use fonts designed to work well together as we did above when using Lucida fonts. When visiting `https://fonts.google.com/` once you select a font, you can press the rightmost (at this time) icon to display the whole family of related fonts. You will notice that some modern font families are available at many different weights between the lightest (*hairline*) and the boldest (*black*) as well as in multiple character widths between *condensed* and *wide*. From a design

perspective, the recomendation is not to mix many weights in the same typeset text, but rather choose a suitable pair of weights for `"regular"` and `"bold"` font faces.

Here we use two decorative fonts, `"Permanent Marker"` and `"Courgette"`, and `"Lato"`, a more formal font very popular for web pages. All fonts in the Lato family support for more than 100 Latin-based languages, more than 50 Cyrillic-based languages as well as Greek and IPA phonetics.

```
## Loading Google fonts (http://www.google.com/fonts)
font_add_google(name = "Permanent Marker", family = "Marker")
font_add_google(name = "Courgette")
font_add_google(name = "Lato")
```

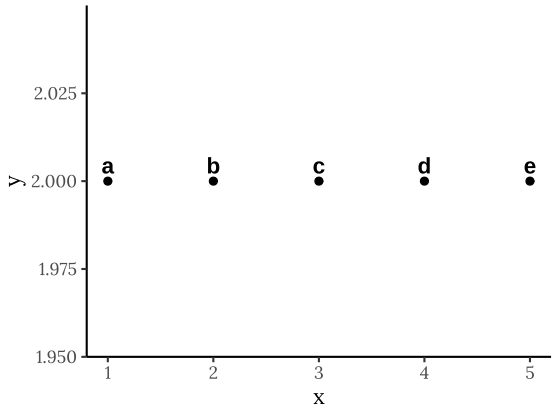
```
ggplot(NULL, aes(x = 1, y = 1)) +
ylim(0.8, 1.2) +
    annotate("text", 1, 1.1, label = 'Courgette "Hello, world!"',
             family = "Courgette", size = 4) +
    annotate("text", 1, 0.9, label = 'Permanent Marker "Hello, world!"',
             family = "Marker", size = 4) +
    labs(title = "Some Google fonts") +
    theme(text = element_text(family = "Lato"))
```



Some modern fonts are parameterized so that the weight can be set numerically. Here we show for font Lora how weights can be assigned to font faces, in this case resulting in a large (actually excessive) difference in weight between `"regular"` and `"bold"` font faces in the plot.

```
font_add_google(name = "Lora", regular.wt = 400, bold.wt = 700)
font_families()
##  [1] "sans"               "serif"              "mono"
##  [4] "wqy-microhei"       "Lucida.Sans"        "Lucida.Bright"
##  [7] "Lucida.Bright.Math" "Marker"             "Courgette"
## [10] "Lato"               "Lora"

ggplot(my.data, aes(x, y, label = label)) +
  geom_text(vjust = -0.5,
            fontface = "bold",
```

```
          size = 4) +
geom_point() +
theme_classic(base_size = 11, base_family = "Lora")
```



In all the examples above we used `geom_text()`. However, `geom_label()` can be used instead, as well as geometries from packages like 'ggrepel' and 'ggpmic' that render text using the same approach as 'ggplot2'.

Practice using system and Google fonts, adding new families and their corresponding font faces, and using them in plots created with 'ggplot2'. Consider both the R coding and graphic design related aspects. Be aware that depending on the font or the operating system in use, the non-ASCII characters available can vary.

10.5 Markdown and HTML

Package 'ggtext' implements support for Markdown and a small subset of HTML. It provides new geometries as well as an element that can replace `element_text()` in theme definitions. This means that geometries from 'ggplot2' and other extension packages will remain with their functionality unchanged, although still usable. At this time only some statistics from package 'ggpmisc' can on request generate Markdown-encoded character strings for use as text labels with the new geometries. Using Markdown-encoded titles and axis labels is controlled the theme settings and will not interfere with any geometry or statistic.

Package 'ggtext' is new and changes can be expected as well as support for additional HTML features to be added. Package 'ggtext' provides an alternative to the use of *plotmath* expressions, described in section 7.12).

Markdown was designed as a very simple language to encode formatting

of text to be displayed in web pages. It is normally translated into HTML before deployment and it supports embedding of HTML markup. Embedded HTML is passed through unchanged during translation into HTML. Markdown has become popular as it is both terser and simpler than HTML markup.

Package 'ggtext' implements only a subset of the HTML by means of package 'gridtext' resulting in a text encoding suitable for 'grid', one of R's graphics systems.

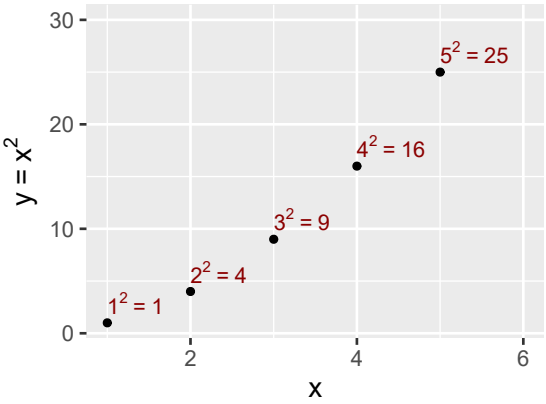
By using 'ggtext' we gain the possibility of using a simpler encoding of text formatting with which R users may be already familiar. We also gain some features that are missing from *plotmath* like highlighting with colour individual words in a title or label, wrapping of long text lines, and the possibility of embedding bitmaps and graphic elements in place of text. On the other hand the ability to nicely format complex mathematical equations available with *plotmath* and LaTeX is missing. So, LaTeX remains the most powerful approach, supporting best and flexible formatting of both text and maths, but requires the use of a specific graphic driver (see section 10.6).

We create some data. To generate character strings to use as text labels we use `sprintf()` taking advantage that it supports vectors as arguments. We define `x` and `y` in the user environment (before calling `data.frame()`) so that `x` and `y` are available when for calling `sprintf()`.

```
x <- 1:5
y <- x^2
my.labels.data <-
  data.frame(x,
             y,
             mkdwn.labs = sprintf("%.0f<sup>2</sup> = %.0f", x, y),
             plotmath.labs = sprintf("%.0f^{2}~`=`~%.0f", x, y),
             latex.labs = sprintf("$%.0f^{2} = %.0f$", x, y))
```

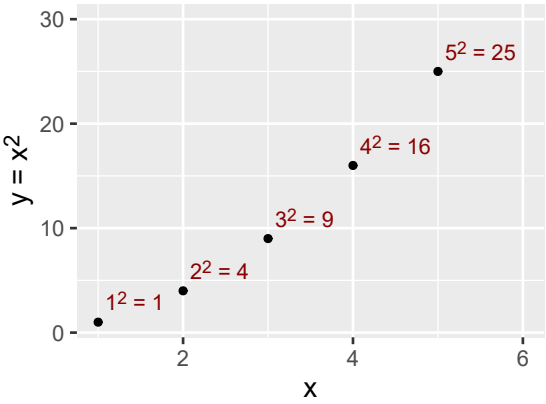
A plot using *plotmath* syntax as described in section 7.12 on page 282.

```
ggplot(my.labels.data,
       aes(x, y, label = plotmath.labs)) +
  geom_text(parse = TRUE, vjust = -0.4, hjust = 0, colour = "darkred") +
  geom_point() +
  expand_limits(x = 6, y = 30) +
  labs(y = expression(y~`=`~x^{2}))
```



The same plot using Markdown syntax. There are some differences: while rendering of R expressions drops trailing zeros, rendering of Markdown does not. In addition, `geom_richtext()` is similar in to `geom_label` in having a box, which we need to make invisible by passing `NA` as arguments. To use Markdown in the axis title we need to override the default in of the theme definition in use.

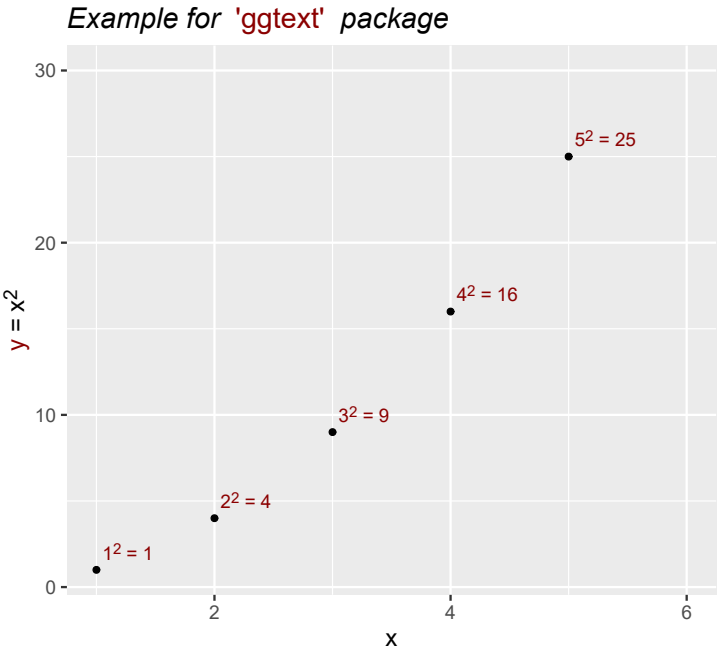
```
ggplot(my.labels.data,
       aes(x, y, label = mkdwn.labs)) +
  geom_richtext(vjust = 0, hjust = 0, colour = "darkred",
                fill = NA, label.size = NA) +
  geom_point() +
  expand_limits(x = 6, y = 30) +
  labs(y = "y = x<sup>2</sup>") +
  theme(axis.title.y = element_markdown())
```



The next example shows some new features introduced by 'ggtext'.

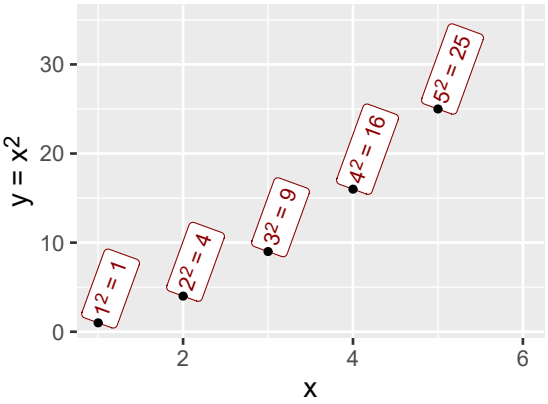
```
ggplot(my.labels.data,
       aes(x, y, label = mkdwn.labs)) +
  geom_richtext(vjust = 0, hjust = 0, colour = "#8B0000",
                fill = NA, label.size = NA) +
  geom_point() +
  expand_limits(x = 6, y = 30) +
  labs(y = "<span style = 'color:#8B0000;'>y</span> = x<sup>2</sup>",
       title = "_Example for_ <span style = 'color:#8B0000;'>'ggtext'</span> _package_",
```

```
         caption = "This is a _caption_. It is encoded using **Markdown** syn-
tax so that we can add some <span style = 'color:#8B0000;'>red colour</span> to it.") +
  theme(axis.title.y = element_markdown(),
        plot.title = element_markdown(),
        plot.caption =
          element_textbox_simple(
            padding = margin(5.5, 5.5, 5.5, 5.5),
            margin = margin(0, 0, 5.5, 0)))
```



While `geom_label()` does not yet support rotation, `geom_richtext()` does.

```
ggplot(my.labels.data,
       aes(x, y, label = mkdwn.labs)) +
  geom_richtext(vjust = 0.5, hjust = 0, colour = "darkred",
                angle = 70) +
  geom_point() +
  expand_limits(x = 6, y = 35) +
  labs(y = "y = x<sup>2</sup>") +
  theme(axis.title.y = element_markdown())
```

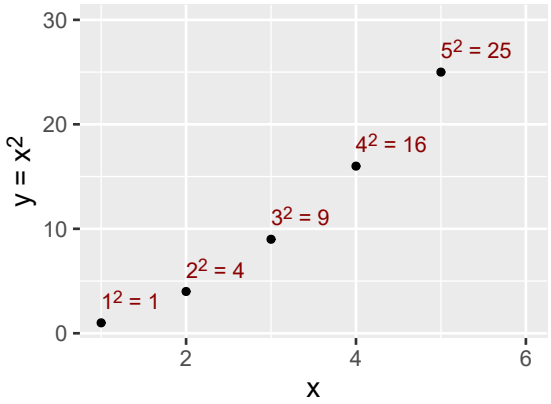


To avoid having to override the defaults repeatedly, we can define a wrapper on `geom_richtext()` with new defaults. In this example we try to match `geom_text()`, so in addition to setting `fill = NA` and `label.size = NA` as above, we set the padding to zero so that justification is based on the boundary of the text like in `geom_text()`. We name the new geometry as `geom_mkdwntext()`.

```
geom_mkdwntext <- function(fill = NA,
                           label.size = NA,
                           label.padding = grid::unit(rep(0, 4), "lines"),
                           ...) {
  geom_richtext(fill = fill,
                label.size = label.size,
                label.padding = label.padding,
                ...)
}
```

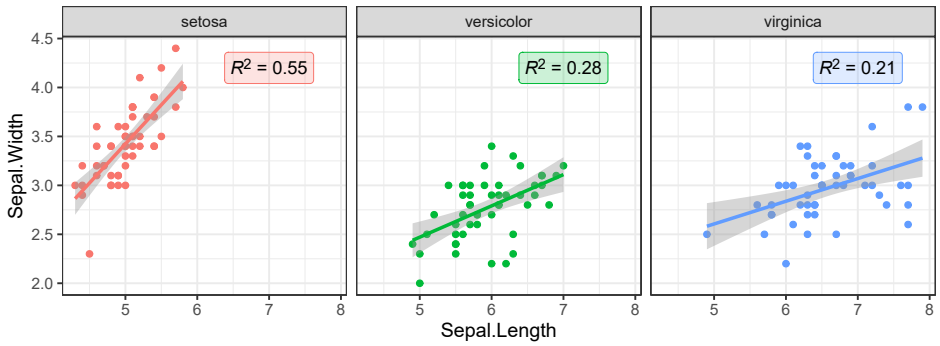
We can now rewrite the first example of the use of Markdown from page 376 as follows. We can use the same justification as with `geom_text()` because we have removed the padding.

```
ggplot(my.labels.data,
       aes(x, y, label = mkdwn.labs)) +
  geom_mkdwntext(vjust = -0.4, hjust = 0, colour = "darkred") +
  geom_point() +
  expand_limits(x = 6, y = 30) +
  labs(y = "y = x<sup>2</sup>") +
  theme(axis.title.y = element_markdown())
```



Package 'ggpmisc' as (described in section 7.5.3, page 244) makes it possible to automate model annotations. We here show an alternative version of an example from the vignette of package 'ggtext' avoiding the need to separately fit the models by using `geom_poly_eq()` from 'ggpmisc'. The geometries `geom_poly_eq()` and `geom_quant_eq()` can automatically generate text labels with model equations and other estimates encoded as *plotmath*, Markdown, LaTeX or plain text. Here we use Markdown and `geom_richtext()`.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width, colour = Species)) +
    geom_point() +
    stat_poly_line() +
    stat_poly_eq(aes(fill = after_scale(alpha(colour, .2))),
                 label.x = 7, label.y = 4.2, text.colour = "black",
                 geom = "richtext") +
    facet_wrap(~Species) +
    theme_bw(12) +
    theme(legend.position = "none")
```



We have only provided some examples of the use of Markdown in ggplots. Current versions of Markdown and HTML markup are described at `https://daringfireball.net/projects/markdown/syntax` and `https://html.spec.whatwg.org/multipage/`, however none of them are fully supported by 'ggtext'. See `https://wilkelab.org/ggtext/` for up to date documentation and further examples.

10.6 LaTeX

LaTeX is the most powerful and stable typesetting engine currently available. I used it to typeset the body text of *Learn R: As a Language* but not for plot labels. 'Tikz' is a LaTeX extension package implementing a language for graphical output. A graphic driver is available for R that renders plots into TikZ commands instead of a bitmap, Postscript or PDF files. The final rendering is done with TeX or one of its derivatives LuaTeX or XeTeX, in most cases after automatically including it into the LaTeX-encoded source file for a manuscript.

The examples in this section use the 'knitr' option `driver="Tikz"` and require R package 'TikzDevice' to be installed. For R users already familiar with LaTeX or needing one of the many alphabets only supported by LaTeX this is the best alternative. Markdown is easy to write but unsuitable for all but the simplest math expressions while R expressions lack the font choices provided by LaTeX.

If a book or article is typeset using LaTeX, this approach not only allows access to fonts suitable for most currently used languages and even some historical ones, but also allows using the use of fonts from the same family in the body of the text and in illustrations enhancing the aesthetics of the typeset document.

```
try(detach(package:tikzDevice))
try(detach(package:ggtext))
try(detach(package:showtext))
try(detach(package:ggpmisc))
try(detach(package:ggpp))
try(detach(package:ggplot2))
try(detach(package:magrittr))
try(detach(package:tibble))
try(detach(package:learnrbook))
```

11

Grammar of Graphics: Color palettes

> Show data variation, not design variation.
>
> Edward Tufte
> *The Visual Display of Quantitative Information*, 1983

11.1 Aims of this chapter

In this chapter I describe some advanced features of package 'ggplot2' and a selection packages that extend it following the same grammar of graphics. Some extensions provide additional *graphical designs*, others *extend the grammar* to additional types of plots and others *wrap frequently used features* into easy to use functions. The update of 'ggplot2' to version 2.0.0 made it straightforward to write these extensions. To keep up-to-date with the release of new extensions I recommend to regularly check the site 'ggplot2 Extensions' (maintained by Daniel Emaasit and colaborators) at `https://exts.ggplot2.tidyverse.org/`.

11.2 Packages used in this chapter

The list of packages used in the current chapter is very long. However, there are only few incompatibilities among them. Even if one is not attempting to use functions from different packages within the same plot, one may want to use them to create different figures within the same document when using the literate approach to programming and scripting (see 3.2.4 on page 90), with incompatibilities requiring additional coding to work around them.

If the packages used in this chapter are not yet installed in your computer, you can install them with the code shown below, as long as package 'learnrbook' (>= 1.0.3) is already installed.

```
packages_at_cran <-
  setdiff(learnrbook::pkgs_ch_ggplot_extra, learnrbook::pkgs_at_github)
```

```
install.packages(packages_at_cran)
for (p in learnrbook::pkgs_at_github) {
  devtools::install_github(p)
}
```

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(learnrbook)
library(tibble)
library(magrittr)
library(lubridate)
library(ggplot2)
library(viridis)
library(pals)
library(ggsci)
```

We set a font of larger size than the default

```
theme_set(theme_grey(14))
```

11.3 Colours and colour palettes

In addition to the scales in 'ggplot2' (see section ?? on page ?? for the basic concepts), there are packages that implement additional colour and fill scales. Package 'viridis' provides an older implementation of the *Viridis* palettes not available in 'ggplot2' 2, with additional flexibility and a different interface through `scale_colour_viridis()` and `scale_fill_viridis()`.

