Data visualization with ggplot2

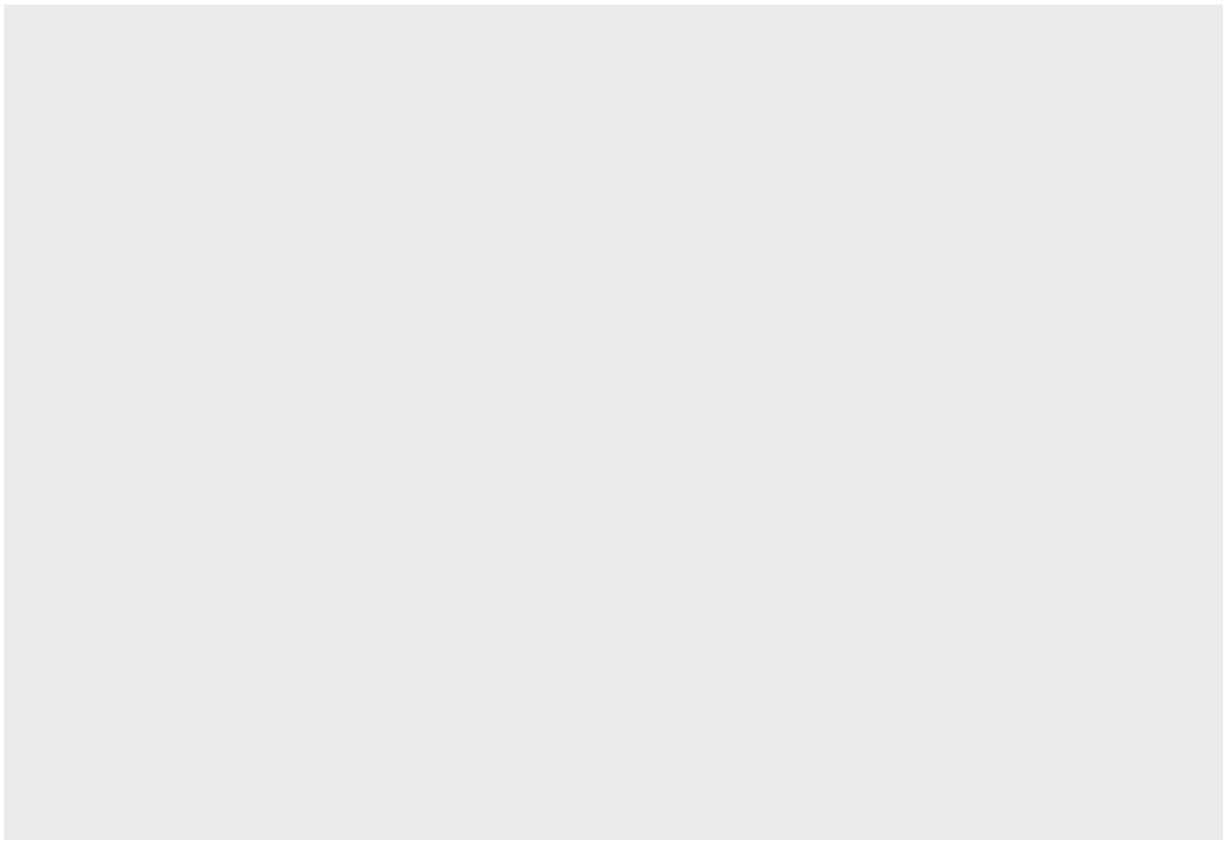
David García-Callejas

There are a variety of options for generating high-quality graphics in R. The base installation already possesses a comprehensive toolset, and other frameworks also exist. `Ggplot2` is arguably the most used visualization package due to its intuitive design, the capabilities it provides to modify virtually every aspect of a graphic through a consistent grammar, the variety of graphics it can produce, and the wide set of complementary packages that provide even more functionality. In this session, we will learn how to produce different types of graphics in R using `ggplot2`. This package is, in addition, part of the `tidyverse`, which makes it totally integrated with `tidyverse` functions and semantics.

