1 GeoPandas

Lab Objective: *GeoPandas is a package designed to organize and manipulate geographic data, It combines the data manipulation tools of pandas with the geometric capabilities of the Shapely package. In this lab, we explore the basic data structures of GeoSeries and GeoDataFrames and their functionalities.*

Installation

GeoPandas is a new package designed to combine the functionality of pandas with Shapely, a package used for geometric manipulation. Using GeoPandas with geographic data is very useful as it allows the user to not only compare numerical data, but also geometric attributes. GeoPandas can be installed via `pip`:

```
>>> pip install geopandas
```

However, Geopandas can be notoriously difficult to install. This is especially the case if the python environment is not carefully maintained. Some of its dependencies can also be very difficult to install on certain systems. Because of this, using Colab for this lab is recommended; its environment is set up in a way that makes GeoPandas very easy to install. Otherwise, the GeoPandas documentation contains some additional options that can be used if installation difficulties occur: `https://geopandas.org/install.html`.

GeoSeries

A GeoSeries is a pandas Series where each entry is a set of geometric objects. There are three classes of geometric objects inherited from the Shapely package:

1. Points / Multi-Points
2. Lines / Multi-Lines
3. Polygons / Multi-Polygons

A point is used to identify objects like coordinates, where there is one small instance of the object. A line could be used to describe objects such as roads. A polygon could be used to identify regions, such

as a country. Multipoints, multilines, and multipolygons contain lists of points, lines, and polygons, respectively.

Since each object in the GeoSeries is also a Shapely object, the GeoSeries inherits many methods and attributes of Shapely objects. Some of the key attributes and methods are listed in Table 1.1. These attributes and methods can be used to calculate distances, find the sizes of countries, and determine whether coordinates are within country's boundaries. The example below uses the attribute `bounds` to find the maximum and minimum coordinates of Egypt in a built-in GeoDataFrame.

Method/Attribute	Description
`distance(other)`	returns minimum distance from GeoSeries to `other`
`contains(other)`	returns `True` if shape contains `other`
`intersects(other)`	returns `True` if shape intersects `other`
`area`	returns shape area
`convex_hull`	returns convex shape around all points in the object
`bounds`	returns the bounding x- and y-coordinates of the object

Table 1.1: Attributes and Methods for GeoSeries

```
>>> import geopandas as gpd
>>> world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
# Get GeoSeries for Egypt
>>> egypt = world[world['name']=='Egypt']

# Find bounds of Egypt
>>> egypt.bounds
         minx    miny         maxx      maxy
47   24.70007    22.0     36.86623  31.58568
```

Creating GeoDataFrames

The main structure used in GeoPandas is a GeoDataFrame, which is similar to a pandas DataFrame. A GeoDataFrame has one special column called `geometry`, which must be a GeoSeries. This GeoSeries column is used when a spatial method, like `distance()`, is used on the GeoDataFrame.

A GeoDataFrame can be made from a pandas DataFrame. At least one of the columns in the DataFrame should contain geometric information. This column containing geometric information can be converted to a GeoSeries using the `apply()` method. At this point, the Pandas DataFrame can be cast as a GeoDataFrame. Assign which column will be the `geometry` using either the `geometry` keyword in the constructor or the `set_geometry()` method afterwards.

```
>>> import pandas as pd
>>> import geopandas as gpd
>>> from shapely.geometry import Point, Polygon

# Create a Pandas DataFrame
>>> df = pd.DataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
```

```
...                    'Latitude': [37.57, -12.05, -26.20],
...                    'Longitude': [126.98, -77.04, 28.04]})

# Create geometry column
>>> df['Coordinates'] = list(zip(df.Longitude, df.Latitude))

# Make geometry column Shapely objects
>>> df['Coordinates'] = df['Coordinates'].apply(Point)

# Cast as GeoDataFrame
>>> gdf = gpd.GeoDataFrame(df, geometry='Coordinates')

# Equivalently, specify the geometry after construction
# Note that set_geometry() returns a new GeoDataFrame
>>> gdf = gpd.GeoDataFrame(df)
>>> gdf = gdf.set_geometry('Coordinates')

# Display the GeoDataFrame
>>> gdf
           City       Country  Latitude  Longitude                  Coordinates
0         Seoul   South Korea     37.57     126.98   POINT (126.98000 37.57000)
1          Lima          Peru    -12.05     -77.04  POINT (-77.04000 -12.05000)
2  Johannesburg  South Africa    -26.20      28.04   POINT (28.04000 -26.20000)

# Create a polygon with all three cities as points
>>> city_polygon = Polygon(list(zip(df.Longitude, df.Latitude)))
```

