****

Prepared by: The Aquaya Institute

Last updated: May 2023

SanPlan: The Sanitation Planning Tool

Operating Manual

# Table of Contents

[Table of Contents 1](#_Toc135073443)

[Introduction to SanPlan 2](#_Toc135073444)

[KEY FUNCTIONALITIES 2](#_Toc135073445)

[Architecture overview 3](#_Toc135073446)

[Overall structure 3](#_Toc135073447)

[Technologies used 4](#_Toc135073448)

[Flow of information 4](#_Toc135073449)

[Detailed components 6](#_Toc135073450)

[Getting started 7](#_Toc135073451)

[Installing the codebase 7](#_Toc135073452)

[Technologies used 7](#_Toc135073453)

[Key steps 7](#_Toc135073454)

[Data treatment environment 8](#_Toc135073455)

[Technologies used 8](#_Toc135073456)

[Key steps 8](#_Toc135073457)

[Updating the site: Standard Operating Procedures 9](#_Toc135073458)

[Appendix I: Data treatment scripts 17](#_Toc135073459)

[Defining settlement boundaries (non-African countries) 17](#_Toc135073460)

[Extract “towns” and “cities” 19](#_Toc135073461)

[Compute “distance to towns”, “distance to roads”, and “rural typologies” 22](#_Toc135073462)

[Compute settlement, pixel, and administrative boundary zonal statistics 27](#_Toc135073463)

[Add administrative identifiers 33](#_Toc135073464)

[Add pixel and administrative boundary data to settlement layer 35](#_Toc135073465)

[Appendix II: Table of scripts 36](#_Toc135073466)

# Introduction to SanPlan

USAID’s Water, Sanitation, and Hygiene Partnerships and Learning for Sustainability #2 (WASHPaLS #2) project is a 5-year contract awarded to Tetra Tech ARD that aims to strengthen USAID’s sanitation programming and enhance global learning and adoption of evidence-based strategies towards sustainable development goal (SDG) 6.2. The objective of WASHPaLS #2 is to generate and facilitate WASH sector research and learning that result in sustainable, at-scale, and equitable improvements in key sanitation services, behaviors, and environmental conditions at the community and household levels. WASHPaLS #2 builds on the results and learning from WASHPaLS 1, including in the area of sanitation planning.

As governments, donors, and implementing partners collaborate to achieve open-defecation-free (ODF) districts and countries, there is growing recognition within the sector that different contexts call for different approaches to encourage the construction and sustained use of improved sanitation. WASHPaLS 1 demonstrated how existing datasets could help identify local conditions influencing CLTS outcomes (Stuart, et al., 2021). Analyzing and applying information on local conditions can help design more cost-effective rural sanitation interventions.

In an effort to make existing information on local conditions widely available, The Aquaya Institute (Aquaya) developed the Sanitation Planning (SanPlan) tool as part of WASHPaLS 1. SanPlan is an interactive web application that harmonizes data from multiple publicly available sources so that users can create custom maps and download data filtered to their specifications. The SanPlan tool is intended to help sanitation practitioners (program implementers, funders, government institutions, and researchers) design and execute sanitation programs by allowing them to explore highly localized, contextual, spatial data. The primary function of the tool is the visualization of rural typologies (i.e., rural remote, rural on-road, rural mixed, and urban) overlaid with other metrics that are thought to influence sanitation program success, including socioeconomic indicators, accessibility, and estimated levels of WASH access in 18 countries. SanPlan allows implementers to retrieve information about their geographic areas of interest and can aid in determining which interventions best suit them.

An introductory webinar, including a tutorial and discussion, is available at (<https://aquaya.org/sanplan-the-sanitation-planning-tool/>).

## KEY FUNCTIONALITIES

There are five key analysis functionalities embedded in SanPlan:

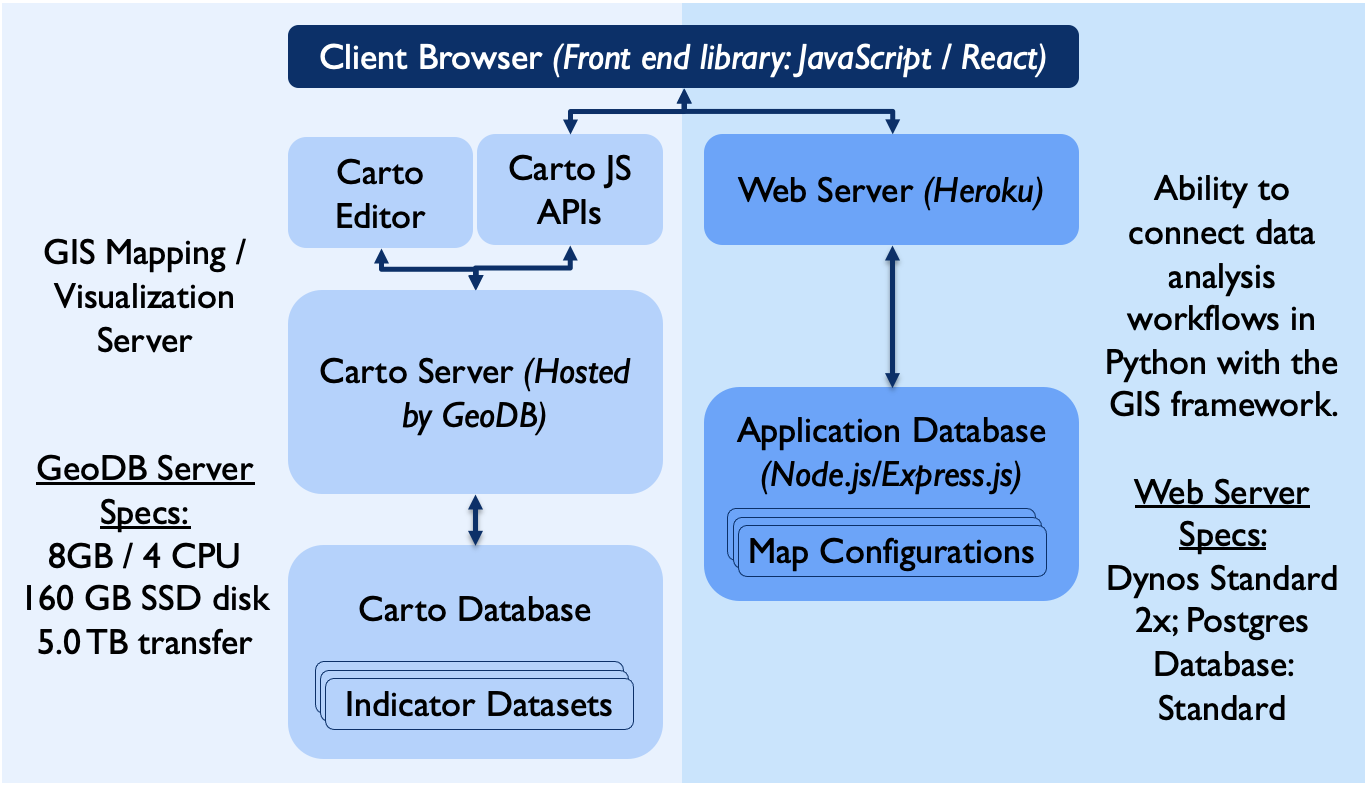
1. **Rural Typology Classification**: All areas within a country have been categorized as either urban, rural mixed, rural on-road, or rural remote. Typologies were determined using population, roadways, and travel time data. Urban areas were defined as those within cities of 50,000 or more people. Rural mixed areas combine urban and rural characteristics; they include peri-urban areas and small towns of at least 5,000 people. Rural on road areas are the remaining areas that are within 1.5-km of a main (trunk, primary, secondary, or tertiary) roadway. Rural remote areas are all those left over, i.e., far from roads, small towns, and cities. The rural typology layer is available at both 1-km and 5-km resolution.
2. **5 km pixel-level filtering**: All variables are combined into a single 5-km pixel dataset that the user can filter to visualize the pixels that meet specified criteria. Users can utilize more than one filter at once to identify locations based on a range of indicators.
3. **Administrative-level filtering** (e.g., district-level in Ghana): All variables are combined into a single administrative dataset that the user can filter to visualize administrative areas (e.g., districts) that meet specified criteria. Users can utilize more than one filter at once to identify locations based on a range of indicators.
4. **User-added communities**: Users can upload and visualize community locations from a CSV file. Other data in the uploaded CSV are also displayed when the user clicks the community marker on the map. The user can export data on all indicators (e.g., rural typology) extracted at uploaded community locations as a CSV file (SanPlan data is appended to the original CSV and downloaded onto the user’s desktop).
5. **Estimated settlement mapping** (Beta): Users can plot the boundaries of estimated settlement areas on top of their customized map to better understand the number and locations of communities. For each settlement, the map also displays all indicators available at sufficiently fine resolution (< 1-km). For the current version, these include population, distance to roads, distance to towns, travel time to cities, and rural typology.

# Architecture overview

## Overall structure

The web application requires servers, databases, and visualization technology to function. These are purchased through third-party vendors and incur monthly or annual costs. The current application architecture and specific information on the technology stack are mapped in Figure 1.

SanPlan is a React application that relies on Carto (<https://carto.com/>) for spatial visualizations and analytics. Carto is a location intelligence, cloud computing platform that provides Geographic Information System (GIS), web mapping, and spatial data science tools. Carto’s codebase is open source, which allows developers to set up their own private server with Carto installed. Carto is also available as a Software as a Service (SaaS), through a paid subscription plan. Due to the high costs associated with both Carto services and setting up a private server, SanPlan currently utilizes a third-party Carto server hosted by GeoDB ([www.getGeoDB.com](http://www.getGeoDB.com)). Future product managers have the option to change the source of the Carto server if they prefer not to use GeoDB.



**Figure 1. SanPlan application architecture and technology stack**. JS = “JavaScript”, API = “Application Programming Interface”.

The JavaScript/React codebase is stored in a GitHub repository, and datasets are stored on GeoDB (Table 1). These datasets are produced from processing raw data files via a series of Python scripts that rely on a consistent naming convention and file tree framework. A Standard Operating Procedure (SOP) for processing the raw data (e.g., for the purpose of adding a country) is provided in Appendix 1.

