THE SEA RISE MAP

HUBERT Margot . LENOIR Rachel . ROMBAUT Doriane . CHATELUS Eolia

General Presentation

This project is a geographical and demographical visualization of the sea level rise and its impacts. It offers an interactive interface that allows users to explore how different IPCC climate scenarios and years affect the emerging land across the globe depending on the year (past and future). Users can also visualize the profile view of France with the elevation of the emerged land and estimate the number of people who could become climate refugees in each continent.
In order to provide an accurate model for the user to visualize the impact of sea level rise, we used real data from GIEC reports (the IPCC scenarios) and the geographical coordinates as well as the population density data come from reliable websites.

Libraries and tools used

· **CustomTkinter** for GUI design (more esthetic than usual Tkinter)

· **netCDF4** and **NumPy** to process data

· **Pandas** for structured data manipulation.

· **Shapely** for geometric operations (polygon creation and testing with simple shapes)

· **Matplotlib** and **PIL** to model the map and manipulate the images

Project Architecture Overview

The project is organized using a Model-View-Controller structure, as we studied in class
- The **Model classes** (including ElevationData, SeaLevel, and refugee estimation) are used for the computations data manipulation.
- The **View classes** (including MainView, ProfileView, and SecondaryView) allow the display of the maps, the profile view and handle the user interactions
- The **Controller class** is used to coordinate all the previous classes : it connects the user actions from the GUI with data computation and visualization.

Usage instruction (to be found in read me)

To run the program

1. To install all the necessary libraries write *‘**pip install customtkinter netCDF4 numpy pandas pillow shapely matplotlib**’* restart the kernel
2. execute the MainView class. This will start the main user interface of the simulation.

To display the map of emerged land

1. **Year Selection :** Use the slider or the arrow buttons to select a year. The selected year is automatically rounded to the nearest multiple of 5.
2. **Selecting a Climate Scenario :** choose one of the four available IPCC scenarios. Each scenario corresponds to a different sea level rise model.
3. **Generating the Map :** Click on the "Generate Map" button to display the global map showing areas still above sea level, based on the selected year and scenario. This process can be repeated for any other year or scenario.
4. **Zooming and Navigating the Map :** Use the mouse scroll wheel to zoom in or out on the map. This allows for closer inspection of specific regions. The image will automatically pan to remain centered during resizing or zooming.

To display the profile view of France

1. Click on France on the map to access a **profile view** showing emerged land with respect to the sea level. This view includes a vertical scale with elevation and a sea level line.
2. You can change the year thanks to the slider while in this view to observe variations over time.
3. To return to the main map, click the **"Quit"** button.

To compute the number of climate refugees

1. Click the **“Show Refugees”** button (available only for years beyond 2022). The application calculates and displays the number of people displaced due to land loss from sea level rise, calculated by multiplying the submerged land area on each continent by its average population density.

To exit the application

1. To close the application, simply close the window (top-right "X" button).

Class Description

1. ElevationData

Purpose: This class manages coordinate and elevation data, click detection (using polygons), and builds data for profile visualizations. It is basically the data provider for the Profile view and it detects clicks from the user.

Methods and purpose :

- **__init__**: Loads elevation data from NetCDF files and polygon coordinates from CSVs.
- **create_elevation()**: Converts raw NetCDF elevation data into a dictionary of the form (latitude, longitude, elevation) for fast access.
- **create_polygon(file)**: Reads coordinate data from a CSV and creates a Polygon object using shapely.geometry.polygon.
- **test_if_point_in(coords, polygon)**: Checks whether a given latitude and longitude is inside a polygon.
- **build_dico_per_long()**: Groups French elevation data by longitude and calculates the average elevation for each longitude. The output dictionary is used in Profile view class

---

2. MainView

Purpose: This class handles user interface interactions and layout using CustomTkinter. It displays map views, receives input from users (choice of the year and scenario, clicks…). It also calls the Profile, secondary view or refugee calculations.

Methods:

- **get_user_year() and get_ipcc_value()**: get the current user selection.
- **show_map() and generate_map_canvas()**: Create and display maps from SecondaryView.
- **count_refugees()**: Calculate and display the number of climate refugees.
- **exit_profile_view() and change_mode_value()**: Handle transitions between map and profile views.

---

3. ProfileView

Purpose: Displays the profile elevation view of a France, showing how much of the land is emerged as compared to the sea level. This class could have worked for other countries, but it would have required a great amount of data computations, which makes the code slower.

Methods:

- **draw_profile()**: Prepares and displays profile.
- **redraw()**: Re-display the image if canvas is resized.
- **on_resize()**: Re-displays automatically when window size changes.

- **load_sky_image()**: Loads the optional background.

---

4. SeaLevel

Purpose: Provides sea level height based on the selected year and IPCC scenario. Data can be precomputed (from a CSV) or dynamically generated using fitted polynomial functions. It is called both by main view and secondary view to get the sea level.

Methods:

- **load_data_sea_level()**: Loads sea level rise data.
- **retrieve_sea_level(year, scenario)**: Chooses the proper sea level computation method based on user scenario.
- **Scenario Models (compute_sea_level_1/2/3/4)**: Polynomial functions that simulate different IPCC trajectories.

---

5. SecondaryView

Purpose: displays the main world map showing emerged and submerged areas using pixel RGB data from NetCDF.

Methods:

- **generate_base_image()**: Prepares the base image only once or when sea level changes.
- **redraw()**: Scales and displays the map.
- **on_zoom()**: Manages zooming while maintaining cursor focus.
- **on_resize()**: Centers the image.
- **on_click()**: Calculates image coordinates and notifies the controller.
- **create_map()**: Sets up and binds the canvas for map display.

