RA Application (Joaquin Vilchez)

Part I — Setting up the spatial framework for Punjab

The first step of my project focused on defining the spatial reference for the entire analysis. I started by using the GADM 4.1 database, which provides official administrative boundaries for India. The goal was to extract the state of Punjab, determine its geographic limits, and align these limits with the ERA5 meteorological grid, which operates on a 0.25° × 0.25° spatial resolution.

To make this process fully reproducible, I structured the code as a clean Python module with a main() function that executes all steps in sequence. The script begins by importing essential geospatial libraries such as geopandas, shapely, and zipfile, as well as core utilities like math, os, and pathlib. These packages allow the program to handle vector data, manipulate geometries, and manage files systematically.

Then, I defined key constants at the top of the script, such as the URL for downloading GADM, the ERA5 grid size (0.25°), and a padding margin of ±0.5°. This margin ensures that edge areas around Punjab are not excluded from the analysis. I also used os.makedirs() to automatically create the folder structure (data/ for raw files and outputs/ for processed results). This structure keeps the workflow organized and reproducible.

To download the GADM data, I implemented a custom function called download_with_retry(). This function was important because it prevents the code from failing due to temporary network issues. It tries to download the file up to three times, and if the ZIP file is incomplete or corrupted, it deletes it and starts again. This makes the workflow stable and avoids errors common in large geospatial projects.

Once the download is validated, the script extracts the ZIP file and searches for the correct shapefile corresponding to the level-1 administrative units (states). Instead of hardcoding filenames, I created a function called find_level1_shp() that automatically detects the right file inside the ZIP folder. This small design decision ensures that if GADM changes its file naming conventions in future versions, the code will still work without manual changes.

After the shapefile is loaded into a GeoDataFrame, I converted its coordinate reference system to WGS84 (EPSG:4326), the standard for geographic coordinates in degrees. I then filtered the dataset to select only the rows where the field "NAME_1" equals "Punjab". Before moving forward, I included a check to confirm that this column exists and that the polygon was successfully retrieved. This type of validation helps avoid silent errors later in the analysis.

Finally, I calculated Punjab's minimum and maximum latitude and longitude, printed them to the console for transparency, and saved the polygon as a GeoJSON file. I also applied the "snapping" process: I expanded the bounding box by ±0.5° and aligned the edges to the ERA5 grid using floor and ceil functions. This ensures that my area of study fits perfectly with the meteorological dataset and that no rounding errors occur. The output shows the final coordinates in both bounding-box and geographic (North, West, South, East) formats, which confirms the exact region I used in the following steps.

Part II — Downloading and preparing ERA5 wind data

After defining the spatial extent of Punjab, I moved to the meteorological data collection stage. This part of the project aimed to download monthly wind data (u and v components) for 2023 using the Copernicus Climate Data Store (CDS) API.

To do this, I created a configuration block that defines the study year, output paths for the NetCDF and CSV files, and the bounding box calculated earlier. The code requests data at two pressure levels — 700 hPa and 850 hPa — because these represent lower-atmospheric layers relevant to air transport and pollution dispersion.

To ensure the download process was reliable, I wrote a helper function called download_nc_safely(). This function downloads the dataset into a temporary .part file and then renames it once the transfer completes successfully. This avoids situations where partial downloads could cause corrupted files. The same function also handles cases where the target file might be locked by another process, providing a safe fallback method.

Once the NetCDF file is available, I opened it using the xarray library with the netcdf4 engine. Before doing any calculations, the code performs a series of checks to confirm that all required variables (u and v) and dimensions (latitude, longitude, time, and pressure_level) exist in the dataset. If any of these are missing, the script stops immediately and raises a clear error message. This step acts like an “early warning system” for potential issues.

With the structure verified, I used NumPy to calculate wind speed and wind direction (from where the wind blows). The wind speed is computed using np.hypot(u, v), which avoids floating-point errors, and the direction is obtained with

$$direction_{from} = (\text{degrees}(\arctan 2(-u, -v)) + 360) \mod 360.$$

This formula is standard in meteorology and converts the mathematical “to” direction into a meteorological “from” direction (for example, a wind coming from the north has 0°, from the east 90°, etc.).