### Technologies used

|  |  |
| --- | --- |
| **Technology** | **Description** |
| React | Front-end JavaScript library for component-based user interfaces. |
| JavaScript | Programming language. |
| Carto (GeoDB) | Spatial analytics server hosted through GeoDB. GeoDB is a fully-managed geospatial stack built with PostgreSQL, CartoDB and Jupyter. |
| GitHub | Internet hosting service for software development and Git, a version control system that tracks changes in any set of computer files. |
| Heroku | Cloud platform as a service that enables developers to build, run, and operate applications entirely in the cloud. |

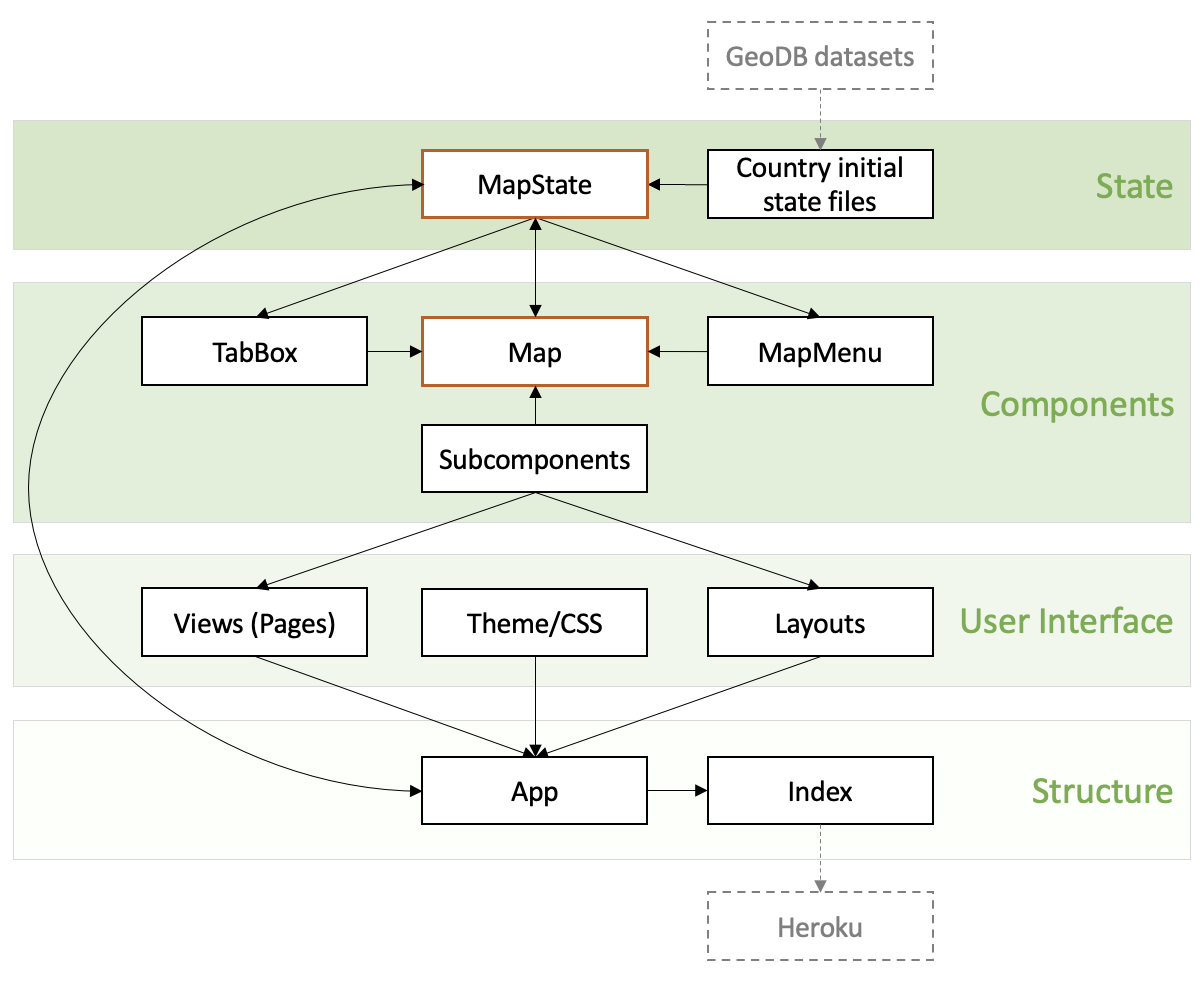
**TABLE 1. PRIMARY TECHNOLOGIES USED IN SANPLAN DEPLOYMENT**

## Flow of information

Information flows through the website from the processed datasets stored on GeoDB to the codebase (Figure 2). The data displayed upon the initial load is predetermined by the country-specific files. These files are updated when the user makes a change. The user selections are stored in the MapState file, which ultimately determines what data is displayed on the map page. The MapState file also stores decisions relating to the user interface (UI), e.g., whether a menu is open or closed, whether a layer is visible or not, etc.

The “components” files, Map, TabBox, MapMenu, and subcomponents make up the building blocks of the site. They design the menus, buttons, filters, export features, etc. The Map file is where all the components are linked. Map.js and MapState.js work together to display the map page and ensure it reacts to user clicks.

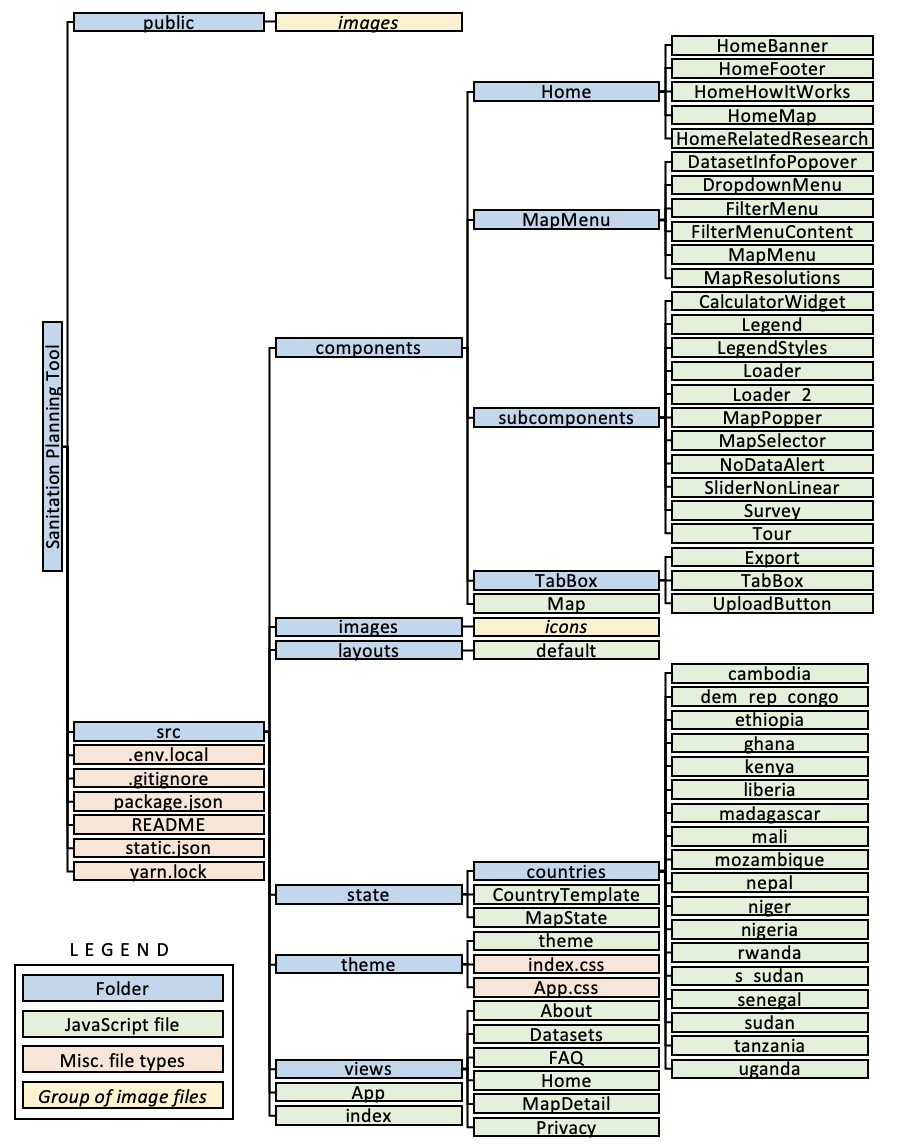
The user interface (UI) files include high-level frameworks of each page and determine style components such as colors, fonts, font sizes, etc. The App and Index files take all the UI, Map, and MapState components and compile the data into a React application website. This packed site is hosted on GitHub and linked to Heroku. Heroku pulls the codebase from GitHub and publishes the site to the web.



**FIGURE 2. SANPLAN INFORMATION FLOW SCHEME.**

## Detailed components

A table with descriptions of each file is located in Appendix 2.



**FIGURE 3. THE SANPLAN CODEBASE FILE TREE.**

# Getting started

## Installing the codebase

### Technologies used

|  |  |
| --- | --- |
| **Technology** | **Description** |
| Node.js | Back-end JavaScript runtime environment, which runs on the V8 JavaScript Engine and executes JavaScript code outside a web browser (https://nodejs.org/en). |
| NVM (Node Version Manager) | A POSIX-compliant bash script to manage multiple active node.js versions. allows you to quickly install and use different versions of node via the command line (https://github.com/nvm-sh/nvm). |
| Yarn | JavaScript package manager for the Node.js JavaScript runtime environment (https://yarnpkg.com/). |
| VSCode | (Optional) Visual Studio Code is a code editor for building and debugging modern web and cloud applications (https://code.visualstudio.com/). |

**TABLE 2. BACK-END TECHNOLOGIES USED FOR SANPLAN DEPLOYMENT.**

### Key steps

1. Clone the repository from GitHub onto your computer’s drive
2. Install a code editor of your choice (e.g., VSCode, Table 2)
3. Install Node.js via a version manager (e.g., NVM)
4. Install Yarn
5. Create a local “environment file” at the root of the repository. Name it `.env.local`. This file is listed in the `.gitignore` script and will be ignored by Git, keeping it out of the repository.
6. In this file, insert the following API and restart the server

```

REACT\_APP\_CARTO\_DEV\_API\_KEY=<DEV\_API\_KEY\_FROM\_CARTO>

REACT\_APP\_CARTO\_USERNAME=<USERNAME>

```

1. In the terminal of the code editor (VSCode), install all the dependencies by running “yarn install”
2. Each time to start the development server, run “yarn start” to open a browser tab which will reload as you edit the codebase.
3. Once you are ready to publish changes to the site, push your local changes to GitHub and redeploy the site via Heroku.

