geoplot: cartography in gretl

Allin Cottrell and Jack Lucchetti

February 28, 2024

This document describes the geoplot addon package for gretl, introduced in version 2020c. We have found it preferable, for reasons of efficiency, to implement its core mapping functionality via built-in functions, coded in C. For basic usage, therefore, it's not necessary to load the package explicitly via "`include geoplot.gfn`", as one would usually expect for an addon.

Section 1 in this document is meant to give the reader a quick introduction to the main cartographic features we offer. In order to produce more refined maps, however, we provide more specialized facilities that are explained in the rest of the document.

Section 2 below covers some preliminaries which we hope will help the reader understand the basics of cartography via gretl. Section 3 describes the basic workflow for producing a map image. Section 4 then provides a simple worked example and section 5 addresses a potential stumbling block. Section 6 goes over in detail the options to the core `geoplot` function; section 7 discusses some spcifics of plotting qualitative data; section 8 tells you what's available via gretl's graphical interface; and section 9 explains some "expert" refinements. Three appendices cover some technical points that may be of interest to expert users.

1 Quick start

The sample file `us2020.geojson` contains the main results of the 2020 presidential elections in the US. If you open it via the command

```
open us2020.geojson --frompkg=geoplot
```

you will see a dataset that looks very much like an ordinary gretl dataset:

	name	region	dem	biden	rep
1	Maryland	South	0.6536	1	0.3215
2	Minnesota	Midwest	0.5240	1	0.4528
3	Montana	West	0.4055	0	0.5692
4	North Dakota	Midwest	0.3178	0	0.6512
5	Hawaii	West	0.6373	1	0.3427
6	Idaho	West	0.3307	0	0.6384
		⋮			

In fact this dataset is special, since it also contains cartographic information "in the background". The question of how to create such a file will be dealt with in Sections 2 and 3, but for the moment let's just skip this point.

To plot a thematic map for the share of votes received by the Republicans in each state, you could simply open a console and type

```
geoplot(rep)
```

and a plot like the one shown in Figure 1 should appear on your screen. Achieving the same via the GUI is quite easy; we'll deal with this in Section 8.

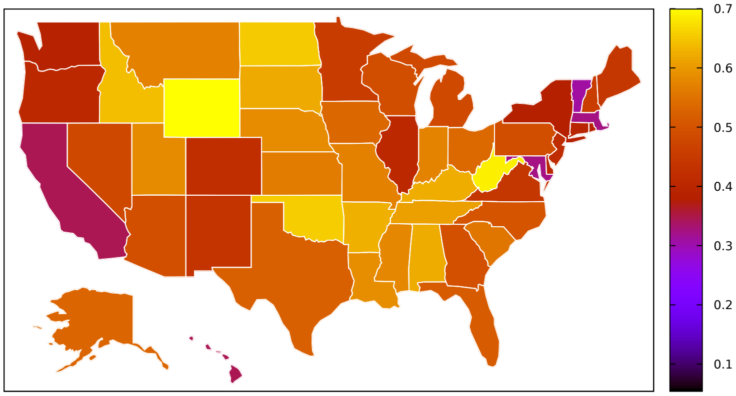


Figure 1: Output of `geoplot(rep)`

The interpretation of the map in Figure 1 should be obvious, with "hotter" colors for the states where the Republican party enjoyed more support, as shown by the legend to the right of the map. If we try the same with the `region` variable via

```
geoplot(region)
```

we get a map similar to Figure 2. In this case the colors are the same for states in the same US region: note that the format of the legend has also changed appropriately. The difference is due to the fact that `rep` is a quantitative variable, while `region` is qualitative (and string-valued). Gretl is aware of the difference and plots the map with the appropriate adjustments. A similar example that you can try by yourself is plotting a map for the binary variable `biden`. More on this in Section 7.

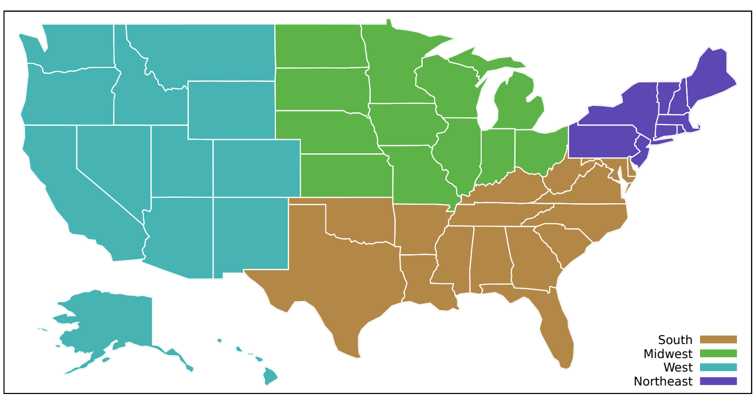


Figure 2: Output of `geoplot(region)`

Many details of the maps—such as their color scheme, size, and so on—can be customized to the user's needs. To this aim, you can supply an options bundle to the `geoplot` function to inflect its behavior. For example, the line

```
geoplot(dem, _(palette="blues"))
```

produces something like Figure 3. In this case, we use the `palette` option to represent the degree of Democratic support by shades of blue. The available options and their meaning are explained in section 6.

And now, let's take a deeper look under the hood.

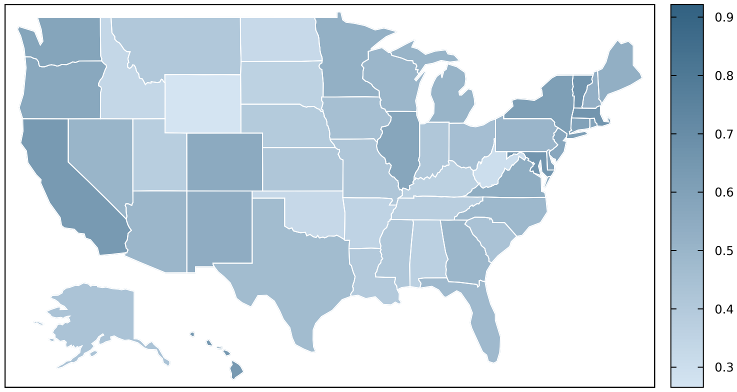


Figure 3: Output of `geoplot(dem, _(palette="blues"))`

2 Preliminaries

Given gretl's user base and vocation, we assume that people will be primarily interested in "thematic" maps, in which geographical areas get different colors according to some variable of interest (for example regions of a country are colored according to their unemployment rates). Fancier maps are out of scope for the present. From here on we'll simply refer to maps of this sort as "maps" and the geographical entities they contain (which may be countries, states, counties, länder or whatever) as "regions". Plotting a map typically involves drawing a number of polygons, filled with appropriate colors, to some device (the screen, or a file).

The essential ingredients for doing this are

1. A geometrical description of the regions.
2. The data for coloring the polygons.
3. Appropriate software for producing the map.

2.1 The geometry

Let's say we have n regions, indexed by i. Region i is represented geometrically as a collection of k_i polygons (think islands in an archipelago), indexed by j. Each polygon is defined by $h_{i,j}$ coordinates. Typically, each coordinate vector has two elements, latitude and longitude.