Finally, I exported two versions of the dataset: a raw NetCDF file (for reproducibility) and an enriched NetCDF with the derived variables (wind_speed, wind_direction_from). Additionally, I flattened the dataset into a tidy CSV file, which includes only the essential columns needed for later merges. This makes it easier to visualize or verify data in Excel or pandas without opening large binary files.

Part III — Processing satellite fire detections (FIRMS)

The next step focused on building the fire panel for Punjab using NASA’s FIRMS API. I wrote this code carefully to handle large data requests efficiently. FIRMS restricts queries to about ten days per request, so I designed a loop that iterates through the entire year in 10-day chunks. Each iteration calls the API for both VIIRS sensors (SNPP and NOAA-20) and appends the responses to a list of DataFrames. After the loop finishes, I concatenate everything into a single dataset containing all fires detected in 2023.

To maintain transparency, I saved this consolidated CSV file in the data/firms folder. Then, to ensure that I was analyzing only fires within Punjab, I clipped the detections using the official SHRUG district boundaries. This clipping step replaces the rough bounding box with precise administrative boundaries, ensuring that no fires from neighboring states (like Haryana or Himachal Pradesh) are accidentally included.

Because different satellites can detect the same fire multiple times, I implemented a de-duplication algorithm. It rounds latitude and longitude coordinates to three decimal places (≈100 meters) and groups detections into 30-minute time intervals. Within each group, it keeps the record with the highest confidence score and drops the others. This method removes redundant detections and improves data quality without losing real fire events.

Finally, I grouped the data by year and month to produce monthly fire totals for all of Punjab — a quick diagnostic to check seasonality. These totals were then merged with the subgrid polygons created in the previous step, assigning each fire to the correct subgrid. After aggregation, the code fills missing months with zeros to ensure that each subgrid has twelve months of data. This guarantees a balanced panel, which is essential for fixed-effects econometric models later in the analysis.

Part IV — Constructing subgrids and attaching wind data

In this part, I created the ERA5 subgrids — the building blocks of the spatial analysis. Since ERA5 provides data at grid centers (latitude, longitude), I first calculated the edges of each cell by taking midpoints between adjacent coordinates. Using these edges, I constructed polygons that represent each ERA5 grid cell.

Then, I subdivided each of these polygons into 25 smaller subgrids (5×5) to achieve higher spatial detail. This allows me to capture more local variation in fires and wind direction. I only kept subgrids that intersect with the Punjab polygon to reduce unnecessary computation.

Next, I attached wind information to each subgrid. To do this efficiently, I mapped each subgrid to its parent ERA5 cell using grid indices (grid_i, grid_j). Each subgrid inherits its parent cell's u, v, and speed values. This step avoids interpolation errors and keeps the values physically consistent with the ERA5 data.

Finally, I saved the subgrid wind time series as a CSV file that includes all relevant information for each subgrid, month, and pressure level. I also implemented an optional visualization step where I built wedge-shaped "sectors" to represent the wind direction graphically. These sectors are generated using the shapely library in a metric CRS (UTM 43N), which allows for accurate area and distance measurements. The output GeoPackages can be opened in QGIS to visually confirm that the wind fields align properly with Punjab's geography.

Part V — Constructing the "Downwind" metric for each district

Once I had the subgrid wind and fire datasets, the next step was to build a spatial indicator that identifies whether each grid cell or district was "downwind" or "upwind" during a given month. This metric is crucial for the difference-in-differences (DiD) analysis because it acts as the *treatment variable*—it tells us whether a location was exposed to airflows carrying potential pollutants from elsewhere.

The script begins by loading the SHRUG district boundaries, filtering them so that only districts with pc11_state_id == "03" (Punjab) are kept. Using numerical IDs rather than text matching prevents spelling errors or encoding mismatches. Both the districts and subgrids are then projected to a metric coordinate reference system (UTM Zone 43N, EPSG:32643), which allows for accurate geometric calculations in meters rather than degrees. Working in a metric CRS is essential because all area-based operations—such as calculating intersections and buffers—require linear units.

Next, I attached each subgrid to a single district. The matching was done using the centroid-within rule, meaning that the centroid (center point) of each subgrid must fall inside a district's boundary. This ensures a one-to-one relationship between grids and districts. If a centroid happened to fall exactly on a border (a rare case), the script used a fallback method: it searched for the nearest district polygon and assigned it accordingly. This logic avoids ambiguous cases and ensures no subgrid remains unclassified.

