**Beautiful maps in a beautiful world**

Maps are used in a variety of fields to express data in an appealing and  
interpretive way. Data can be expressed into simplified patterns, and  
this data interpretation is generally lost if the data is only seen  
through a spread sheet. Maps can add vital context by incorporating many  
variables into an easy to read and applicable context. Maps are also  
very important in the information world because they can quickly allow  
the public to gain better insight so that they can stay informed. It’s  
critical to have maps be effective, which means creating maps that can  
be easily understood by a given audience. For instance, maps that need  
to be understood by children would be very different from maps intended  
to be shown to geographers.

Knowing what elements are required to enhance your data is key into  
making effective maps. Basic elements of a map that should be considered  
are polygon, points, lines, and text. Polygons, on a map, are closed  
shapes such as country borders. Lines are considered to be linear shapes  
that are not filled with any aspect, such as highways, streams, or  
roads. Finally, points are used to specify specific positions, such as  
city or landmark locations. With that in mind, one need to think about  
what elements are required in the map to really make an impact, and  
convey the information for the intended audience. Layout and formatting  
are the second critical aspect to enhance data visually. The arrangement  
of these map elements and how they will be drawn can be adjusted to make  
a maximum impact.

**A solution using R and its ecosystem of packages**

Current solutions for creating maps usually involves GIS software, such  
as ArcGIS, QGIS, eSpatial, etc., which allow to visually prepare a map,  
in the same approach as one would prepare a poster or a document layout.  
On the other hand, R, a free and open-source software development  
environment (IDE) that is used for computing statistical data and  
graphic in a programmable language, has developed advanced spatial  
capabilities over the years, and can be used to draw maps  
programmatically.

R is a powerful and flexible tool. R can be used from calculating data  
sets to creating graphs and maps with the same data set. R is also free,  
which makes it easily accessible to anyone. Some other advantages of  
using R is that it has an interactive language, data structures,  
graphics availability, a developed community, and the advantage of  
adding more functionalities through an entire ecosystem of packages. R  
is a scriptable language that allows the user to write out a code in  
which it will execute the commands specified.

Using R to create maps brings these benefits to mapping. Elements of a  
map can be added or removed with ease — R code can be tweaked to make  
major enhancements with a stroke of a key. It is also easy to reproduce  
the same maps for different data sets. It is important to be able to  
script the elements of a map, so that it can be re-used and interpreted  
by any user. In essence, comparing typical GIS software and R for  
drawing maps is similar to comparing word processing software (e.g.  
Microsoft Office or LibreOffice) and a programmatic typesetting system  
such as LaTeX, in that typical GIS software implement a WYSIWIG approach  
(“What You See Is What You Get”), while R implements a WYSIWYM approach  
(“What You See Is What You Mean”).

The package ggplot2 implements the grammar of graphics in R, as a way  
to create code that make sense to the user: The grammar of graphics is a  
term used to breaks up graphs into semantic components, such as  
geometries and layers. Practically speaking, it allows (and forces!) the  
user to focus on graph elements at a higher level of abstraction, and  
how the data must be structured to achieve the expected outcome. While  
ggplot2 is becoming the de facto standard for R graphs, it does not  
handle spatial data specifically. The current state-of-the-art of  
spatial objects in R relies on Spatial classes defined in the package  
[sp](https://cran.r-project.org/package=sp), but the new package  
[sf](https://cran.r-project.org/package=sf) has recently implemented  
the “simple feature” standard, and is steadily taking over sp.  
Recently, the package ggplot2 has allowed the use of simple features  
from the package sf as layers in a graph[1](https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html#fn:1). The combination of  
ggplot2 and sf therefore enables to programmatically create maps,  
using the grammar of graphics, just as informative or visually appealing  
as traditional GIS software.

**Drawing beautiful maps programmatically with R, sf and ggplot2**

In this part, we will cover the fundamentals of mapping using ggplot2  
associated to sf, and presents the basics elements and parameters we  
can play with to prepare a map.