There are two key aspects to choosing a colour palette for a plot: aesthetics and readability. Both aspects concern the design of information graphics. The *Viridis* palettes are special in that they have been designed to be meaningfully readable under various types of colour blindness as well as when printed or displayed as a grey scale. They are also visually uniform, which is crucial for objectivity: no colours overwhelm others becoming accidentally emphasized. Recent versions of 'ggplot2' include an implementation of discrete and continuous scales based on *Viridis* palettes in `scale_colour_viridis_d()`, `scale_colour_viridis_c()`, `scale_fill_viridis_d()`, and `scale_fill_viridis_c()`. Other colour and fill scales in 'ggplot2', frequently used by default, are not as good in these respects as the *Viridis* ones. The higher the number of distinct colours used, the more difficult it becomes to ensure that they can be distinguished.

Packages exist that include colour palettes, e.g., 'ggsci' that reproduces palettes used by some scientific journals, data analysis software and even TV series, 'hrbrthemes' that defines a few colour and fill scales used in the new themes included in it and 'pals'. Modifying and subsetting palettes needs to be done with care, and package 'pals' provides in addition to palette definitions, the tools to assess the performance of palettes.

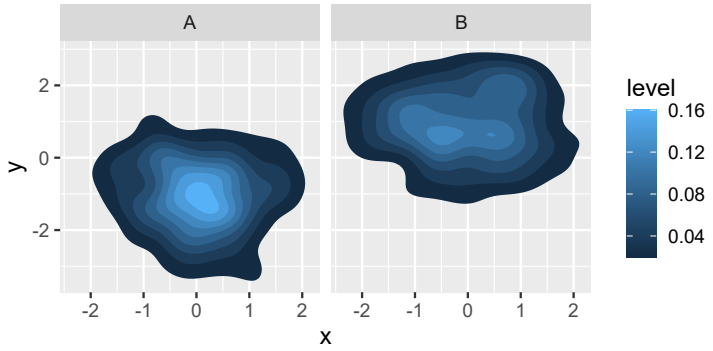
Some of the continuous scales from 'ggplot2' such as `scale_fill_gradient()`

and `scale_fill_gradientn()` allow to easily use of colours chosen by users. We use these scales in the examples below. When a factor is mapped to `color` or `fill` aesthetics, we need to use discrete scales such `scale_color_brewer()` or `scale_fill_brewer()` instead of continuous ones.

```
set.seed(56231)
my.data <-
  tibble(x = rnorm(500),
         y = c(rnorm(250, -1, 1), rnorm(250, 1, 1)),
         group = factor(rep(c("A", "B"), c(250, 250))) )
```

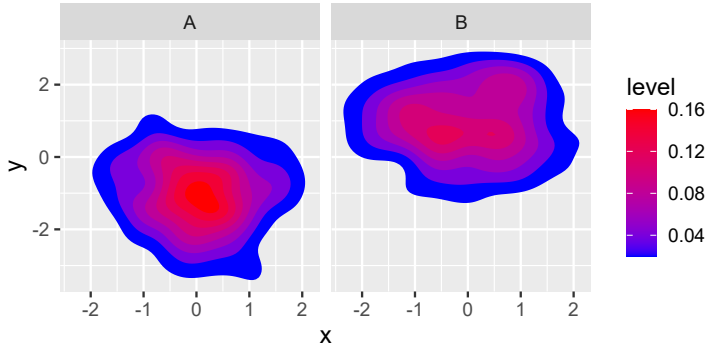
First we use 'ggplot2''s default, based on luminance and saturation of the same color.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradient()
```



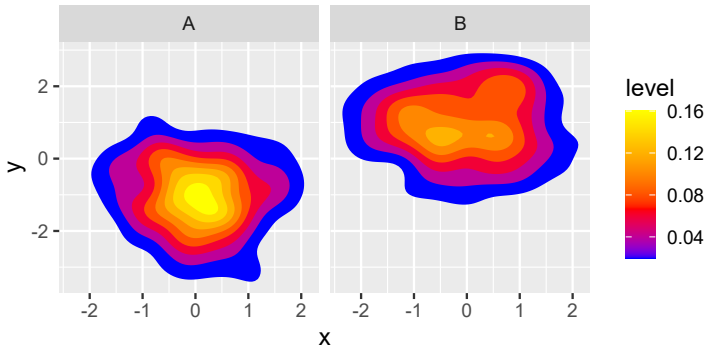
A gradient between two arbitrary colours is not necessarily good, as shown here.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradient(low = "blue", high = "red")
```



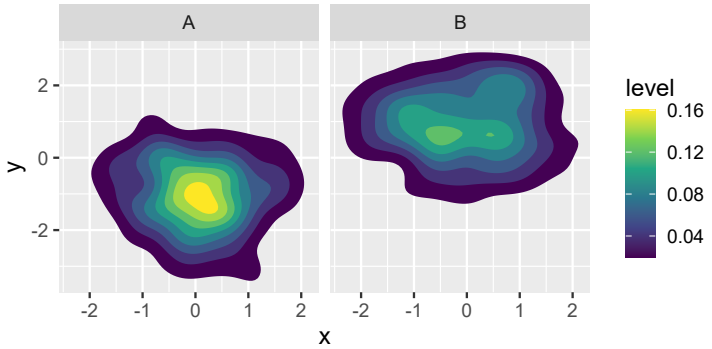
We can also define a gradient based on more than two colour definitions.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradientn(colors = c("blue", "red", "orange", "yellow"))
```



Or use a scale based on an specific palette out of a set, here the default *Viridis* palette.

```
ggplot(my.data, aes(x, y)) +
    stat_density_2d(aes(fill = ..level..), geom = "polygon") +
    facet_wrap(~group) +
    scale_fill_viridis_c()
```



As shown in the next section, instead of passing a manually created vector to `scale_fill_gradientn()`, we can generate it using a palette builder.

Modify the example based on the `iris` data from page 379 playing with discrete color and fill scales. Add `scale_fill_discrete()` and `scale_color_discrete()` and adjust the colours and fill in a way that enhances the appearance and/or the readability of the plot.

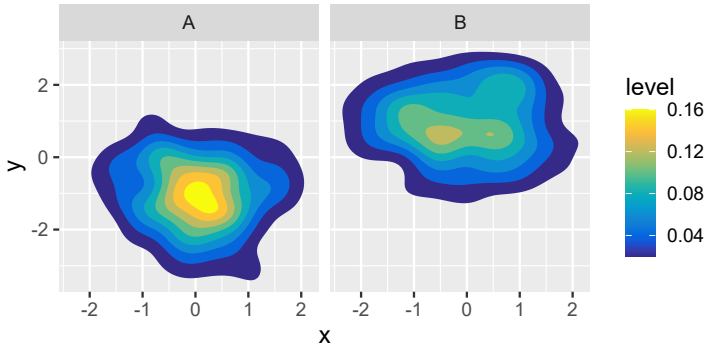
11.3.1 'pals'

```
citation(package = "pals")
```

Package 'pals' provides definitions for palettes and color maps, and also palette evaluation tools. Being a specialized package, we describe it briefly and recommend readers to read the vignette and other documentation included with the package.

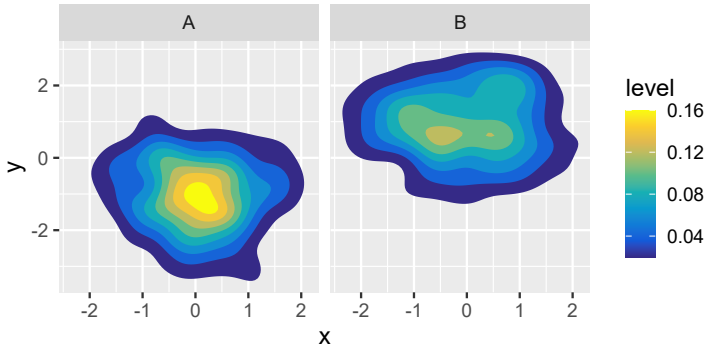
First we reproduce the last example above using palette `parula()`.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradientn(colors = parula(100), guide = "colourbar")
```



The biggest advantage is that we can in the same way use any color map and palette and, in addition, choose how smooth a color map we use. We here use a much shorter vector as argument passed to `colors`. The plot remains almost identical because interpolation is applied.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradientn(colours = parula(10), guide = "colourbar")
```

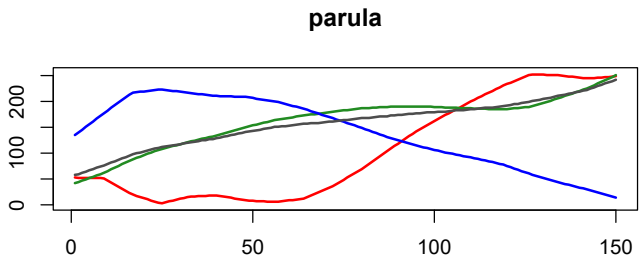


We can compare different color maps with `pal.bands()`, here those used in the last two examples above.

```
pal.bands(parula(100), parula(10))
```

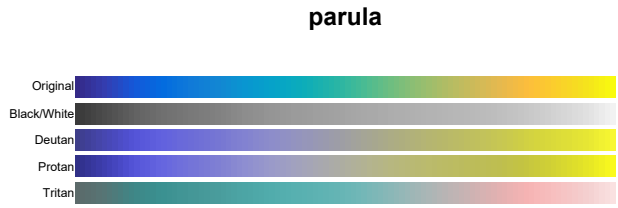
How does the luminance of the red, green and blue colour channels vary along the palette or color map gradient? We can see this with `pal.channels()`.

```
pal.channels(parula, main = "parula")
```



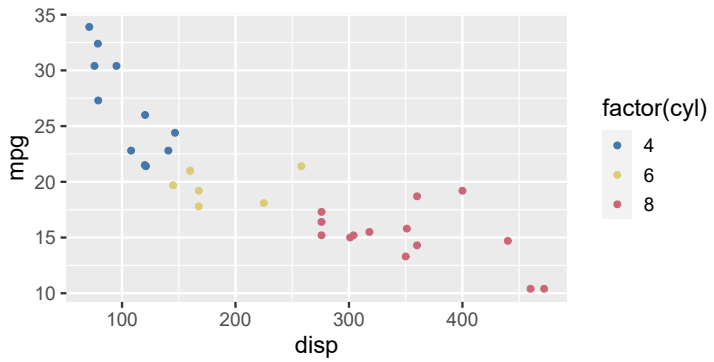
How would `viridis` look in monochrome, and to persons with different kinds of color blindness? We can see this with `pal.safe()`.

```
pal.safe(parula, main = "parula")
```



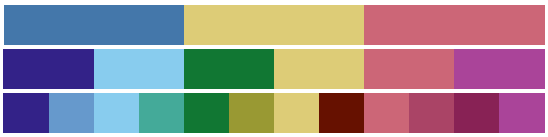
A brief example with using the discrete palette `tol()` from package 'pals'.

```
ggplot(data = mtcars,
    aes(x = disp, y = mpg, color = factor(cyl))) +
    geom_point() +
    scale_color_manual(values = tol(n = 3))
```



Parameter `n` gives the number of discrete values in the palette. Discrete palettes have a maximum value for `n`, in the case of tol, 12 discrete steps.

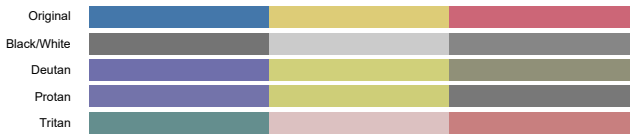
```
pal.bands(tol(n = 3), tol(n = 6), tol())
```



Play with the argument passed to `n` to test what happens when the number of values in the scale is smaller or larger than the number of levels of the factor mapped to the color *aesthetic*.

Is this palette safe?

```
pal.safe(tol(n = 3))
```



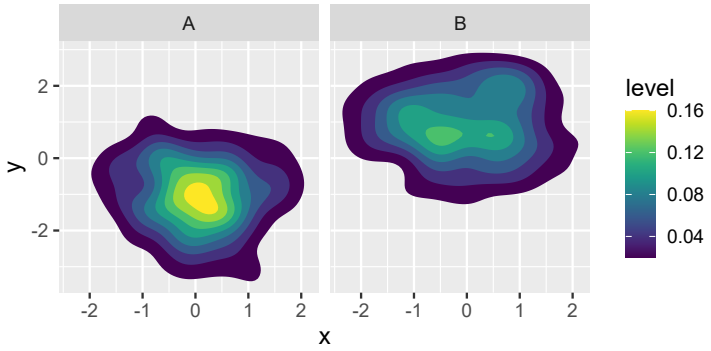
Explore the available palettes until you find a nice one that is also safe with three steps. Be aware that color maps, like `viridis()` can be used to define a discrete color scale using `scale_color_manual()` in exactly the same way as palettes like `tol()`. Colormaps, however, may be perceived as gradients, rather than un-ordered discrete categories, so care is needed.

11.3.2 'viridis'

```
citation(package = "viridis")
```

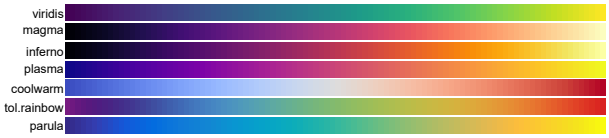
First we simply reproduce the example from page **??**, obtaining the same plot as with `scale_fill_viridis_c()` but using a palette builder function to create a vector of color definitions of length 100.

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_gradientn(colors = viridis(100), guide = "colourbar")
```



Next we compare palettes defined in package 'viridis' with some of the palettes defined in package 'pals'.

```
pal.bands(viridis, magma, inferno, plasma, coolwarm, tol.rainbow, parula)
```



11.3.3 'ggsci'

```
citation(package = "ggsci")
##
## To cite package 'ggsci' in publications use:
##
##   Nan Xiao (2018). ggsci: Scientific Journal and Sci-Fi Themed Color
##   Palettes for 'ggplot2'. R package version 2.9.
##   https://CRAN.R-project.org/package=ggsci
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {ggsci: Scientific Journal and Sci-Fi Themed Color Palettes for
## 'ggplot2'},
##     author = {Nan Xiao},
##     year = {2018},
##     note = {R package version 2.9},
##     url = {https://CRAN.R-project.org/package=ggsci},
##   }
```

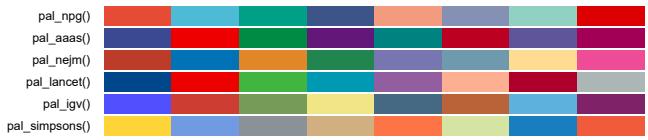
I list here package 'ggsci' as it provides several *color palettes* (and color maps) that some users may like or find useful. They attempt to reproduce the those used by several publications, films, etc. Although visually attractive, several of them are not safe, in the sense discussed in section 11.3.1 on page 386. For each palette, the package exports a corresponding *scale* for use with package 'ggplot2'.

Here is one example, using package 'pals', to test if it is "safe".

```
pal.safe(pal_uchicago(), n = 9)
```

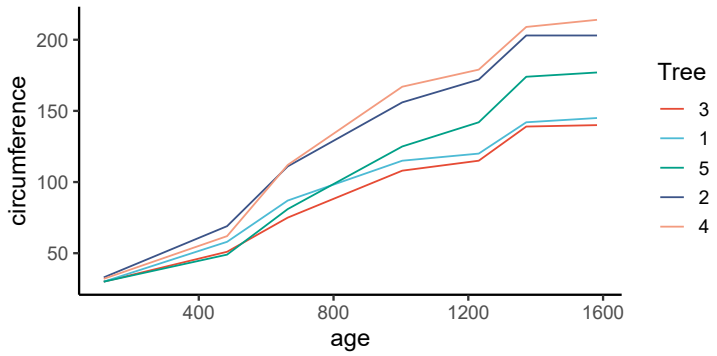
A few of the discrete palettes as bands, setting `n` to 8, which is the largest value supported by the smallest of these palettes.

```
pal.bands(pal_npg(),
          pal_aaas(),
          pal_nejm(),
          pal_lancet(),
          pal_igv(),
          pal_simpsons(),
          n = 8)
```



And a plot using a palette mimicking the one used by Nature Publishing Group (NPG).

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  scale_color_npg() +
  theme_classic(14)
```



11.4 Plotting color patches

For choosing colours when designing plots, or scales used in them, an indexed colour patch plot is usually very convenient (see section 7.7.6.1 on page 267. We can produce such a chart of colors with subsets of colors, or colours re-ordered compared to their position in the value returned by `colors()`. As the present chapter is on package 'ggplot2' we use this package in this example. As this charts are likely to be needed frequently, I define here a function `ggcolorchart()`.

```
ggcolorchart <- function(colors,
                         ncol = NULL,
                         use.names = NULL,
                         text.size = 2) {
    # needed if the argument passed is subset with [ ]!
    force(colors)
```

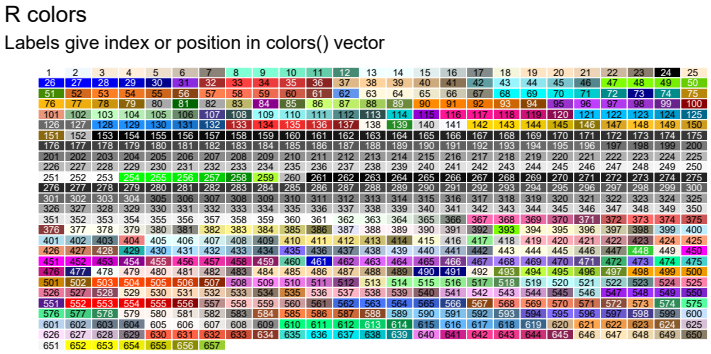
```
    len.colors <- length(colors)
    # by default we attempt to use
    if (is.null(ncol)) {
      ncol <- max(trunc(sqrt(len.colors)), 1L)
    }
    # default for when to use color names
    if (is.null(use.names)) {
      use.names <- ncol < 8
    }
    # number of rows needed to fit all colors
    nrow <- len.colors %/% ncol
    if (len.colors %% ncol != 0) {
      nrow <- nrow + 1
    }
    # we extend the vector with NAs to match number of tiles
    if (len.colors < ncol*nrow) {
      colors[(len.colors + 1):(ncol*nrow)] <- NA
    }
    # we build a data frame
    colors.df <-
      data.frame(color = colors,
                 text.color =
                   ifelse(sapply(colors,
                                 function(x){mean(col2rgb(x))}) > 110,
                                 "black", "white"),
                 x = rep(1:ncol, nrow),
                 y = rep(nrow:1, rep(ncol, nrow)),
                 idx = ifelse(is.na(colors),
                              "",
                              format(1:(ncol * nrow), trim = TRUE)))
  # we build the plot
  p <- ggplot(colors.df, aes(x, y, fill = color))
  if (use.names) {
    p <- p + aes(label = ifelse(is.na(colors), "", colors))
  } else {
    p <- p + aes(label = format(idx, width = 3))
  }
  p <- p +
     geom_tile(color = "white") +
     scale_fill_identity() +
     geom_text(size = text.size, aes(color = text.color)) +
     scale_color_identity()
  p + theme_void()
}
```

After reading the use examples below, review the definition of the function, section by section, trying to understand what is the function of each section of the code. You can add print statements at different steps to look at the intermediate data values. Once you think you have grasped the purpose of a given statement, you can modify it in some way that modifies the output. For example, changing the defaults, for the shape of the tiles, e.g. so that the number of columns is about 1/3 of the number of rows. Although you may never need exactly this function, studying its code will teach you some *idioms* used by R programers. This function, in contrast to some other R code examples for

plotting color tiles, does not contain any loop. It returns a `ggplot` object, which be added to and/or modified.

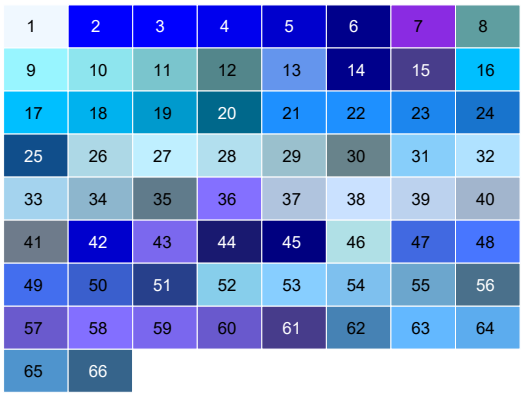
We first the predefined colors available in R.