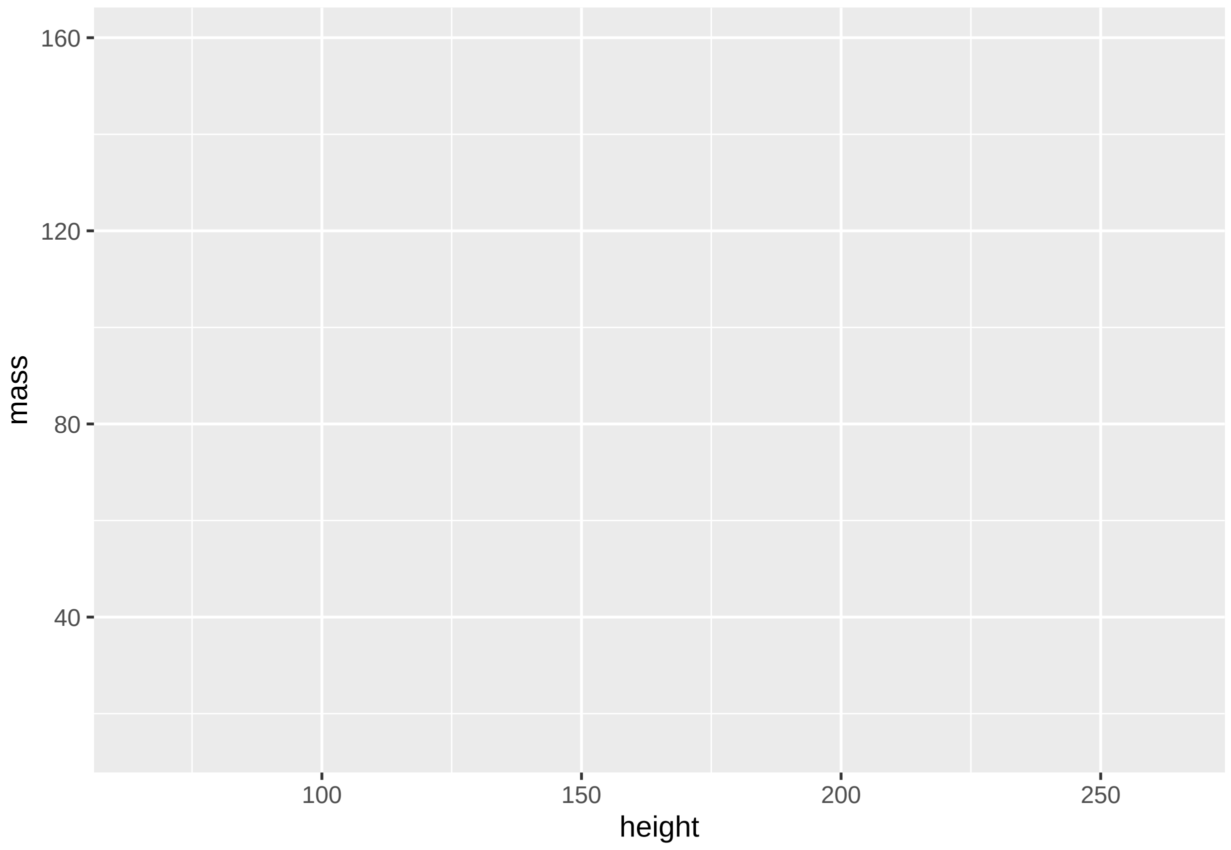
The name of the package is an acronym for "Grammar of Graphics Plots". This so-called "Grammar of Graphics" is a set of grammar rules to structure the design of graphics. The basis of these rules is that graphics are divided in *layers* which contain different visual information. These layers, as we are about to see, are arguably the main idea behind the `ggplot2` framework.

Let's start with a first example, a scatterplot in which we plot the relationship between body mass and height of Star Wars characters.