The information on each region has two components:

Metadata At minimum this should include the region's identifier(s), as strings and/or numerical codes. Other information, such as land area, may also be included. You can think of this as a dataset with n observations and several variables, possibly string-valued.

Polygons A representation of the region's shape on the map, in the form of one or more polygons, each taking the form of an array of X–Y pairs, typically latitude and longitude. You can think of this as an array of arrays of 2-column matrices: the outer array is of size n; inner array i contains k_i matrices, each with two columns.

Several file formats can be used for storing the geometry information.[1] Gretl supports what are probably the two most common formats:

- GeoJSON files: these are plain JSON files with an internal structure specified by RFC 7946. Such a file takes the form of an array of regions (or "features"), with each element containing the metadata under the key `properties` and the polygons under the key `geometry`.

[1]The site https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/administrative-units-statistical-units/nuts offers a nice collection for European NUTS regions (NUTS = Nomenclature of Territorial Units for Statistics).

- ESRI shapefiles: these come as collections of several files, usually zipped together. The essential components are an xBase file, with `dbf` extension, holding the metadata; the shapefile proper with extension `shp`, holding the polygons; and an index file with extension `shx`, used to speed up operations when reading the data. The website https://gadm.org/ offers a huge collection of such files.

2.2 The payload data

By "payload" we mean the data used for coloring the regions. We assume that the payload is available as a gretl series. This typically means that the user has a data file (in native `gdt` format or some other format gretl can read) in which each line represents a region, as illustrated in Table 1. Note in particular the "id" column. We assume that the map metadata contain sufficient information to establish a correspondence with the dataset containing the payload: either a numerical code or a suitable string-valued variable. One thing one quickly learns in exploring a variety of `geojson` and `shp` files is that there's no telling how the regions will be ordered; one *cannot* assume that they occur in what one might think of as "standard" order (e.g. alphabetical order for US states).

id	State	Pop2019	Pop2010
1	Alabama	4903185	4779736
2	Alaska	731545	710231
101	American Samoa	55641	55519
3	Arizona	6278717	6392017
4	Arkansas	3017825	2915918
5	California	39512223	37254523
6	Colorado	5758736	5029196
7	Connecticut	3565287	3574097
	⋮		

Table 1: Typical format for a payload dataset

2.3 The software backend

In principle one could represent maps using any one of the many plotting libraries available, but gretl uses gnuplot, which we already use for all other kinds of plot. Some insight into the gnuplot commands we use can be gained from Appendix A.

Given the geometry data and the payload, writing a gnuplot script for producing the map is straightforward. And in most cases gnuplot can produce a plot in short order. You might have to wait a little if there are many regions, of complex shapes, represented in high precision in the source map file.

3 The workflow

Typical workflow for producing a thematic map in gretl is likely to be as follows.

1. You `open` the map-datafile as a gretl dataset; this reads in the metadata so gretl's `$nobs` will be equal to the number of regions, n.
2. You add the payload data, via the `append` or `join` commands or in some other way.
3. You decide on some details of your map (appearance, format, etc.), with sensible defaults being available.
4. You create the map.

5. Optionally, you can save the data in gretl's native `gdt` format (but see below).

Point 1 is handled by using the `open` command on the map-datafile. The filename extensions recognized for this purpose are `json` or `geojson` for GeoJSON files, and `dbf` or `shp` for ESRI shapefiles.[2] Point 2 is also handled by standard gretl commands.

Points 3 and 4 are handled by the `geoplot` function, which can be called in either of two ways, corresponding to these two signatures:

```
function void geoplot(const series payload[null], const bundle options[null])

function void geoplot(string mapfile, const series payload[null],
                      const bundle options[null])
```

The first case is applicable if the map to be shown has already been loaded as a gretl dataset. The second case is required if you want to use a map file which has not been so loaded: then you need to give its name. The `payload` argument is the (optional) series with which to colorize the polygons, and `options` is an (optional) bundle to contain one or more elements governing the appearance or destination of the plot.

- If the `payload` argument is given as `null` or omitted then the map is drawn "as is", without any colorization. This can be useful if you just want to see what the polygons look like.
- If the `options` bundle is omitted all options are set to their default values—which means, among other things, that you see the map on screen but nothing is saved. For full details on the available options see section 6.

Note that once a map is loaded as a dataset you can retrieve its filename using the `$mapfile` accessor. This is convenient if you wish to read the full content of the file (including the polygons) as a bundle:

```
bundle b = bread($mapfile)
```

Finally, a cautionary note on point 5 above: the saved `gdt` file will *not* contain the geometry information, but rather a pointer to the appropriate file (GeoJSON or shapefile). As a consequence, as long as the file holding the polygon info remains in the same place, you can simply re-open your `gdt` file and create new maps. However, you can't send it to someone else and expect it to work for map creation, unless you also supply them with the original geometry file.

4 An example

For this example we'll produce a map showing GDP per capita of the six founder countries of the EU, using three files: script `founders.inp`, data file `founders.csv` and map `founders.geojson`. In this case the required files are small enough to be readily inspected by hand. The content of `founders.csv`, which holds what will be the payload, is shown in Listing 1.

The JSON file is too big to show here in full but small enough to examine in any text editor; listing 2 contains a representative excerpt. Note that while Belgium is a single polygon France is an array of polygons ("`MultiPolygon`"), because of Corsica.

The `founders.inp` script is shown in Listing 3, and what we get after opening the `geojson` file in gretl is in Listing 4.

Next, we perform a `join` operation with `FID` (from the JSON file) as the inner key and `code` (from the CSV file) as the outer key. Finally, we call the `geoplot` function to create an on-screen map. We specify the `payload` to colorize and via the `options` argument we add a couple of points: the `gnuplot` input

[2] In principle we could read the polygons at this point and store them in RAM, but for now we don't. We just read in the metadata, but store the path to the associated geometry file internally.

```
Name,code,pop,area,gdp
Belgium,BE,11365834,30528,534230
France,FR,67024633,632833,2833687
Germany,DE,82437641,357386,3874437
Italy,IT,61219113,301338,2147744
Luxembourg,LU,589370,2586.4,65683
Netherlands,NL,17220721,41543,880716
```

Listing 1: Content of `founders.csv`


```
{"type": "FeatureCollection", "features": [
 {"geometry": {"type": "Polygon", "coordinates": [[[40.40360,
     30.79039], [40.59686, 30.49366], [40.65087, 30.29746], ... ]]},
   "type": "Feature", "properties": {"CNTR_NAME": "Belgique",
     "ISO3_CODE": "BEL", "CNTR_ID": "BE", "NAME_ENGL": "Belgium",
     "FID": "BE"}, "id": "BE"},
 {"geometry": {"type": "MultiPolygon", "coordinates": [[[[40.18497,
     29.45664], [40.23634, 29.39875], [40.57754, 29.35021], ...],
     [[[42.66689, 20.70300], [42.57348, 20.41660], ...]]},
   "type": "Feature", "properties": {"CNTR_NAME": "France",
     "ISO3_CODE": "FRA", "CNTR_ID": "FR", "NAME_ENGL": "France",
     "FID": "FR"}, "id": "FR"},
  ...
```