```
ggcolorchart(colors()) +
  ggtitle("R colors",
          subtitle = "Labels give index or position in colors() vector")
```



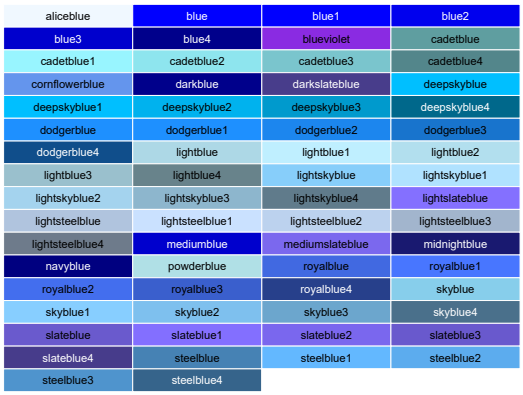
We subset those containing “blue” in the name, using the default number of columns.

```
ggcolorchart(grep("blue", colors(), value = TRUE), text.size = 3)
```



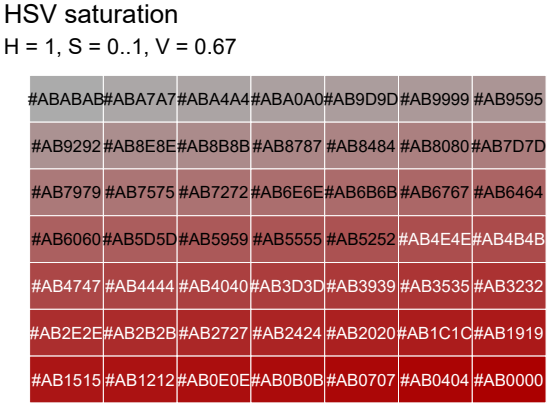
We reduce the number of columns and obtain rectangular tiles. The default for `use.names` depends on the number of tile columns, triggering automatically the change in labels.

```
ggcolorchart(grep("blue", colors(), value = TRUE), ncol = 4)
```

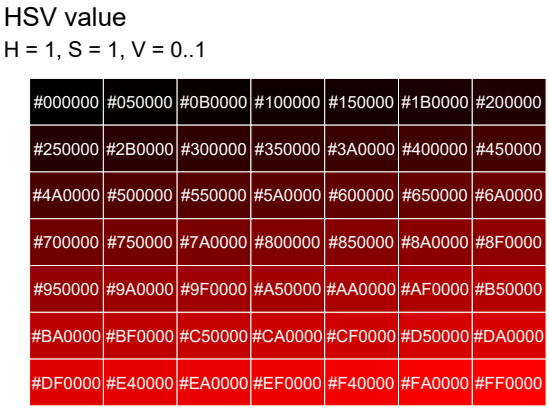


We demonstrate how perceived colors are affected by the hue, saturation and value in the HSV colour model.

```
ggcolorchart(hsv(1, (0:48)/48, 0.67), text.size = 3) +
  ggtitle("HSV saturation", "H = 1, S = 0..1, V = 0.67")
```



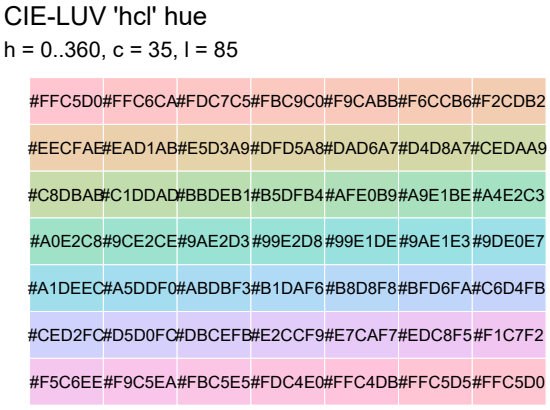

```
ggcolorchart(hsv(1, 1, (0:48)/48), text.size = 3) +
  ggtitle("HSV value", "H = 1, S = 1, V = 0..1")
```



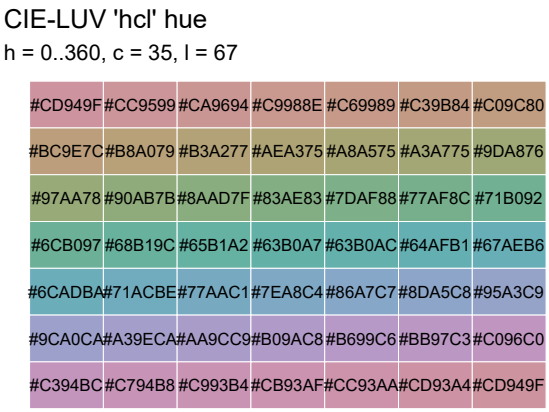
```
ggcolorchart(hsv((0:48)/48, 1, 1), text.size = 3) +
  ggtitle("HSV hue", "H = 0..1, S = 1, V = 1")
```

We demonstrate how perceived colors are affected by the hue, chroma and luminance in the HCL colour model.

```
ggcolorchart(hcl((0:48)/48 * 360), text.size = 3) +
  ggtitle("CIE-LUV 'hcl' hue", "h = 0..360, c = 35, l = 85")
```




```
ggcolorchart(hcl((0:48)/48 * 360, l = 67), text.size = 3) +
  ggtitle("CIE-LUV 'hcl' hue", "h = 0..360, c = 35, l = 67")
```




```
ggcolorchart(hcl((0:48)/48 * 360, c = 67), text.size = 3) +
    ggtitle("CIE-LUV 'hcl' hue", "h = 0..360, c = 67, l = 85")
```

The default order of the different colors in the vector returned by `colors()` results in a rather unappealing color tile plot (see page 394). Use functions `col2rgb()`, `rgb2hsv()` and `sort()` or `order()` to rearrange the tiles into a more pleasant arrangement, but still using for the labels the indexes to the positions of the colors in the original unsorted vector.

```
try(detach(package:ggsci))
try(detach(package:pals))
try(detach(package:viridis))
try(detach(package:ggplot2))
try(detach(package:magrittr))
try(detach(package:tibble))
try(detach(package:learnrbook))
```

Part II

Connecting R to other languages

12

If and when R needs help

> Improving the efficiency of your S [or R] functions can be well worth some effort. ...But remember that large efficiency gains can be made by using a better algorithm, not just by coding the same algorithm better.
>
> Patrick J. Burns
> *S Poetry*, 1998

12.1 Aims of this chapter

In this chapter I discuss briefly a more advanced subject: execution speed or performance of R code and how to enhance it. A few books cover this subject from different perspectives, including *R Packages* (Wickham 2015), *Advanced R* (Wickham 2019), *The Art of R Programming: A Tour of Statistical Software Design* (Matloff 2011), *Extending R* (Chambers 2016) from a practical programming perspective, and, *S Poetry* (Burns 1998) and *The R Inferno* (Burns 2011) from a more conceptual and language design oriented perspective.

12.2 Packages used in this chapter

```
install.packages(learnrbook::pkgs_ch_performance)
```

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(Rcpp)
library(inline)
# library(rPython)
library(rJava)
library(microbenchmark)
```

```
library(profr)
library(tibble)
library(ggplot2)
```

12.3 R's limitations and strengths

12.3.1 Introduction to R code optimization

Some constructs like `for` and `while` loops execute slowly in R, as they are interpreted. Byte compiling and Just-In-Time (JIT) compiling of loops (enabled by default in R >= 3.4.0) should decreases this burden. However, base R as well some packages define several *apply* functions. Being compiled functions, written in C or C++, using apply functions instead of explicit loops *can* provide an improvement in performance while keeping user's code fully written in R. Even more effective is using vectors for indexing in operations so as to eliminate iteration loops. Preallocating memory, rather than growing a vector or array at each iteration can help. One little known problem is related to consistency tests when 'growing' data frames. If we add one by one variables to a large data frame the overhead is in many cases huge. This can be easily reduced in many cases by assembling the object as a list, and once assembled converting it into a data frame. These common cases are used as examples in this chapter.

You may ask, how can I know, where in the code is the performance bottleneck. During the early years of R, this was quite a difficult task. Nowadays, good code profiling and code benchmarking tools are available. With interactive profiling now also implemented in the RStudio IDE. Profiling consists in measuring how much of the total runtime is spent in different functions, or even lines of code. Benchmarking consists in timing the execution of alternative versions of some piece of code, to decide which one should preferred.

There are some rules of style, and common sense, that should be always applied, to develop good quality program code. However, as in some cases, high performance comes at the cost of a more complex program or algorithm, optimizations should be applied only to the sections of the code that are limiting overall performance. Usually even when the requirement of high performance is known in advance, it is best to start with a simple implementation of a simple algorithm. Get this first solution working reliably, and use this as a reference both for performance and accuracy of returned results while attempting performance optimization.

The book *The Art of R Programming: A Tour of Statistical Software Design* (Matloff 2011) is very good at presenting the use of R language and how to profit from its peculiar features to write concise and efficient code. Studying the book *Advanced R* (Wickham 2019) will give you a deep understanding of the R language, its limitations and good and bad approaches to its use. If you aim at writing R packages, then *R Packages* (Wickham 2015) will guide you on how to write your own packages, using modern tools. Finally, any piece of software, benefits from thorough and consistent testing, and R packages and scripts are no exception.

Building a set of test cases simplifies enormously code maintenance, as they help detect unintended changes in program behaviour (Cotton 2016; Wickham 2015).

Interpreters and compilers Computer programs and scripts are nowadays almost always written in a high level language that is readable to humans, and that relies on a grammar much more complex than that understood by the hardware processor chip in the computer or device. Consequently one or more translation steps are needed. An interpreter, translates user code at the time of execution, and consequently parts of the code that are executed repeatedly are translated multiple times. A native compiler translates the user code into machine code in a separate step, and the compiled machine code can be stored and executed itself as many times as needed. On the other hand, compiled code can be executed only on a given hardware (processor, or processors from a given family). A byte-code compiler, translates user code into an intermediate representation, which cannot be directly executed by any hardware, and which is independent of the hardware architecture, but easier/faster to translate into machine code. This second interpreter is called a "virtual machine", as it is not dependent on a real hardware processor architecture.

An interpreter adds flexibility and makes interactive use possible, but results in slower execution compared to compiled executables. Nowadays, byte compiling is part of the R program, and used by default in some situations or under user control. Just-in-time (JIT) compiling is a relatively new feature in R, and consists in compiling on-the-fly code that is repeatedly evaluated within a single run of a script.

Functions or subroutines that have been compiled to machine code can be called from within R, but currently not written in the R language itself, as no native compiler exists for the R language. It is common to call from within R code, compiled functions or use whole libraries coded in languages such a C, C++ and FORTRAN when maximum execution speed is needed. The calls are normally done from within an R package, so that they appear to the user not different any other R function. Functions and libraries written in other interpreted and/or byte-compiled languages like Java and Python can also be called from R.

In addition, R exposes a programming interface (API) and many R functions can be called from within programs or scripts written in other languages such a Python and Java, also database systems and work sheets. This flexibility is one of the reasons behind R's popularity.

12.4 Measuring and improving performance

In this section we present simple "toy" examples of how execution speed of R code can be improved. These examples demonstrate the use of benchmarking and profiling tools and of the R built-in compiler to improve performance of R code. As I

wrote these examples some time ago, I was surprised by how much the benchmark results for the same code have changed since a couple of years ago.

The performance of several base R functions has improved dramatically in recent R versions even turning around the performance lead that equivalent functions from purportedly higher performance packages had. This highlights the need to regularly benchmark code if the best possible performance is aimed at, and to do the bench marking with data sets as large and complex as those to be used in production. It also highlights, that one should not assume *a priori* that base R functions will consistently under perform those from the 'tidyverse' and similar extensions to the R. In recent versions of R, in many cases the opposite is true.

Within base R there are huge differences in how different approaches to carry out a given operation perform. Usually, there is ample room for performance improvement within R code. In any case, the first step is to identify the segments of code that take most time to run, as it is only for these parts of the code that it is worthwhile spending effort in improving performance. So, the first step for improving existing code is to profile it. On the other hand when comparing the performance of different approaches to coding a given algorithm or comparing algorithms, performance *benchmarking* using test cases is the approach to use.

12.4.1 Benchmarking

We call benchmarking to the measurement of overall performance of a piece of code, in the examples below a statement calling a user-defined function. Benchmarking returns a summaries of the time of execution derived from consecutive repeated runs of the same code, by default 100 `times`. The examples below show how different approaches to a simple computation affect performance. We use package function `microbenchmark` from 'microbenchmark' to assess how effective our attempts at improving performance are.

```
library(microbenchmark)
```

We define a function using a `for` loop with only default optimization for performance. For loops have a fairly similar syntax in multiple programming languages, so usually are the first to be used.

```
my.fun01 <- function(x) {
  y <- numeric()
  for (i in seq(along.with = x[-1])) {
    y[i] <- x[i] * x[i+1] / log10(123456)
  }
  y
}
```

We time it. We are mainly interested in the *median* time. We use as input a numeric vector of length 1000. By default a suitable time unit is used, but we set

it to milliseconds with `unit = "ms"` to make sure that the same units are used in all the examples.

```
microbenchmark(my.fun01(runif(1e3)), unit = "ms")
## Unit: milliseconds
##                   expr   min      lq    mean median     uq     max neval
##  my.fun01(runif(1000)) 0.635 0.65485 0.80822 0.6698 0.7034 10.1011   100
```

Before continuing reading, study carefully the definition of `my.fun01()` and try to discover ways of improving its performance.

One approach is to reserve storage space for the result vector `y` ahead of the iterations through the loop as this avoids reallocation of memory and copying of the partial vector multiple times. We simply replace `y <- numeric()` by `y <- numeric(length(x))`.

```
my.fun02 <- function(x) {
  y <- numeric(length(x))
  for (i in seq(along.with = x[-1])) {
    y[i] <- x[i] * x[i+1] / log10(123456)
  }
  y
}
```

Benchmarking the new function shows that run time has decreased by 46%.

```
microbenchmark(my.fun02(runif(1e3)), unit = "ms")
## Unit: milliseconds
##                   expr    min      lq     mean median      uq   max neval
##  my.fun02(runif(1000)) 0.3922 0.40395 0.483658 0.4097 0.41585 7.489   100
```

Within the loop we have the calculation of the logarithm of a constant, a value that does not change from iteration to iteration, in technical terms, a *loop invariant computation*. We move this invariant calculation out of the loop.

```
my.fun03 <- function(x) {
  y <- numeric(length(x))
  k <- log10(123456)
  for (i in seq(along.with = x[-1])) {
    y[i] <- x[i] * x[i+1] / k
  }
  y
}
```

Benchmarking reveals that runtime is now decreased by almost 65% in this step, and by more than 80% from our starting point.

```
microbenchmark(my.fun03(runif(1e3)), unit = "ms")
## Unit: milliseconds
##                   expr    min     lq     mean median     uq   max neval
##  my.fun03(runif(1000)) 0.1579 0.1631 0.237668 0.1644 0.1656 7.406   100
```

However, we can use subscripting with vectors and avoid using iteration in an explicit loop.

```
my.fun04 <- function(x) {
  i <- seq(along.with = x[-1])
  x[i] * x[i+1] / log10(123456)
}
```

Now execution is really fast! We have decreased runtime to 4.5% of the initial implementation of the function, i.e., the performance-optimized version runs 22 times faster than the initial one.

```
microbenchmark(my.fun04(runif(1e3)), unit = "ms")
## Unit: milliseconds
##                   expr    min      lq     mean median      uq    max neval
##  my.fun04(runif(1000)) 0.0355 0.03735 0.104274 0.0384 0.03955 6.4731   100
```

 a) Benchmark the first three code chunks from section 3.3.5 starting at page 105. How big is the difference in execution time? b) Rerun the same benchmarks with 10, 100 and 1000 outer iterations. How do the differences among approaches depend on the number of iterations?

 The examples above show that in many cases performance can be improved a lot without recourse to complex code. In fact the code for our optimized function is a lot simpler than the initial one. Having read the book as far as this section, you must surely have some code of your own that could be optimized for performance. Make use of the ideas in this section to improve your own code.

 The example code used in this section needs also to be tested at boundary conditions. Try with any of the functions to run the following code.

```
my.fun01(1)
my.fun01(numeric(0))
my.fun01(NA)
my.fun01(NULL)
```

Repeat these tests for all the four functions. Do they all return the same values in each of these cases?

Finally validate the output of `my.fun04()` against `my.fun01()` with numeric arguments of different lengths.

12.4.2 Profiling

Profiling is the estimation of how much different parts of the code contribute to the total execution time. For example a fine grained profiling would be based on individual statements in the code. We will here use a different example. The approach we use is naive and expected to execute slowly and leave room for performance enhancements.

```
my.fun11 <- function(row = 100, col = 100) {
    df <- data.frame(1:row)
    for (i in 2:col)
      df <- cbind(df, 1:row)
  df
}
```

We check that our code is correct and that it returns the expected value, in this case a 5×5 data frame.

```
my.fun11(5, 5)
##   X1.row 1:row 1:row 1:row 1:row
## 1      1     1     1     1     1
## 2      2     2     2     2     2
## 3      3     3     3     3     3
## 4      4     4     4     4     4
## 5      5     5     5     5     5
```

One interesting piece of information is how the time spent increases with increasing size of the problem. There are two aspects to this: the shape of the curve, e.g. linear or exponential, and the problem-size-independent time overhead.

```
microbenchmark(my.fun11(row = 10, col = 10), unit = "ms")
## Unit: milliseconds
##                          expr    min      lq     mean  median      uq    max
##  my.fun11(row = 10, col = 10) 1.0834 1.13695 1.204857 1.16555 1.28315 1.4273
##  neval
##    100

microbenchmark(my.fun11(row = 100, col = 100), unit = "ms")
## Unit: milliseconds
##                            expr     min      lq     mean  median      uq
##  my.fun11(row = 100, col = 100) 11.1758 11.7058 12.61254 11.8936 12.2974
##      max neval
##  21.1652   100

microbenchmark(my.fun11(row = 1000, col = 1000), unit = "ms")
## Unit: milliseconds
##                              expr      min      lq     mean   median       uq
##  my.fun11(row = 1000, col = 1000) 125.2134 133.192 138.7893 138.1931 142.9165
##       max neval
##  165.3887   100
```

In the next two code chunks we find out whether adding columns or rows is more expensive in terms of run time.

```
microbenchmark(my.fun11(row = 500, col = 100), unit = "ms")
```

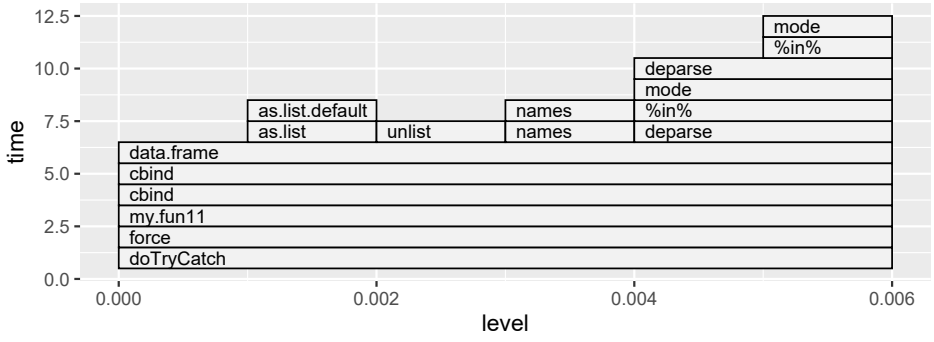
```
## Unit: milliseconds
##                              expr     min       lq     mean   median       uq
##  my.fun11(row = 500, col = 100) 11.1087 11.61065 12.51274 11.79405 12.49135
##      max neval
##  20.353   100
```

```
microbenchmark(my.fun11(row = 100, col = 500), unit = "ms")
## Unit: milliseconds
##                              expr    min      lq     mean   median       uq
##  my.fun11(row = 100, col = 500) 58.543 61.4713 65.19533 64.31825 68.12475
##      max neval
##  76.518   100
```

We now know that what is expensive is to add columns. We look into this problem by profiling the code to find where most time is being spent. We need to adjust the sampling interval to match the number statements the code under test has and how long it takes to run them. We can use the plot as a guide, if resolution is too little with too few levels and functions we need to decrease the interval, while if the plot looks like an unintelligible comb, the interval needs to be increases.

```
microbenchmark(my.fun11(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                              expr     min      lq     mean  median      uq
##  my.fun11(row = 200, col = 200) 23.3505 23.7321 28.06478 25.1578 27.4513
##       max neval
##  212.9722   100
```

```
prof.df <- profr(my.fun11(row = 500, col = 500), interval = 0.001)

ggplot(prof.df)
```



Modify the value passed as argument to parameter `interval`, and find a good value that gives more detail than the one used above. Be aware that profiling is done by sampling, and the output may vary between runs.