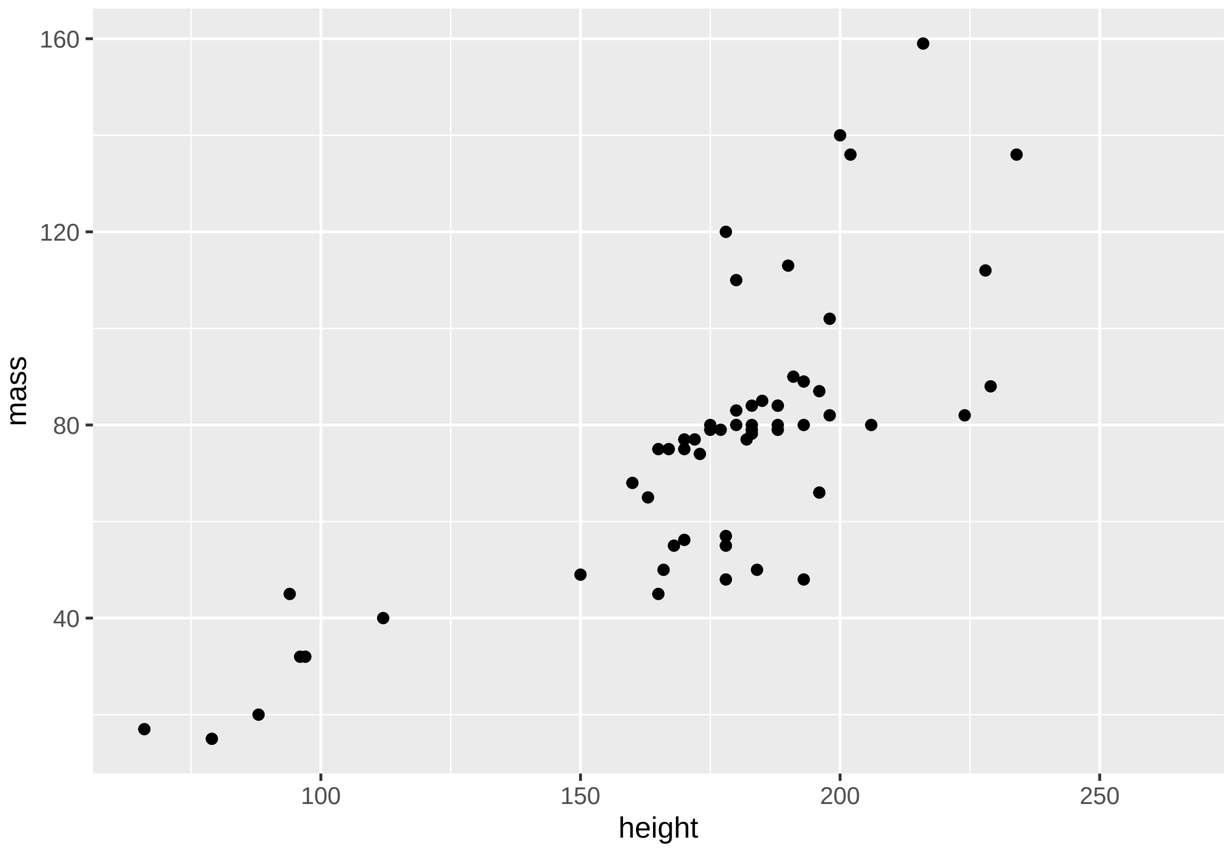
```r
# loading the full tidyverse will also load ggplot2
library(tidyverse)

# read the dataframe
sw <- read.csv2(file = "../data/starwars_info_characters.csv")

# we want to create a figure showing height and weight of every character,
# where each character is a point

# the first thing to understand is that every plot is an object, just like
# a numeric variable, a matrix, or a dataframe. Therefore, we can store
# a plot by assigning it to a variable.
# For creating a plot, we use the function "ggplot". At the very least,
# we need to specify which dataset we will use for this figure

f1 <- ggplot(data = sw)

# what is contained in this object?
f1 # in Rstudio, this will produce a grey square in the plots tab.
```



```
class(f1) # it's an object of class "ggplot"

# at this point, you can think of this object as an empty canvas to which
# we can start adding layers of information
# what would be the first piece of information to add?
# In ggplot, we can start by giving information about the axis of our figure.
# We can do so directly in the ggplot function:
f1 <- ggplot(data = sw,aes(x = height, y = mass))
f1
```



```r
# Look at the syntax. We have specified the x and y axes within the ggplot function,
# and inside another function, aes. Aes is acronym for "aesthetics". Many details
# of ggplot2 figures will be "aesthetic" details that need to be specified within
# the "aes" function.

# Now, you should see a figure with explicit axes, labels, and ticks on it. Over these axes,
# our first layer of information, we will add another layer with the points representing
# the characters in our dataset.
# Layers in ggplot2 are added through functions with a common structure,
# "geom_TYPE", where TYPE is the type of information we want to add (point, line, etc).
# In our case, we go with "geom_point". Since we have already specified the axes of
# the figure, we don't need any further argument. Adding layers in ggplot2 is done
# simply with the addition symbol.

f1 <- f1 + geom_point()
f1
```