Listing 2: Excerpt of `founders.geojson`

```
open founders.geojson --frompkg=geoplot

join founders.csv gdp pop --ikey=FID --okey=code
series gdppc = 1000*gdp/pop

opts = _(plotfile = "GDPpc.plt", inlined = 1)
geoplot(gdppc, opts)
```

Listing 3: Content of `founders.inp`

```
     CNTR_NAME    ISO3_CODE      CNTR_ID    NAME_ENGL          FID

1     Belgique          BEL           BE      Belgium           BE
2       France          FRA           FR       France           FR
3  Deutschland          DEU           DE      Germany           DE
4       Italia          ITA           IT        Italy           IT
5    Luxemburg          LUX           LU   Luxembourg           LU
6    Nederland          NLD           NL  Netherlands           NL
```

Listing 4: The “founders” metadata

file should be saved under the name `GDPpc.plt`, and the geometry data should be “inlined” in this file, making it self-contained.

Running the script will produce a gnuplot file resembling the following:

```
set term wxt persist

unset key
set cbrange [33.3288:117.018]
set xrange [31.7826:51.4213]
set yrange [14.7701:36.0553]
$MapData << EOD
40.4036 30.79039 47.00315
40.59686 30.49366 47.00315

[...]

38.92268 31.40258 51.142806
38.59809 31.50169 51.142806
38.59737 31.60855 51.142806

EOD
plot for [i=0:*] $MapData index i with filledcurves fc palette, \
  $MapData using 1:2 with lines lc "white" lw 1
```

and feeding the above into gnuplot yields the map shown in Figure 4.

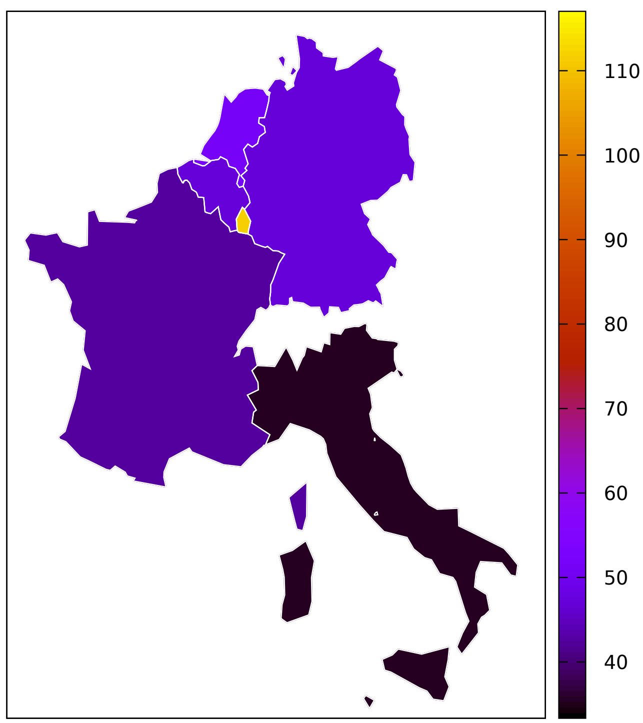


Figure 4: Output of `founders.inp` (default gnuplot palette)

5 Ensuring correct alignment

In order to produce a correct map it is essential that everything be aligned properly: the map metadata, the payload series, and the geometries of the regions. "Region i" must have the same referent in all three contexts.

If the source map (GeoJSON or shapefile) is not broken we can assume that the original metadata and the geometries are indeed aligned correctly. But problems may arise (a) in aligning the payload and (b) if one wishes to exclude some regions from the map. We discuss these issues in turn.

5.1 Aligning the payload

If the payload you wish to plot is already included in the map metadata, there's no problem. But when the map data and the payload come from different sources it may be tricky to get them aligned properly. This problem is not specific to the mapping apparatus—it's a more general issue concerning the matching of data from different sources, addressed at length in the chapter titled "Joining data sources" in the the *Gretl User's Guide*—but it may be helpful to offer a few comments here.

As mentioned above, the relevant tools provided by gretl are `append` and `join`. A simple `append` will work only if the regions appear in the same order in the map and payload datasets. This is fairly easily checked if the number of regions is small and each dataset contains readily comparable identifiers. Otherwise—if the orders clearly differ or it's hard to tell—it will be necessary to use `join`.

Look back at Listing 3. In that case the map and payload datasets contained the same set of two-letter identifiers for the countries—albeit under different names, `FID` and `code`—so `join` using the `--ikey` and `--okey` options worked fine. In a different case, however, the respective identifiers may not match up. For example, region names might be in English in one dataset and in, say, Italian in the other. Then you'll have to exercise your intelligence, but one idea is to create an intermediate "Rosetta stone" file, maybe as CSV, giving the mapping between the two identifiers, as in:

```
# rosetta.csv
ID_en,ID_it
Apulia,Puglia
Sardinia,Sardegna
...
```

Then you can `join` the Rosetta file to the map dataset, hence adding the required identifier that's initially lacking.

5.2 Sub-sampling

In some cases one may wish to leave out certain outlying regions. For example, it's quite common to produce thematic maps of the USA that omit Hawaii, and perhaps Alaska. In principle leaving out regions threatens to break the required alignment of payload and geometry, but this is handled as follows: if you sub-sample the map dataset using the `smpl` command, `geoplot` automatically drops the associated polygons from the plot.

This is illustrated in Listing 5. Figure 5 shows the results with and without exclusion of Alaska and Hawaii.[3]

A related case is where the payload value is missing (`NA`) for one or more regions. Here you have a choice. By default, regions whose payload is `NA` are shown in outline, not colored, but either of two alternatives can be selected by passing a string under the key `missvals` in the options bundle: if the value is `"skip"` the affected regions are omitted; if it's `"fill"` they are colored gray. (The value `"outline"` may be given, confirming the default.)

[3] Given the role of this example we don't bother adding a real payload, but just simulate data using the `normal` function.

```
open us-states.geojson --quiet --frompkg=geoplot
x = normal() # fake up some data!
opts = _(plotfile = "us0.plt", palette = "blues")

# show the entire USA
opts.title = "USA (complete)"
geoplot(x, opts)

# skip Alaska and Hawaii
smpl postal != "AK" && postal != "HI" --restrict
opts.title = "USA (mainland)"
opts.plotfile = "us1.plt"
geoplot(x, opts)
```