The problem seems to be with data frame and column-binding. We try a different approach to building a data frame: building first a list and converting it into

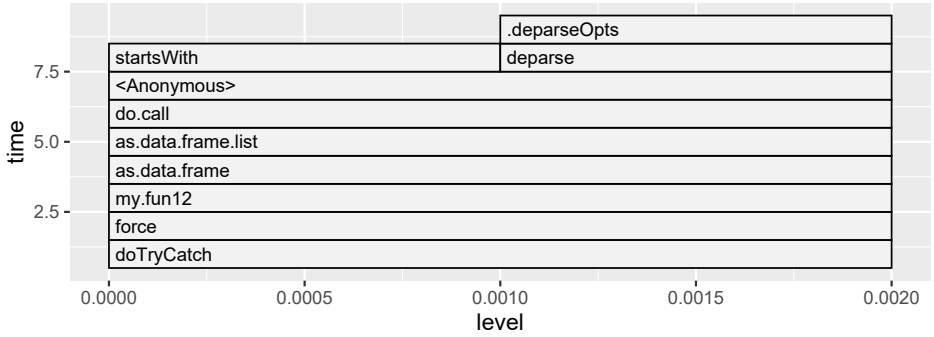
a data frame once assembled. The wisdom behind this is that the overhead of assembling a list is less than that of assembling a data frame, as the latter involves checking for the length of the vectors.

```
my.fun12 <- function(row = 200, col = 200) {
lst <- list()
    for (i in 1:col)
      lst <- c(lst, list(1:row))
    as.data.frame(lst)
}
```

We achieve a 40% reduction in execution time.

```
microbenchmark(my.fun12(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                          expr     min       lq     mean  median      uq
##  my.fun12(row = 200, col = 200) 12.6096 13.22485 14.79672 14.2827 15.4389
##      max neval
##  21.5371   100
```

```
prof12.df <- profr(my.fun12(row = 500, col = 500), interval = 0.001)

ggplot(prof12.df)
```



An optimization specific to this case—taking advantage that all values are numeric—is to use `rep()` to create a long vector, convert it into a matrix and the matrix into a data frame.

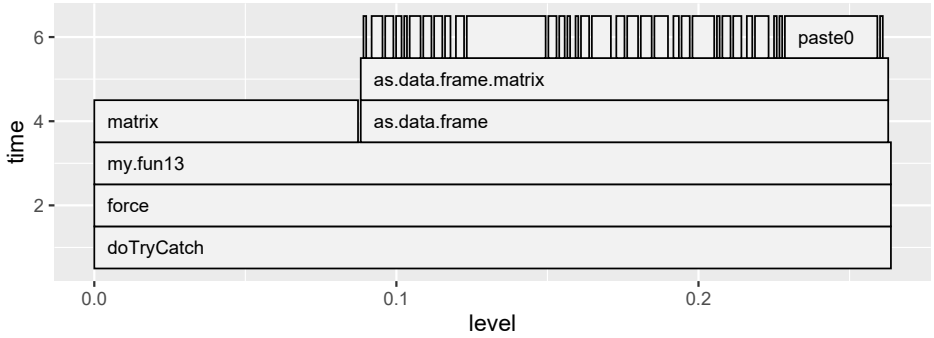
```
my.fun13 <- function(row = 200, col = 200) {
  mtx <- matrix(rep(1:row, col), ncol = col)
  as.data.frame(mtx)
}
```

A round about but very effective way of improving performance with a runtime that is only 1.6% of the starting one, or an execution speed 62.5 times faster than in the original naive approach used in `my.fun11()`.

```
microbenchmark(my.fun13(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                          expr    min     lq     mean median     uq    max
##  my.fun13(row = 200, col = 200) 0.6479 0.6579 0.770019 0.6669 0.6885 6.4999
##  neval
##    100
```

Unless we consider the current version fast enough, we will want to know where time is being spent in the improved version of the function. As the code runs now a lot faster it helps to decrease the interval used for profiling.

```
prof13.df <- profr(my.fun13(row = 500, col = 500), interval = 0.0009)

ggplot(prof13.df)
```



Can we still improve it?

```
my.fun14 <- function(row = 200, col = 200) {
  mtx <- matrix(rep(1:row, col), ncol = col)
  as_tibble(mtx, .name_repair = "minimal")
}
```

While some time ago using `as_tibble()` instead of `as.data.frame()` halved runtime, now it increases execution by 60% even with variable name repair disabled!

```
microbenchmark(my.fun14(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                            expr    min     lq     mean median      uq    max
##  my.fun14(row = 200, col = 200) 1.0092 1.0359 1.261248 1.0586 1.13805 7.5991
##  neval
##    100
```

The results from these examples show that we can obtain large performance improvements within base R code, and that the improvement obtained by using functions from the tidyverse can be sometimes marginal or even nonexistent. Differences are currently small enough for direction of differences to easily depend on the size of the data sets, computer hardware, compilers and operating system. The take home message is then that *if you do need* improved performance, in most cases it is worthwhile to assess where the bottlenecks are and deal with them within R, as R updates are unlikely to break existing code, while updates to packages in the 'tidyverse' rather frequently include code breaking changes.

Investigate how the size of the problem affects this heavily optimized code, as it was done with the initial version of the function at the start of this section.


We have gone very far in optimizing the function. In the last version the function returns a `tibble` instead of a `data.frame`. This can be expected to affect the performance of different operations, from indexing and computations to summaries when applied to the returned object. Use bench marking to assess these differences, both for cases with substantially more columns than rows, and more rows than columns. Think carefully a test case that makes heavy use of indexing, calculations combining several columns, and sorting.


Profile, using `profvis::profvis()` or `proftools::profr()` the first three code chunks from section 3.3.5 starting at page 105. In which functions is most of the execution time spent? What are the differences between the three implementations?

Profiling tools such as those provided by package 'proftools' can be useful when dealing with both simple and complex pieces of code. Furthermore, package 'profvis' provides an interactive user interface within RStudio, which makes it extremely easy to navigate to the code sections from the plot displaying the corresponding timings.

12.4.3 Compiling R functions

Although the current version of R uses by default the compiler quite frequently, we will demonstrate its manual use. We can see if a function is compiled, by printing it and looking if it contains a pointer to byte code.

To test what speed-up compiling can achieve for this small function we switch-off default compiling momentarily with function `enableJIT()`—JIT is an abbreviation for Just In Time compiler. Possible values of `levels` range from 0 to 3. Zero disables the compiler, while 1, 2 and 3 indicate use of the compiler by default in an increasing number of situations. In current versions of R, the default is 3.

We define, using a different name, the same function as earlier, and we check that it has not been compiled. Then we compile it.

```
old.level <- compiler::enableJIT(level = 0L)

my.fun11nc <- function(row = 200, col = 200) {
    df <- data.frame(1:row)
    for (i in 2:col)
      df <- cbind(df, 1:row)
  df
}

my.fun11nc
## function(row = 200, col = 200) {
##     df <- data.frame(1:row)
##     for (i in 2:col)
##       df <- cbind(df, 1:row)
##   df
```

```
## }

my.fun11c <- compiler::cmpfun(my.fun11nc)

my.fun11c
## function(row = 200, col = 200) {
##     df <- data.frame(1:row)
##     for (i in 2:col)
##       df <- cbind(df, 1:row)
##   df
## }
## <bytecode: 0x00000000275e6500>

microbenchmark(my.fun11nc(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                              expr     min      lq     mean   median       uq
##  my.fun11nc(row = 200, col = 200) 23.6347 23.9568 26.38908 24.94015 28.53015
##      max neval
##  35.6888   100

microbenchmark(my.fun11c(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                             expr     min       lq     mean  median       uq
##  my.fun11c(row = 200, col = 200) 23.3072 26.64975 29.85656 28.8888 32.25485
##      max neval
##  47.8868   100

compiler::enableJIT(level = old.level)
## [1] 0

microbenchmark(my.fun11nc(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                              expr     min       lq    mean  median       uq
##  my.fun11nc(row = 200, col = 200) 24.6973 27.91085 33.0632 31.7987 37.34015
##      max neval
##  53.5302   100

microbenchmark(my.fun11c(row = 200, col = 200), unit = "ms")
## Unit: milliseconds
##                             expr    min      lq     mean  median       uq
##  my.fun11c(row = 200, col = 200) 23.507 24.0358 27.10816 25.7045 29.20905
##      max neval
##  40.3067   100
```

For this small function pre-compilation yields no significant reduction in execution time.

12.5 R is great, but not always best

12.5.1 Using the best tool for each job

Frequently, optimizing R code for performance can yield more than an order of magnitude decrease in runtime. In many cases this is enough, and the most cost-effective solution. There are both packages and functions in base R, that if properly

used can make a huge difference in performance. In addition, efforts in recent years to optimize the overall performance of R itself have been successful. Some of the packages with enhanced performance have been described in earlier chapters, as they are easy enough to use and have also an easy to learn user interfaces. Other packages like 'data.table' although achieving very fast execution, incur the cost of using a user interface and having a behaviour alien to the "normal way of working" with R.

Sometimes, the best available tools for a certain job have not been implemented in R but are available in other languages. Alternatively, the algorithms or the size of the data are such that performance is poor when implemented in the R language, and can improved using a different language.

One extremely important feature leading to the success of R is extensibility. Not only by writing packages in R itself, but by allowing the development of packages containing functions written in other computer languages. The beauty of the package loading mechanism, is that even if R itself is written in C, and compiled into an executable, packages containing interpreted R code, and also compiled C, C++, FORTRAN, or other languages, or calling libraries written in Java, Python, etc. can be loaded and unloaded at runtime.

Most common reasons for using other programming languages, are the availability of libraries written in FORTRAN, C and C++ that are well tested and optimized for performance. This is frequently the case for numerical calculations and time-consuming data manipulations like image analysis. In such cases the R code in packages is just a wrapper (or "glue") to allow the functions in the library to be called from R.

In other cases we may diagnose a performance bottleneck, and decide to write a few functions within a package otherwise written in R, in a compiled language like C++. In such cases is a good idea to use bench marking to compare implementations, as the use of a different language does not necessarily provide a worthwhile performance enhancement. The reason behind this is that different languages do not always store data in computer memory in the same format. Differences among languages can add overhead to function calls across them, specially when they execute very quickly when called from R and/or when large amounts of data need to be shuffled back and forth between R and functions written in other languages. The R program itself is written in the C language.

12.5.2 C++

Nowadays, thanks to package 'Rcpp', using C++ code mixed with R language, is fairly simple (Eddelbuettel 2013). This package does not only provide R code, but a C++ header file with macro definitions that reduces the writing of the necessary "glue" code to the use of a simple macro in the C++ code. Although, this mechanism is most frequently used as a component packages, it is also possible to define a function written in C++ at the R console, or in a simple user's script. Of course for these to work all the tools needed to build R packages from source are needed, including a suitable compiler and linker.

An example taken from the 'Rcpp' documentation follows. This is an example of how one would define a function during an interactive session at the R console,

or in a simple script. When writing a package, one would write a separate source file for the function, include the `rcpp.h` header and use the C++ macros to build the R code side. Using C++ inline requires package 'inline' to be loaded in addition to 'Rcpp'.

First we save the source code for the function written in C++, taking advantage of types and templates defined in the `Rccp.h` header file.

```
src <- '
 Rcpp::NumericVector xa(a);
 Rcpp::NumericVector xb(b);
 int n_xa = xa.size(), n_xb = xb.size();

 Rcpp::NumericVector xab(n_xa + n_xb - 1);
 for (int i = 0; i < n_xa; i++)
 for (int j = 0; j < n_xb; j++)
 xab[i + j] += xa[i] * xb[j];
 return xab;
'
```

The second step is to compile and load the function, in a way that it can be called from R code and indistinguishable from a function defined in R itself.

```
# need to check maximum number of dll in Windows!
# and also compare to previous version of Rcpp
fun <- cxxfunction(signature(a = "numeric", b = "numeric"), src, plugin = "Rcpp")
```

We can now use it as any other R function.

```
fun(1:3, 1:4)
## [1]  1  4 10 16 17 12
```

As we will see below, this is not the case when calling Java and Python functions, cases where although the integration is relatively tight, special syntax is used when calling the "foreign" functions and/or methods. The advantage of Rcpp in this respect is very significant, as we can define functions that have exactly the same argument signature, use the same syntax and behave in the same way, using either the R or C++ language. This means that at any point during development of a package a function defined in R can be replaced by an equivalent function defined in C++, or vice versa, with absolutely no impact on user's code, except possibly for faster execution of the C++ version.

12.5.3 FORTRAN and C

In the case of FORTRAN and C, the process is less automated as the R code needed to call the compiled functions needs to be explicitly written (See *Writing R Extensions* in the R documentation, for up-to-date details). Once written, the building and installation of the package is automatic. This is the way how many existing libraries are called from within R and R packages.

12.5.4 Python

Package 'reticulate' allows calling Python functions and methods from R code. Under RStudio even the help from Python libraries and auto-completion are available

when writing R code. Obviously, Python should be installed and available. Nowadays, mixing and matching functions written in R and Python is easy and fairly automatic. See `https://rstudio.github.io/reticulate/` for detailed documentation.

It is also possible to call R functions from Python. However, this is outside the scope of this book.

12.5.5 Java

Although native Java compilers exist, most frequently Java programs are compiled into intermediate byte code and this is interpreted, and usually the interpreter includes a JIT compiler. For calling Java functions or accessing Java objects from R code, the solution is to use package 'rJava'. One important point to remember is that the Java SE Development Kit (JDK) must be installed for this package to work. The usually installed Java runtime environment (JRE) is not enough.

We need first to start the Java Virtual Machine (the byte-code interpreter).

```
.jinit()
```

The code that follows is not that clear, and merits some explanation.

We first create a Java array from inside R.

```
a <- .jarray( list(
                   .jnew( "java/awt/Point", 10L, 10L ),
                   .jnew( "java/awt/Point", 30L, 30L )
                   ) )
print(a)
## [1] "Java-Array-Object[Ljava/lang/Object;:[Ljava.lang.Object;@7852e922"

mode(a)
## [1] "S4"

class(a)
## [1] "jarrayRef"
## attr(,"package")
## [1] "rJava"

str(a)
## Formal class 'jarrayRef' [package "rJava"] with 2 slots
##   ..@ jobj  :<externalptr>
##   ..@ jclass: chr "[Ljava/lang/Object;"
##   ..@ jsig  : chr "[Ljava/lang/Object;"
```

Then we use base R function `lapply()` to apply a user-defined R function to the elements of the Java array, obtaining as returned value an R array.

```
b <- sapply(a,
            function(point){
              with(point, {
                (x + y )^2
              } )
            })
print(b)
## [1]  400 3600
```

```
mode(b)
## [1] "numeric"

class(b)
## [1] "numeric"

str(b)
##  num [1:2] 400 3600
```

Although more cumbersome than in the case of 'Rcpp' one can manually write wrapper code to hide the special syntax and object types from users.

It is also possible to call R functions from within a Java program. This is outside the scope of this book.

12.5.6 sh, bash

Thesh bash operating system shell can be accessed from within R and the output from programs and shell scripts returned to the R session. This is useful, for example for pre-processing raw data files with tools like AWK or Perl scripts. The problem with this approach is that when it is used, the R script cannot run portably across operating systems, or in the absence of the tools or sh or bash scripts. Except for code that will never be reused (i.e., used once and discarded) it is preferable to use R built-in commands whenever possible, or if shell scripts are used, to make the shell script the master script from within which the R scripts are called, rather than the other way around. The reason for this is mainly making clear the developer's intention: that the code as a whole will be run in a given operating system using a certain set of tools, rather hiding shell calls inside the R script. In other words, keep the least portable bits in full view.

12.6 Web pages, and interactive interfaces

There is a lot that could be written on using R to create web pages, interactive widgets and gadgets for both local and remote user interaction. This is an area currently under intense development. One example is the 'shiny' package and Shiny server https://shiny.rstudio.com/. This package allows the creation of interactive displays to be viewed through any web browser.

There are other packages for generating both static and interactive graphics in formats suitable for on-line display, as well as package 'knitr' https://yihui.name/knitr/ used for writing the present book, which when using R Markdown for markup (package 'rmarkdown' http://rmarkdown.rstudio.com, 'bookdown' https://bookdown.org/ or 'blogdown' https://blogdown.org/ can output self-contained HTML files. 'rmarkdown' and 'bookdown' can in addition to HTML generate RTF and PDF files.

Several books have been published on the use of Markdown in R, *R Markdown* (Xie et al. 2018), *bookdown: Authoring Books and Technical Documents with R Markdown* (Xie 2016) and *Blogdown : creating websites with R Markdown* (Xie 2018).

12.7 A final thought on package writing

Some R packages provide some fundamentally new capabilities, but many packages provide new user interfaces to functionality that already exists in base R or contributed packages. In many cases the improved usability provided by a well thought and consistent user interface can make packages that are to a large extent wrappers on existing functions and methods still worthwhile developing.

If you attempt to improve or extend existing functionality, do study existing code. R itself and all packages archived in CRAN are open-source and published under licences that allow re-use and modification as long you acknowledge all authors and abide to the requirement of releasing your version under the same or similar licences as the original code.

If you study the code of package 'ggpmisc', you will quickly realize that for writing many of the methods and functions I have used code from package 'ggplot2' as a template, building upon it with new code that calls functions defined in base R and very frequently also in other packages. Of course, other packages may do the opposite: to respect the user interface of existing functions but provide improved performance.