**Getting started**

Many R packages are available from [CRAN](https://cran.r-project.org/),  
the Comprehensive R Archive Network, which is the primary repository of  
R packages. The full list of packages necessary for this series of  
tutorials can be installed with:

install.packages(c("cowplot", "googleway", "ggplot2", "ggrepel",

"ggspatial", "libwgeom", "sf", "rworldmap", "rworldxtra"))

We start by loading the basic packages necessary for all maps, i.e.  
ggplot2 and sf. We also suggest to use the classic dark-on-light  
theme for ggplot2 (theme\_bw), which is appropriate for maps:

library("ggplot2")

theme\_set(theme\_bw())

library("sf")

The package rworldmap provides a map of countries of the entire world;  
a map with higher resolution is available in the package rworldxtra.  
We use the function getMap to extract the world map (the resolution  
can be set to "low", if preferred):

library("rworldmap")

library("rworldxtra")

world <- getMap(resolution = "high")

class(world)

## [1] "SpatialPolygonsDataFrame"

## attr(,"package")

## [1] "sp"

The world map is available as a SpatialPolygonsDataFrame from the  
package sp; we thus convert it to a simple feature using st\_as\_sf  
from package sf:

world <- st\_as\_sf(world)

class(world)

## [1] "sf" "data.frame"

**General concepts illustrated with the world map**

**Data and basic plot (ggplot and geom\_sf)**

First, let us start with creating a base map of the world using  
ggplot2. This base map will then be extended with different map  
elements, as well as zoomed in to an area of interest. We can check that  
the world map was properly retrieved and converted into an sf object,  
and plot it with ggplot2:

ggplot(data = world) +

geom\_sf()



This call nicely introduces the structure of a ggplot call: The first  
part ggplot(data = world) initiates the ggplot graph, and indicates  
that the main data is stored in the world object. The line ends up  
with a + sign, which indicates that the call is not complete yet, and  
each subsequent line correspond to another layer or scale. In this case,  
we use the geom\_sf function, which simply adds a geometry stored in a  
sf object. By default, all geometry functions use the main data  
defined in ggplot(), but we will see later how to provide additional  
data.

Note that layers are added one at a time in a ggplot call, so the  
order of each layer is very important. All data will have to be in an  
sf format to be used by ggplot2; data in other formats (e.g. classes  
from sp) will be manually converted to sf classes if necessary.

**Title, subtitle, and axis labels (ggtitle, xlab, ylab)**

A title and a subtitle can be added to the map using the function  
ggtitle, passing any valid character string (e.g. with quotation  
marks) as arguments. Axis names are absent by default on a map, but can  
be changed to something more suitable (e.g. “Longitude” and “Latitude”),  
depending on the map:

ggplot(data = world) +

geom\_sf() +

xlab("Longitude") + ylab("Latitude") +

ggtitle("World map", subtitle = paste0("(", length(unique(world$NAME)), " countries)"))



**Map color (geom\_sf)**

In many ways, sf geometries are no different than regular geometries,  
and can be displayed with the same level of control on their attributes.  
Here is an example with the polygons of the countries filled with a  
green color (argument fill), using black for the outline of the  
countries (argument color):

ggplot(data = world) +

geom\_sf(color = "black", fill = "lightgreen")



The package ggplot2 allows the use of more complex color schemes, such  
as a gradient on one variable of the data. Here is another example that  
shows the population of each country. In this example, we use the  
“viridis” colorblind-friendly palette for the color gradient (with  
option = "plasma" for the plasma variant), using the square root of  
the population (which is stored in the variable POP\_EST of the world  
object):

ggplot(data = world) +

geom\_sf(aes(fill = POP\_EST)) +

scale\_fill\_viridis\_c(option = "plasma", trans = "sqrt")



**Projection and extent (coord\_sf)**

The function coord\_sf allows to deal with the coordinate system, which  
includes both projection and extent of the map. By default, the map will  
use the coordinate system of the first layer that defines one (i.e.  
scanned in the order provided), or if none, fall back on WGS84  
(latitude/longitude, the reference system used in GPS). Using the  
argument crs, it is possible to override this setting, and project on  
the fly to any projection. This can be achieved using any valid PROJ4  
string (here, the European-centric ETRS89 Lambert Azimuthal Equal-Area  
projection):

ggplot(data = world) +

geom\_sf() +

coord\_sf(crs = "+proj=laea +lat\_0=52 +lon\_0=10 +x\_0=4321000 +y\_0=3210000 +ellps=GRS80 +units=m +no\_defs ")