Listing 5: US maps, complete vs contiguous states

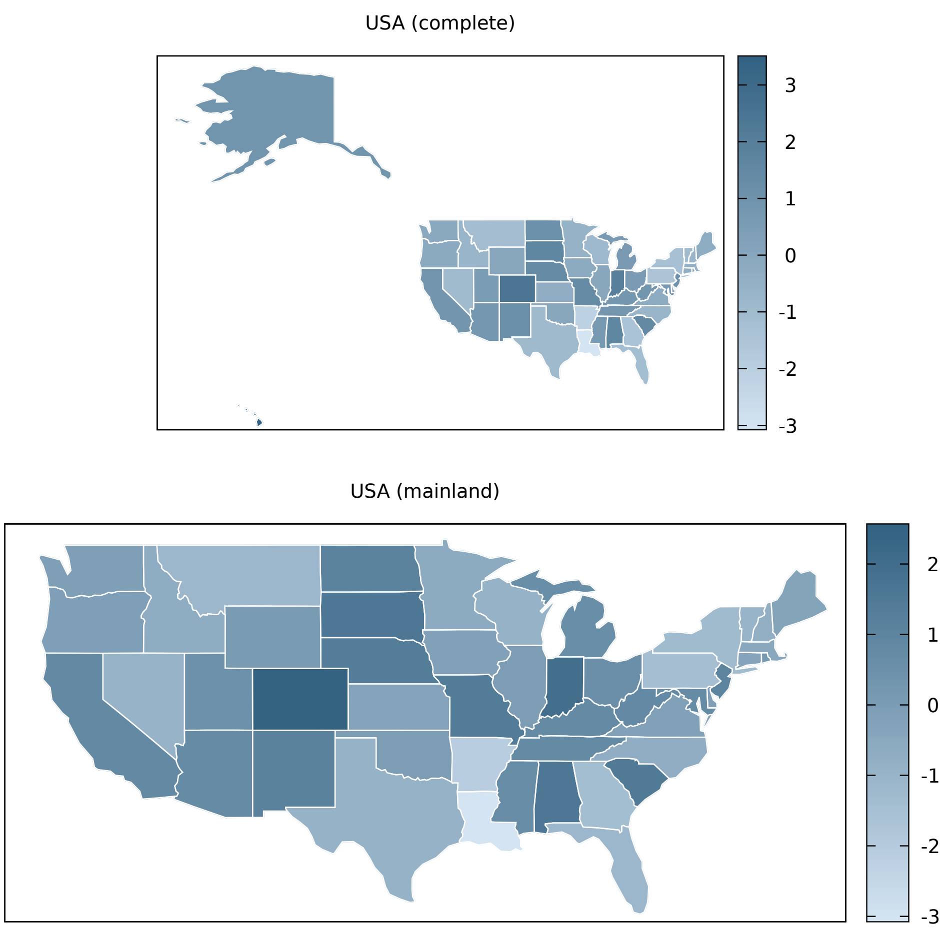


Figure 5: Output of Listing 5

6 Options for the `geoplot` function

We first present all the currently supported options in alphabetical order. Below the listing we give some further explanation of the usage of `plotfile`, `show` and `inlined`.

`border:` boolean, show a rectangular border around the map. Default: true.

`height:` scalar, giving the height of the plot in pixels. Default: 600. Even if the desired output is a vector graphic (PDF or EPS) rather than a bitmap (see `plotfile` below), setting this value relative to the default can be used to adjust the size of the plot. For example `height = 400` will give a PDF graphic that's two-thirds of the default size.

`inlined:` boolean, to have the polygon data written directly into the gnuplot file. Default: false, the data are read from a separate file.

`keypos:` string, specifying the position of the key for discrete plot colors (relevant only when plotting qualitative data; see Section 7). This string must be a valid argument to gnuplot's `set key` command. The most common usage takes the form of two words: `left`, `right` or `center` along with `top`, `bottom` or `center` (in either order). These words can be preceded by `outside` to place the key outside of the plot area. For example, `keypos="bottom center"` or `keypos="outside top left"`. The default position is bottom right.

`linecolor:` string, naming the color in which to draw the borders of the regions. By default this is white if a payload is plotted, black if only outlines are shown.

`linewidth:` scalar, giving the width of the lines representing the borders of the regions. Setting this to 0 suppresses those lines (unless no payload is supplied). Default: 1.0.

`literal:` string containing gnuplot commands, for insertion before the actual `plot` command.

`logscale:` boolean, use log scale for the payload. Default: false.

`missvals:` string, see section 5.2.

`plotfile:` filename, allowing the user to direct output, either to a specified gnuplot command file or to a graphic file. If a command file is wanted the extension should be `plt`. If a graphic file is wanted you should give one of the following extensions: `png`, `pdf`, `eps`, `emf`, `svg` or `html`. If `plotfile` is given, but not as a full path, the file is written to the user's working directory.

`projection:` string, see Appendix B.

`palette:` string, the exact specification of which depends on whether the payload data are quantitative or qualitative. For the qualitative case see Section 7. In the quantitative case the string should give either a gnuplot `set palette` command or a predefined option, of which there are currently three: `blues`, `oranges` and `green-to-red`. For example, the syntax

```
options.palette = "set palette defined (0 '#D4E4F2', 1 'steelblue')"
```

will give you a pleasing blue gradient—which also happens to be what you get by giving

```
options.palette = "blues"
```

If no such string is given you get the default built-in gnuplot palette.

`show:` boolean, should the plot should be shown on-screen right away? Default: true. (Otherwise a file of some sort is written but not displayed—see `plotfile` above.)

`tics:` boolean, for turning on the printing of X (longitude) and Y (latitude) tics. Default: tics are suppressed, unless `geoplot` is invoked without any payload.

`title:` string specifying a title for the plot. Default: no title.

xrange: 2-element vector, containing the longitude range used on the plot (see Appendix B). Default: automatically determined.

yrange: 2-element vector, containing the latitude range used on the plot (see Appendix B). Default: automatically determined.

It may be helpful to run through various `geoplot` scenarios with an eye to usage of the options.

1. You just want to see the map on-screen. Then don't give `plotfile` (or set it to `null`) and accept the default of `show` = 1.
2. You want to see the map on-screen but also save the plot command file that generated it (maybe you want to edit the commands or pass them to `gnuplot` independently of gretl). Then specify `plotfile` with a `plt` extension.
3. As in case 2 but you don't care to see the map on-screen: add `show` = 0.
4. You want to generate a graphic file (maybe for inclusion in a document or web page). Then give `plotfile` with one of the recognized graphic format extensions. In this case `show` is automatically turned off.

Note that if you set `show` to 0 and do *not* specify `plotfile` that is tantamount to saying "Don't do anything!", which is regarded as an error.

6.1 Inlined data or not?

As mentioned above, `geoplot` can pass the map coordinates to `gnuplot` either by writing them directly into the plot command file (`inlined` = 1), or by writing them to a separate data file whose name is recorded in the command file (`inlined` = 0).

From the user's point of view, this distinction makes a difference only if you're saving the plot command file (as in scenario 1 or 2 above). In that context, the advantage of having the data inlined is that, being all in one file, the information required to generate a map cannot easily become "unstuck". The disadvantage is that the command file may be very large, and perhaps not so easy to edit; besides the actual `gnuplot` commands it may contain many thousands of lines of coordinates data (which in general *should not be touched*, on pain of breaking the map). At present `inlined` is set to 0 by default but that may change; we recommend making an explicit choice via the `options` bundle.