Some of the packages in the 'tidyverse' developed by Hadley Wickham and collaborators evolved so that early versions focused on an improved user interaction (improved grammars) and only later performance was honed until it became as good or better than that of the earlier alternatives and R itself. The development of 'data.table' seems to have started by focusing on performance, and much of the unusual user interface reflects this. The packages in the 'tidyverse' are becoming very popular, while 'data.table' remains a rather specialized tool for cases where performance is extremely important. Package 'dtplyr' implements the syntax of 'dplyr' using package 'data.table' as a back end for the operations.

When developing R packages, including a good coverage of test cases as part of the package itself simplifies code maintenance enormously, helps in maintaining consistency of behaviour across versions, and reveals in good time problems triggered by updates to R or packages depended upon.

So, my recommendation is to spend time designing and testing the user interface first. Go through design and test cycles, until you are sure the interface is user friendly, consistent and complete. Be consistent not only within your own code, but with R itself and/or popular packages. Avoid name clashes except when the functions or methods you develop are intended to directly replace those they make inaccessible. Be very careful with new names for methods, functions and operators, make sure they fully describe their function. If equivalent methods exist in R or in the packages your own code expands upon, if possible define method specializations of the existing methods for the new classes you define. Try to create as few new *words* for the user to learn. However, keep the future in mind, making the user interface flexible and expandable. Even if you will be the only user, having a good interface will make it easier for you to remember how to use your own package.

```
try(detach(package:ggplot2))
try(detach(package:tibble))
try(detach(package:profr))
try(detach(package:microbenchmark))
# try(detach(package:rJava))
# try(detach(package:rPython))
try(detach(package:inline))
try(detach(package:Rcpp))
```

Part III

Example Cases

A

Case: Timeline plots

> What this means is that we shouldn't abbreviate the truth but rather get a new method of presentation.
>
> Edward Tufte
> *Q+A - Edward Tufte*, 2007

A.1 Aims of this chapter

The chapters in this appendix exemplify specific use cases. The present chapter shows how to produce nice and accurate timeline plots as ggplots. It is based on an article written by the author currently *in press* in the *UV4Plants Bulletin* (`https://bulletin.uv4plants.org`) issue 2021:1.

A.2 Packages used in this chapter

If the packages used in this chapter are not yet installed in your computer, you can install them with the code shown below, as long as package 'learnrbook' (>= 1.0.4) is already installed.

```
packages_at_cran <-
  setdiff(learnrbook::pkgs_ch_ggplot_extra, learnrbook::pkgs_at_github)
install.packages(packages_at_cran)
for (p in learnrbook::pkgs_at_github) {
  devtools::install_github(p)
}
```

For executing the examples listed in this chapter you need first to load the following packages from the library:

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

```
library(ggpp)
library(ggrepel)
library(lubridate)
library(photobiology)
```

We set a font of larger size than the default.

```
theme_set(theme_grey(14))
```

A.3 What is a timeline plot?

The earliest examples of timeline plots date from the 1 700's (Koponen and Hildén 2019). Initially used to depict historical events, timeline plots are useful to describe any sequence of events and periods along a time axis. To be useful the events should be positioned proportionally, i.e., the distance along the timeline at which events are marked, should be proportional to the time interval separating these events (Koponen and Hildén 2019).

Timelines provide a very clear, unambiguous and concise way of describing a sequence of historical events, the time course of scientific experiments, the timing of steps in laboratory protocols and even the progress of people's careers in science, sports, politics, etc. I suspect they are used less frequently than they could because drafting them can seem like a lot of work. But is it necessarily so?

Let's think what a time-line really is: it is in essence a one-dimensional data plot, where a single axis represents time and events or periods are marked and usually labelled. So the answer on how to easily and accurately draw a timeline plot is to use data plotting software instead of "free-hand" drafting software. I will show examples using R package 'ggplot2' and some extensions to it. The beauty of this approach is that there is little manual fiddling, the time-line plot is created from data, and the code can be reused with few changes for other timelines by plotting different data. As this is the only question for this issue, I provide an extended answer with multiple examples.

A.4 A simple timeline

The first example is a timeline plot showing when each past issue of the UV4Plants Bulletin was published. We use package 'ggplot2' for the plotting, and package 'lubridate' to operate on dates.

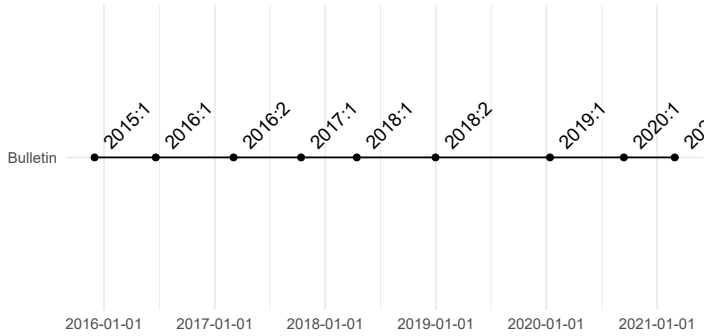
We write the data into a data frame.

```
issues.tb <-
  data.frame(what = c("2015:1", "2016:1", "2016:2", "2017:1", "2018:1",
                      "2018:2", "2019:1", "2020:1", "2020:2"),
             when = ymd(c("2015-12-01", "2016-06-20", "2017-03-04",
```

```
                          "2017-10-13", "2018-04-15", "2018-12-31",
                          "2020-01-13", "2020-09-13", "2021-02-28")),
            event.type = "Bulletin")
```

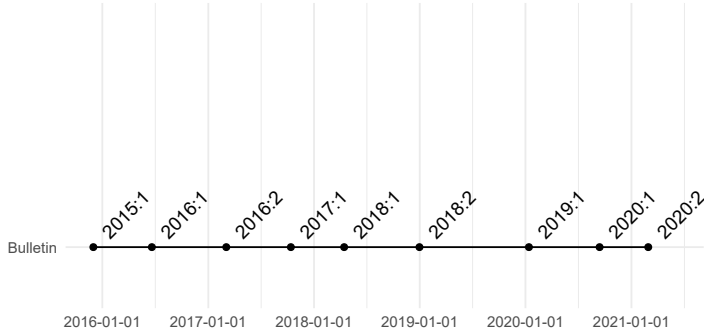
A timeline with sparse events is easy to plot once we have the data. The plots are displayed in this chaapter as narrow horizontal strips. The width and height of the plots is decided when they are printed or exported (code not visible here).

```
ggplot(issues.tb, aes(x = when, y = event.type, label = what)) +
  geom_line() +
  geom_point() +
  geom_text(hjust = -0.3, angle = 45) +
  scale_x_date(name = "", date_breaks = "1 years") +
  scale_y_discrete(name = "") +
  theme_minimal()
```



In the plot above, using defaults, there is too much white space below the timeline and the rightmost text label is not shown in full. We expand the x axis on the right and remove some white space from the y axis by adjusting the scales' expansion (the default is `mult = 0.05`).

```
ggplot(issues.tb, aes(x = when, y = event.type, label = what)) +
  geom_line() +
  geom_point() +
  geom_text(hjust = -0.3, angle = 45) +
  scale_x_date(name = "", date_breaks = "1 years",
               expand = expansion(mult = c(0.05, 0.1))) +
  scale_y_discrete(name = "",
                   expand = expansion(mult = c(0.01, 0.04))) +
  theme_minimal()
```



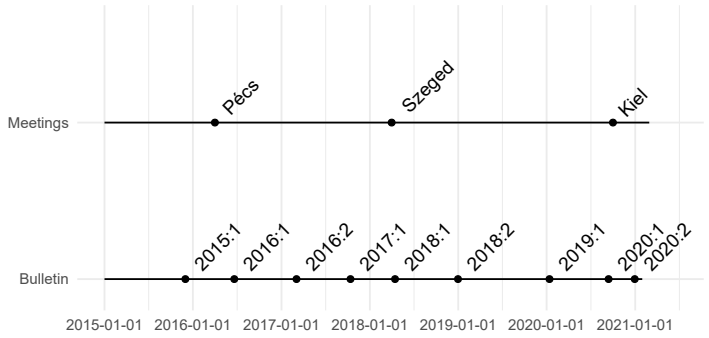
A.5 Two parallel timelines

Let's add a second parallel timeline with the UV4Plants Network Meetings and extend both ends of the lines. We use package 'dplyr' to filter part of the data on the fly so that no points are plotted at the ends of the lines.

```
uv4plants.tb <-
  data.frame(what = c("", "2015:1", "2016:1", "2016:2", "2017:1", "2018:1",
                      "2018:2", "2019:1", "2020:1", "2020:2", "",
                      "", "Pécs  ", "Szeged", "Kiel  ", ""),
             when = ymd(c("2015-01-01", "2015-12-01", "2016-06-20",
                          "2017-03-04", "2017-10-13", "2018-04-15",
                          "2018-12-31", "2020-01-13", "2020-09-13",
                          "2020-12-30", "2021-01-30",
                          "2015-01-01", "2016-04-01", "2018-04-01",
                          "2020-10-01", "2021-02-28")),
             event.type = c(rep("Bulletin", 11), rep("Meetings", 5)))
```

Compared to a single timeline the main change is in the data. The code remains very similar. We do need to adjust the expansion of the y-axis (by trial and error).

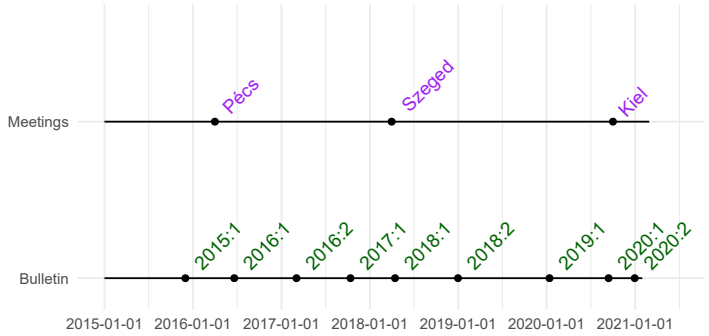
```
ggplot(uv4plants.tb, aes(x = when, y = event.type, label = what)) +
  geom_line() +
  geom_point(data = . %>% filter(what != "")) +
  geom_text(hjust = -0.3, angle = 45) +
  scale_x_date(name = "", date_breaks = "1 years",
               expand = expansion(mult = c(0.05, 0.1))) +
  scale_y_discrete(name = "",
                   expand = expansion(mult = c(0.2, 0.75))) +
  theme_minimal()
```



We add colours by adding `aes(colour = event.type)` to the call to `geom_text()`. To override the default colours we use `scale_colour_manual()` and as we do not need a key to indicate the meaning of the colours, we add to the call `guide = "none"`. In this example the colour is only applied to the text labels, but we can similarly add colour to the lines and points.

```
ggplot(uv4plants.tb, aes(x = when, y = event.type, label = what)) +
  geom_line() +
  geom_point(data = . %>% filter(what != "")) +
  geom_text(aes(colour = event.type), hjust = -0.3, angle = 45) +
```

```
  scale_x_date(name = "", date_breaks = "1 years",
               expand = expansion(mult = c(0.05, 0.1))) +
  scale_y_discrete(name = "",
                   expand = expansion(mult = c(0.2, 0.75))) +
  scale_colour_manual(values = c(Bulletin = "darkgreen", Meetings = "purple"),
                      guide = "none") +
  theme_minimal()
```



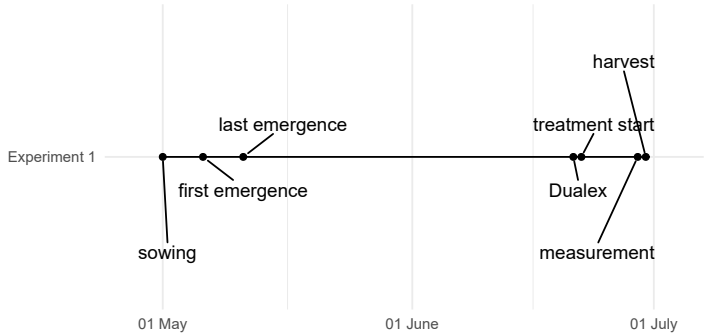
A.6 Crowded timeline

Let's assume an experiment with plants, and create some data. As the labels are rather long and we want to keep the text horizontal, we will use package 'ggrepel' which provides geoms that implement repulsion of labels to automatically avoid overlaps.

```
plants.tb <-
  data.frame(what = c("sowing", "first emergence", "last emergence", "Dualex",
                      "treatment start", "measurement", "harvest"),
             when = ymd(c("2020-05-01", "2020-05-06", "2020-05-11", "2020-06-21",
                          "2020-06-22", "2020-06-29", "2020-06-30")),
             series = "Experiment 1")
```

Now the labels would overlap, so we let R find a place for them using `geom_text_repel()` instead of `geom_text()`.

```
ggplot(plants.tb, aes(x = when, y = series, label = what)) +
    geom_line() +
    geom_point() +
    geom_text_repel(direction = "y",
                    point.padding = 0.5,
                    hjust = 0,
                    box.padding = 1,
                    seed = 123) +
    scale_x_date(name = "", date_breaks = "1 months", date_labels = "%d %B",
                 expand = expansion(mult = c(0.12, 0.12))) +
    scale_y_discrete(name = "") +
    theme_minimal()
```

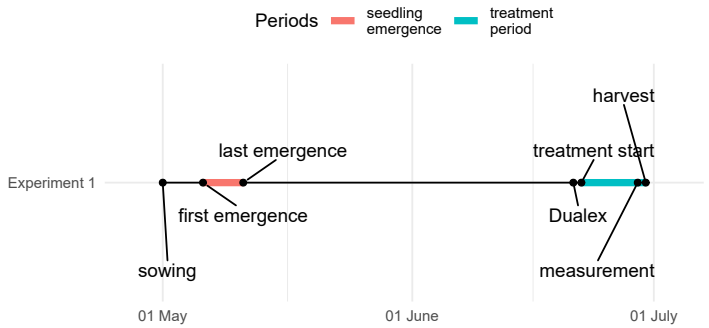


As germination and treatments are periods, we can highlight them more elegantly. For this we will create a second data frame with data for the periods.

```
plants_periods.tb <-
  data.frame(Periods = c("seedling\nemergence",
                         "treatment\nperiod"),
             start = ymd(c("2020-05-06", "2020-06-22")),
             end = ymd(c("2020-05-11", "2020-06-30")),
             series = "Experiment 1")
```

We highlight two periods using colours, and move the corresponding key to the top.

```
ggplot(plants.tb, aes(x = when, y = series)) +
  geom_line() +
  geom_segment(data = plants_periods.tb,
               mapping = aes(x = start, xend = end,
                             y = series, yend = series,
                             colour = Periods),
               size = 2) +
  geom_point() +
  geom_text_repel(aes(label = what),
                  direction = "y",
                  point.padding = 0.5,
                  hjust = 0,
                  box.padding = 1,
                  seed = 123) +
  scale_x_date(name = "", date_breaks = "1 months", date_labels = "%d %B",
               expand = expansion(mult = c(0.12, 0.12))) +
  scale_y_discrete(name = "") +
  theme_minimal() +
  theme(legend.position = "top")
```



A.7 Timelines of graphic elements

Finally an example where the "labels" are inset plots. The aim is to have a timeline of the course of the year with plots showing the course of solar elevation through specific days of the year. This is a more complex example where we customize the plot theme.

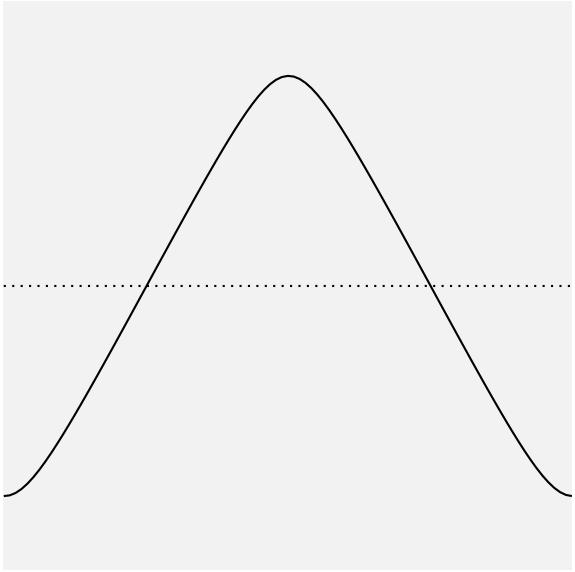
We will use package 'ggpp' that provides a geom for easily insetting plots into a larger plot and package 'photobiology' to compute the solar elevation.