To approximate, the script uses a large rectangular polygon projected from each subgrid's centroid, oriented according to the wind direction. This rectangle extends several thousand kilometers forward (5,000,000 m in my case) and sideways, forming an artificial "downwind" zone that mimics the half-space in front of the wind vector. I chose a rectangle instead of a wedge because rectangles are computationally more stable—wedges can cause degenerate geometries and unreliable area measurements near the apex.

For each subgrid-month combination, the script intersects this rectangle with the corresponding district polygon. The intersection gives the downwind area, while the remainder of the district polygon corresponds to the upwind area.
I then compute:

$$share_{down} = \frac{\text{downwind area}}{\text{downwind area + upwind area}}$$

and set the binary variable downup = 1 if share_down > 0.5, meaning that most of the district area lies downwind of that grid's centroid.
All these calculations were parallelized using the Joblib library, which distributes the computations across multiple CPU cores. This significantly reduced runtime, since Punjab contains thousands of grid-month combinations.

At the end of this process, the script produces three datasets:

1. downup_by_grid_month_district.csv — detailed results at the grid–month level;
2. downup_district_month_majority.csv — a district–month summary where a district is labeled downwind if more than half of its subgrids were downwind;
3. downup_district_month_mean_share.csv — an alternative summary based on the mean share of downwind areas per district.

Finally, the code performs a sanity check: for each district, it verifies that the sum of downwind and upwind areas equals the district's total area (within negligible rounding error). The standard deviation of this total area was nearly zero, confirming that the geometric logic and CRS projections were consistent.

Part VI — Visualizing the Downwind–Upwind patterns

After generating the "downwind" classification, I wanted to visualize the temporal evolution of exposure for each district in a clear, interpretable format.
For that, I wrote a short script in matplotlib that produces a district × month heatmap, summarizing whether each district was mostly downwind or upwind in every month of 2023.

The code first checks if the district-level mean-share file exists (downup_district_month_mean_share.csv). If it does, the script uses it because it better

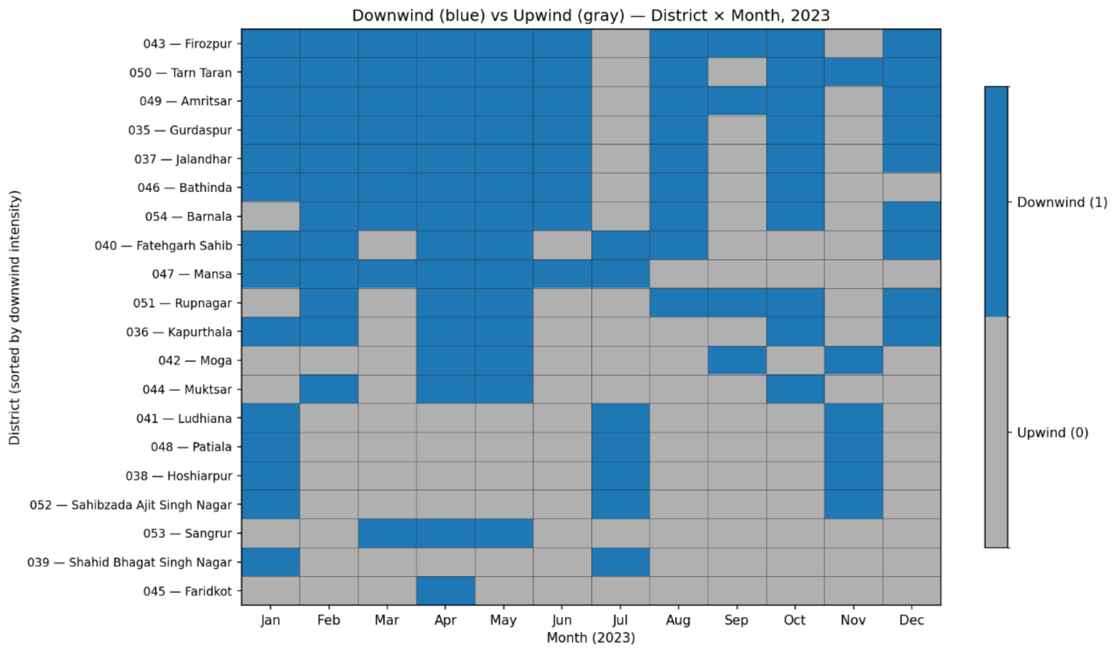
represents the *intensity* of downwind exposure. Otherwise, it falls back to the majority rule file (downup_by_grid_month_district.csv).
In either case, the output is standardized into a table where each district-month combination is labeled as 1 (downwind) or 0 (upwind).