Congratulations, you created your first ggplot figure! Now, you will likely want to store this figure in your hard drive. There are at least two main ways to do that. Visually, Rstudio allows to export any graphic from the Plots tab by clicking on the "Export" button. Then, programatically, you can use the function `ggsave`, which automates the process and gives you more control on the output.

```
# ggsave by default uses inches as units
# but font size relative to the figure may vary.
# I usually try different sizes until I am happy with the ratio and size
ggsave(filename = "mi_first_plot.png",plot = f1,width = 5,height = 5)
```

Before we move on to other types of graphics, let's see how we can improve this first figure. A first idea is to colour points according to the species of the characters.

```
# for adding variable colours, we specify it within its geom_ layer.
# As this is an aesthetic detail, we add it inside a call to "aes"
# Note that our data is already in an appropriate format, where each observation
# has a species associated to it.
f2 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species))

# This figure is valid, but there are far too many species for it to be effective.
# Note also how ggplot automatically assigns a colour palette and generates a
# nicely formatted legend.

# How can we improve the figure? we may group some of the species together
# to decrease the number of categories to show.
# First, let's see how many characters of each species we have:
table(sw$species)
```

```r
# Most species are represented by a single character. We will group most of these,
# and leave humans, droids, and of course Yoda, as distinct categories.

# First, we duplicate the species column
sw$species_group <- sw$species
# and we group those that are not "Human", "Droid" or "Yoda's species" in a single category
sw$species_group[which(!sw$species %in% c("Human",
                                          "Droid",
                                          "Yoda's species"))] <- "Other"
table(sw$species_group)

# let's redraw the figure with the new grouping
f2 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group))

# we may assign a fixed color to all points, instead of a variable one.
# You may specify colours by their names
# (e.g. https://www.datanovia.com/en/blog/awesome-list-of-657-r-color-names/)
# or by their hexadecimal code.
f2.2 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(color = "darkgreen")
```

By including "colour" or any other argument within an `aes` call, we are effectively making that argument vary with respect to another column, "species_group" in our example. On the other hand, directly specifying a value inside `geom_point` makes all points take that value. Common attributes to modify include, in addition to colour, shape, size, or fill.

```r
f2.3 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(color = "darkgreen", size = 3, aes(shape = species_group))
```

Point shapes in R are given by a numeric code, that you can check here: http://www.cookbook-r.com/Graphs/Shapes_and_line_types/

In plotting the figure, ggplot automatically discards NA values. This is why when running this code you will se a warning in the console about rows with NA values in them.

Now, let's see a few examples of other types of plots we can create:

```r
# bar plots: how many individuals per group of species?
# for this figure we do not need to specify the vertical axes, as it corresponds
# not to any column, but simply to the count of each group.
# we specify that with the argument "stat = "count"" within geom_bar
f3 <- ggplot(sw, aes(x = species_group)) +
  geom_bar(stat="count")

# as before, we can colour each group.
# To colour whole areas, we use the argument "fill".
f3 <- ggplot(sw, aes(x = species_group)) +
  geom_bar(stat="count", aes(fill = species_group))

# boxplot, specifying both axes
f4 <- ggplot(sw, aes(x = species_group, y = height)) +
  geom_boxplot(aes(fill = species_group))

# distribution of points
```

```r
f5 <- ggplot(sw, aes(x = species_group, y = height)) +
  geom_jitter(aes(color = species_group))

# points and a boxplot
f6 <- ggplot(sw, aes(x = species_group, y = height)) +
  geom_jitter(aes(color = species_group), alpha = .8) +
  geom_boxplot(aes(fill = species_group), alpha = .5)

# histogram - again, no need to specify vertical axes
f7 <- ggplot(sw, aes(x = height)) +
  geom_histogram(bins = 30)

# density distribution
f8 <- ggplot(sw, aes(x = height)) +
  geom_density()
```

We can also add regression lines to scatterplots very easily, using the `geom_smooth` function. This serves as another example of building a complete figure by stacking layers of information.

```r
# By default, geom_smooth fits a loess regression
f9 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth()

# bu we can specify different models
f10 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm")

# we can plot one regression line per group
f11 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm", aes(color = species_group))

# and tweak many other details
f12 <- ggplot(sw, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm",
              aes(color = species_group),se = FALSE)
```