As we will need to make several, very similar plots, we first define a function that returns a plot and takes as arguments a date and a geocode. The intention is to create plots that are almost like icons, extremely simple but still comparable and conveying useful information. To achieve these aims we need to make sure that irrespective of the actual range of solar elevations the limits of the y-axis are -90 and +90 degrees. We use a pale gray background instead of axes to show this range, but making sure that the default expansion of the axis limits is not applied.

```
make_sun_elevation_plot <- function(date, geocode) {
  # 97 points in time from midnight to midnight
  t <- rep(date, 24 * 4 + 1) +
         hours(c(rep(0:23, each = 4), 24)) +
         minutes(c(rep(c(0, 15, 30, 45), times = 24), 0))
  e <- sun_elevation(time = t, geocode = geocode)
  ggplot(data.frame(t, e), aes(t, e)) +
    geom_hline(yintercept = 0, linetype = "dotted") +
    geom_line() +
    expand_limits(y = c(-90, 90)) +
    scale_x_datetime(date_labels = "%H:%m", expand = expansion()) +
    scale_y_continuous(expand = expansion()) +
    theme_void() +
    theme(panel.background = element_rect(fill = "grey95",
                                          color = NA),
          panel.border = element_blank())
}
```

```
make_sun_elevation_plot(date = ymd("2020-06-21"),
                        geocode = tibble(lon = 0,
```

```
                                    lat = 0,
                                    address = "Equator"))
```

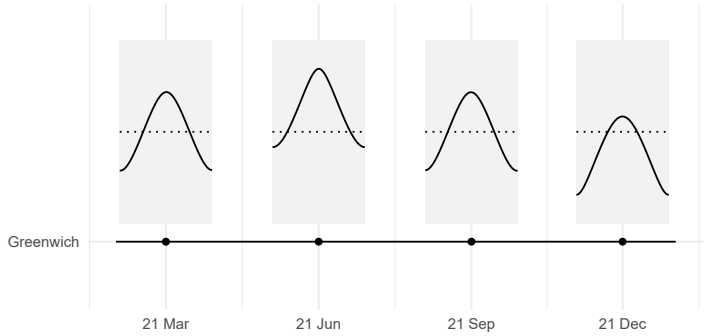


Now that we have a working function, assembling the data for the timeline is not much more complex than in earlier examples. We add two dates, one at each

```
geocode <- tibble(lon = 0, lat = 51.5, address = "Greenwich")
dates <- ymd(c("2020-02-20", "2020-03-21", "2020-6-21",
               "2020-09-21", "2020-12-21", "2021-01-22"))
date.ticks <- dates[2:5]
date.ends <- dates[c(1, 6)]
sun_elevation.tb <-
  tibble(when = dates,
         where = geocode$address,
         plots = lapply(dates, make_sun_elevation_plot, geocode = geocode))
```

The code used to plot the timeline follows the same pattern as in the examples above, except that we replace `geom_text()` with `geom_plot()`. This also entails overriding the default size (`vp.height` and `vp.width`) and justification (`vjust` and `hjust`) of the insets .

```
ggplot(sun_elevation.tb, aes(x = when, y = where, label = plots)) +
    geom_line() +
    geom_point(data = . %>% filter(day(when) == 21)) +
    geom_plot(data = . %>% filter(day(when) == 21),
              inherit.aes = TRUE,
              vp.width = 0.15, vp.height = 0.6, vjust = -0.1, hjust = 0.5) +
    scale_x_date(name = "", breaks = date.ticks, date_labels = "%d %b") +
    scale_y_discrete(name = "",
                     expand = expansion(mult = c(0.1, 0.35))) +
    theme_minimal()
```

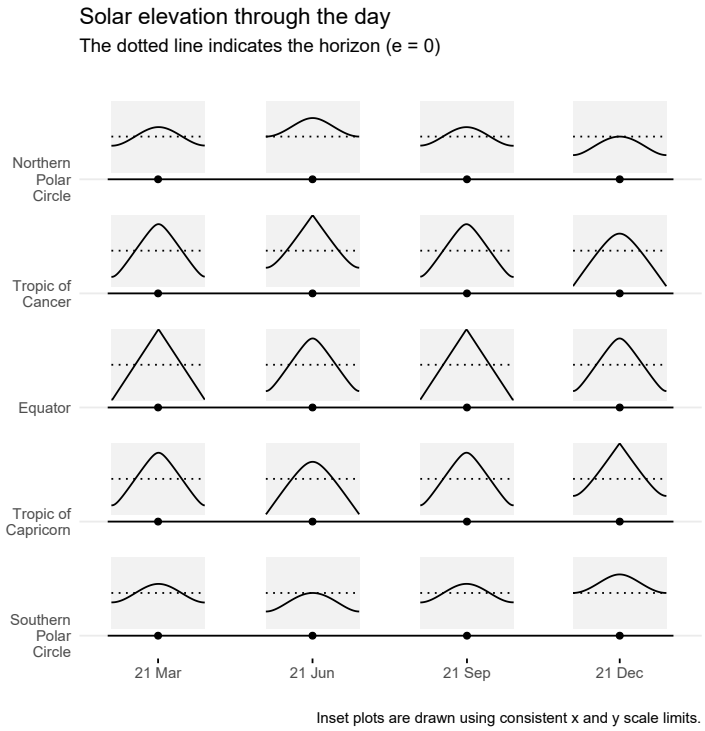


And to finalize, we plot three parallel timelines, each for a different latitude. For this we can reuse the function defined above, passing as argument geocodes for three different locations. We reuse the dates defined above, but use `rep()` to repeat this sequence for each location.

```
geocodes <-
  tibble(lon = c(0, 0, 0, 0, 0),
         lat = c(66.5634, 23.4394, 0, -23.4394, -66.5634),
         address = c("Northern\nPolar\nCircle", "Tropic of\nCancer", "Equator",
                     "Tropic of\nCapricorn", "Southern\nPolar\nCircle"))
sun_elevation.tb <-
  tibble(when = rep(dates, nrow(geocodes)),
         where = rep(geocodes$address, each = length(dates)),
         plots = c(lapply(dates, make_sun_elevation_plot, geocode = geocodes[1, ]),
                   lapply(dates, make_sun_elevation_plot, geocode = geocodes[2, ]),
                   lapply(dates, make_sun_elevation_plot, geocode = geocodes[3, ]),
                   lapply(dates, make_sun_elevation_plot, geocode = geocodes[4, ]),
                  lapply(dates, make_sun_elevation_plot, geocode = geocodes[5, ])))
sun_elevation.tb$where <-
  factor(sun_elevation.tb$where, levels = rev(geocodes$address))
```

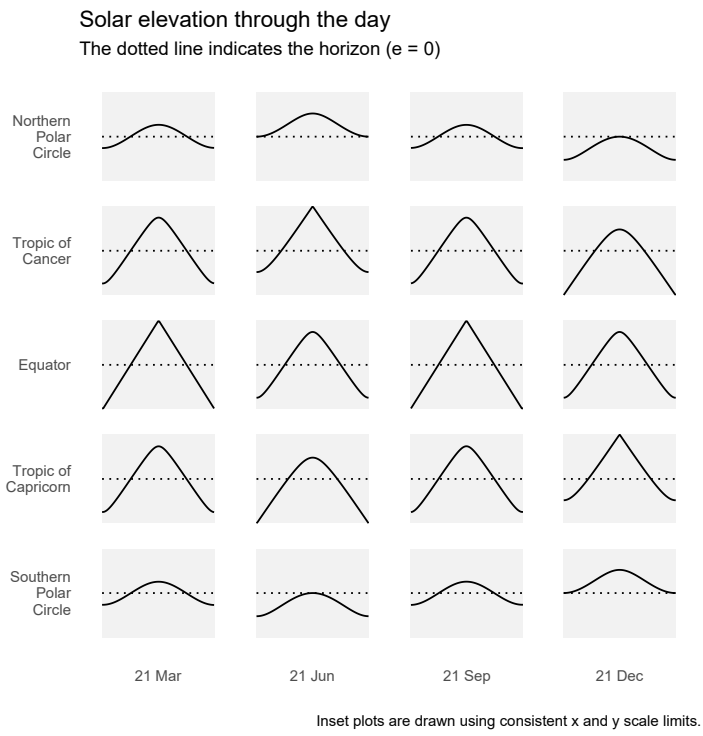
The only change from the code used above to plot a single timeline is related to the vertical size of the inset plots as it is expressed relative to the size of the whole plot. We also add a title, a subtitle and a caption, and tweak the theme of the main plot.

```
ggplot(sun_elevation.tb, aes(x = when, y = where, label = plots)) +
  geom_line() +
  geom_point(data = . %>% filter(day(when) == 21)) +
  geom_plot(data = . %>% filter(day(when) == 21),
            inherit.aes = TRUE,
            vp.width = 0.15, vp.height = 0.12, vjust = -0.1, hjust = 0.5) +
  scale_x_date(name = "", breaks = date.ticks, date_labels = "%d %b") +
  scale_y_discrete(name = "", expand = expansion(mult = c(0.05, 0.25))) +
  labs(title = "Solar elevation through the day",
       subtitle = "The dotted line indicates the horizon (e = 0)",
       caption = "Inset plots are drawn using consistent x and y scale limits.") +
  theme_minimal() +
  theme(panel.grid.minor.x = element_blank(),
        panel.grid.major.x = element_blank(),
        axis.ticks.x.bottom = element_line())
```



The five parallel timelines become rather crowded so an option is to build a matrix of plots. We achieve this by editing the code for the previous plot.

```
ggplot(sun_elevation.tb, aes(x = when, y = where, label = plots)) +
  geom_plot(data = . %>% filter(day(when) == 21),
            inherit.aes = TRUE,
            vp.width = 0.18, vp.height = 0.15, vjust = 0.5, hjust = 0.5) +
  scale_x_date(name = "", breaks = date.ticks,
               limits = date.ends,
               date_labels = "%d %b") +
  scale_y_discrete(name = "") +
  labs(title = "Solar elevation through the day",
       subtitle = "The dotted line indicates the horizon (e = 0)",
       caption = "Inset plots are drawn using consistent x and y scale limits.") +
  theme_minimal() +
  theme(panel.grid.minor = element_blank(),
        panel.grid.major = element_blank())
```



A.8 Related plots

Package 'ggpp' provides in addition to `geom_plot()`, `geom_table()` and `geom_grob()`. The first of them makes it possible to inset a data frame as a table, and the second any graphic object supported by package 'grid' (`grob` for short), which is part of R itself. These graphic objects can be vector graphics or bitmaps (or raster images) converted into `grob`s. Examples of how to convert bitmaps and vector graphics read from files of various formats is described in the documentation of packages 'grid', 'magick' and more briefly in package 'ggpp'. See the book *R Graphics* (Murrell 2011) for details.

Using photographs converted to grobs one can, for example, create phenological time lines. Another variation could to be to use a similar approach to represent geographic or topographic transects. In this last case instead of only using the x-axis to map time, one could map distance to the x-axis and elevation to the y-axis. Furthermore, one can use package 'patchwork' to assemble a multi-panel figure in which one panel is a timeline plot and other panels display other types of plots or even tables.

```
try(detach(package:photobiology))
try(detach(package:lubridate))
try(detach(package:ggrepel))
try(detach(package:ggpp))
try(detach(package:ggplot2))
try(detach(package:dplyr))
```

B

Case: Regression in plots

B.1 Aims of this chapter

The chapters in this appendix exemplify specific use cases. The present chapter discusses how to represent the results from fitted regression models in plots and how to implement them using packages 'ggplot2', 'ggpp' and 'ggpmisc'. We consider models linear- and non-linear in their parameters, and ordinary least squares (OLS) and other fitting criteria. We expand the discussion in section ?? on page ??.

B.2 Packages used in this chapter

If the packages used in this chapter are not yet installed in your computer, you can install them with the code shown below, as long as package 'learnrbook' (>= 1.0.4) is already installed.

```
packages_at_cran <-
  setdiff(learnrbook::pkgs_ch_ggplot_extra, learnrbook::pkgs_at_github)
install.packages(packages_at_cran)
for (p in learnrbook::pkgs_at_github) {
  devtools::install_github(p)
}
```

The examples in this chapter make use of features added to package 'ggplot2' in version 3.3.0, to package 'ggpp' in version 0.4.3 and to package 'ggpmisc' in version 0.4.4. For executing the examples listed in this chapter you need first to load recent enough versions of the following packages from the library:

```
library(ggplot2)
library(ggpmisc)
library(broom)
```

We set a font of larger size than the default.

```
theme_set(theme_grey(14) + theme(legend.position = "bottom"))
```

B.3 Introduction

Some years ago I was asked if I knew a simple way of adding fitted linear model coefficients as an equation in a ggplot. I was also shown quite complex code in answers in StackOverflow to achieve this. This was how package 'ggpmisc' was born, answering a colleague's question. Quite many written and re-written lines of code later 'ggpmisc' has become a comprehensive tool for such annotations and related ones. In the year since the publication of *Learn R: As a Language* I made further enhancements to package 'ggpmisc'. Model-fit-based annotations are briefly described in section 7.5.2 on page 239. Here we describe them in more detail and taking advantage of enhancements to the packages in versions released after the book went to press.

Polynomial, including linear, regression are very frequently used. However, in many cases they are not necessarily the best choice. So, we start here with a very brief discussion of different models that we may like to fit to data in a plot to summarize them and why. We will not deal in this chapter with the fitting of `loess` and other cubic-spline based smoothers. We discuss linear and non-linear models fitting by OLS approaches as well as major axis regression (also called Model II regression) and quantile regression approaches. The results of these fits can be represented in plots as lines and bands but also by equations and inset tables added as plot annotations.

The grammar of graphics and its limitations. As described in chapter 7 starting on page 203, each layer in a ggplot is based on a *statistics* chained to a *geometry*. This provides a clean and simple syntax based on passing a data frame with data and mapping the variables to *aesthetics*. This, however, creates a limitation on what data can be conveyed from a *statistics* to a *geometry* and also implies that the data computed within a *statistic* can be passed to a single *geometry*. This makes it necessary if we want to avoid stepping outside the grammar of graphics to sometimes fit the same model to the same data more than once when rendering a given plot. Consequently, as I aimed to retain simplicity and to remain true to the grammar of graphics, the layer functions in package 'ggpmisc' may result in repeated model fit computations.

B.4 Linear models

As shown in section ?? on page ?? linear models fitted to continuous variables mapped to `x` and `y` aesthetics can be fitted and the predicted line with a confidence band added as a layer to a ggplot with `stat_smooth()` by setting `method = "lm"`. Statistic `stat_poly_line()` is similar and even though it has enhancements as described below, it also supports a smaller set of fitting methods, in particular, it

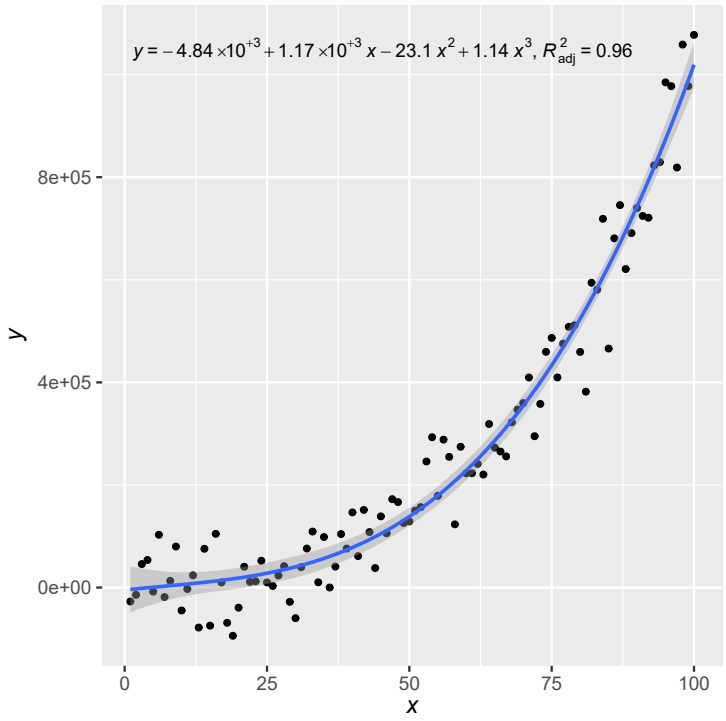
currently does not support fitting of smooth splines. Statistic `stat_poly_eq()`, is consistent in interface and supported fit methods with `stat_poly_line()` but instead of adding a line and band, it can be used to add annotations based on coefficient estimates, such as an equation, and estimates for most other parameters that can be extracted from model fit objects.

```
set.seed(4321)
# generate artificial data
x <- 1:100
y <- (x + x^2 + x^3) + rnorm(length(x), mean = 0, sd = mean(x^3) / 4)
my.data <- data.frame(x,
                      y,
                      group = c("A", "B"),
                      y2 = y * c(0.5,2),
                      block = c("a", "a", "b", "b"),
                      wt = sqrt(x))
```

```
formula <- y ~ poly(x, 3, raw = TRUE)
```

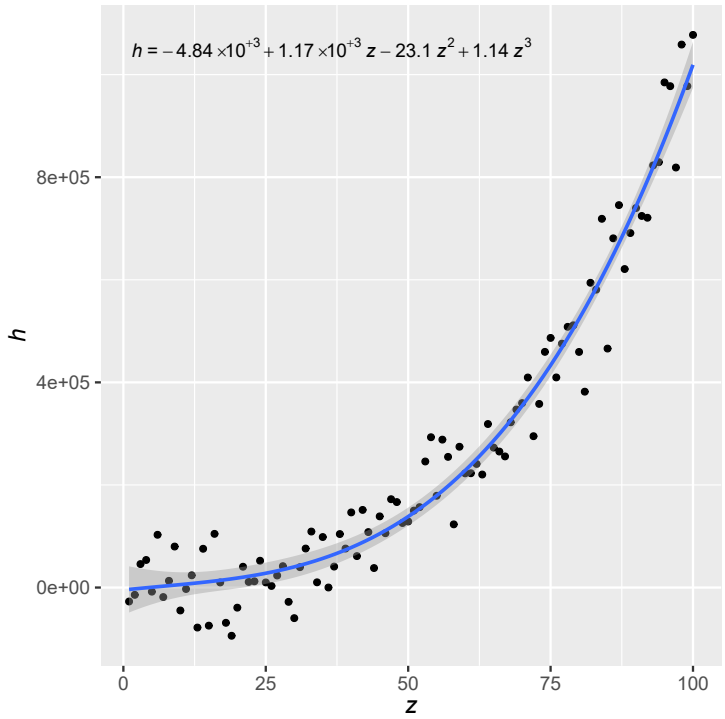
Within `aes()` it is also possible to compute new labels based on those returned plus "arbitrary" text. The labels returned by default are meant to be parsed into R expressions, so any text added should be valid for a string that will be parsed. When using `paste()`, inserting a comma plus white space in an expression requires some trickery in the argument passed to `sep`. Do note the need to escape the embedded quotation marks as `\"`. Combining the labels in this way ensures correct alignment. To insert only white space `sep = "~"` can be used, with each tilde character, `~`, adding a rather small amount of white space. We show herein addition how to change the axis labels to ensure consistency with the typesetting of the equation.

```
ggplot(my.data, aes(x, y)) +
  geom_point() +
  stat_poly_line(formula = formula) +
  stat_poly_eq(aes(label =  paste(stat(eq.label), stat(adj.rr.label),
                                  sep = "*\", \"*")),
               formula = formula) +
  labs(x = expression(italic(x)), y = expression(italic(y)))
```



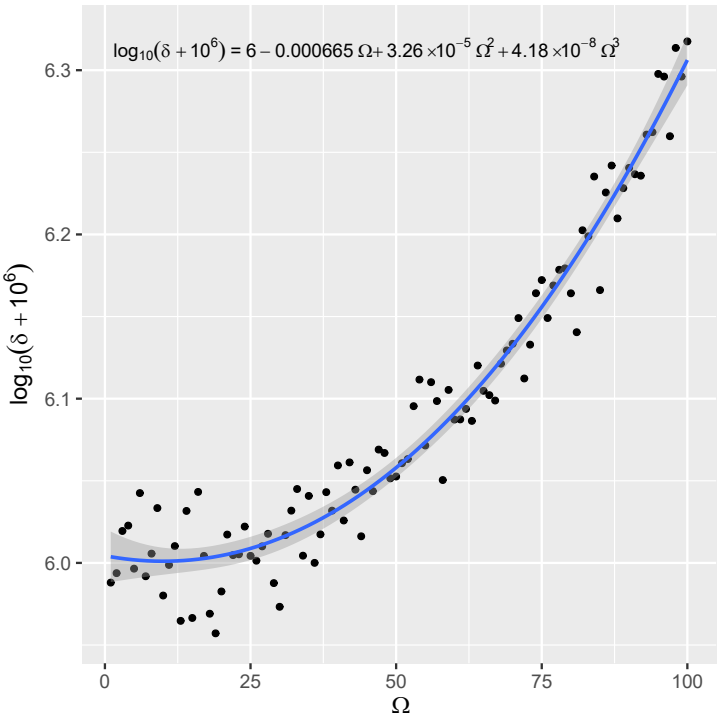
A more advanced example replaces x and y in the equation to match simple axis labels.

```
ggplot(my.data, aes(x, y)) +
  geom_point() +
  stat_poly_line(formula = formula) +
  stat_poly_eq(aes(label = stat(eq.label)),
               eq.with.lhs = "italic(h)~`=`~",
               eq.x.rhs = "~italic(z)",
               formula = formula) +
  labs(x = expression(italic(z)), y = expression(italic(h)))
```



The replacements for x and y can be character strings to be parsed to complex R expressions.