## Data treatment environment

### Technologies used

|  |  |
| --- | --- |
| **Technology** | **Description** |
| Python | Programming language |
| QGIS | QGIS is a desktop geographic information system application that supports viewing, editing, printing, and analysis of geospatial data (https://qgis.org/en/site/). |
| Carto | Spatial analytics server hosted through GeoDB. |
| GitHub | Internet hosting service for software development and Git, a version control system that tracks changes in any set of computer files. |

**TABLE 3. TECHNOLOGIES USED FOR SANPLAN DATA TREATMENT.**

### Key steps

SanPlan currently includes five types of compiled map layers for each country:

1. Country boundary
2. 1-km rural typology layer
3. 5-km multi-indicator layer
4. Administrative boundary layer(s) with multi-indicator data
5. Estimated settlement areas.

The 5-km and administrative boundary layers include all available indicators aggregated to the pixel or boundary area (Table 4). Boundary layers are sub-national administrative zones, e.g., provinces, regions, districts, or counties. The estimated settlement areas layer includes indicators available at < 1 km resolution. All indicators are aggregated using the arithmetic mean except for population estimate, which is summed, and rural typology, which is defined as the most urban typology contained in the feature (pixel, boundary, or settlement area). A step-by-step protocol for data treatment (adding a new country to the tool) is found in Appendix 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Raw Indicator** | **Res.** | **Cov.** | **Source** | **Year** |
| Cholera risk | 20 km | Africa | Infectious Disease Dynamics | 2018 |
| Diarrhea in children under five | 5 km | LMIC | IHME | 2017 |
| Men’s educational attainment | 5 km | LMIC | IHME | 2017 |
| Mortality in children under five | 5 km | LMIC | IHME | 2017 |
| Population estimate | 100 m | Global | WorldPop | 2020 |
| Prevalence of open defecation | 5 km | LMIC | IHME | 2017 |
| Reliance on unimproved drinking water | 5 km | LMIC | IHME | 2017 |
| Reliance on unimproved sanitation | 5 km | LMIC | IHME | 2017 |
| Time to cities | 1 km | Global | Malaria Atlas Project | 2015 |
| Women’s educational attainment | 5 km | LMIC | IHME | 2017 |
| Roads | Vector | Global | Open Street Maps | 2020 |
| Settlement extents | Polygon | Africa | GRID3 | 2020 |
| **Processed Indicators** | **Res.** | **Cov.** | **Source(s) / Reference variable(s)** | **Year** |
| Rural typology | 1 km | Global | Distance to towns, distance to roads, time to cities, population estimate | 2015 |
| Distance to roads | 200 m | Global | Roads | 2020 |
| Distance to towns | 200 m | Global | Estimated settlement areas | 2020 |
| Estimated settlement areas (African countries) | -- | Africa | Settlement extents, population estimate | 2020 |
| Estimated settlement areas (Non-African countries) | -- | Global | Population estimate | 2020 |

**Table 4 SanPlan variables.** Original resolutions (Res.), geographic coverage (Cov.), and data sources. LMIC = “Low- and Middle-Income Countries.” IHME = “Institute of Health Metric and Evaluation.”

# Updating the site: Standard Operating Procedures

The SOP below provides a detailed guide to expanding the SanPlan tool by creating, treating, and adding datasets for additional countries. This guide does not involve any changes to current features or content available on the tool. Changes to the tool’s functionality may require alterations to this data treatment process, either in the data production or the script used in step 7, below, when loading the data to the website.

There are six data treatment scripts used to produce the datasets that are loaded to the data server and encoded into the website (Table 5). Four are run in the Python console in QGIS (which can be adapted to run from Python, itself), and one in Python. The series of scripts are run for a single country at a time.

If you are adding a new country, you will need to run each script, beginning the SOP at step 1, for just the new country (all resolutions: settlement, pixel, district, etc.). To add a new indicator, you will need to run the SOP from Step 5c, running each script for all relevant countries (all resolutions). To add a new resolution to a country (e.g., a second district-level boundary), you will need to run the SOP from Step 5c, for only the new resolution level for the relevant country (but still including Step 5g).

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Inputs** | **Outputs** | **Software** |
| 0\_settlements\_  noGRID3\_2023.py | * Raster of population (WP) | * Shapefile of settlement boundaries | QGIS |
| 1\_towns\_construct\_  GRID3\_2023.py | * Geodatabase of built-up areas, small settlement areas, hamlet areas (GRID3) * Shapefile of settlement boundaries (script 0) | * Shapefile of cities * Shapefile of towns * Shapefile of settlements | QGIS |
| 2\_class\_dr\_dt\_  construct\_2023.py | * Shapefile of cities (script 1) * Shapefile of towns (script 1) * Raster of time to cities (MAP) * Shapefile of country boundary (HDX) * Shapefile of roads (OSM) | * Raster of distance to towns * Raster of distance to roads * Raster of community classifications * GeoJSON of community classifications | QGIS |
| 3\_zonal\_stats\_  final\_2023.py | * Shapefile of sub-country boundary * Raster of distance to towns (script 2) * Raster of distance to roads (script 2) * Raster of community classifications (script 2) * Rasters of additional desired raw indicators\* | * Multivariable shapefile (or GeoJSON) by settlement, 5x5km pixels or by sub-national boundary (separate runs) * CSV of column minimums/maximums | Python |
| 4\_add\_admin\_  2023.py | * Multivariable shapefile (or GeoJSON) by 5x5km or sub-country boundary (script 3) * Smallest relevant sub-country boundary shapefile (HDX) | * Multivariable shapefile (or GeoJSON) by settlement, 5x5km or sub-country boundary with boundary names | QGIS |
| 5\_comms\_join\_  2023.py | * Multivariable shapefile of settlements with boundary names (script 4) * All additional multivariable shapefiles (or GeoJSONs) with boundary names (script 4) | * Multivariable shapefile of settlements with boundary names and all other boundary data | QGIS |
| *\* Current script indicators listed in Table 4..* | | | |

**Table 5. Summary of the stepwise data treatment scripts.** WP = “WorldPop.” MAP = “Malaria Atlas Project.” HDX = “Humanitarian data exchange.” OSM = “open street maps.”

The data treatment scripts rely on importing and exporting data to dynamic file paths, therefore the structure of the file tree and the name conventions, including country codes and variable codes are very important.

In the following SOP, folders within the computer drive’s directory are highlighted blue. Variables defined in python scripts are highlighted green. Variables defined in javascript scripts are highlighted yellow.

1. Create file tree within a master folder (e.g. “datasets”)
   1. Create a “global” folder (skip after initial setup)
      1. Download global datasets from Table 6. Alter file names if needed to match Table 6.

|  |  |  |
| --- | --- | --- |
| **Raw Indicator** | **Link** | **Filename** |
| Cholera risk | https://www.iddynamics.jhsph.edu/resources/suspected-cholera-incidence-africa-2010-2016 | IDD\_cholera.tif |
| Diarrhea in children under five | https://ghdx.healthdata.org/record/ihme-data/global-under-5-diarrhea-incidence-prevalence-mortality-geospatial-estimates-2000-2019 | IHME\_Dia.tif |
| Men’s educational attainment | https://ghdx.healthdata.org/record/ihme-data/lmic-education-geospatial-estimates-2000-2017 | IHME\_EDU\_M\_00-17.tif |
| Women’s educational attainment | IHME\_EDU\_W\_00-17.tif |
| Mortality in children under five | https://ghdx.healthdata.org/record/ihme-data/lmic-under5-mortality-rate-geospatial-estimates-2000-2017 | IHME\_U5M\_00-17.tif |
| Prevalence of open defecation | https://ghdx.healthdata.org/record/ihme-data/lmic-wash-access-geospatial-estimates-2000-2017 | IHME\_OD.tif |
| Access to any improved drinking water | IHME\_W\_IMP.tif |
| Access to a piped drinking water | IHME\_W\_PIPED.tif |
| Access to any improved sanitation | IHME\_S\_IMP.tif |
| Access to piped sanitation | IHME\_S\_PIPED.tif |
| Reliance on surface drinking water | IHME\_W\_SURFACE.tif |
| Reliance on other improved drinking water | IHME\_W\_IMP\_OTHER.tif |
| Reliance on other improved sanitation | IHME\_S\_IMP\_OTHER.tif |
| Reliance on unimproved drinking water | IHME\_W\_UNIMP.tif |
| Reliance on unimproved sanitation | IHME\_S\_UNIMP.tif |
| Time to cities | https://malariaatlas.org/project-resources/accessibility-to-healthcare/ | MAP\_timecities.tif |

**Table 6. GLOBAL DATASET SOURCES AND NAMING CONVENTIONS.**

* 1. Create a country folder.

The name of the folder should be the three-letter country code as defined by [IBAN](https://www.iban.com/country-codes)

* + 1. Within the country folder, create empty folders with the following names:
       1. class
       2. comms
       3. country
       4. dist
       5. distroads
       6. disttowns
       7. pop
       8. prov
       9. roads
       10. timecities

1. Gather country-level datasets and place in appropriate file tree folder
   1. Source administrative boundary shapefiles from OCHA on HDX. Find the dataset by navigating to [HDX](https://data.humdata.org/) and searching for “[country name] administrative boundaries”. Select the link for the appropriate subnational boundaries published by OCHA. The updated data should be within the last year.

*Sourcing from OCHA ensures consistent column names within the shapefiles (“ADM1\_EN”, “ADM2\_EN”). We have been transitioning from using DIVA-GIS, which has over time shown to not update as boundaries change OCHA datasets are updated annually. Therefore, in the scripts, you will file relicts of older column names (e.g., “NAME\_1”, “NAME\_2”).*

* + 1. Extract country level (adm0) -> country folder
    2. Create a zipped folder for the country-level shapefile files, rename as “countrycode\_adm0” (e.g., “gha\_adm0.zip” for Ghana), keep in country folder.
    3. Extract desired sub-country level (typically adm2 or adm3) -> dist folder

*Occasionally a country may have up to 5-6 boundary levels. At minimum, you should not use levels where zones are close to or smaller than 5x5km. It is best to consider the local government functions and determine which level is most useful for implementers.*

* 1. Source population raster data from [WorldPop](https://www.worldpop.org/geodata/listing?id=78) -> pop folder

1. Select the 2020 constrained estimates for the country of interest.
2. Ensure the filename follows “countrycode\_ppp\_2020\_constrained.tif”
   1. Source roads shapefiles from [Humanitarian Data Exchange](https://data.humdata.org/) -> roads folder

*Easiest found by searching the country name + “roads”. Select the dataset from Open Street Maps Humanitarian Team (HOTOSM). This is important for consistent file names coded into the scripts. If using data from a different source, be sure to match the file name to the appropriate scripts.*

* + 1. Of the file type options, download the lines\_shp.zip folder
  1. Source settlements geodatabase data from [GRID3](https://grid3.org/resources/data) -> comms folder

*This dataset will only be available if the desired country is in Africa. If there is no GRID3 settlements dataset, see Step 4.*

* + 1. On the website, click the “settlements” tab, locate the desired country and download the associated “settlements extent” geodatabase.
    2. Rename the geodatabase folder to be “countrycode\_GRID3.gdb” (e.g. “gha\_GRID3.gdb” for Ghana).