Spatial Reference System Identifier (SRID) or an European Petroleum  
Survey Group (EPSG) code are available for the projection of interest,  
they can be used directly instead of the full PROJ4 string. The two  
following calls are equivalent for the ETRS89 Lambert Azimuthal  
Equal-Area projection, which is EPSG code 3035:

ggplot(data = world) +

geom\_sf() +

coord\_sf(crs = "+init=epsg:3035")

ggplot(data = world) +

geom\_sf() +

coord\_sf(crs = st\_crs(3035))

The extent of the map can also be set in coord\_sf, in practice  
allowing to “zoom” in the area of interest, provided by limits on the  
x-axis (xlim), and on the y-axis (ylim). Note that the limits are  
automatically expanded by a fraction to ensure that data and axes don’t  
overlap; it can also be turned off to exactly match the limits provided  
with expand = FALSE:

ggplot(data = world) +

geom\_sf() +

coord\_sf(xlim = c(-102.15, -74.12), ylim = c(7.65, 33.97), expand = FALSE)



**Scale bar and North arrow (package ggspatial)**

Several packages are available to create a scale bar on a map (e.g.  
prettymapr, vcd, ggsn, or legendMap). We introduce here the  
package ggspatial, which provides easy-to-use functions…

scale\_bar that allows to add simultaneously the north symbol and a  
scale bar into the ggplot map. Five arguments need to be set manually:  
lon, lat, distance\_lon, distance\_lat, and distance\_legend. The  
location of the scale bar has to be specified in longitude/latitude in  
the lon and lat arguments. The shaded distance inside the scale bar  
is controlled by the distance\_lon argument. while its width is  
determined by distance\_lat. Additionally, it is possible to change the  
font size for the legend of the scale bar (argument legend\_size, which  
defaults to 3). The North arrow behind the “N” north symbol can also be  
adjusted for its length (arrow\_length), its distance to the scale  
(arrow\_distance), or the size the N north symbol itself  
(arrow\_north\_size, which defaults to 6). Note that all distances  
(distance\_lon, distance\_lat, distance\_legend, arrow\_length,  
arrow\_distance) are set to "km" by default in distance\_unit; they  
can also be set to nautical miles with “nm”, or miles with “mi”.

library("ggspatial")

ggplot(data = world) +

geom\_sf() +

annotation\_scale(location = "bl", width\_hint = 0.5) +

annotation\_north\_arrow(location = "bl", which\_north = "true",

pad\_x = unit(0.75, "in"), pad\_y = unit(0.5, "in"),

style = north\_arrow\_fancy\_orienteering) +

coord\_sf(xlim = c(-102.15, -74.12), ylim = c(7.65, 33.97))

## Scale on map varies by more than 10%, scale bar may be inaccurate



Note the warning of the inaccurate scale bar: since the map use  
unprojected data in longitude/latitude (WGS84) on an equidistant  
cylindrical projection (all meridians being parallel), length in  
(kilo)meters on the map directly depends mathematically on the degree of  
latitude. Plots of small regions or projected data will often allow for  
more accurate scale bars.

**Country names and other names (geom\_text and annotate)**

The world data set already contains country names and the coordinates  
of the centroid of each country (among more information). We can use  
this information to plot country names, using world as a regular  
data.frame in ggplot2. We first check the country name information:

head(world[, c("NAME", "LON", "LAT")])

## Simple feature collection with 6 features and 3 fields

## geometry type: MULTIPOLYGON

## dimension: XY

## bbox: xmin: -70.06164 ymin: -18.0314 xmax: 74.89231 ymax: 60.48075

## epsg (SRID): 4326

## proj4string: +proj=longlat +datum=WGS84 +no\_defs

## NAME LON LAT geometry

## 1 Aruba -69.97345 12.51678 MULTIPOLYGON (((-69.87609 1...

## 2 Afghanistan 66.00845 33.83627 MULTIPOLYGON (((71.02458 38...

## 3 Angola 17.56405 -12.32934 MULTIPOLYGON (((13.98233 -5...

## 4 Anguilla -63.05667 18.22432 MULTIPOLYGON (((-63.0369 18...

## 5 Albania 20.05399 41.14258 MULTIPOLYGON (((20.06496 42...

## 6 Aland 19.94429 60.22851 MULTIPOLYGON (((19.91892 60...

The function geom\_text can be used to add a layer of text to a map  
using geographic coordinates. The function requires the data needed to  
enter the country names, which is the same data as the world map. Again,  
we have a very flexible control to adjust the text at will on many  
aspects:

* The size (argument size);
* The alignment, which is centered by default on the coordinates  
  provided. The text can be adjusted horizontally or vertically using  
  the arguments hjust and vjust, which can either be a number  
  between 0 (right/bottom) and 1 (top/left) or a character (“left”,  
  “middle”, “right”, “bottom”, “center”, “top”). The text can also be  
  offset horizontally or vertically with the argument nudge\_x and  
  nudge\_y;
* The font of the text, for instance its color (argument color) or  
  the type of font (fontface);
* The overlap of labels, using the argument check\_overlap, which  
  removes overlapping text. Alternatively, when there is a lot of  
  overlapping labels, the package ggrepel provides a  
  geom\_text\_repel function that moves label around so that they do  
  not overlap.

Additionally, the annotate function can be used to add a single  
character string at a specific location, as demonstrated here to add the  
Gulf of Mexico:

ggplot(data = world) +

geom\_sf() +

geom\_text(aes(LON, LAT, label = NAME), size = 4, hjust = "left",

color = "darkblue", fontface = "bold", check\_overlap = TRUE) +

annotate(geom = "text", x = -90, y = 26, label = "Gulf of Mexico",

fontface = "italic", color = "grey22", size = 6) +

coord\_sf(xlim = c(-102.15, -74.12), ylim = c(7.65, 33.97), expand = FALSE)



**Final map**

Now to make the final touches, the theme of the map can be edited to  
make it more appealing. We suggested the use of theme\_bw for a  
standard theme, but there are many other themes that can be selected  
from (see for instance ?ggtheme in ggplot2, or the package  
[ggthemes](https://cran.r-project.org/package=ggthemes) which provide  
several useful themes). Moreover, specific theme elements can be tweaked  
to get to the final outcome:

* Position of the legend: Although not used in this example, the  
  argument legend.position allows to automatically place the legend  
  at a specific location (e.g. "topright", "bottomleft", etc.);
* Grid lines (graticules) on the map: by using panel.grid.major and  
  panel.grid.minor, grid lines can be adjusted. Here we set them to  
  a gray color and dashed line type to clearly distinguish them from  
  country borders lines;
* Map background: the argument panel.background can be used to color  
  the background, which is the ocean essentially, with a light blue;
* Many more elements of a theme can be adjusted, which would be too  
  long to cover here. We refer the reader to the documentation for the  
  function theme.

ggplot(data = world) +

geom\_sf(fill = "antiquewhite1") +

geom\_text(aes(LON, LAT, label = NAME), size = 4, hjust = "left",

color = "darkblue", fontface = "bold", check\_overlap = TRUE) +

annotate(geom = "text", x = -90, y = 26, label = "Gulf of Mexico",

fontface = "italic", color = "grey22", size = 6) +

annotation\_scale(location = "bl", width\_hint = 0.5) +

annotation\_north\_arrow(location = "bl", which\_north = "true",

pad\_x = unit(0.75, "in"), pad\_y = unit(0.5, "in"),

style = north\_arrow\_fancy\_orienteering) +

coord\_sf(xlim = c(-102.15, -74.12), ylim = c(7.65, 33.97), expand = FALSE) +

xlab("Longitude") + ylab("Latitude") +

ggtitle("Map of the Gulf of Mexico and the Caribbean Sea") +

theme(panel.grid.major = element\_line(color = gray(.5),

linetype = "dashed", size = 0.5),

panel.background = element\_rect(fill = "aliceblue"))



**Saving the map with ggsave**

The final map now ready, it is very easy to save it using ggsave. This  
function allows a graphic (typically the last plot displayed) to be  
saved in a variety of formats, including the most common PNG (raster  
bitmap) and PDF (vector graphics), with control over the size and  
resolution of the outcome. For instance here, we save a PDF version of  
the map, which keeps the best quality, and a PNG version of it for web  
purposes:

ggsave("map.pdf")

ggsave("map\_web.png", width = 6, height = 6, dpi = "screen")