A `GeoDataFrame` can also be made directly from a dictionary. If the dictionary already contains geometric objects, the corresponding column can be directly set as the `geometry` in the constructor. Otherwise, a column containing geometry data can be created as in the above example and then set as the `geometry` with the `set_geometry()` method.

```
# Both of these methods create the same GeoDataFrame as above
# Directly create the GeoDataFrame from the dictionary
>>> gdf = gpd.GeoDataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
...                    'Latitude': [37.57, -12.05, -26.20],
...                    'Longitude': [126.98, -77.04, 28.04]})
# Create geometry column and set as the geometry
>>> gdf['Coordinates'] = list(zip(df.Longitude, df.Latitude))
>>> gdf['Coordinates'] = df['Coordinates'].apply(Point)
# inplace=True modifies gdf itself rather than returning a copy
>>> gdf.set_geometry('Coordinates', inplace=True)

# Equivalently, using a dictionary that already contains geometry objects
>>> gdf = gpd.GeoDataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
...                    'Coordinates': [Point(126.98,37.57),
...                        Point(-77.04,-12.05), Point(28.04,-12.05)]},
```

```
...                     geometry='Coordinates')
```

> **Note**
>
> Longitude is the angular measurement starting at the Prime Meridian, 0°, and going to 180° to the east and −180° to the west. Latitude is the angle between the equatorial plane and the normal line at a given point; a point along the Equator has latitude 0, the North Pole has latitude +90° or 90°N, and the South Pole has latitude −90° or 90°S.

Plotting GeoDataFrames

Information from a GeoDataFrame is plotted based on the geometry column. Data points are displayed as geometry objects. The following example plots the shapes in the `world` GeoDataFrame.

```
# Plot world GeoDataFrame
>>> world.plot()
```

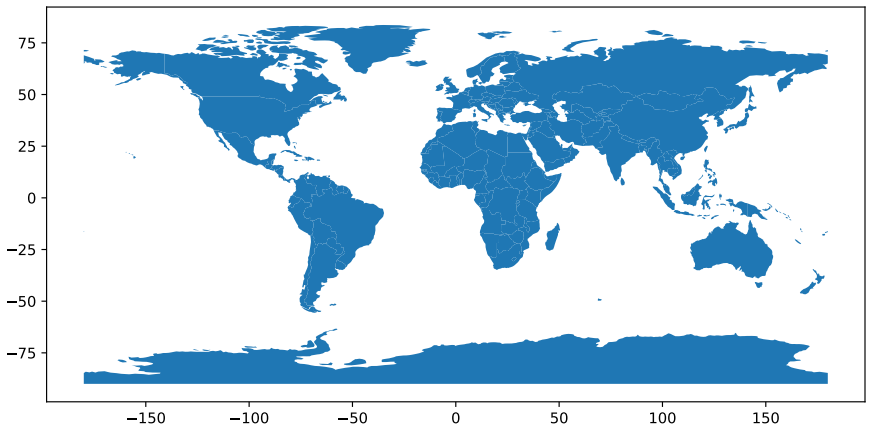

Figure 1.1: World map

Multiple GeoDataFrames can be plotted at once. This can be done by by setting one GeoDataFrame as the base of the plot and ensuring that each layer uses the same axes. In the following example, the file `airports.csv`, containing the coordinates of world airports, is loaded into a GeoDataFrame and plotted on top of the boundary of the `world` GeoDataFrame.

```
# Set outline of world countries as base
>>> fig,ax = plt.subplots(figsize=(10,7), ncols=1, nrows=1)
>>> base = world.boundary.plot(edgecolor='black', ax=ax, linewidth=1)

# Load airport data and convert to a GeoDataFrame
>>> airports = pd.read_csv('airports.csv')
>>> airports['Coordinates'] = list(zip(airports.Longitude, airports.Latitude))
```

```
>>> airports['Coordinates'] = airports.Coordinates.apply(Point)
>>> airports = gpd.GeoDataFrame(airports, geometry='Coordinates')

# Plot airports on top of world map
>>> airports.plot(ax=base, marker='o', color='green', markersize=1)
>>> ax.set_xlabel('Longitude')
>>> ax.set_ylabel('Latitude')
>>> ax.set_title('World Airports')
```