1. Produce custom layers (automated Python/QGIS scripts – Appendix 1, download required packages as needed)
2. If a GRID3 dataset does not exist for the desired country, run settlement script.

*If a GRID3 dataset exists, skip to step 5.*

* 1. Run Script 0 (0\_settlements\_noGRID3.py) in QGIS
     1. BEFORE RUNNING:
        1. Update/Check the directory (“dir”) pathname to lead to your master folder (e.g. “datasets”)
        2. Update/Check the country code (“cc”) is correct for the desired country
     2. AFTER RUNNING:
        1. Do a visual inspection of resulting settlement map with satellite imagery to evaluate the accuracy of the settlement boundaries. If many clusters of buildings are missed and not included in a settlement, or if the settlement boundaries are often too large and encapsulate multiple communities, adjust DBSCAN clustering and concave hull parameters and re-run script.

1. Run data treatment scripts
   1. Run Script 1 (1\_towns\_construct\_GRID3\_2023.py) in QGIS
      1. BEFORE RUNNING:
         1. Update/check the directory (“dir”) pathname to lead to your master folder (e.g. “datasets”).
         2. Update/check the country code (“cc”) is correct for the desired country.
   2. Run Script 2 (2\_class\_dr\_dt\_construct\_2023.py) in QGIS
      1. BEFORE RUNNING:
         1. Update/check the directory (“dir”) pathname to lead to your master folder (e.g. “datasets”).
         2. Update/check the country code (“cc”) is correct for the desired country.
      2. AFTER RUNNING:
         1. Make sure classification layer has exactly 1-4 categories.
   3. Run Script 3 (3\_zonal\_stats\_final\_2023.py) for either pixel- or sub-country-level in Python
      1. BEFORE RUNNING:
         1. Update/check the directory (“os.chdir”) pathname to lead to your master folder (e.g. “datasets”).
         2. Update/check the country code (“cc”) is correct for the desired country.
         3. Update/check the variable (“var”) is correct for the desired output dataset.

*Options are “comms” for the settlement boundary layer, “pixel” for the 5x5km pixel layer, “dist” for the middle (2nd or 3rd) sub-country boundary (“dist” is used REGARDLESS of whether the boundary name is “district”), or “prov” for the 1st sub-country boundary (“prov” is used REGARDLESS of whether the boundary name is “province”).*

* + - 1. Update/check the suffix (“suffix”) is correct for the desired output dataset.

*Leave this blank unless you are running for a repeated subnational boundary of the same resolution (e.g., add health districts as well as admin disticts. In this case, add a suffix such as "\_health" when running for health districts.). The suffix can be anything, as long as it is consistent when running across scripts for the same country.*

* + - 1. Update/check the admin boundary filename (“file\_path”) is correct and matches the file from Step 2.a.iii.

*Depending of the boundaries available for the desired country the boundary level may change, it is typically 2 or 3.*

* + - 1. Update/check the admin boundary column names are all listed and match the file from Step 2.a.iii.
* *The number of column names will match the admin boundary level number*
* *If sourced from OCHA, will take the structure: “ADM#\_EN”. For example, if the file name is “gha\_adm2”, the code should be: “dist = dist[[“ADM1\_EN”, “ADM2\_EN”, “geometry”]]”. If it is “lbr\_adm3”: the code should be “dist = [[“ADM1\_EN”, “ADM2\_EN”, “ADM3\_EN”, “geometry”]]”*
* *If sourced from DIVA-GIS, will take the structure: “NAME\_#”. For example, if the file name is “gha\_adm2”, the code should be: “dist = dist[[“NAME\_1”, “NAME\_2”, “geometry”]]”. If it is “lbr\_adm3”: the code should be “dist = dist[[“NAME\_1”, “NAME\_2”, “NAME\_3”, “geometry”]]”*
  + 1. AFTER RUNNING:
       1. Look at table of values to inspect if numbers look reasonable, particularly look at null values for potential errors.
       2. Plot in QGIS to ensure it maps correctly.
  1. Repeat previous step (5.c) for the other variable option (pixel or sub-country)
  2. Run Script 4 (4\_add\_admin\_2023.py) for either pixel- or sub-country-level in QGIS
     1. BEFORE RUNNING:
        1. Update/check the directory (“os.chdir”) pathname to lead to your master folder (e.g. “datasets”).
        2. Update/check the country code (“cc”) is correct for the desired country.
        3. Update/check the variable (“var”) is correct for the desired output dataset.

*Options are “comms” for the settlement boundary layer, “pixel” for the 5x5km pixel layer, “dist” for the middle (2nd or 3rd) sub-country boundary (“dist” is used REGARDLESS of whether the boundary name is “district”), or “prov” for the 1st sub-country boundary (“prov” is used REGARDLESS of whether the boundary name is “province”).*

* + - 1. Update/check the sub-country boundary file name (“dist”) is correct for the desired country.

*Typically, this will be either “adm2” or “adm3”, but should match the value defined in Step 2.a.iii.*

* + - 1. Update/check the suffix (“suffix”) is correct for the desired output dataset.

*Leave this blank unless you are running for a repeated subnational boundary of the same resolution (e.g., add health districts as well as admin disticts. In this case, add a suffix such as "\_health" when running for health districts.). The suffix can be anything, as long as it is consistent when running across scripts for the same country.*

* + - 1. Update/check the admin boundary column names (“join\_fields“) are all listed and match the file from Step 2.a.iii.
* *The number of column names will match the admin boundary level number.*
* *If sourced from OCHA, will take the structure: “ADM#\_EN”.*
* *If sourced from DIVA-GIS, will take the structure: “NAME\_#”.*
  1. Repeat previous step (4.e) for the other variable options (pixel = “pixel”, settlements = “comms”, sub-country = “dist”)
  2. Run Script 5 (5\_comms\_join\_2023.py) for either pixel- or sub-country-level in QGIS
     1. BEFORE RUNNING:
        1. Update/check the directory (“os.chdir”) pathname to lead to your master folder (e.g. “datasets”).
        2. Update/check the country code (“cc”) is correct for the desired country.
        3. Update/check the sub-country multivariable file names (“infn”) are correct for the desired country.

*Typically, these will end with "\_multivariable\_noadmin\_pixel.geojson" and “\_multivariable\_noadmin\_dist.shp”. If you have included additional boundary levels (e.g., province level or a second district level, include the sptial index and column join commands for each of these (repeat the sections as noted within the script).*

1. Load all layers to GeoDB server: country-level boundary shapefile, 1-km rural typology GEOJSON (or shapefile), settlement-level multivariable shapefile, 5-km multivariable shapefile, all additional boundary-level multivariable shapefiles.
2. Update codebase
   1. Copy/paste the content of the country template (Sanitation planning tool > src > state > CountryTemplate.js) into a new javascript file within the MapState folder.
   2. Update and add missing information
      1. Name the file the full country name, in lowercase.
      2. Replace “country” in line 1 with the country name.
      3. Add the country name to “name” in line 2, capitalized.
      4. Add the country name to “mapID” in line 3, lowercase.
      5. Use Google Maps to approximate the centroid of the country, place the lat/longs in lines 4-5 (as numbers).
      6. Replace “cc” in “carto\_tableName” with the country code for each layer.
      7. Add the appropriate admin boundary names in the “name” field of layer 4 and the filter names of layers as needed, ensure “column\_names” for these are correct.
      8. Variable minimums/maximums and values for all filters listed in all layers

*Use “countrycode\_variable\_varvals.csv” file which is exported in steps 5.c and 5.d to facilitate the filter updates.*

* + - 1. For the population estimate filter(s), and potentially others, determine if you need to use a non-linear scale (filters are not useful due to skewed variable distributions). This often occurs with population estimate where many pixels have small values, but large cities skew the filter scale making if difficult for the user to filter small values. If needed, use the max value to build logical steps for a non-linear scale. If you implement a non-linear scale needed, change “type” to “non-linear”
    1. Add layers as needed for additional boundary-levels and repeat steps vi-viii (copy and paste the district-level layer and update accordingly).
    2. DO NOT alter:
       1. Layer-level:
          1. carto\_source, carto\_layer, carto\_style, visible, accessCounter, washCounter, socioCounter, healthCounter,
       2. Filter-level:
          1. name (except boundary names), unit, type, column\_name, subcategory
          2. For classification layer, do not alter the value field
  1. Add the country name to the list of “maps” in the initial state (sanitation planning tool > src > state > MapState.js).
  2. Check Homepage that the country appears on the map selector.
  3. Check country dropdown that the name appears.
  4. Check map for functionality.