Now it makes sense -hopefully- why ggplot, and the tidyverse in general, prefer data in long format. Here, categorical variables (groups) are organized in columns, instead of distributed across rows. This makes it straightforward to assign grouping variables to aesthetics.

```r
head(sw)

sw.example <- sw[,c("name","height","mass","species_group")]
sw.example$val <- TRUE

sw.example.wide <- pivot_wider(data = sw.example,
                               id_cols = c(name,height,mass),
                               names_from = species_group,
                               values_from = val,
                               values_fill = list(val = FALSE))
head(sw.example.wide)
```

By dividing species across columns, it would be impossible to assign that category: what column would be use to group observations?

```
sw.wide.plot <- ggplot(sw.example.wide, aes(x = height, y = mass)) +
  geom_point(aes(color = Human)) # this only differentiates humans from other categories
```

To create line plots we will use the earthquake data. Let's plot the number of observed earthquakes per century. This example includes the data wrangling previous to the plotting itself, but the compactness of the tidyverse idioms makes it a breeze.

```
eq <- read.csv2(file = "../data/earthquakes.csv")

head(eq)

# we need to create a "century" column, assign the century of each observation,
# and we will also convert it to roman numerals
eq.century <- eq %>%
  mutate(century = as.character(as.roman(as.numeric(substr(eq$Year,1,2)) + 1))) %>%
  group_by(century) %>%
  summarise(eq.count = n())

# we want the centuries to appear sorted, not alphabetically,
# so we make it a factor and specify the ordering
eq.century$century <- factor(eq.century$century,levels = c("XI",
                                                           "XII","XIII","XIV",
                                                           "XV","XVI","XVII",
                                                           "XVIII","XIX","XX"))
eq.century
```

Now that we have the data ready, let's plot it

```
# R, and ggplot, have some quirks. When plotting lines, we often will need
# to specify another argument, the grouping of the points to be connected.
# This is better explained in the ggplot2 cookbook (link below):
# "For line graphs, the data points must be grouped so that it knows
# which points to connect.
# In this case, it is simple - all points should be connected, so group=1".
f13 <- ggplot(eq.century, aes(x = century,
                              y = eq.count, group = 1)) +
  geom_line()
```

Just as we did with points, we can easily specify different line parameters (again, check http://www.cookbook-r.com/Graphs/Shapes_and_line_types/)

```
f14 <- ggplot(eq.century, aes(x = century,
                              y = eq.count,
                              group = 1)) +
  geom_line(linetype = "dashed", size = 3, color = "darkred")
```

Instead of grouping observations by colour, we may need a different panel for each data category. Going back to the Star Wars example:

```
f15 <- ggplot(sw, aes(x = height, y = mass))+
  geom_point() +
  geom_smooth(method = "lm") +
  facet_grid(species_group ~ .)
```

The `facet_grid` layer divides our figure according to the specified columns, both vertically and horizontally.

The syntax is “vertical_grouping ~ horizontal_grouping”. If we just want a single axis of division, as in the example above, we use a dot for the other axis.

```
f16 <- ggplot(sw, aes(x = height, y = mass))+
  geom_point() +
  facet_grid(gender~species_group)
```

These divisions can of course be combied with other attributes, such as colours, fills, etc.

```
f17 <- ggplot(sw, aes(x = height, y = mass))+
  geom_point(aes(color = species_group, shape = gender)) +
  facet_grid(gender~species_group)
```

In this last example we see that, for each grouping attribute in `aes`, a new legend is created. You will have noticed that here, the colour legend is reiterative with the horizontal grouping, whereas the gender legend is reiterative with the vertical grouping. We are going to take this example as a template to improve a bit the figure.

```
# first, let's get rid of NAs
sw2 <- subset(sw, !is.na(height) & !is.na(mass) & !is.na(gender))

f18 <- ggplot(sw2, aes(x = height, y = mass))+
  geom_point(aes(color = species_group, shape = gender), size = 2) +
  facet_grid(gender ~ species_group) +
  # a new layer for providing title and subtitle
  ggtitle(label = "Height and mass of Star Wars characters",
          subtitle = "across species and males/females") +
  # another layer specifying axis labels
  labs(x = "height (cm)", y = "mass (kg)") +
  # legends
  # colour
  scale_color_manual(name = "Species type",
                     labels = c("droid","human", "other", "Yoda"),
                     values = c("grey30","darkorange",
                                "darkred","darkgreen")) +
  # shape
  scale_shape_manual(name = "gender",
                     labels = c("female", "male", "other/none"),
                     values = c(1,5,6))
```

The layers that control attributes are given by calls of the form `scale_attribute_type`. In this example, we modify both `color` and `shape` attributes. Since we specify manually the values we want for these attributes, we need to embed those in a call to `scale_colour_manual` and to `scale_shape_manual`.