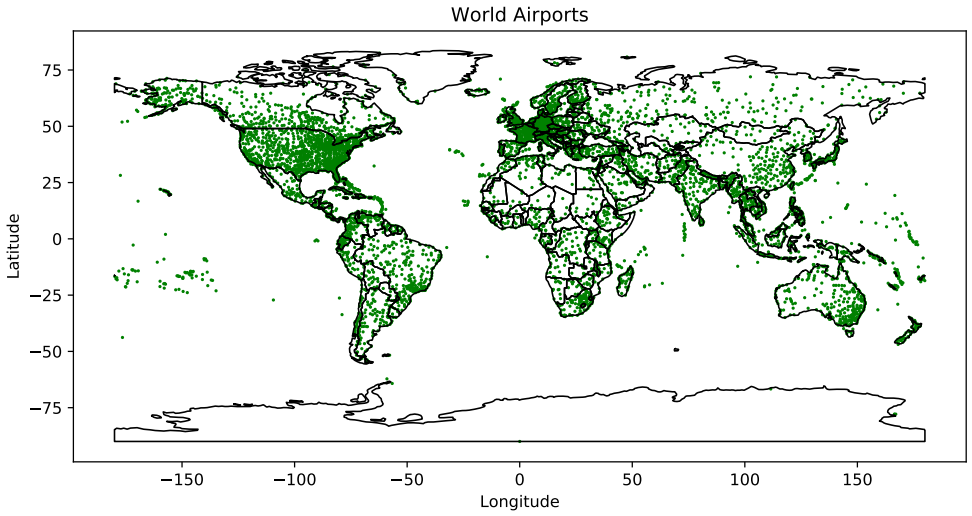

Figure 1.2: Airport map

Problem 1. Read in the file `airports.csv` as a pandas DataFrame. Create three convex hulls around the three sets of airports listed below. This can be done by passing in lists of the airports' coordinates to a `shapely.geometry.Polygon` object.

Create a new GeoDataFrame with these three Polygons as entries. Plot this GeoDataFrame on top of an outlined world map.

- Maio Airport, Scatsta Airport, Stokmarknes Skagen Airport, Bekily Airport, K. D. Matanzima Airport, RAF Ascension Island
- Oiapoque Airport, Maio Airport, Zhezkazgan Airport, Walton Airport, RAF Ascension Island, Usiminas Airport, Piloto Osvaldo Marques Dias Airport
- Zhezkazgan Airport, Khanty Mansiysk Airport, Novy Urengoy Airport, Kalay Airport, Biju Patnaik Airport, Walton Airport

Working with GeoDataFrames

As previously mentioned, GeoDataFrames contain many of the functionalities of pandas DataFrames. For example, to create a new column, define a new column name in the GeoDataFrame with the needed information for each GeoSeries.

```
# Create column in the world GeoDataFrame for gdp_per_capita
>>> world['gdp_per_cap'] = world.gdp_md_est / world.pop_est
```

GeoDataFrames can utilize many pandas functionalities, and they can also be parsed by geometric manipulations. For example, a useful way to index GeoDataFrames is with the `cx` indexer. This splits the GeoDataFrame by the coordinates of each geometric object. It is used by calling the method `cx` on a GeoDataFrame, followed by a slicing argument, where the first element refers to the longitude and the second refers to latitude.

```
# Create a GeoDataFrame containing the northern hemisphere
>>> north = world.cx[:, 0:]

# Create a GeoDataFrame containing the southeastern hemisphere
>>> south_east = world.cx[0:, :0]
```

GeoSeries objects in a GeoDataFrame can also be dissolved, or merged, together into one GeoSeries based on their geometry data. For example, all countries on one continent could be merged to create a GeoSeries containing the information of that continent. The method designed for this is called `dissolve`. It receives two parameters, `by` and `aggfunc`. `by` indicates which column to dissolve along, and `aggfunc` tells how to combine the information in all other columns. The default `aggfunc` is `first`, which returns the first application entry. In the following example, we use `sum` as the `aggfunc` so that each continent is the combination of its countries.

```
>>> world = world[['continent', 'geometry', 'gdp_per_cap']]

# Dissolve world GeoDataFrame by continent
>>> continent = world.dissolve(by = 'continent', aggfunc='sum')
```

Projections and Coloring

When plotting, GeoPandas uses the CRS (coordinate reference system) of a GeoDataFrame. This reference system indicates how coordinates should be spaced on a plot. Two of the most commonly used CRSs are EPSG:4326 and EPSG:3395. EPSG:4326 is the standard latitude-longitude projection used by GPS. EPSG:3395, also known as the Mercator projection, is the standard navigational projection.