# Appendix I: Data treatment scripts

These scripts are also available at (https://github.com/Aquaya-Institute/SPT-data)

## Defining settlement boundaries (non-African countries)

##### Settlements generation SOP #####

from qgis.core import \*

import processing

import qgis.utils

import os # This is is needed in the pyqgis console also

dir = "/Users/karastuart/Dropbox (Aquaya)/WASHPaLS\_RProjects/PEFO/SPT-data/datasets/" # UPDATE

directory = os.fsencode(dir)

cc = "idn" # country code #UPDATE

### Pixels to Points

def pixels\_points(in\_fn,out\_fn):

processing.run("native:pixelstopoints", {'INPUT\_RASTER':in\_fn,\

'RASTER\_BAND':1,'FIELD\_NAME':'VALUE','OUTPUT':out\_fn})

in\_fn = dir+cc+"/pop/"+cc+"\_ppp\_2020\_constrained.tif"

out\_fn = dir+cc+"/comms/"+cc+"\_raspoints.shp"

pixels\_points(in\_fn,out\_fn)

### DBSCAN Clustering

def cluster(in\_fn, min\_clus, dist, out\_fn):

processing.run("native:dbscanclustering", {'INPUT':in\_fn,\

'MIN\_SIZE':min\_clus,'EPS':dist,'DBSCAN\*':False,\

'FIELD\_NAME':'CLUSTER\_ID','OUTPUT':out\_fn})

in\_fn = dir+cc+"/comms/"+cc+"\_raspoints.shp"

min\_clus = 2

dist = 0.005

out\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus.shp"

cluster(in\_fn, min\_clus, dist, out\_fn)

### Remove NULL cluster IDs

def extract(in\_fn,out\_fn):

processing.run("native:extractbyattribute", {'INPUT':in\_fn,\

'FIELD':'CLUSTER\_ID','OPERATOR':9,'VALUE':'','OUTPUT':out\_fn})

in\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus.shp"

out\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus\_nonull.gpkg"

extract(in\_fn, out\_fn)

### Buffer clusters

def buff(in\_fn,out\_fn):

processing.run("native:buffer", {'INPUT':in\_fn,\

'DISTANCE':0.0015,'SEGMENTS':5,'END\_CAP\_STYLE':0,\

'JOIN\_STYLE':0,'MITER\_LIMIT':2,'DISSOLVE':False,'OUTPUT':out\_fn})

in\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus\_nonull.gpkg"

out\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus\_nonull\_buff.gpkg"

buff(in\_fn, out\_fn)

### Concave hull

def polygon(in\_fn,out\_fun):

processing.run("qgis:minimumboundinggeometry", {'INPUT':in\_fn,\

'FIELD':'CLUSTER\_ID','TYPE':3,'OUTPUT':out\_fn})

in\_fn = dir+cc+"/comms/"+cc+"\_raspoints\_clus\_nonull\_buff.gpkg"

out\_fn = dir+cc+"/comms/"+cc+"\_clus\_concave.gpkg"

polygon(in\_fn, out\_fn)

#%%

## Extract “towns” and “cities”

from qgis.core import \*

import processing

import qgis.utils

import os

dir = "/Users/karastuart/Dropbox (Aquaya)/WASHPaLS 2/PEFO/SPT-data/datasets/" # UPDATE

directory = os.fsencode(dir)

cc = "sdn" # country code # UPDATE

grid3 = "yes" # UPDATE to "no" if non-African country

def mergelayers(layers, out\_fn):

processing.run("native:mergevectorlayers", {'LAYERS':layers,\

'CRS':None,\

'OUTPUT':out\_fn})

##### Separate out hamlets (too many for efficient use) #####

### USE if settlements raw dataset is downloaded geodatabase folder (old version) rather than shapefile

#if(grid3=="yes"):

# bua = dir+cc+"/comms/"+cc+"\_GRID3.gdb|layername=bua\_extents"

# ssa = dir+cc+"/comms/"+cc+"\_GRID3.gdb|layername=ssa\_extents"

# layers = [bua, ssa]

# outfn=dir+cc+"/comms/"+cc+"\_buassa.shp"

# mergelayers(layers, outfn)

### USE if settlements raw dataset is NOT downloaded as geodatabase folder (old version)

def extract\_settlementtype(in\_fn, out\_fn, field, val):

processing.run("native:extractbyattribute", {'INPUT':in\_fn,\

'FIELD':field,'OPERATOR':val,'VALUE':'Hamlet',\

'OUTPUT':out\_fn})

### Remove halmets, save all bigger regions

val = 1

outfn = dir+cc+"/comms/"+cc+"\_buassa.shp"

infn=dir+cc+"/comms/"+cc+"\_grid3raw.shp" # CHECK FILENAME OF RAW DATA

field='type'

extract\_settlementtype(infn, outfn, field, val)

### Remove everything BUT halmets, save as hamlets

val = 0

outfn = dir+cc+"/comms/"+cc+"\_ham.shp"

infn=dir+cc+"/comms/"+cc+"\_grid3raw.shp"

field='type'

extract\_settlementtype(infn, outfn, field, val)

### Buffer settlement boundaries to capture nearby population data and simplify geometry

def bufferlayer(in\_fn, out\_fn, dist):

processing.run("native:buffer", {'INPUT':in\_fn,

'DISTANCE':dist,'SEGMENTS':5,'END\_CAP\_STYLE':0,'JOIN\_STYLE':0,

'MITER\_LIMIT':2,'DISSOLVE':False,'OUTPUT':out\_fn})

if(grid3=="yes"):

infn = dir+cc+"/comms/"+cc+"\_buassa.shp"

else:

infn = dir+cc+"/comms/"+cc+"\_clus\_concave.gpkg"

outfn = dir+cc+"/comms/"+cc+"\_buassa\_b.shp"

dist = 0.001

bufferlayer(infn, outfn, dist)

### Sum population per settlement boundary

def zonal(in\_ras,in\_vec):

processing.run("qgis:zonalstatistics", {'INPUT\_RASTER':in\_ras,

'RASTER\_BAND':1,'INPUT\_VECTOR':in\_vec,

'COLUMN\_PREFIX':'\_','STATISTICS':[1]})

in\_ras = dir+cc+"/pop/"+cc+"\_ppp\_2020\_constrained.tif"

in\_vec = dir+cc+"/comms/"+cc+"\_buassa\_b.shp"

zonal(in\_ras,in\_vec)

#Select 5,000+, save as "towns"

def extract\_towns(in\_fn, out\_fn, field, val):

processing.run("native:extractbyattribute", {'INPUT':in\_fn,\

'FIELD':field,'OPERATOR':3,'VALUE':val,\

'OUTPUT':out\_fn})

val = "5000"

outfn = dir+cc+"/disttowns/"+cc+"\_towns.shp"

infn=dir+cc+"/comms/"+cc+"\_buassa\_b.shp"

field='\_sum'

extract\_towns(infn, outfn, field, val)

#Select 50,000+, save as "cities"

val = "50000"

outfn = dir+cc+"/disttowns/"+cc+"\_cities.shp"

infn=dir+cc+"/comms/"+cc+"\_buassa\_b.shp"

field='\_sum'

extract\_towns(infn, outfn, field, val)

if(grid3=="yes"):

#Select large hamlets, save as

val = ".0000045" # Customize if needed

outfn = dir+cc+"/comms/"+cc+"\_hambig.shp"

infn=dir+cc+"/comms/"+cc+"\_ham.shp"

field='SHAPE\_Area'

extract\_towns(infn, outfn, field, val)

#Buffer large hamlets

infn = dir+cc+"/comms/"+cc+"\_hambig.shp"

outfn = dir+cc+"/comms/"+cc+"\_hambig\_b.shp"

dist = 0.0005

bufferlayer(infn, outfn, dist)

#Population of large hamlets

in\_ras = dir+cc+"/pop/"+cc+"\_ppp\_2020\_constrained.tif"

in\_vec = dir+cc+"/comms/"+cc+"\_hambig\_b.shp"

zonal(in\_ras,in\_vec)

#Merge all settlements

buassa = dir+cc+"/comms/"+cc+"\_buassa\_b.shp"

ham = dir+cc+"/comms/"+cc+"\_hambig\_b.shp"

layers = [buassa, ham]

outfn=dir+cc+"/comms/"+cc+"\_comms\_b.shp"

mergelayers(layers, outfn)

##Map generation SOP

## Compute “distance to towns”, “distance to roads”, and “rural typologies”

#Map generation SOP

from qgis.core import \*

import qgis.utils

from qgis.utils import iface

import processing

import os

os.environ["PROJ\_LIB"]="/Applications/QGIS.app/Contents/Resources/proj" # CHECK

dir = "/Users/karastuart/Dropbox (Aquaya)/WASHPaLS 2/PEFO/SPT-data/datasets/" # UPDATE

directory = os.fsencode(dir)

cc = "sdn" # country code # UPDATE

####Get Country Extent (deg)

country = QgsVectorLayer(dir+cc+"/country/"+cc+"\_adm0.shp", cc+"\_country", "ogr")

QgsProject.instance().addMapLayer(country)

country = iface.activeLayer()

ext = country.extent()

xmin = ext.xMinimum()

xmax = ext.xMaximum()

ymin = ext.yMinimum()

ymax = ext.yMaximum()

coords = "%f,%f,%f,%f" %(xmin, xmax, ymin, ymax) # this is a string that stores the coordinates

extent = coords+' [EPSG:4326]'

QgsProject.instance().removeMapLayer(country)

####Get Country Extent (m)

#Reproject for geometric units

def reproject(in\_fn, out\_fn, crs):

processing.run("native:reprojectlayer", {'INPUT':in\_fn,\

'TARGET\_CRS':crs,

'OPERATION':'+proj=pipeline +step +proj=unitconvert +xy\_in=deg +xy\_out=rad +step +proj=webmerc +lat\_0=0 +lon\_0=0 +x\_0=0 +y\_0=0 +ellps=WGS84',\

'OUTPUT':out\_fn})

infn = dir+cc+"/country/"+cc+"\_adm0.shp"

outfn = dir+cc+"/country/"+cc+"\_adm0p.shp"

crs = QgsCoordinateReferenceSystem('EPSG:3857')

reproject(infn, outfn, crs)

country = QgsVectorLayer(dir+cc+"/country/"+cc+"\_adm0p.shp", cc+"\_country", "ogr")

QgsProject.instance().addMapLayer(country)

country = iface.activeLayer()

ext = country.extent()

xmin = ext.xMinimum()

xmax = ext.xMaximum()

ymin = ext.yMinimum()

ymax = ext.yMaximum()

coords = "%f,%f,%f,%f" %(xmin, xmax, ymin, ymax) # this is a string that stores the coordinates

extent\_p = coords+' [EPSG:3857]'

QgsProject.instance().removeMapLayer(country)

####ROADS

#Extract cat 1-3 roads

#Select 1-3 roads, save as

def extract\_roads(in\_fn, out\_fn, exp):

processing.run("native:extractbyexpression", {'INPUT':in\_fn,\

'EXPRESSION':exp,\

'OUTPUT':out\_fn})

infn = dir + cc + "/roads/hotosm\_"+cc+"\_roads\_lines.shp" # CHECK RAW FILENAME

outfn=dir + cc + "/roads/"+cc+"\_roads\_123.shp"

exp = ' \"highway\" = \'primary\' OR \"highway\" = \'secondary\' OR \"highway\" = \'tertiary\' OR \"highway\" = \'trunk\' OR \"highway\" = \'primary\_link\' OR \"highway\" = \'secondary\_link\' OR \"highway\" = \'tertiary\_link\' OR \"highway\" = \'trunk\_link\' '

extract\_roads(infn, outfn, exp)