Note that if you specify `plotfile` as a `gnuplot` command file, but not `inlined`, you'll get two output files: the specified `plt` file plus a data file named by adding the extension `dat`. For example you might get `mygeo.plt` and `mygeo.plt.dat`, while with `inlined` = 1 you'd just get a big `mygeo.plt`.

7 Quantitative versus qualitative payloads

In the foregoing we have assumed that if a payload is supplied the data are quantitative—either continuous, or if discrete then at least ordinal. However, one may wish to plot qualitative (categorical) data, and that calls for a different sort of color palette. In the quantitative case one probably wants a graduated scheme—either shades of a single hue or perhaps a "heat map" comprising more than one hue. If the data are categorical, clearly distinct colors are likely wanted.

How can `geoplot` tell the difference? Well, in a gretl dataset a categorical variable is likely represented by a string-valued series, but may take the form of a numeric series where the numbers are a pure encoding with no quantitative significance. String-valued series (and also 0/1 dummy series) will be recognized by `geoplot` as qualitative automatically, but otherwise a numeric series called `cvals` which serves as an encoding should be marked as such, using the command

```
setinfo cvals --coded
```

If a payload series is recognized as qualitative with k values, the default `geoplot` palette is not the gnuplot default (which is of the heat-map type) but rather an automatic selection of k distinct colors. And the values accepted under the `palette` key in the `options` bundle are simpler than in the quantitative case. We expect either the name of an array of k strings, each of which contains a color specification comprehensible by gnuplot, or the keyword `auto` (which just confirms automatic color selection). Here's an example, suitable for a binary payload variable ($k = 2$):

```
strings mycolors = defarray("#D22532", "#244999")
opts.palette = "mycolors"
```

In addition a second, comma-separated element can be given when the payload data are qualitative, namely an array of strings to appear in the "key" or legend for the plot. If the payload series is string-valued its strings will be used by default for this purpose, but you may wish to supply abbreviated or translated strings. This facility can also be used to give suitable strings for a 0/1 binary series (which cannot have string values since that requires a minimum numeric value of 1 or greater). Extending the example above, we might do:

```
strings mycolors = defarray("#D22532", "#244999")
strings zlabels = defarray("Republican", "Democratic")
opts.palette = "mycolors,zlabels"
```

The position of the legend can be adjusted by using the `keypos` option (see Section 6).

Examples of maps showing qualitative data are provided by the scripts `swiss-langs.inp` and `us-2020.inp`. You can find these scripts and others via the item Resource from addon under the File menu in the gretl GUI. Figure 6 shows two variants of a map representing the main language of each Swiss canton (via a string-valued series named `mainlang`). The customizations in the second map are achieved via

```
strings langcolors = defarray("#E87E7E", "#9DA8E0", "#85E1C3", "#E1C385")
strings zlabels = defarray("Deutsch", "Français", "Rumantsch", "Italiano")
opts.palette = "langcolors,zlabels"
```

Give a palette string of (e.g.) `"auto,zlabels"` if you want to adjust the strings but use the default colors.

8 Maps via the GUI

To this point we have referred exclusively to executing commands and calling functions. You can, of course, execute commands and call functions in the GUI program via script or via the gretl console, but what about point-and-click? Well, there is a certain amount you can do in that way.

First, you can open a shapefile or GeoJSON file using the menu item /File/Open data/User file. In the bottom right-hand corner of the "open file" dialog, use the pull-down list to select "Shapefiles" or "GeoJSON files". You can also drag-and-drop such files onto the main gretl window to the same effect.

Once map metadata are loaded in this way, the option Display map becomes available via the context ("right-click") menu in the main window, and also under /View/Graph specified vars; it will trigger a window like the one in Figure 7.

If the current dataset contains one or more series that seem to gretl to be plausible "payload" variables,[4] invoking Display map will produce a dialog box that allows you to select one, in which case you'll get a choice of color palette to represent it. Otherwise you just get to view the map outlines. The options `border`, `linewidth`, `logscale` and `height` (see Section 6) can also be selected via the GUI.

Further, the plot window shown by Display map offers a right-click menu that allows saving the map (as PDF, EPS, PNG or EMF), copying it to the clipboard, or saving it "as an icon".

Note that the options offered by the GUI are, at the moment, quite limited compared to the ones available via scripting. We may at some point expand the Display map dialog to offer more of the choices available via the `options` argument to the `geoplot` function.

[4] Admittedly, the heuristic employed for this purpose is not terribly clever.

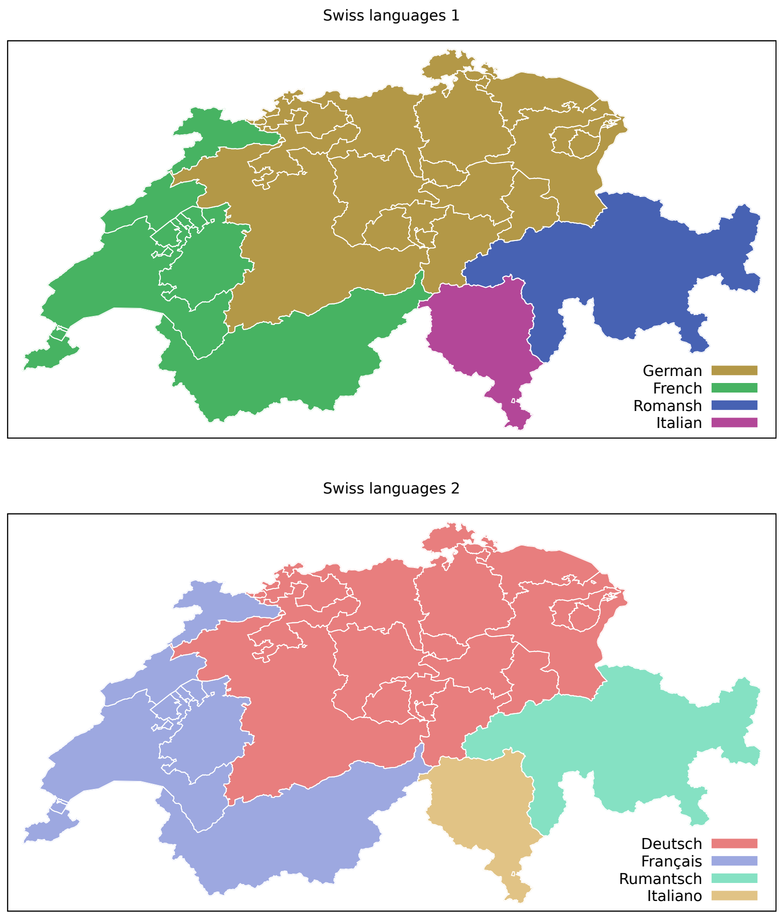


Figure 6: Main language per Swiss canton: the upper plot uses the default palette and strings; the lower one uses a customized palette and translated strings.