When creating a new GeoDataFrame, it is important to set the `crs` attribute of the GeoDataFrame. This allows any plots to be shown correctly. Furthermore, GeoDataFrames being layered need to have the same CRS. To change the CRS, use the method `to_crs()`.

```
# Check CRS of world GeoDataFrame
>>> print(world.crs)
epsg:4326
```

```
# Change CRS of world to Mercator
# inplace=True ensures that we modify world instead of returning a copy
>>> world.to_crs(3395, inplace=True)
>>> print(world.crs)
epsg:3395
```

GeoPandas accepts many different CRSs; a reference can be found at `www.spatialreference.org`. Additionally, inspecting a given CRS object in the terminal without using `print()` or `str()` can be used to get additional information about a specific CRS:[1]

```
>>> world.crs
<Projected CRS: EPSG:3395>
Name: WGS 84 / World Mercator
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: World between 80°S and 84°N.
- bounds: (-180.0, -80.0, 180.0, 84.0)
Coordinate Operation:
- name: World Mercator
- method: Mercator (variant A)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

GeoDataFrames can also be plotted using the values in the the other attributes of the GeoSeries. The map plots the color of each geometry object according to the value of the column selected. This is done by passing in the parameter `column` into the `plot()` method.

```
>>> fig, ax = plt.subplots(1, figsize=(10,4))
# Plot world based on gdp
>>> world.plot(column='gdp_md_est', cmap='OrRd', legend=True,  ax=ax)
>>> ax.set_title('World Map based on GDP')
>>> ax.set_xlabel('Longitude')
>>> ax.set_ylabel('Latitude')
>>> plt.show()
```

[1] This can also be accomplished using `print(repr(crs))`.

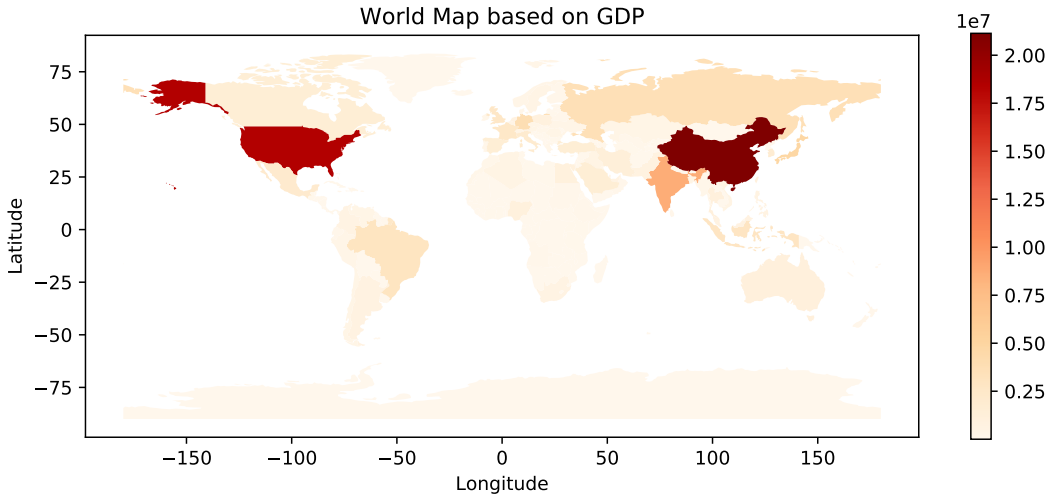

Figure 1.3: World Map Based on GDP

Problem 2. The file `county_data.gpkg.zip` contains information about US counties.[a] After unzipping, use the command `gpd.read_file('county_data.gpkg')` to create a GeoDataFrame of this information. Each county's shape is stored in the `geometry` column. Use this to plot all US counties two times, first using the default CRS and then using EPSG:5071.

Next, create a new GeoDataFrame that merges all counties within a single state. Drop regions with the following STATEFP codes: 02, 15, 60, 66, 69, 72, 78. Plot this GeoDataFrame to see an outline of all 48 contiguous states. Ensure a CRS of EPSG:5071.

[a] Source: `http://www2.census.gov/geo/tiger/GENZ2016/shp/cb_2016_us_county_5m.zip`

Note

`.gpkg` files are actually structured as a directory that contains several files that each contain parts of the data. For instance, `county_data.gpkg` consists of the files `county_data.cpg`, `county_data.dbf`, `county_data.prj`, `county_data.shp`, and `county_data.shx`. Be sure that these files are placed directly in the first level of folders, and not in further subdirectories.
To use this file in Google Colab, upload the zipped file and extract it with the following code:

```
county = files.upload()
!unzip county_data.gpkg.zip
```

It then can be loaded:

```
county_df = gpd.read_file('county_data.gpkg')
```

Merging GeoDataFrames

Just as multiple pandas DataFrames can be merged, multiple GeoDataFrames can be merged with attribute joins or spatial joins. An attribute join is similar to a merge in pandas. It combines two GeoDataFrames on a column (not the geometry column) and then combines the rest of the data into one GeoDataFrame.