Then, I built a tidy "panel" by pivoting the data with districts as rows and months as columns. Missing entries (e.g., no data for a given month) are preserved as NaN, which are displayed as white cells in the heatmap. Before plotting, I computed a "downwind intensity" score for each district, defined as the mean of its binary values across all months. This score was used to sort the rows from the most frequently downwind district to the least. That sorting step makes the figure much more intuitive.

The plotting section uses matplotlib's imshow() with a discrete color map:

- Gray (0) → upwind;
- Blue (1) → downwind;
- White → missing values.
  The color normalization is defined by BoundaryNorm([-0.5, 0.5, 1.5]) to keep the colors purely categorical. The figure automatically adjusts its height to fit the number of districts, ensuring that all labels are readable. Thin gridlines ("rayitas") are added for visual separation of cells.

The final heatmap shows the pattern clearly: districts such as Firozpur, Tarn Taran, Amritsar, and Gurdaspur appear predominantly blue, meaning they were downwind most of the year, while Faridkot, Sangrur, and Shahid Bhagat Singh Nagar show alternating gray and blue cells, indicating more mixed exposure.



Part VII — Difference-in-Differences (DiD) estimation at the subgrid level

With the wind exposure and fire counts aligned on the same subgrid-month panel, I applied a Difference-in-Differences (DiD) regression to test whether being downwind increases the

number of fires.
The statistical model is:

$$fires_{g,m} = \alpha + \beta \cdot downup_{g,m} + \mu_g + \lambda_m + \varepsilon_{g,m},$$

where:

- $fires_{g,m}$is the number of fires in grid $g$during month m;
- $downup_{g,m}$is 1 if the grid is downwind that month;
- $\mu_g$and $\lambda_m$are grid and month fixed effects.

I implemented this model using the PanelOLS class from the linearmodels package, which handles two-way fixed effects efficiently.
The data were first merged on (subgrid_id, year, month) to align fires and treatment status. Fire counts were filled with zeros to keep the panel balanced. Then, I set a multi-index [subgrid_id, month] to define the panel structure.

The model automatically controls for all time-invariant spatial differences (through μ_g) and for any shocks affecting all grids simultaneously (through λ_m).
I also applied two-way clustered standard errors—by grid and by month—to make inference robust against both spatial and temporal correlation.

The results show a coefficient of –0.27 with a standard error of 0.19 and a p-value of 0.15. The coefficient's negative sign means that, on average, grids that were downwind had *slightly fewer fires*, but the effect is not statistically significant. In other words, we cannot conclude that being downwind has a measurable causal effect on fire activity at the subgrid level.

This finding may reflect that the physical link between local wind direction and fire ignition is weak at the monthly scale, or that fires are primarily driven by crop cycles and policies rather than short-term meteorological changes.

Part VIII — District-level DiD: "Self-pollution" analysis

As a robustness check, I replicated the DiD logic at the district-month level, where the treatment variable (SelfDownwind) indicates whether a district was downwind of its own area—that is, whether local winds were blowing from one part of the district toward another.
The model estimated is:

$$fires_{d,m} = \alpha + \beta \cdot SelfDownwind_{d,m} + \mu_d + \lambda_m + \varepsilon_{d,m},$$

with fixed effects for districts (μ_d) and months (λ_m), and two-way clustered standard errors (district, month).

This regression used 240 observations (20 districts × 12 months).
The coefficient on SelfDownwind was –175.4, with a standard error of 177.4 and a p-value of 0.32.
As in the previous model, the effect is negative but statistically insignificant. This means that when a district is downwind of itself, it does not experience a systematically higher or lower number of fires compared to other months.

Economically and environmentally, this result makes sense: intra-district winds may influence pollution dispersion, but not necessarily fire ignition.