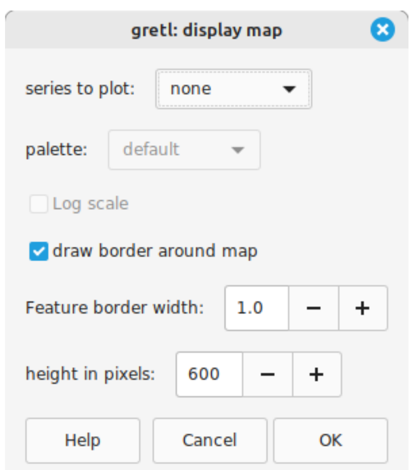


Figure 7: Graphical interface to the `geoplot` command.

9 "Expert" refinements

To recap, we have explained how map metadata can be brought in via the `open` command (or via the GUI); how to add "payload" data; and how to generate a map using the `geoplot` function. So far so good, but that leaves open some questions that might occur to ambitious users. Can I open a map file, add a payload series, and save a modified version of the map file including the payload? And in relation to a map of the USA (for example), is there a way to include Alaska and Hawaii, but "tuck them underneath" the continental US, like I see in graphics on the web?

Short answer: Yes. You can import a GeoJSON file (or shapefile) as a gretl "bundle" by means of the `bread` function; make changes to the bundle; then save it as GeoJSON using `bwrite`.

To get control over this it's necessary to understand the structure of the bundle that `bread` produces when fed map input. This mimics the structure of a GeoJSON file (even if the input comes from a shapefile), as shown in Listing 6; the labeling of elements is as in GeoJSON, with gretl types in parentheses.

```
FeatureCollection (bundle)
  features (array of n bundles)
    features[i] (bundle)
      features[i].properties (bundle)
      features[i].geometry   (bundle)
        features[i].geometry.type (string)
        features[i].geometry.coordinates (array)
```

Listing 6: Structure of map data, gretl types in parentheses

9.1 Injecting payload data

First let's look at the case of injecting a payload series into a map file. Listing 7 revisits the EU founders map discussed in section 4. As before, we start by opening the metadata and "joining" the GDP per capita data. But now we open the GeoJSON as a bundle, and for each `feature` (country) we augment its `properties` bundle with the corresponding value of the `gdppc` series (under the key "`gdppc`"). Finally, we write the modified data to file.

9.2 Rearranging regions

Now how about rearranging regions of a given country (or more generally, `features` within a given `FeatureCollection`)? Here we use the function `geoplot_translate_feature`. Unlike the `geoplot` function this is not "built in" so before calling it one must do

```
include geoplot.gfn
```

The signature of the feature-translation function is

```
void geoplot_translate_feature(bundle *b, int f,
                               matrix shift,
                               matrix center[null],
                               matrix scale[null])
```

You pass in a map bundle obtained via `bread` (in pointer form); the sequential index, `f`, of the feature to translate; and a 2-vector `shift` giving displacement in the X and Y directions. If in addition you want to rescale the feature you pass two more 2-vectors: `center` holds the coordinates of the feature's centroid and `scale` the scale factors to apply in the two directions.

An example script is shown in Listing 8 and the result of plotting the modified GeoJSON file in Figure 8. We obtain the second argument to pass to the translator by inspection of the map dataset: Alaska is feature 48 and Hawaii feature 5. In this case we decide to move Alaska 34° East and 35° South and

```
open founders.geojson --quiet --frompkg=geoplot
join founders.csv gdp pop --ikey=FID --okey=code
series gdppc = 1000*gdp/pop

# open full GeoJSON as bundle
bundle b = bread($mapfile)

# add GDP per capita to properties
loop i=1..nelem(b.features)
   b.features[i].properties.gdppc = gdppc[i]
endloop

# save modified geojson file
bwrite(b, "founders_mod.json")
```

Listing 7: Adding payload data to a map file: `founders_mod.inp`

```
include geoplot.gfn

open us-states.geojson --quiet --frompkg=geoplot
bundle b = bread($mapfile)

# Shrink Alaska and place underneath the "lower 48"
matrix shift = {34, -35}
matrix center = {-150.885, 62.5503}
matrix scale = {0.3, 0.35}
geoplot_translate_feature(&b, 48, shift, center, scale)

# Shift Hawaii alongside Alaska
shift = {51, 5}
geoplot_translate_feature(&b, 5, shift)

# save modified geojson file
bwrite(b, "us_modified.json")
```

Listing 8: Moving Alaska and Hawaii

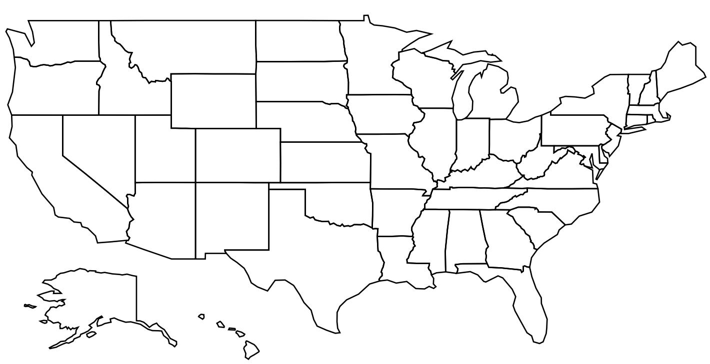


Figure 8: Alaska and Hawaii moved

shrink it substantially. Getting the effect one wants is likely to take some trial and error, but two geoplot features can be helpful.

First, if you plot just the map outlines (no payload) then you should see the X and Y values on the axes, giving you at least a rough idea of the shifts you might want. Second, geoplot contains the function `geoplot_describe_json` which gives you a good deal of relevant information. The signature of this function is

```
bundle geoplot_describe_json (const bundle jb, int verbose[1])
```

You pass a map bundle and a verbosity level, as in

```
include geoplot.gfn
bundle us = bread("us-states.geojson")
geoplot_describe_json(us, 3)
```

On searching the `verbose` = 3 output for Alaska one finds (here's a small snippet):

```
  48: geometry type = MultiPolygon, no id
        ...
        name: Alaska
        ...
        Extents: X = {-171.791,-129.98}; Y = {54.4042,70.6964}
```

The `Extents` data enable you to figure out plausible values for the `center` of a feature.

10 Coda

In the foregoing we have mostly kept things simple with toy examples. In concluding, we'll show off with a "real" example: Figure 9 shows the distribution of COVID cases across Italian provinces as of 2020-05-15. The appearance of this plot was tuned using the following option settings:

```
string cmds = sprintf("set colorbox user origin 0.9,0.45 size 0.03,0.4\n")
cmds ~= sprintf("set xrange [6.4:20.5]")
bundle opts = _(plotfile = "covid.pdf")
opts.logscale = 1
opts.border = 0
opts.linewidth = 0.4
opts.palette = "green-to-red"
opts.literal = cmds
opts.height = 900
```

As you will have seen in the previous figures, the default gnuplot "colorbox" is quite large, occupying the full height of the plot. This doesn't look so great for a tall, skinny country like Italy, so we used the `literal` option to pass in commands to make the colorbox smaller and reposition it; to prevent the colorbox from going off the right edge of the plot we also made the `xrange` a little wider than the default. We were able to determine what the revised `xrange` should look like by examining a plot with no payload, showing the latitude values on the axes.

11 Change log

2024-02-28: Fix obsolete use of `--write` with the `outfile` command.