```
>>> world = gpd.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
>>> cities = gpd.read_file(geopandas.datasets.get_path('naturalearth_cities'))

# Create subsets of the world and cities GeoDataFrames
>>> world = world[['continent', 'name', 'iso_a3']]
>>> cities = cities[['name', 'iso_a3']]

# Merge the GeoDataFrames on their iso_a3 code
>>> countries = world.merge(cities, on='iso_a3')
```

A spatial join merges two GeoDataFrames based on their geometry data. The function used for this is `sjoin`. `sjoin` accepts two GeoDataFrames and then direction on how to merge. It is imperative that two GeoDataFrames have the same CRS. In the example below, we merge using an `inner` join with the option `intersects`. The `inner` join means that we will only use keys in the intersection of both geometry columns, and we will retain only the left geometry column. `intersects` tells the GeoDataFrames to merge on GeoSeries that intersect each other. Other options include `contains` and `within`.

```
# Combine countries and cities on their geographic location
>>> countries = gpd.sjoin(world, cities, how='inner', op='intersects')
```

Problem 3. Load in the file `nytimes.csv`[a] as a DataFrame. This file includes county-level data for the cumulative cases and deaths of Covid-19 in the US, starting with the first case in Snohomish County, Washington, on January 21, 2020. Begin by converting the `date` column into a `DatetimeIndex`.

Next, use county FIPS codes to merge your GeoDataFrame from Problem 2 with the DataFrame you just created. A FIPS code is a 5-digit unique identifier for geographic locations. Ignore rows in the Covid-19 DataFrame with unknown FIPS codes as well as all data from Hawaii and Alaska.

Note that the `fips` column of the Covid-19 DataFrame stores entries as floats, but the county GeoDataFrame stores FIPS codes as strings, with the first two digits in the `STATEFP` column and the last three in the `COUNTYFP` column.

Once you have completed the merge, plot the cases from March 21, 2020 on top of your state outline map from Problem 2, using the CRS of EPSG:5071. Finally, print out the name of the county with the most cases on March 21, 2020 along with its case count.

[a] Source: `https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv`

Logarithmic Plotting Techniques

The color scheme of a graph can also help to communicate information clearly. A good list of available colormaps can be found at `https://matplotlib.org/3.2.1/gallery/color/colormap_reference.html`. Note also that you can reverse any colormap by adding `_r` to the end. The following example demonstrates some plotting features, using country GDP as in Figure 1.3.

```
>>> fig, ax = plt.subplots(figsize=(15,7), ncols=1, nrows=1)
>>> world.plot(column='gdp_md_est', cmap='plasma_r',
...         ax=ax, legend=True, edgecolor='gray')

# Add title and remove axis tick marks
>>> ax.set_title('GDP on Linear Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