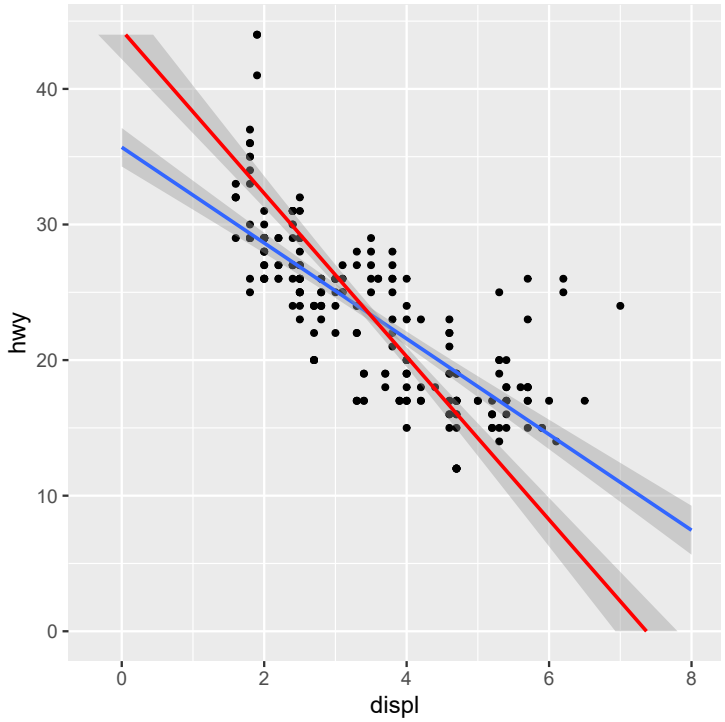
```
ggplot(my.data, aes(x, log10(y + 1e6))) +
  geom_point() +
  stat_poly_line(formula = formula) +
  stat_poly_eq(aes(label = stat(eq.label)),
               eq.with.lhs = "plain(log)[10](italic(delta)+10^6)~`=`~",
               eq.x.rhs = "~Omega",
               formula = formula) +
  labs(y = expression(plain(log)[10](italic(delta)+10^6)), x = expression(Omega))
```



Add ROBUST REGRESSION examples with rlm!!

If we desire to consider the variable mapped to y as the independent variable in the model fit, we can indicate this with a suitable `formula`. In this case we show that for ordinary least square (OLS) fits this change yields a very different fitted line. We expand the scale limits and extrapolate the fitted line to cover the full range of the scales.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  stat_poly_line(formula = y ~ x, fullrange = TRUE) +
  stat_poly_line(formula = x ~ y, fullrange = TRUE, colour = "red") +
  expand_limits(x = c(0, 8), y = 0)
```

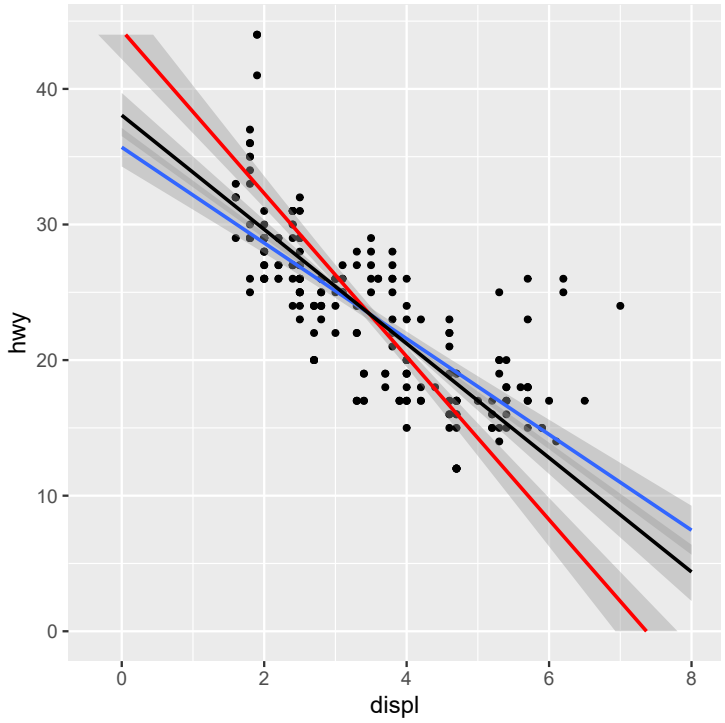


In this example, it is clear that we should prefer `formula = y ~ x` as the engine displacement is expected to be the cause of differences in highway fuel use. There are cases, as shown next, where both variables can be considered as independent.

B.5 Major axis regression

Before switching to a data set that requires the use of major axis regression, we add the line for ranged major axis (RMA) regression to the last plot above. We can see that it falls in-between the two OLS regressions.

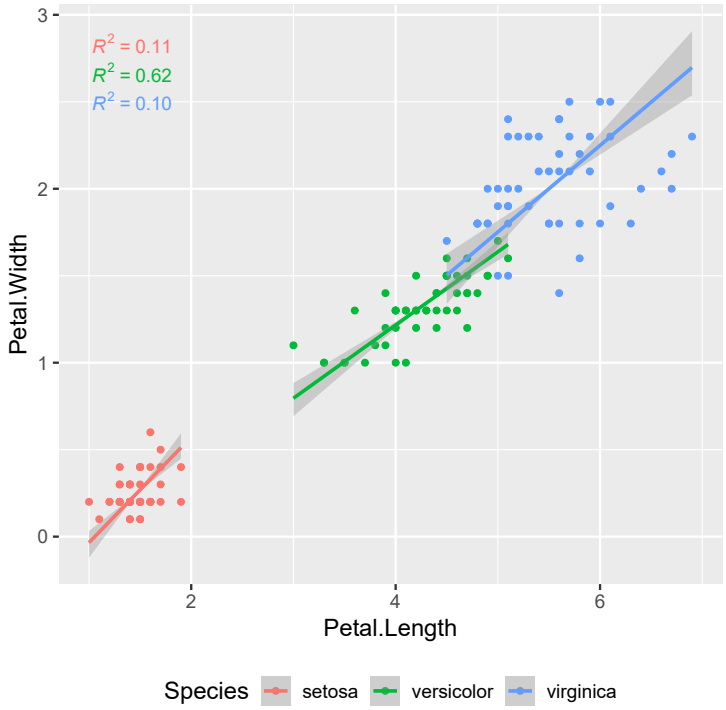
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  stat_poly_line(formula = y ~ x, fullrange = TRUE) +
  stat_poly_line(formula = x ~ y, fullrange = TRUE, colour = "red") +
  stat_ma_line(fullrange = TRUE, method = "RMA",
               range.y = "relative", range.x = "relative", colour = "black") +
  expand_limits(x = c(0, 8), y = 0)
```



 There are three methods available, `"MA"`, `"SMA"` and `"RMA"`. They differ in how the residuals in x and y are weighted. The use of method `"MA"` makes sense when the range and variation in both x and y are similar; standard major axis `"SMA"` regression balances the weights based on ???? while ranged major axis `"RMA"`, balances the weights based on one of two criteria, `relative` or `interval`. Relative cannot be used if any observation has a negative value.

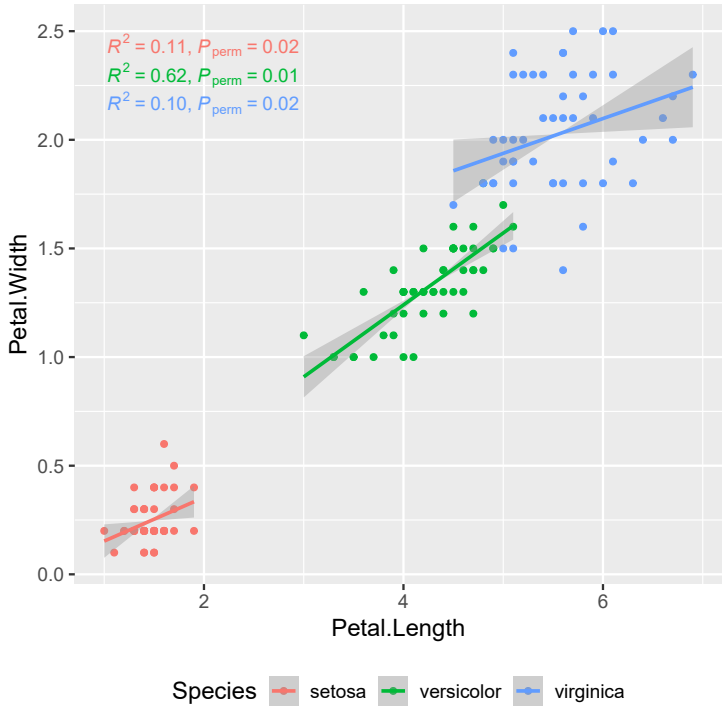
A frequent use of MA regression is to compute the slope in allometric ratios. We will use the `iris` data for the next example. As the width of the petals is about 1/3 their length, MA regression is unsuitable. We use SMA, assuming that the relationship is significant.

```
ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) +
  geom_point() +
  stat_ma_line(method = "SMA") +
  stat_ma_eq(method = "SMA")
```



Method `"OLS"` as implemented in package 'lmodel2' differs from `lm()` in that the *P*-value is estimated by permutation, which makes it more likely that it remains valid.

```
ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) +
  geom_point() +
  stat_ma_line(method = "OLS") +
  stat_ma_eq(aes(label = paste(after_stat(rr.label),
                               after_stat(p.value.label),
                               sep  = "*\", \"*")),
             method = "OLS")
```



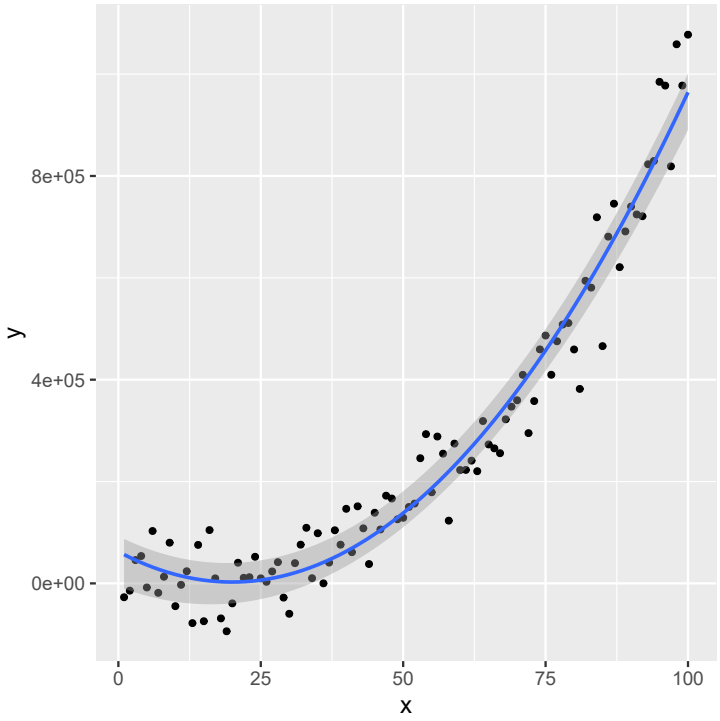
Please, see section B.4 above for examples of the use of `stat_poly_line()` and `stat_poly_eq()`, which follow a similar syntax and can be adapted to `stat_ma_line()` and `stat_ma_eq()`.

B.6 Quantile regression

Median regression differs from OLS linear models in a similar way as the median of a distribution differs from the mean of a distribution. The median corresponds to quantile $\tau = 0.5$. Just as it is possible to compute point estimates for any quantile τ in $0 \leq \tau \leq 1$, quantile regression models can be also fit using τ values in this same range. Three statistics from package 'ggpmisc' implement support for quantile regression: `stat_quant_line()`, `stat_quant_band()` and `stat_quant_eq()`. Package 'ggplot2' provides `stat_quantile()` which, currently, provides less comprehensive support.

Here we use Using `stat_quant_band()` with the default values for quantiles, c(0.25, 0.50, 0.75), equivalent to those used for the box of box plots, and we obtain a plot with a line for median regression and a band highlighting the space between the quartiles. The default geometry is `geom_smooth()` from package 'ggplot2'.

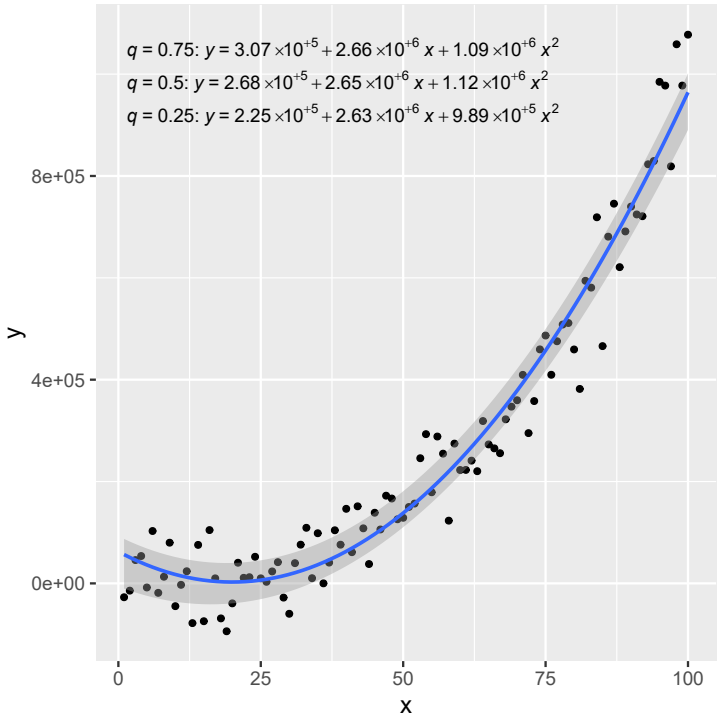
```
ggplot(my.data, aes(x, y)) +
  geom_point() +
  stat_quant_band(formula = y ~ poly(x, 2))
```



 As this is different from a linear regression and its confidence band, one should be careful to avoid misinterpretations. Be clear in the caption and override the default `colour` and `fill` when needed.

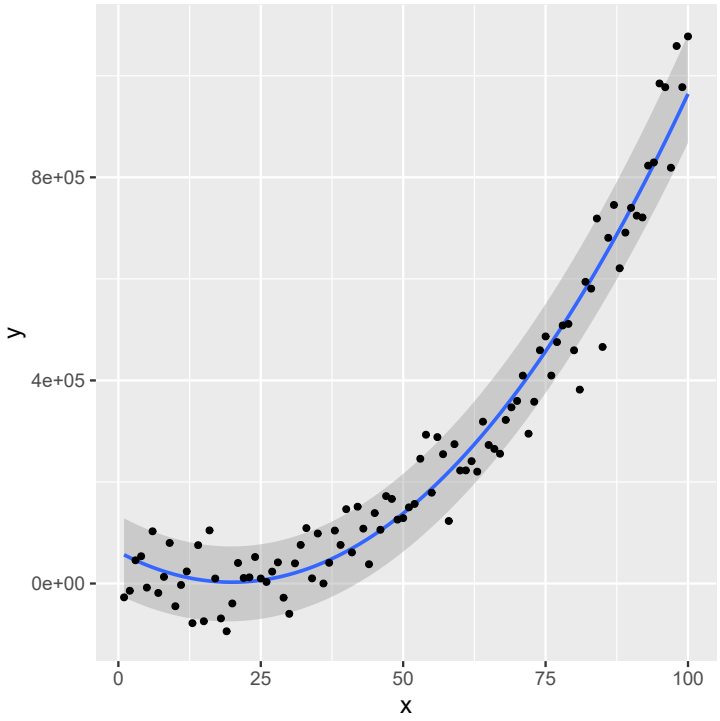
As shown for linear models, we can add the fitted model equations, and label them with the quantile values used.

```
ggplot(my.data, aes(x, y)) +
  geom_point() +
  stat_quant_band(formula = y ~ poly(x, 2)) +
  stat_quant_eq(aes(label = paste(stat(grp.label), "*\": \"*",
                                  stat(eq.label), sep = "")),
                formula = y ~ poly(x, 2))

## Warning in rq.fit.br(x, y, tau = tau, ci = TRUE, ...): Solution may be nonunique
```



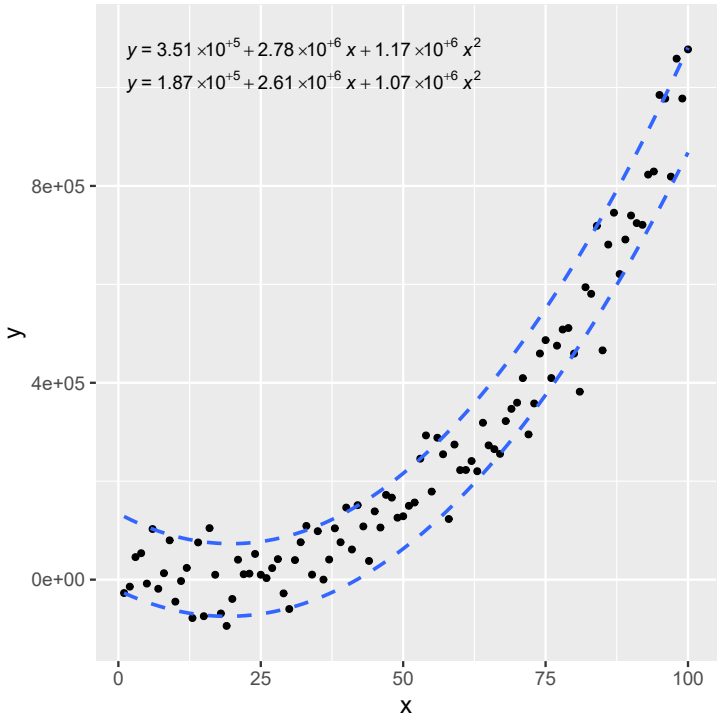
Using `stat_quant_band()` and overriding the default values for quantiles, we find the median plus a boundary containing 90% of the observations.

```
ggplot(my.data, aes(x, y)) +
  stat_quant_band(formula = y ~ poly(x, 2),
                  quantiles = c(0.1, 0.5, 0.9)) +
  geom_point()
```



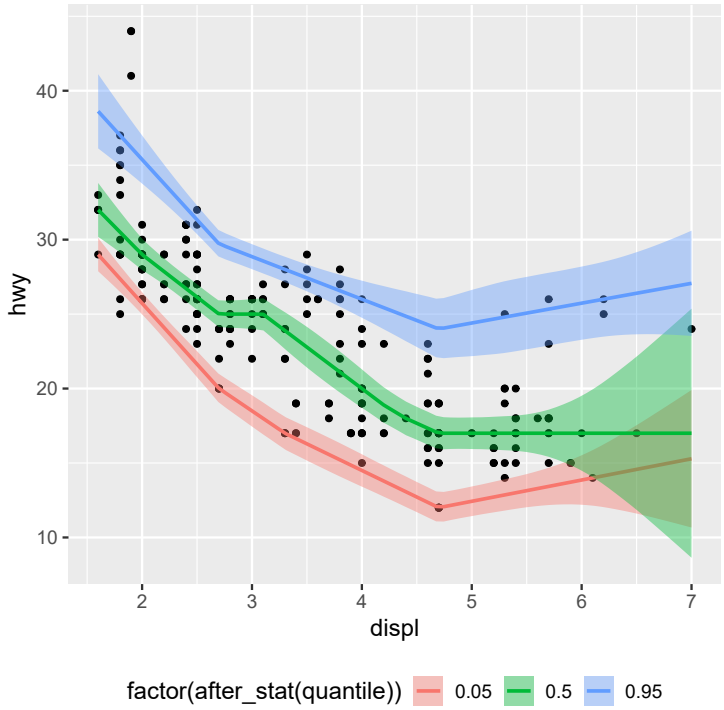
Using `stat_quant_line()` instead of Using `stat_quant_band()` and passing as arguments only two quantiles we can highlight the same boundary with lines, here dashed. Here we also add the fitted model equations.

```
ggplot(my.data, aes(x, y)) +
  geom_point() +
  stat_quant_eq(formula = y ~ poly(x, 2),
                quantiles = c(0.1, 0.9)) +
  stat_quant_line(formula = y ~ poly(x, 2),
                  quantiles = c(0.1, 0.9),
                  linetype = "dashed")

## Warning in rq.fit.br(x, y, tau = tau, ci = TRUE, ...): Solution may be nonunique
```