Many more options can be tweaked, as you would expect. The general appearance of the figures (background/foreground colours, grid rectanges, text size/color/font, etc) are bundled in what ggplot2 calls “themes”. Ggplot2 comes with a series of pre-loaded themes, that can be directly applied (you can check the looks of the default themes in https://ggplot2.tidyverse.org/reference/ggtheme.html)

```
f19 <- f18 +
  theme_bw()
```

In addition to dividing any plot across categories, we can append independent plots in a single figure. For doing so, we use the `patchwork` package:

```
#install.packages("patchwork") install and load the package if you have not done so
library(patchwork)
```

```
f.combined <- f18 + f19
```

Lastly, we are going to scratch the surface of the possibilities that ggplot2 offers to draw maps and areas. The package includes a function `map_data` that serves as a wrapper of the `maps` package, and loads the specified area in a format usable with ggplot.

```
# we need to load the "maps" package
library(maps)

# world emerged land boundaries
world_map <- map_data("world")
```

The `map_data` function can load any map provided by the `maps` package (see https://cran.r-project.org/web/packages/maps/maps.pdf). One may, for example, specify a given region of the world:

```
sp_map <- map_data("world",region = "Spain")
```

These ggplot2-friendly contours are then simply drawn as polygons, using the `geom_polygon` function:

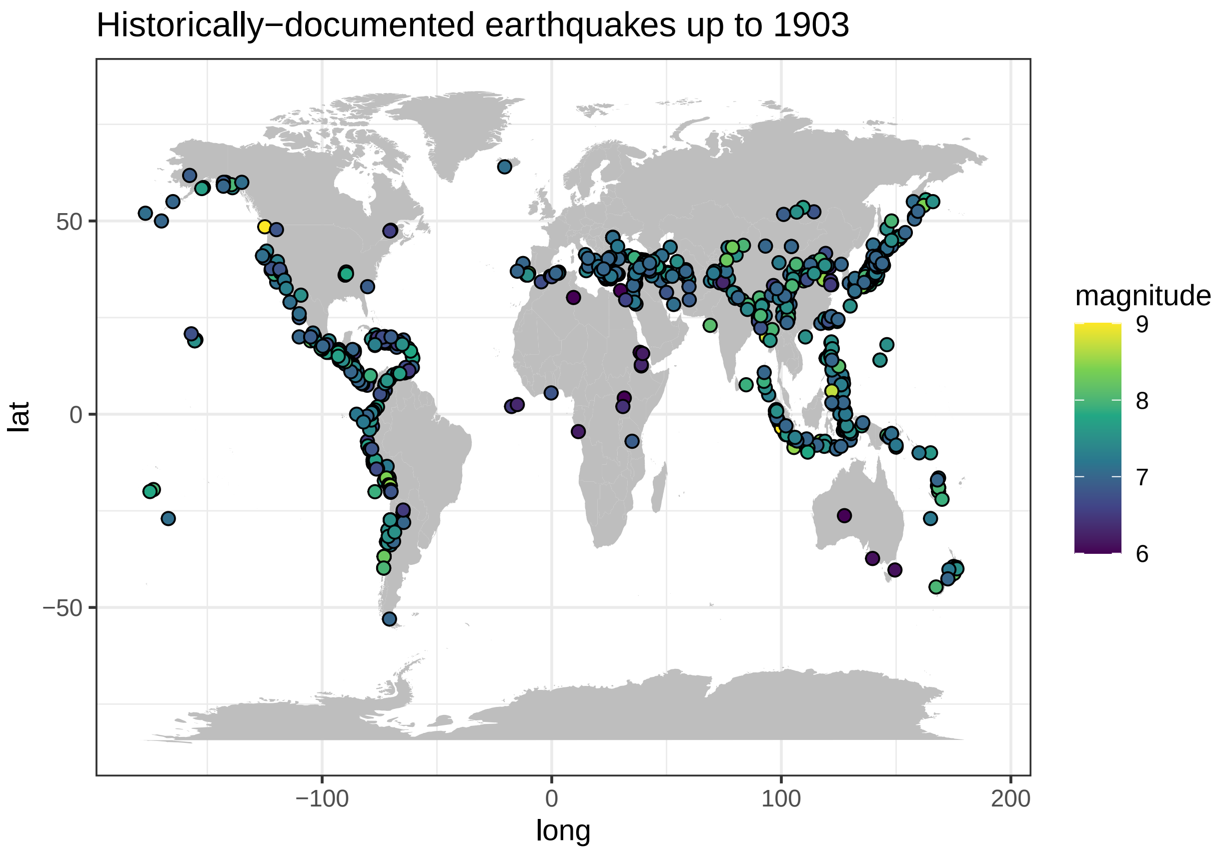
```
# lat/lon for the axes!
world.plot <- ggplot(world_map, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill="gray") +
  theme_bw()

sp.plot <- ggplot(sp_map, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill="gray") +
  theme_bw()
```