#Reproject for geometric units

def reproject(in\_fn, out\_fn, crs):

processing.run("native:reprojectlayer", {'INPUT':in\_fn,\

'TARGET\_CRS':crs,'OPERATION':'+proj=pipeline +step +proj=unitconvert +xy\_in=deg +xy\_out=rad +step +proj=webmerc +lat\_0=0 +lon\_0=0 +x\_0=0 +y\_0=0 +ellps=WGS84',\

'OUTPUT':out\_fn})

filename = os.fsdecode(cc+"/roads/"+cc+"\_roads\_123.shp")

outfn = dir + cc + "/roads/"+cc+"\_roads\_123p.shp"

infn = os.path.join(os.fsdecode(directory),filename)

crs = QgsCoordinateReferenceSystem('EPSG:3857')

reproject(infn, outfn, crs)

#Rasterize

def rasterize\_roads(in\_fn, out\_fn, extent\_temp):

processing.run("gdal:rasterize", {'INPUT':in\_fn,\

'FIELD':'','BURN':1,'UNITS':1,'WIDTH':200,'HEIGHT':200,\

'EXTENT':extent\_temp,\

'NODATA':None,'OPTIONS':'','DATA\_TYPE':5,'INIT':None,'INVERT':False,'EXTRA':'',\

'OUTPUT':out\_fn})

filename = os.fsdecode(cc+"/roads/"+cc+"\_roads\_123p.shp")

outfn = dir + cc + "/roads/"+cc+"\_roads\_123p.tif"

infn=os.path.join(os.fsdecode(directory),filename)

extent\_temp = extent\_p

rasterize\_roads(infn, outfn, extent\_temp)

#Proximity

def prox\_roads(in\_fn, out\_fn, max):

processing.run("gdal:proximity", {'INPUT':in\_fn,\

'BAND':1,'VALUES':'','UNITS':0,'MAX\_DISTANCE':None,'REPLACE':None,'NODATA':None,'OPTIONS':'','EXTRA':'','DATA\_TYPE':2,\

'OUTPUT':out\_fn})

infn= dir + cc + "/roads/"+cc+"\_roads\_123p.tif"

outfn = dir + cc + "/distroads/"+cc+"\_distroadsp.tif"

max='None'

prox\_roads(infn, outfn, max)

def assign\_crs(in\_fn):

processing.run("gdal:assignprojection", {'INPUT':in\_fn,'CRS':QgsCoordinateReferenceSystem('EPSG:3857')})

infn = dir + cc + "/distroads/"+cc+"\_distroadsp.tif"

assign\_crs(infn)

#Warp

def warp(in\_fn, out\_fn, crs):

processing.run("gdal:warpreproject", {'INPUT':in\_fn,\

'SOURCE\_CRS':None,'TARGET\_CRS':crs,\

'RESAMPLING':0,'NODATA':None,'TARGET\_RESOLUTION':None,'OPTIONS':'','DATA\_TYPE':0,'TARGET\_EXTENT':None,\

'TARGET\_EXTENT\_CRS':None,'MULTITHREADING':False,'EXTRA':'','OUTPUT':out\_fn})

infn = dir + cc + "/distroads/"+cc+"\_distroadsp.tif"

outfn = dir + cc + "/distroads/"+cc+"\_distroads.tif"

crs = QgsCoordinateReferenceSystem('EPSG:4326')

warp(infn, outfn, crs)

####TOWNS

#Reproject

infn=dir + cc + "/disttowns/"+cc+"\_towns.shp"

outfn = dir + cc + "/disttowns/"+cc+"\_townsp.shp"

crs=QgsCoordinateReferenceSystem('EPSG:3857')

reproject(infn, outfn, crs)

#Rasterize

def rasterize\_towns(in\_fn, out\_fn, extent\_temp):

processing.run("gdal:rasterize", {'INPUT':in\_fn,\

'FIELD':'','BURN':1,'UNITS':1,'WIDTH':200,'HEIGHT':200,\

'EXTENT':extent\_temp,\

'NODATA':None,'OPTIONS':'','DATA\_TYPE':5,'INIT':None,'INVERT':False,'EXTRA':'',\

'OUTPUT':out\_fn})

infn=dir + cc + "/disttowns/"+cc+"\_townsp.shp"

outfn = dir + cc + "/disttowns/"+cc+"\_townsp.tif"

extent\_temp = extent\_p

rasterize\_towns(infn, outfn, extent\_temp)

#Proximity

def prox\_towns(in\_fn, out\_fn, max):

processing.run("gdal:proximity", {'INPUT':in\_fn,\

'BAND':1,'VALUES':'','UNITS':0,'MAX\_DISTANCE':max,'REPLACE':None,'NODATA':None,'OPTIONS':'','EXTRA':'','DATA\_TYPE':5,\

'OUTPUT':out\_fn})

infn=dir + cc + "/disttowns/"+cc+"\_townsp.tif"

outfn = dir + cc + "/disttowns/"+cc+"\_disttownsp.tif"

max=None

prox\_towns(infn, outfn, max)

infn = dir + cc + "/disttowns/"+cc+"\_disttownsp.tif"

assign\_crs(infn)

#Warp

infn = dir + cc + "/disttowns/"+cc+"\_disttownsp.tif"

outfn = dir + cc + "/disttowns/"+cc+"\_disttowns.tif"

crs = QgsCoordinateReferenceSystem('EPSG:4326')

warp(infn, outfn, crs)

####CITIES

#Reproject

infn=dir + cc + "/disttowns/"+cc+"\_cities.shp"

outfn = dir + cc + "/disttowns/"+cc+"\_citiesp.shp"

crs=QgsCoordinateReferenceSystem('EPSG:3857')

reproject(infn, outfn, crs)

#Rasterize

infn=dir + cc + "/disttowns/"+cc+"\_citiesp.shp"

outfn = dir + cc + "/disttowns/"+cc+"\_citiesp.tif"

extent\_temp = extent\_p

rasterize\_towns(infn, outfn, extent\_temp)

#Warp

infn = dir + cc + "/disttowns/"+cc+"\_citiesp.tif"

outfn = dir + cc + "/disttowns/"+cc+"\_cities.tif"

crs = QgsCoordinateReferenceSystem('EPSG:4326')

warp(infn, outfn, crs)

####TIME TO CITIES

def cliplayer(in\_fn, out\_fn, masklayer):

processing.run("gdal:cliprasterbymasklayer", {'INPUT': in\_fn,'MASK':masklayer,\

'SOURCE\_CRS':None,'TARGET\_CRS':None,'NODATA':None,'ALPHA\_BAND':False,'CROP\_TO\_CUTLINE':True,\

'KEEP\_RESOLUTION':False,'SET\_RESOLUTION':False,'X\_RESOLUTION':None,'Y\_RESOLUTION':None,\

'MULTITHREADING':False,'OPTIONS':'','DATA\_TYPE':0,'EXTRA':'','OUTPUT':out\_fn})

infn = dir + "/global/MAP\_timecities.tif"

outfn=dir + cc + "/timecities/"+cc+"\_timecities.tif"

masklayer = dir + cc + "/country/"+cc+"\_adm0.shp"

cliplayer(infn, outfn, masklayer)

####CLASSIFICATION

#Raster calculator

def rastercalc\_class(out\_fn, exp, layers, extent\_temp):

processing.run("qgis:rastercalculator", {'EXPRESSION':exp,\

'LAYERS':layers,'CELLSIZE':None,\

'EXTENT':extent\_temp,'CRS':None,\

'OUTPUT':out\_fn})

disttowns = QgsRasterLayer(dir + cc + "/disttowns/"+cc+"\_disttowns.tif","disttowns")

timecities = QgsRasterLayer(dir + cc + "/timecities/"+cc+"\_timecities.tif","timecities")

distroads = QgsRasterLayer(dir + cc + "/distroads/"+cc+"\_distroads.tif","distroads")

cities = QgsRasterLayer(dir + cc + "/disttowns/"+cc+"\_cities.tif","cities")

outfn = dir + cc + "/class/"+cc+"\_class\_nc.tif"

QgsProject.instance().addMapLayer(disttowns)

QgsProject.instance().addMapLayer(distroads)

QgsProject.instance().addMapLayer(timecities)

QgsProject.instance().addMapLayer(cities)

layers = [timecities]

extent\_temp = extent

exp = '(\"cities@1\" > 0 OR \"timecities@1\"<= 10)\*4+(( \"cities@1\" < 1 AND \"timecities@1\">10) AND (\"disttowns@1\" <= 800 OR \"timecities@1\" <= 25))\*3+((\"cities@1\" < 1) AND (\"disttowns@1\" > 800 AND \"timecities@1\">25) AND \"distroads@1\" <= 1500)\*2+((\"cities@1\" < 1) AND (\"disttowns@1\" > 800 AND \"timecities@1\">25) AND \"distroads@1\" > 1500)\*1'

rastercalc\_class(outfn, exp, layers, extent\_temp)

QgsProject.instance().removeMapLayer(disttowns)

QgsProject.instance().removeMapLayer(distroads)

QgsProject.instance().removeMapLayer(timecities)

QgsProject.instance().removeMapLayer(cities)

#Clip

infn = dir + cc + "/class/"+cc+"\_class\_nc.tif"

outfn=dir + cc + "/class/"+cc+"\_class.tif"

masklayer = dir + cc + "/country/"+cc+"\_adm0.shp"

cliplayer(infn, outfn, masklayer)

iface.addRasterLayer(dir + cc + "/class/"+cc+"\_class.tif","class")

def polygonize(in\_fn, out\_fn):

processing.run("gdal:polygonize", {'INPUT':in\_fn,\

'BAND':1,'FIELD':'classes','EIGHT\_CONNECTEDNESS':False,'EXTRA':'',\

'OUTPUT':out\_fn})

infn = dir + cc + "/class/"+cc+"\_class.tif"

outfn= dir + cc + "/class/"+cc+"\_class.geojson"

polygonize(infn, outfn)

## Compute settlement, pixel, and administrative boundary zonal statistics

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created on Tue Mar 30 14:24:30 2021

@author: karastuart

"""

import os

import pandas as pd

import geopandas as gpd

from rasterstats import zonal\_stats

from shapely.geometry import Polygon

import numpy as np

import math

os.chdir("/Users/karastuart/Dropbox (Aquaya)/WASHPaLS 2/PEFO/SPT-data") # UPDATE

###Set Variables

# Varibale-specfic variables

cc = 'cod' # country code #UPDATE

var = 'pixel' # variable name, select either 'pixel', 'dist', or 'prov' # UPDATE

suffix = "" # UPDATE - leave blank unless this is a second subnational boundary of the same resolution (e.g., add health districts as well as admin disticts. In this case, add a suffix such as "\_health" when running for health districts.)