2023-05-01: Add `keypos` option for adjusting the key position when plotting qualitative data.

2023-01-17: Update the documentation to explain how to handle qualitative payload data; fix bug whereby output to an image file could fail.

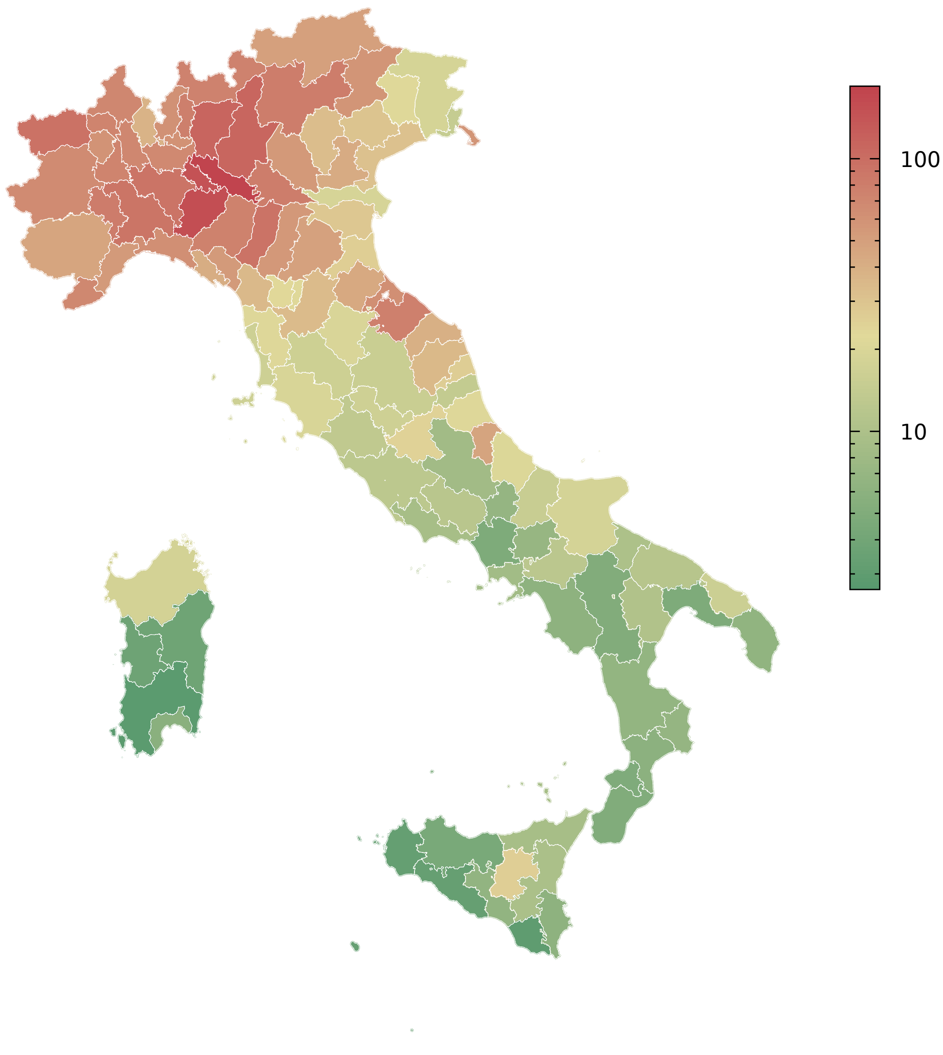


Figure 9: COVID-19 cases per 10 000 by province as of 2020-05-15, log scale

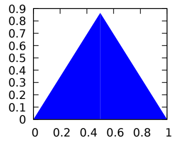
2021-02-12: Fix some incorrect type specifiers in `geoplot_utils.inp`, revealed by improved type-checking following the gretl 2021a release.

2020-05-27: Initial release

Appendix A Representation of polygons in gnuplot

The following gnuplot code

```
unset key
plot '-' using 1:2:3 with filledcurves fillcolor "blue"
0 0 0
0.5 0 0.866
1 0 0
e
```

produces an equilateral triangle: 

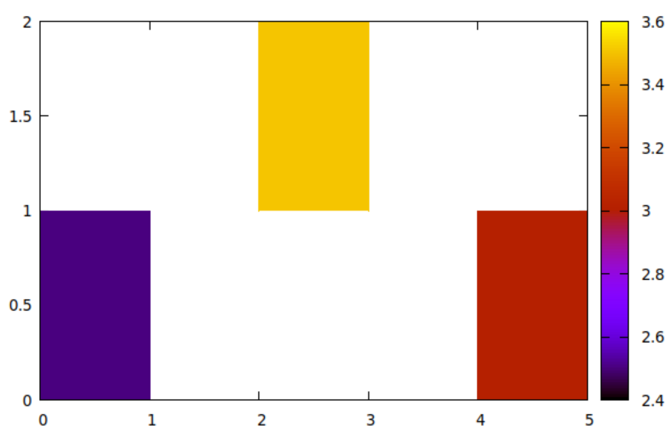
The internal coloring of polygons can be set using "`fillcolor palette`". For example,

```
unset key
$coords << EOD
0 0 2.5
0 1 2.5
1 1 2.5
1 0 2.5

2 1 3.5
2 2 3.5
3 2 3.5
3 1 3.5

4 0 3
4 1 3
5 1 3
5 0 3
EOD
plot for [i=0:*] $coords index i with filledcurves fillcolor palette
```

(where the third column of the data indexes into the palette) produces



A nice way to customize the palette is via "`set palette defined`" (see the gnuplot manual; also see section 6 above).

Appendix B Projections

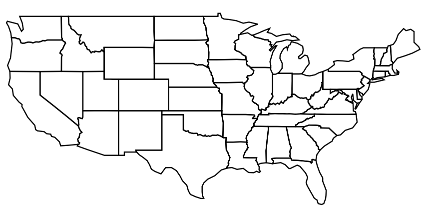
Geoplot assumes (mostly, but see below) that incoming map coordinates are given as degrees of latitude (Y) and longitude (X). This is mandated by RFC 7946, which has governed GeoJSON since 2016; it also appears to be the most common case for ESRI shapefiles.

Everyone knows that the Earth is not actually a sphere, but let's pretend it is for simplicity. Then a degree of latitude is always the same length on the ground: 1/360 of the planet's circumference. But the length of a degree of longitude varies, from 1/360 of Earth's circumference at the equator to zero at the poles. So imagine that we pass the X–Y pairs to our plotting engine on the assumption that "a degree" is always the same size: the result will be more or less OK close to the equator but at higher or lower latitudes features will be seriously stretched horizontally (or squashed vertically) relative to what we're used to seeing. To avoid this effect some sort of projection is required.