On top of these polygons (or any other you plug in, see example below) you can add any type of spatial information, be it other polygons, lines, points… in this example, we will plot the spatial coordinates of all the earthquakes included in our table.

```
# we need two dataframes. First, the world_map polygon data;
# second, the coordinates of each earthquake.
# We can "feed" ggplot with different datasets by moving the data argument
# from the ggplot function to the "geom_" idioms.

# In this example, projections match, so no need to modify them.
eq.map <- ggplot() +
  geom_polygon(data = world_map,aes(x = long,
                                    y = lat,
                                    group = group),
               fill = "gray") +
  geom_point(data = eq, aes(x = Lon, # careful with column names
                            y = Lat),
             color = "darkred", size = 1.2) +
  theme_bw()
```

Can we improve this map? Let's try a few additions. In particular, we will colour-code the magnitude of each earthquake, and then add a number of additional details.

```
eq.map2 <- ggplot() +
  # map
  geom_polygon(data = world_map,aes(x = long, y = lat, group = group),
               fill = "gray") +
  # points
  geom_point(data = eq, aes(x = Lon, y = Lat, fill = M),
             shape = 21, size = 2) +
  # colour legend
  scale_fill_continuous(name = "magnitude", type = "viridis") +
  # title
  ggtitle("Historically-documented earthquakes up to 1903")+
  # theme
  theme_bw()

eq.map2
```

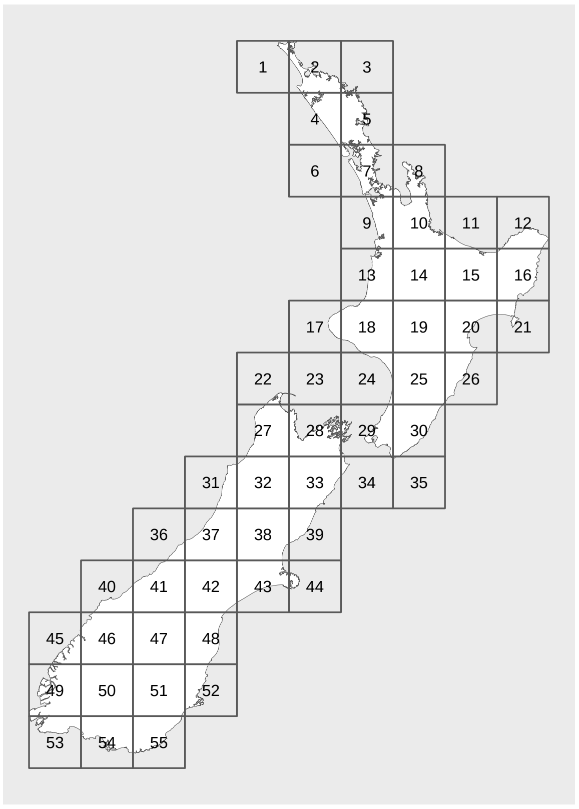


In this figure we used the "viridis" scale to fill points (remember, we have continuous values, and we want to specify values for the fill attribute, so we need `scale_fill_continuous`).