adm\_fn = 'adm3' # UPDATE number to be 2 or 3 as needed

#Define paths

country\_path = os.path.join('./datasets/' + cc + '/')

global\_path = os.path.join('./datasets/global/')

file\_path = 'dist/'+cc+'\_'+adm\_fn+'.shp' # shapefile name

shp\_path = os.path.join(country\_path + file\_path)

### UPDATE IF VAR = DIST

dist = gpd.read\_file(shp\_path)

### UPDATE IF VAR = DIST select the line that includes the correct boundary level names + geometry

### there will be 2 or 3 boundary levels depending on the country

dist = dist[["ADM1\_EN", "ADM2\_EN", "geometry"]]

# dist = dist[["REGION", "DISTRICT", "geometry"]]

# dist = dist[["ADM1\_FR", "ADM2\_FR", "NOM", "geometry"]]

# dist = dist[["NAME\_1", "NAME\_2", "NAME\_3", "geometry"]]

# dist = dist[["NAME\_1", "NAME\_2", "geometry"]]

### UPDATE IF VAR = PROV

# prov = gpd.read\_file(shp\_path)

### UPDATE IF VAR = PROV select the line that includes the correct boundary level names + geometry

### there will be 2 or 3 boundary levels depending on the country

# prov = prov[["NAME\_1", "geometry"]]

# prov = prov[["ADM1\_EN", "geometry"]]

# prov = prov[["REGION", "geometry"]]

# prov = prov[["ADM1\_FR", "geometry"]]

### SET RAW DATA PATHS

od = os.path.join(global\_path + 'IHME\_OD.tif')

timecities = os.path.join(global\_path + 'MAP\_timecities.tif')

dia = os.path.join(global\_path + 'IHME\_Dia.tif')

cholera = os.path.join(global\_path + 'IDD\_cholera.tif')

s\_unimp = os.path.join(global\_path + 'IHME\_S\_UNIMP.tif')

s\_imp = os.path.join(global\_path + 'IHME\_S\_IMP.tif')

s\_imp\_other = os.path.join(global\_path + 'IHME\_S\_IMP\_OTHER.tif')

s\_piped = os.path.join(global\_path + 'IHME\_S\_PIPED.tif')

w\_unimp = os.path.join(global\_path + 'IHME\_W\_UNIMP.tif')

w\_imp = os.path.join(global\_path + 'IHME\_W\_IMP.tif')

w\_imp\_other = os.path.join(global\_path + 'IHME\_W\_IMP\_OTHER.tif')

w\_piped = os.path.join(global\_path + 'IHME\_W\_PIPED.tif')

w\_surface = os.path.join(global\_path + 'IHME\_W\_SURFACE.tif')

edu\_w = os.path.join(global\_path + 'IHME\_EDU\_W\_00-17.tif')

edu\_m = os.path.join(global\_path + 'IHME\_EDU\_M\_00-17.tif')

u5m = os.path.join(global\_path + 'IHME\_U5M\_00-17.tif')

dr = os.path.join(country\_path + 'distroads/'+cc+'\_distroads.tif')

dt = os.path.join(country\_path + 'disttowns/'+cc+'\_disttowns.tif')

classes = os.path.join(country\_path + 'class/'+cc+'\_class.tif')

pop = os.path.join(country\_path + 'pop/'+cc+'\_ppp\_2020\_constrained.tif')

### Define variables depending on resolution and set boundaries to the appropriate regions

if var == "dist":

bounds = dist

vars = [

("od",od),

("timecities",timecities),

("dia",dia),

("cholera",cholera),

("s\_unimp",s\_unimp),

("w\_unimp",w\_unimp),

("edu\_w",edu\_w),

("edu\_m",edu\_m),

("u5m",u5m),

("dr",dr),

("dt",dt),

("classes",classes),

("pop",pop),

("s\_imp", s\_imp),

("s\_imp\_other", s\_imp\_other),

("s\_piped", s\_piped),

("w\_imp", w\_imp),

("w\_imp\_other", w\_imp\_other),

("w\_piped", w\_piped),

("w\_surface", w\_surface),

]

elif var == "prov":

bounds = prov

vars = [

("od",od),

("timecities",timecities),

("dia",dia),

("cholera",cholera),

("s\_unimp",s\_unimp),

("w\_unimp",w\_unimp),

("edu\_w",edu\_w),

("edu\_m",edu\_m),

("u5m",u5m),

("dr",dr),

("dt",dt),

("classes",classes),

("pop",pop),

("s\_imp", s\_imp),

("s\_imp\_other", s\_imp\_other),

("s\_piped", s\_piped),

("w\_imp", w\_imp),

("w\_imp\_other", w\_imp\_other),

("w\_piped", w\_piped),

("w\_surface", w\_surface),

]

elif var=="pixel":

### Create custom grid within shapefile polygon, set as boundaries

crop\_extent = gpd.read\_file(country\_path+'country/'+cc+'\_adm0.shp')

xmin,ymin,xmax,ymax = crop\_extent.total\_bounds

length = 0.04166667170983360396 # set y resolution

wide = 0.04166666509561942067 # set x resolution

cols = list(np.arange(xmin, xmax + wide, wide))

rows = list(np.arange(ymin, ymax + length, length))

polygons = []

for x in cols[:-1]:

for y in rows[:-1]:

polygons.append(Polygon([(x,y), (x+wide, y), (x+wide, y+length), (x, y+length)]))

grid = gpd.GeoDataFrame({'geometry':polygons})

grid\_clip = gpd.clip(grid, crop\_extent)

grid\_clip = grid\_clip[~grid\_clip.is\_empty]

grid\_clip.to\_file(country\_path+var+'/'+cc+'\_'+var+'.shp')

bounds=gpd.read\_file(country\_path+var+'/'+cc+'\_'+var+'.shp')

bounds = bounds[["FID","geometry"]]

vars = [

("od",od),

("timecities",timecities),

("dia",dia),

("cholera",cholera),

("s\_unimp",s\_unimp),

("w\_unimp",w\_unimp),

("edu\_w",edu\_w),

("edu\_m",edu\_m),

("u5m",u5m),

("dr",dr),

("dt",dt),

("classes",classes),

("pop",pop),

("s\_imp", s\_imp),

("s\_imp\_other", s\_imp\_other),

("s\_piped", s\_piped),

("w\_imp", w\_imp),

("w\_imp\_other", w\_imp\_other),

("w\_piped", w\_piped),

("w\_surface", w\_surface),

]

elif var == "comms":

file\_path = var+'/'+cc+'\_comms\_s.shp'

shp\_path = os.path.join(country\_path + file\_path)

bounds = gpd.read\_file(shp\_path)

bounds = bounds[["geometry"]]

vars = [

("timecities",timecities),

("dr",dr),

("dt",dt),

("classes",classes),

("pop",pop)

]

### Calculate zonal statistics for the boundaries

for i in vars:

if i[0] in ('u5m','edu\_w','edu\_m'):

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, band=18, nodata=-339999995214436420000000000000000000000)

elif i[0]=='classes':

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, categorical=True)

elif i[0]=='pop':

tmp=zonal\_stats(bounds, i[1], stats="sum", all\_touched=False, band=1, nodata=-99999)

elif i[0]=='timecities':

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, band=1, nodata=-9999)

elif i[0]=='cholera':

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, band=1, nodata=-339999995214436420000000000000000000000)

elif i[0]=='dr':

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, band=1, nodata=-340282346638528860000000000000000000000)

else:

tmp=zonal\_stats(bounds, i[1], stats="mean", all\_touched=True, band=1, nodata=-999999)

### Format values in attribute table

tmp=gpd.GeoDataFrame(tmp)

tmp=tmp.rename(columns={"mean": i[0]})

tmp=tmp.rename(columns={"sum": i[0]})

if i[0]=='cholera':

tmp=round(100000\*tmp,1)

elif i[0]=='u5m':

tmp=round(tmp\*100,1)

elif i[0] in ('dt','dr'):

tmp=round(tmp/1000,1)

elif i[0]=='dia':

tmp=round(tmp/10,1)

elif i[0]=='pop':

tmp=round(tmp,0)

tmp.fillna(0, inplace = True)

elif i[0]=='classes':

if var == 'dist':

for x in range(tmp.shape[0]):

if tmp.loc[x,1.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 1

elif tmp.loc[x,2.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 2

elif tmp.loc[x,3.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 3

elif tmp.loc[x,4.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 4

elif var == 'prov':

for x in range(tmp.shape[0]):

if tmp.loc[x,1.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 1

elif tmp.loc[x,2.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 2

elif tmp.loc[x,3.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 3

elif tmp.loc[x,4.0]== np.nanmax([tmp.loc[x,1.0],tmp.loc[x,2.0],tmp.loc[x,3.0], tmp.loc[x,4.0]]):

tmp.loc[x,'classes'] = 4

else:

for x in range(tmp.shape[0]):

if (tmp.loc[x,4.0] > 0):

tmp.loc[x,'classes'] = 4

elif (tmp.loc[x,3.0] > 0) & math.isnan(tmp.loc[x,4.0]):

tmp.loc[x,'classes'] = 3

elif (tmp.loc[x,2.0] > 0) & math.isnan(tmp.loc[x,4.0]) & math.isnan(tmp.loc[x,3.0]):

tmp.loc[x,'classes'] = 2

elif (tmp.loc[x,1.0] > 0) & math.isnan(tmp.loc[x,4.0]) & math.isnan(tmp.loc[x,3.0]) & math.isnan(tmp.loc[x,2.0]):

tmp.loc[x,'classes'] = 1

tmp.fillna(0, inplace = True)

tmp["rr"] = round(100\*(tmp[1.0]/(tmp[1.0]+tmp[2.0]+tmp[3.0]+tmp[4.0])),0)

tmp["rrd"] = round(100\*(tmp[2.0]/(tmp[1.0]+tmp[2.0]+tmp[3.0]+tmp[4.0])),0)

tmp["rm"] = round(100\*(tmp[3.0]/(tmp[1.0]+tmp[2.0]+tmp[3.0]+tmp[4.0])),0)

tmp["u"] = round(100\*(tmp[4.0]/(tmp[1.0]+tmp[2.0]+tmp[3.0]+tmp[4.0])),0)

else:

tmp=round(tmp,0)

bounds = pd.concat([bounds, tmp], axis=1)