---

6. Refugee Estimation Logic

Purpose: Estimate the number of climate refugees based on geographical flooding and other climate events. To ease the computations, some assumptions were made : population density and growth are static, only land submerged beyond a certain sea level threshold is considered, and region association is determined by polygon-based inclusion.

Methods:

- **compute_refugees()**: Multiplies submerged land area by population density and growth rate for each continent.
- **estimate_other_climatic_refugees()**: Adds people displaced by droughts, heatwaves, wildfires, and floods.

Implementation

How we proceeded

- we collected the data : NetCDF elevation data from ETOPO (2022 dataset), coordinate CSVs for France and test zones.
- we coded the program with the classes
- we debugged and tested it (the longest part) : the set of tests can be found on on GitHub. We tested that border and outside points are handled properly using shapely.
- we improved the interface by making it more visually appealing (colors, CustopTkinter…).

Difficulties encountered

One of the main challenges we faced during the development of our project was ensuring consistency and compatibility between the different modules coded by each team member. It was essential that the outputs of one function matched the expected inputs of another, both in terms of format and structure. While we initially defined shared interfaces and documented functions with docstrings to guide our collaboration, we often encountered unexpected implementation constraints that required us to adjust the code on one side or another. Despite planning, certain technical details such as data types, object structures, or edge cases led to misalignments. As a result, we had to regularly revisit and revise our functions, communicate actively, and test the interactions between components to ensure consistency. This aspect highlighted the importance of coordination in collaborative coding, especially in a project involving multiple interconnected parts.

Another significant challenge we encountered was gathering and preparing the data required for our application, particularly the geographical coordinate data. In many cases, the available datasets came in complex or specialized formats (such as NetCDF, shapefiles, …), which could not be used directly. This forced us to write conversion codes to transform the raw data into more accessible formats like CSV. Additionally, some of the files, especially those with high spatial resolution, were very large and heavy, which caused serious slowdowns during execution and map rendering. To address this, we had to reduce the resolution or manually simplify certain datasets while trying to preserve the accuracy needed for meaningful results. This step was crucial to make the data both exploitable and compatible with our program.

AI usage

AI was used to improve the function "generate_base_image".

This function associates to each point on Earth a color depending on whether it is above or below the sea level value for a given year. At first, we created this function using a dictionary, in which we loaded the data from the .nc file containing the elevation, latitude, and longitude of all points on Earth.

The .nc file has a resolution of 60 arc minutes, so 1°. The dictionary elevation_dict has the following form: elevation_dict = { elevation1: [(lat_a, long_a), (lat_b, long_b), ...], elevation2: [(lat_c, long_c), (lat_d, long_d), ...] , ... }. So each elevation is associated with a list of tuples corresponding to the coordinates (latitude, longitude) of all points at the given elevation. To create the dictionary elevation_dict, we used a 5° resolution (so extracting every five points from the file) in order to have a loading time not too high.

Then, with our initial function "generate_base_image", for each elevation of the dictionary, if the elevation is below the sea level, on the canvas, around all points of coordinates stored in the associated list, we plot a circle of radius 10 in blue. Else, if the elevation is above sea level, on the canvas, around all points of coordinates stored in the associated list, we plot a circle of radius 10 in green. This allowed us to obtain the following map: (see generate base map file)

We can see that not all the map is uniformly colored, there are blank spaces, and we distinguish the different colored points. However, when we tried to improve the resolution of the dictionary, the code was running indefinitely and it was impossible to place the million points of the dictionary on the Tkinter canvas.

Thus, we asked AI how to have more precision on our map, without having code running too slowly and without giving too many points to place on the Tkinter canvas. We wrote the following prompt: *"I have a .nc file with a 60 arc minutes resolution containing the elevation, latitude and longitude of all points on Earth. I loaded the data in the dictionary associating the elevation to a list of coordinates (lat, long) at this elevation, keeping a resolution of 5°. With a function generate_base_image, I want to color each point in blue or green on a canvas depending if its elevation is below or above the sea level given as parameter. When running my code, I obtain the following map (photo of the map joined). There are blank spaces with the resolution of 5°, but if I improve the resolution, too many points have to be placed on the canvas and it doesn't work. So how can I refine the coloring of the map to have no blank space, but avoid the code being too slow and avoid a bug because there are too many points?"*

The answer of AI was that creating a dictionary to color the map was not optimal and it was not adapted to send thousands of points to the Tkinter canvas one after the other. Since we have a .nc file with all the coordinates, it is better to use arrays to associate each coordinate with a color, then create the image and send the obtained image all at once to the Tkinter canvas. The function generate_base_image in our final code thus uses the function written by AI, which we adapted in order for it to work within our class SecondaryView.

AI was also used to create the functions on_resize and on_zoom in the class SecondaryView. They are respectively used to adapt the size of the canvas containing our map to the size of the window opened by the user, and to allow zooming (or unzooming) on the map when scrolling with the mouse. We didn't know how to write this code, so we asked AI.

We wrote the following prompt: *"Can you provide two functions: one allowing to adapt the size of the canvas in which the map of the Earth (created by the function generate_base_image) is generated depending on the size of the window opened by the user, and the other allowing to zoom on the map where the mouse cursor is placed and unzoom, using the scroll of the mouse?"*

Conclusion

This project combines climate modeling, geospatial data processing, and interactive visualization to simulate the impact of sea level rise. It offers an interface for users to explore complex environmental data and estimate potential climate refugee numbers. Beyond its educational benefits, working on this code taught us the importance of collaboration, data handling, and optimization in building effective programs.