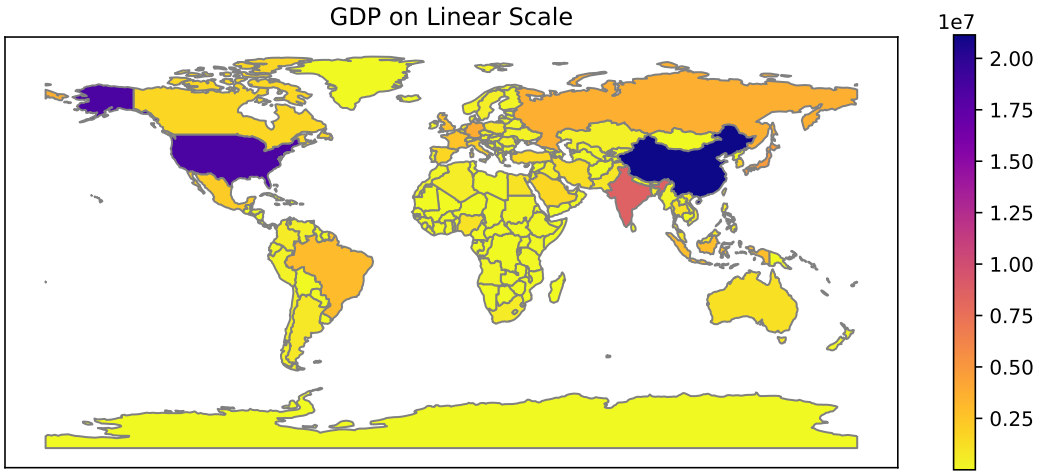

Figure 1.4: World map showing country GDP

Sometimes data can be much more informative when plotted on a logarithmic scale. See how the world map changes when we add a `norm` argument in the code below. Depending on the purpose of the graph, Figure 1.5 may be more informative than Figure 1.4.

```
>>> from matplotlib.colors import LogNorm
>>> from matplotlib.cm import ScalarMappable
>>> fig, ax = plt.subplots(figsize=(15,6), ncols=1, nrows=1)

# Set the norm using data bounds
>>> data = world.gdp_md_est
>>> norm = LogNorm(vmin=min(data), vmax=max(data))

# Plot the graph using the norm
>>> world.plot(column='gdp_md_est', cmap='plasma_r', ax=ax,
```

```
...         edgecolor='gray', norm=norm)

# Create a custom colorbar
>>> cbar = fig.colorbar(ScalarMappable(norm=norm, cmap='plasma_r'),
...         ax=ax, orientation='horizontal', pad=0, label='GDP')

>>> ax.set_title('Country Area on a Log Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

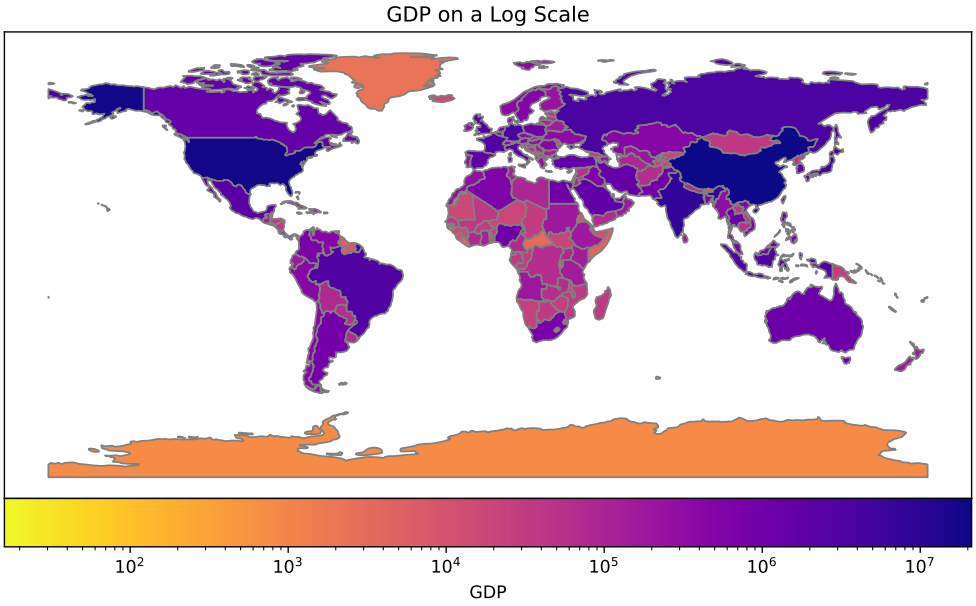

Figure 1.5: World map showing country GDP using a log scale

Problem 4. As in Problem 3, plot your state outline map from Problem 2 on top of a map of the Covid-19 cases from March 21, 2020. This time, however, use a log scale. Use EPSG:5071 for the CRS. Pick a good colormap (the counties with the most cases should generally be darkest) and be sure to display a colorbar.

Problem 5. In this problem, you will create an animation of the spread of Covid-19 through US counties from January 21, 2020 to June 21, 2020. Use a log scale and a good colormap, and be sure that you're using the same norm and colorbar for the whole animation. Use EPSG:5071 for the projection.

As a reminder, below is a summary of what you will need in order to animate this map. You may also find it helpful to refer to the animation section included with the Volume 4 lab

manual.

1. Set up your figure and norm. Be sure to use the highest case count for your `vmax` so that the scale remains uniform.
2. Write your `update` function. This should plot the cases from a given day.
3. Set up your colorbar. Do this outside the `update` function to avoid adding a new colorbar each day.
4. Create the animation and embed it.