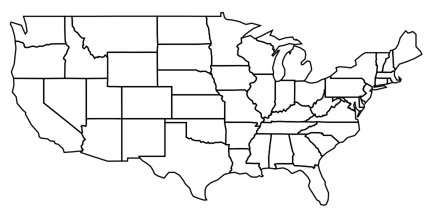
By default geoplot uses what we might call a "quasi-Mercator" projection. In most cases this should produce maps that look quite acceptable and it has the advantage of simplicity. All we do is take the height of the plot as specified by the user (or a default of 600 pixels) and figure out what the width should be to make a degree of longitude the same size as a degree of latitude at the mid-point latitude. However, we offer four alternatives, as follows:

EPSG id	description	option string
3857	standard Mercator	`"Mercator"`
4326	"null" projection	`"EPSG4326"`
2163	U.S. National Atlas Equal Area	`"EPSG2163"`
3035	Europe Equal Area	`"EPSG3035"`

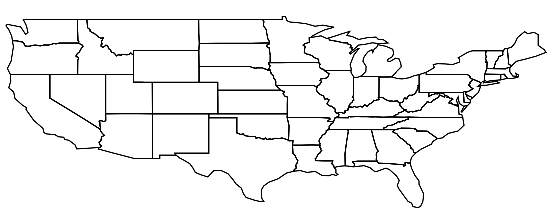
Figure 10 compares the available projections for the contiguous United States. In this case the default geoplot projection and standard Mercator are practically indistinguishable. EPSG:4326, which treats degrees as everywhere the same size, exhibits the horizontal stretching mentioned above. EPSG:2163 gives the impression of looking at the USA on a section of the globe.



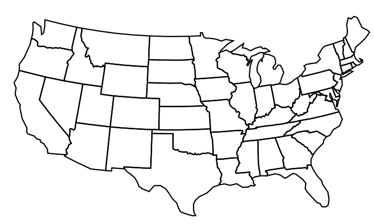
Default



Mercator



EPSG:4326



EPSG:2163

Figure 10: Comparison of projections

To select one of the alternatives, you add the appropriate string to the `options` bundle passed to the `geoplot` function under the key `projection`. Please note that EPSG:2163 is specially tuned for the USA, and will produce weird-looking or non-existent results for other parts of the world. EPSG:3035 is similarly tuned for Europe. They are both so-called Lambert Azimuthal Equal Area projections.

Non-standard coordinates

In certain map files—maybe GeoJSON predating RFC 7946, and perhaps some shapefiles—the X–Y coordinates are not in the expected form of degrees of latitude and longitude. In that case they probably already encode some sort of projection, and so should not be "re-projected". For GeoJSON files, geoplot makes an attempt to determine whether a non-standard coordinate system is used, as was allowed under the obsolete GeoJSON 2008 specification. In that case we automatically cancel projection; we also do this if the X or Y values are out of bounds for representing degrees (that is, $|X| > 360$ or $|Y| > 180$).

Failing such automatic detection, one can try specifying EPSG:4326 to get the X and Y units to be treated as equal in size, in effect canceling projection, as in

```
bundle options
options.projection = "EPSG4326"
```

Limiting the area shown

Section 5.2 explains how to exclude certain features from a map using the `smpl` command. In some cases, however, one may wish to limit what is shown in a different way, by specifying ranges for latitude and longitude. This can be done by supplying 2-vectors in the `geoplot` options bundle under the keys `xrange` (longitude) and `yrange` (latitude). For example, to plot just the area from 40° to 60° North and 10° to 30° East you can do

```
options.yrange = {40,60}
options.xrange = {10,30}
```

Specifying such ranges in degrees works fine if you are using the default `geoplot` projection, Mercator or EPSG:4326. But it doesn't work for azimuthal projections, where neither meridians nor parallels are straight lines. Rather, you should first plot the entire map using the projection you want, with axis tics turned on to show the linearized coordinates. Then specify the ranges in terms of these values. For example, here's how one might restrict a US map to the contiguous states via ranges:

```
open us-states.geojson --frompkg=geoplot
bundle opts
opts.projection = "EPSG2163" # azimuthal
opts.tics = 1
# take a look-see
geoplot(null, opts)
# suitable ranges, by inspection
opts.xrange = {-330,410}
opts.yrange = {-350,130}
geoplot(null, opts)
```

Further reading

For anyone wishing to follow up on this sort of thing, there are many websites presenting information on coordinate systems and projections. Two of the most useful ones, in our experience, are:

Reference materials: https://spatialreference.org/

Explanation: https://source.opennews.org/articles/choosing-right-map-projection/

Appendix C Specialized functions

The functions shown below are implemented in hansl and included in the geoplot addon; to use them you must first do

```
include geoplot.gfn
```

```
bundle geoplot_describe_json (const bundle jb, int verbose[1])
```

Provides a systematic description of the GeoJSON bundle `jb`, the amount of detail depending on the `verbose` setting, which has a maximum of 3. By assigning the return value one can obtain a bundle containing the information but for some purposes the printed output may suffice. See section 9.2.

```
void geoplot_set_properties (bundle *b, list L)
```

Rewrites the `properties` within the bundle representation of a map, `b`, to include all and only the series referenced in the list `L`. This provides a means of adding "payload" data (see section 2.2) and also pruning unwanted metadata. In the example below, `us-states.geojson` originally contains 40 items of metadata per state, most of them unlikely to be of interest.

```
# example
open us-states.geojson --quiet --frompkg=geoplot
join statepop.gdt population --ikey=postal --okey=Code

# select only the properties we actually want
list L = name postal population
bundle b = bread($mapfile)
geoplot_set_properties(&b, L)
bwrite(b, "us_pruned.json")
```

```
void geoplot_translate_feature (bundle *b, int f,
                                matrix shift,
                                matrix center[null],
                                matrix scale[null])
```

Shifts the feature with sequential index `f`, optionally rescaling it. See section 9.2 for an example and explanation.

```
matrix geoplot_seek_feature(const bundle b,
                            string name,
                            bool do_plot[1])
```

Searches the map bundle `b` for features matching `name` (on a case-insensitive basis). If one or more matches are found their 1-based indices are returned in a row vector. If a single match is found metadata for the feature are printed and if `do_plot` is not set to zero a plot of the feature is shown.

```
include geoplot.gfn
open us-states.geojson --frompkg=geoplot --quiet
```

```
map = bread($mapfile)
# no matches
geoplot_seek_feature(map, "nowhere")
# two matches
geoplot_seek_feature(map, "CAROLINA")
# one match, plot shown
geoplot_seek_feature(map, "Florida")
```

`void geoplot_simplify(bundle *b, scalar preserve[0.1:1:0.75])`

Simplifies the polygons in the map bundle `b` using the Visvalingam–Whyatt algorithm.[5] This may be useful if for a certain geography of interest the only map file readily available is at a higher resolution than you need. Smaller values of the `preserve` parameter preserve less detail or in other words simplify the map more radically; the default value of 0.75 may be considered conservative if you start with a very detailed map.

```
include geoplot.gfn
open highres.geojson --quiet
bundle map = bread($mapfile)
geoplot_simplify(&map, 0.5)
bwrite(map, "simplified.geojson")
open simplified.geojson --quiet
# see if the level of detail is OK
geoplot()
```

[5] See https://hull-repository.worktribe.com/output/459275. And for a nice illustration see https://bost.ocks.org/mike/simplify/ (accessed 2023-01-16).