It is possible to fit additive models with `method = "rqss"`, but being splines the equation is not available.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point()+
  stat_quant_line(method="rqss",
                  interval="confidence",
                  se = TRUE,
                  mapping = aes(fill = factor(after_stat(quantile)),
                                color = factor(after_stat(quantile))),
                  quantiles=c(0.05,0.5,0.95))

## Smoothing formula not specified. Using: y ~ qss(x)
```



Please, see section B.4 on page 434 above for examples of the use of `stat_poly_line()` and `stat_poly_eq()`, which follow a similar syntax and can be adapted to `stat_quant_line()`, `stat_quant_band()` and `stat_quant_eq()`.

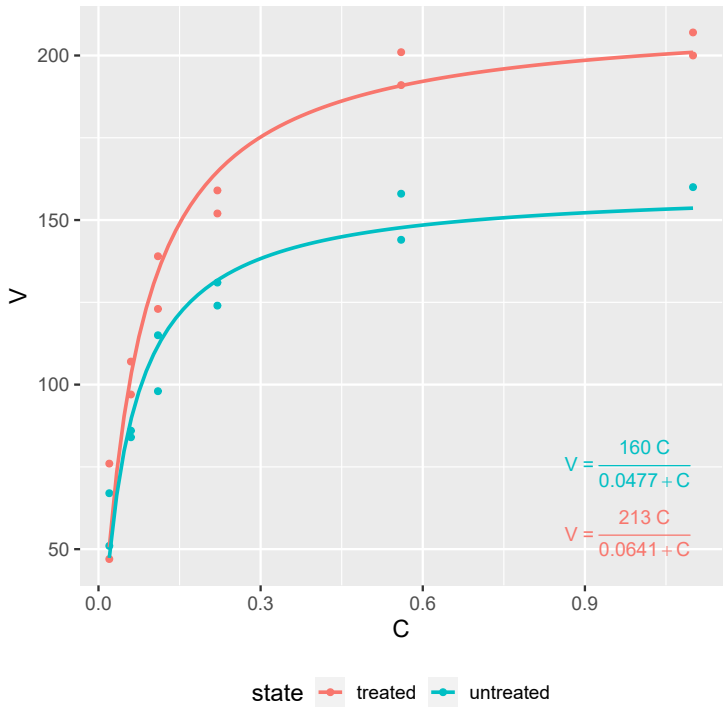
B.7 Non-linear regression

Fitting of models that are non-linear in the parameters is important when such models are grounded on theory. For example the kinetics of chemical reactions is described by the Michaelis-Menthen equation and by fitting this equation we can estimate the kinetic constant of a reaction, i.e, we obtain estimates of parameters that are interpretable according to chemical theory.

It is also possible to fit a non-linear model with method = "nls", and any other model for which a glance() method exists. Do consult the documentation for package 'broom'. Here we fit the Michaelis-Menten equation to reaction rate versus concentration data. This example demonstrates that with slightly more complex code we can fit a non-linear equation and label the plot in a similar way as we did for linear models.

```
micmen.formula <- y ~ SSmicmen(x, Vm, K)
ggplot(Puromycin, aes(conc, rate, colour = state)) +
  geom_point() +
  geom_smooth(method = "nls",
              formula = micmen.formula,
              se = FALSE) +
```

```
  stat_fit_tidy(method = "nls",
                method.args = list(formula = micmen.formula),
                size = 4,
                label.x = "right",
                label.y = "bottom",
                vstep = 0.12,
                  aes(label = paste("V~`=`~frac(", signif(stat(Vm_estimate), dig-
its = 3), "~C,",
                                    signif(stat(K_estimate), digits = 3), "+C)",
                                    sep = "")),
                parse = TRUE) +
  labs(x = "C", y = "V")
```



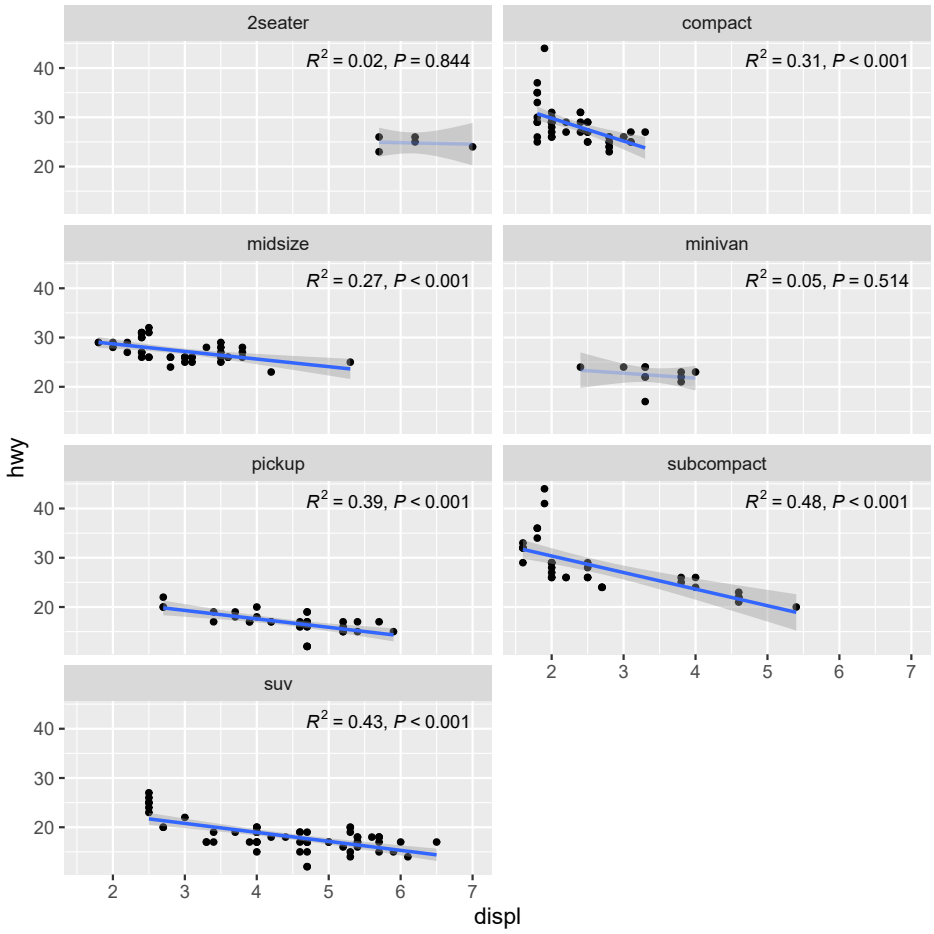
B.8 Conditional display

For the examples in this section we use linear regression fits, but these examples can be easily adjusted to work with other types of model fits, by combining them with the examples shown earlier in this chapter. One frequent situation when dealing with model-fit-derived annotations is the wish to make the display of the fitted model conditional on statistical significance. This can be desirable both in a plot with multiple panels or in code used to plot multiple data sets. We here use plots with panels as they illustrate more clearly the effect of applying the conditions.

One approach, that I find very attractive is to adjust transparency, to make the display of the fitted regression lines only faintly when the *P*-value is above a critical

value. In the examples below we make use of function `stage()`, recently added to 'ggplot2' 2. Here we do not use any grouping within the panels. (Currently these examples trigger what seem to be unimportant warnings.)

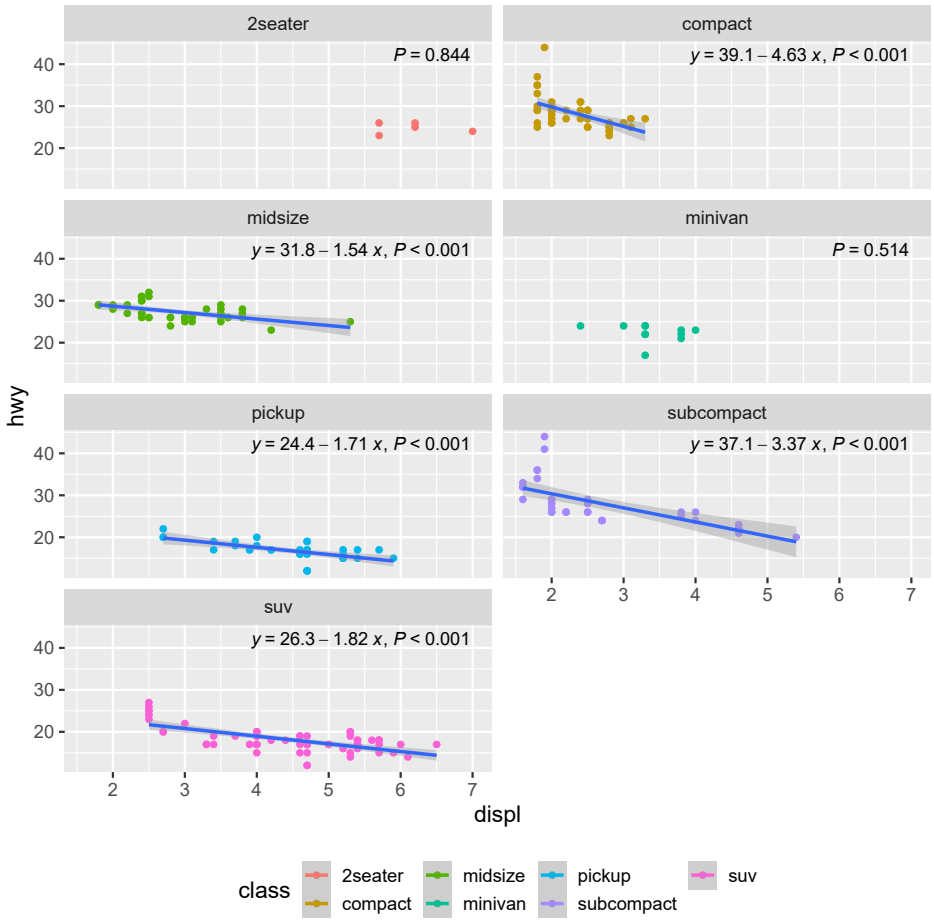
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  stat_poly_eq(aes(label = paste(after_stat(rr.label),
                                 after_stat(p.value.label),
                                 sep = "*\", \"*")),
               label.x = "right") +
  stat_poly_line(aes(colour = stage(after_scale = ifelse(p.value < 0.01,
                                                         alpha(colour, 1),
                                                         alpha(colour, 0.25)))),
                 se = TRUE,
                 mf.values = T) +
  facet_wrap(~class, ncol = 2)
```



We can also hide lines and bands by setting colour and fill to `NA`.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  stat_poly_eq(aes(label = stage(after_stat = ifelse(p.value < 0.01,
                                                     paste(eq.label,
                                                           p.value.label,
```

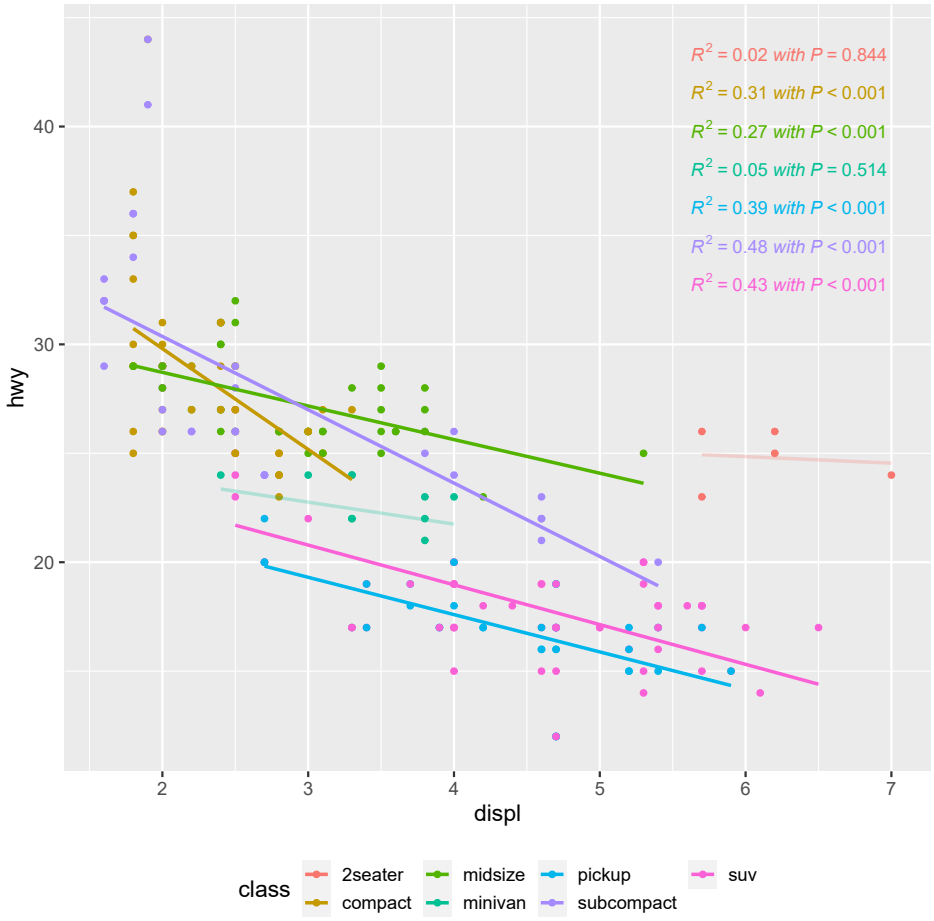
```
                                                      sep = "*\", \"*"),
                                                paste(p.value.label,
                                                      sep = "*\", \"*")))),
                    label.x = "right") +
  stat_poly_line(aes(colour = stage(after_scale = ifelse(p.value < 0.01,
                                                         colour,
                                                         NA)),
                     fill = stage(after_scale = ifelse(p.value < 0.01,
                                                       fill,
                                                       NA))),
                 se = TRUE,
                 mf.values = T) +
  facet_wrap(~class, ncol = 2)

## Warning: Duplicated aesthetics after name standardisation: NA
## Warning: Failed to apply `after_scale()` modifications to legend
```



We can alternatively use grouping by mapping variable `class` to the `colour` aesthetic. In this example, display of both lines and equations is conditional on *P*-value.

```
ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point() +
```

```
  stat_poly_eq(aes(label = paste(after_stat(rr.label),
                                 after_stat(p.value.label),
                                 sep = "*italic(\" with \")*")),
               label.x = "right") +
  stat_poly_line(aes(colour = stage(start  = class,
                                    after_scale = ifelse(p.value < 0.01,
                                                         alpha(colour, 1),
                                                         alpha(colour, 0.25)))),
                 se = FALSE,
                 mf.values = T) +
  theme(legend.position = "bottom")

## Warning: Failed to apply `after_scale()` modifications to legend
```



In these examples when deciding which lines to consider statistically significant we are applying multiple comparisons. In this case it is wise to apply a correction, such as Bonferroni's. This correction can be applied to the threshold by dividing it by the number of model fits (groups or panels in the plot). As a decision criterion, this is equivalent to multiplying the individual *P*-value es-

timates by the same factor. In the examples above, the test $P < 0.01$ ($\alpha = 0.01$) used to display plot elements is equivalent to a Bonferroni-corrected critical value of $\alpha' = 0.01 \times 7 = 0.07$.

```
try(detach(package:broom))
try(detach(package:ggpmisc))
try(detach(package:ggpp))
try(detach(package:ggplot2))
```

Bibliography

Aphalo, P. J. (2020). *Learn R: As a Language*. The R Series. CRC/Taylor & Francis Ltd. 350 pp. ISBN: 036718253X (cit. on p. ix).

Boas, R. P. (1981). "Can we make mathematics intelligible?" In: *The American Mathematical Monthly* 88.10, pp. 727–731.

Burns, P. (1998). *S Poetry* (cit. on p. 401).

— (2011). *The R Inferno*. URL: `http://www.burns-stat.com/pages/Tutor/R_inferno.pdf` (visited on 07/27/2017) (cit. on p. 401).

Chambers, J. M. (2016). *Extending R*. The R Series. Chapman and Hall/CRC. ISBN: 1498775713 (cit. on p. 401).

Cotton, R. J. (2016). *Testing R Code*. The R Series. Chapman and Hall/CRC. ISBN: 1498763650 (cit. on p. 403).

Crawley, M. J. (2012). *The R Book*. Wiley, p. 1076. ISBN: 0470973927 (cit. on p. ix).

Dalgaard, P. (2008). *Introductory Statistics with R*. Springer, p. 380. ISBN: 0387790543 (cit. on p. ix).

Eddelbuettel, D. (2013). *Seamless R and C++ Integration with Rcpp*. Springer, p. 248. ISBN: 1461468671 (cit. on p. 413).

Everitt, B. S. and T. Hothorn (2009). *A Handbook of Statistical Analyses Using R*. 2nd ed. Chapman & Hall, p. 376. ISBN: 1420079336 (cit. on p. ix).

Fox, J. and H. S. Weisberg (2010). *An R Companion to Applied Regression*. SAGE Publications, Inc, p. 472. ISBN: 141297514X (cit. on p. ix).

Holmes, S. and W. Huber (Mar. 1, 2019). *Modern Statistics for Modern Biology*. Cambridge University Press. 382 pp. ISBN: 1108705294 (cit. on pp. ix, 357).

Koponen, J. and J. Hildén (2019). *Data visualization handbook*. Espoo, Finland: Aalto University. ISBN: 9789526074498 (cit. on p. 422).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on pp. 401, 402).

Murrell, P. (2011). *R Graphics*. 2nd ed. CRC Press, p. 546. ISBN: 1439831769 (cit. on p. 431).

Scherer, C. (2019). *A ggplot2 Tutorial for Beautiful Plotting in R*. URL: `https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/` (visited on 07/21/2021) (cit. on p. ix).

Tufte, E. R. (1983). *The Visual Display of Quantitative Information*. Cheshire, CT: Graphics Press. 197 pp. ISBN: 0-9613921-0-X.

Wickham, H. (2015). *R Packages*. O'Reilly Media. ISBN: 9781491910542 (cit. on pp. 401–403).

Wickham, H. (2019). *Advanced R*. 2nd ed. Taylor & Francis Inc. 588 pp. ISBN: 0815384572 (cit. on pp. 401, 402).

Wood, S. N. (2017). *Generalized Additive Models*. Taylor & Francis Inc. 476 pp. ISBN: 1498728332 (cit. on p. ix).

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC. ISBN: 9781138700109 (cit. on p. 416).
— (2018). *Blogdown : creating websites with R Markdown*. Boca Raton: CRC Press, Taylor & Francis, CRC Press. ISBN: 9780815363729 (cit. on p. 416).
Xie, Y., J. J. Allaire, and G. Grolemund (2018). *R Markdown*. Chapman and Hall/CRC. 304 pp. ISBN: 1138359335 (cit. on p. 416).

General index

Index of R names by category

Alphabetic index of R names