For the last example we will take an external shapefile, load it into R, divide it in a rectangular grid of arbitrary size, assign ids to each cell of the new grid, and plot the result. To load and work with spatial data, we will need the `sf` and `stars` packages.

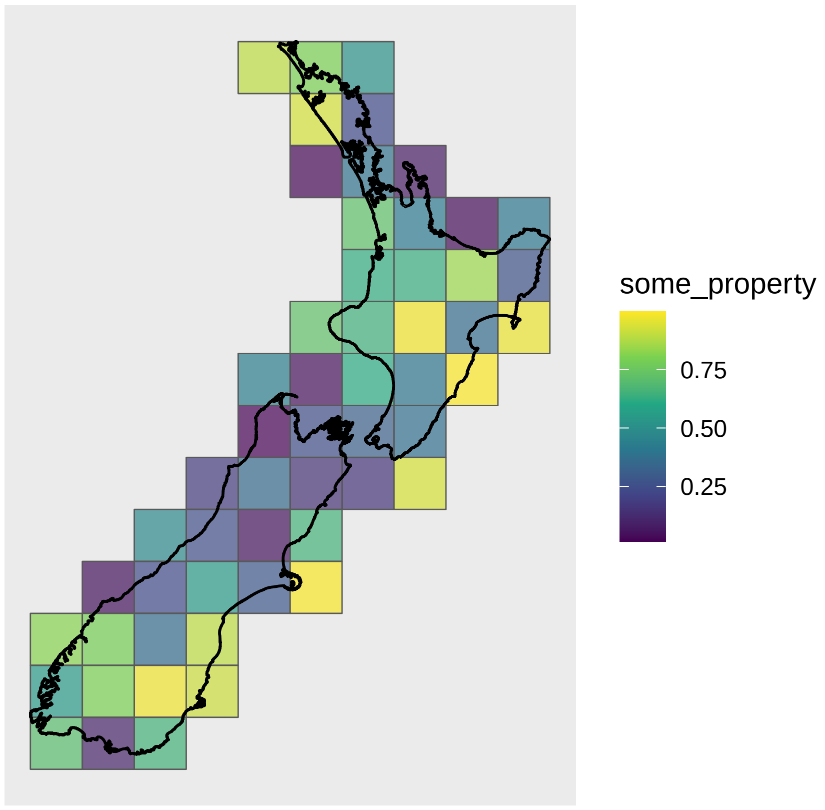
```
library(sf)
library(stars)

NZ <- st_read('../data/NZ_main_islands.shp')

# transform to NZTM2000 projection (https://epsg.io/2193)
NZ2 <- st_transform(NZ, crs= st_crs(2193))

# specify size of the grid cells
grid.size <- 100000 # 100km
```

```r
# generate spatial grid
grid <- st_as_stars(st_bbox(NZ2), dx = grid.size, dy = grid.size)
grid <- st_as_sf(grid)
grid <- grid[NZ2,]
grid$cell_id <- 1:nrow(grid)

# add x-y coords of the centroid
grid_with_labels <- st_centroid(grid) %>% cbind(st_coordinates(.))

# sf objects can be drawn on the fly
# plot(st_geometry(grid), axes = TRUE, reset = FALSE)
# plot(st_geometry(NZ2), border = "darkred", add = TRUE)

# and, of course, with ggplot
nz.grid.plot <-  ggplot() +
  geom_sf(data = NZ2, fill = 'white', lwd = 0.05) + # lwd controls borders
  geom_sf(data = grid, fill = 'transparent', lwd = 0.3) +
  geom_text(data = grid_with_labels,
            aes(x = X, y = Y, label = cell_id),
            size = 2) +
  coord_sf(datum = NA)  +
  labs(x = "") +
  labs(y = "")
nz.grid.plot
```



Now, let's add some random values to the grid, for the sake of the example. We will reverse the order of the layers to not lose the contour of the original shapefile.

```
grid$some_property <- runif(nrow(grid))

nz.grid.plot.2 <- ggplot() +
  geom_sf(data = grid, aes(fill = some_property), alpha = .7, lwd = .25) +
  geom_sf(data = NZ2, fill = 'transparent', color = "black") +
  coord_sf(datum = NA)  + # comment this for showing lat/lon ticks
  scale_fill_continuous(type = "viridis")+
  labs(x = "") +
  labs(y = "")
nz.grid.plot.2
```


some_property
0.75
0.50
0.25