### Filter columns for output

if var=="comms":

bounds = bounds[["geometry",'timecities','dr','dt','classes','rr','rrd','rm','u','pop']]

else:

bounds = bounds[["geometry",'od','timecities','dia','cholera','s\_unimp','s\_imp','s\_imp\_other','s\_piped','w\_unimp','w\_imp','w\_imp\_other','w\_piped','w\_surface','edu\_w','edu\_m','u5m','dr','dt','classes','rr','rrd','rm','u','pop']]

bounds.set\_geometry('geometry', inplace=True

### Output file

bounds.to\_file(country\_path+var+'/'+cc+'\_multivariable\_noadmin\_'+var+suffix+'.shp')

### Create CSV of column minimums and maximums

min = pd.DataFrame(bounds.min())

max = pd.DataFrame(bounds.max())

varvals= pd.concat([min, max], axis=1)

varvals.to\_csv(country\_path+var+'/'+cc+'\_'+var+suffix+'\_varvals.csv')

#%%

## Add administrative identifiers

#Map generation SOP

from qgis.core import \*

import qgis.utils

import processing

import os # This is is needed in the pyqgis console also

dir = "/Users/karastuart/Dropbox (Aquaya)/WASHPaLS 2/PEFO/SPT-data/datasets/" # UPDATE

directory = os.fsencode(dir)

cc = "gha" # country code # UPDATE

var = "comms" # variable name, select either 'pixel', 'dist', or 'prov' # UPDATE

suffix = "" # UPDATE - leave blank unless this is a second subnational boundary of the same resolution (e.g., add health districts as well as admin disticts. In this case, add a suffix such as "\_health" when running for health districts.)

dist = "adm2" # UPDATE number to be 2 or 3 as needed

### Check that a spatial index exists, and create it

def spatialindex(in\_fn):

processing.run("native:createspatialindex", {'INPUT':in\_fn})

infn = dir + cc + "/"+var+"/"+cc+"\_multivariable\_noadmin\_"+var+suffix+".shp"

spatialindex(infn)

##### Set join function and predicate value per variable #####

if var == "pixel" or var == "comms":

infn = dir + cc + "/dist/"+cc+"\_"+dist+suffix+".shp"

spatialindex(infn)

def spatial\_join(in\_fn, join\_fn, join\_fields, out\_fn):

processing.run("native:joinattributesbylocation", {'INPUT':in\_fn,\

'JOIN':join\_fn,'PREDICATE':[0],'JOIN\_FIELDS':join\_fields,'METHOD':2,\

'DISCARD\_NONMATCHING':False,'PREFIX':'','OUTPUT':out\_fn})

joinfn = dir + cc + "/dist/"+cc+"\_"+dist+suffix+".shp"

elif var == "dist":

infn = dir + cc + "/dist/"+cc+"\_"+dist+suffix+".shp"

spatialindex(infn)

def spatial\_join(in\_fn, join\_fn, join\_fields, out\_fn):

processing.run("native:joinattributesbylocation", {'INPUT':in\_fn,\

'JOIN':join\_fn,'PREDICATE':[2],'JOIN\_FIELDS':join\_fields,'METHOD':2,\

'DISCARD\_NONMATCHING':False,'PREFIX':'','OUTPUT':out\_fn})

joinfn = dir + cc + "/dist/"+cc+"\_"+dist+suffix+".shp"

else:

infn = dir + cc + "/prov/"+cc+"\_"+dist+suffix+".shp"

spatialindex(infn)

def spatial\_join(in\_fn, join\_fn, join\_fields, out\_fn):

processing.run("native:joinattributesbylocation", {'INPUT':in\_fn,\

'JOIN':join\_fn,'PREDICATE':[2],'JOIN\_FIELDS':join\_fields,'METHOD':2,\

'DISCARD\_NONMATCHING':False,'PREFIX':'','OUTPUT':out\_fn})

joinfn = dir + cc + "/prov/"+cc+"\_"+dist+suffix+".shp"

##### Set remainder of parameters #####

infn = dir + cc + "/"+var+"/"+cc+"\_multivariable\_noadmin\_"+var+suffix+".shp"

### UPDATE - select the line that includes the correct boundary level names + geometry

### there will be 1 or 2 boundary levels depending on the country

#join\_fields = ['NAME\_1','NAME\_2','NAME\_3']

#join\_fields = ['NAME\_1','NAME\_2']

join\_fields = ['ADM1\_EN','ADM2\_EN']

#join\_fields = ['REGION','DISTRICT']

#join\_fields = ['ADM1\_FR','ADM2\_FR','Nom']

#join\_fields = ['ADM1\_FR']

#join\_fields = ['NAME\_1']

outfn=dir + cc + "/"+var+"/"+cc+"\_multivariable\_"+var+suffix+".shp"

##### Run spatial join #####

spatial\_join(infn, joinfn, join\_fields, outfn)

## Add pixel and administrative boundary data to settlement layer

#Map generation SOP

from qgis.core import \*

import qgis.utils

import processing

import os

dir = "/Users/karastuart/Dropbox (Aquaya)/WASHPaLS 2/PEFO/SPT-data/datasets/" # UPDATE

directory = os.fsencode(dir)

cc = "lbr" # country code # UPDATE

def spatialindex(in\_fn):

processing.run("native:createspatialindex", {'INPUT':in\_fn})

infn = dir + cc + "/comms/"+cc+"\_multivariable\_comms.shp"

spatialindex(infn)

infn = dir + cc + "/pixel/"+cc+"\_multivariable\_noadmin\_pixel.geojson" # CHECK

spatialindex(infn)

##### REPEAT FOR EACH BOUNDARY LEVEL (incl. suffixes) #####

infn = dir + cc + "/dist/"+cc+"\_multivariable\_noadmin\_dist.shp" # CHECK

spatialindex(infn)

def column\_join(in\_fn, join\_fn, join\_fields, prefix, out\_fn):

processing.run("native:joinattributesbylocation", {'INPUT':in\_fn,\

'JOIN':join\_fn,'PREDICATE':[0],'JOIN\_FIELDS':join\_fields,'METHOD':2,\

'DISCARD\_NONMATCHING':False,'PREFIX':prefix,'OUTPUT':out\_fn})

infn = dir + cc + "/comms/"+cc+"\_multivariable\_comms.shp"

joinfn = dir + cc + "/pixel/"+cc+"\_multivariable\_noadmin\_pixel.geojson" # CHECK

join\_fields = []

prefix = "p\_"

outfn=dir + cc + "/comms/"+cc+"\_multivariable\_comms\_join1.shp"

column\_join(infn, joinfn, join\_fields, prefix, outfn)

##### REPEAT FOR EACH BOUNDARY LEVEL #####

infn = dir + cc + "/comms/"+cc+"\_multivariable\_comms\_join1.shp" # IF REPEATED, ADJUST FOR THE EXPORT OF PREVIOUS REPEAT

joinfn = dir + cc + "/dist/"+cc+"\_multivariable\_noadmin\_dist.shp" # CHECK

join\_fields = []

prefix = "\_d" # CUSTOMIZE FOR EACH ADDED BOUNDARY LEVEL

outfn=dir + cc + "/comms/"+cc+"\_multivariable\_comms\_join.shp" # IF REPEATED, USE "\_join2", etc. and keep "\_join" for the file export

column\_join(infn, joinfn, join\_fields, prefix, outfn)

# Appendix II: Table of scripts

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Pathway** | | | | **Filename** | | **Description** |
| Sanitation Planning Tool | public | *images* | | | |  |
| src | compo-nents | Home | | HomeBanner | The background image and title sections of the homepage |
| HomeFooter | The footer of the homepage |
| HomeHowItWorks | The "How it works" section of the homepage |
| HomeMap | The map showing the coverage of the tool on the homepage |
| HomeRelatedResearch | The "Related Research section of the homepage |
| MapMenu | | DatasetInfoPopover | The menu that appears when the information icon on the filter menu is clicked, showing dataset information |
| DropdownMenu | The menu of dropdown selectors that appear when the user clicks "Select area(s)" |
| FilterMenu | The menu containers that appear when the user clicks on an indicator theme |
| FilterMenuContent | The data with indicator filters (range sliders + checkbox selectors) that feed the filter menus |
| MapMenu | The left-hand menu container holding the resolutions, areas, and filter menus - manipulating these filters updates the coverage of the map |
| MapResolutions | The menu that appears when the user clicks on "Set resolution" - this updates the dataset visible on the map |
| subcomponents | | CalculatorWidget | The function that calculates the number of polygons remaining on the map - this appears when the user clicks on "Data export" from the Tab Box |
| Legend | The legend menu |
| LegendStyles | The reference list for color ramp settings per indicator |
| Loader | The loading indicator that appears when the data export is loading |
| Loader\_2 | A function that feeds into the Loader |
| MapPopper | The menu that appears when a user clicks on a map feature - this includes the secondary menu that appears when the user clicks "See more" |
| NoDataAlert | The banner that appears in the top of the map page in countries with data shortages |
| SliderNonLinear | A custom slider feature for range sliders that are not linear. We can set the number of slider values and set custom values for each. |
| Survey | The user survey that appears when the user returns to the homepage after visiting the map page |
| Tour | The function that controls the tour which appears on the user's first visit to a map page - this is also accessed when a user clicks "Restart tour" from the left-hand map menu |
| TabBox | | Export | The export data menu found within the Tab Box |
| TabBox | The Tab Box menu container containing "Communities/Settlements" and "Export data" menus - located on the right side of the map |
| UploadButton | The "Communities/Settlements" menu from the Tab Box - this includes the show/hide settlements toggle (which activates a secondary menu with warning) |
| Map | | | Map.js holds all of the components and subcomponents (including various menus) of the map pages, and configures and manipulates the map. |
| MapSelector | | | The dropdown list found in the navigation bar, and under the map on the homepage - this loads the user-selected country map page |
| images | *icons* | | | Various icon image files used throughout the site |
| layouts | default | | | The navigation bar menu - including the auto-resizing according to the application window width |
| state | countries | | cambodia | Country-specific initial state data, including links to datasets stored on GeoDB |
| dem\_rep\_congo | Country-specific initial state data, including links to datasets stored on GeoDB |
| ethiopia | Country-specific initial state data, including links to datasets stored on GeoDB |
| ghana | Country-specific initial state data, including links to datasets stored on GeoDB |
| kenya | Country-specific initial state data, including links to datasets stored on GeoDB |
| liberia | Country-specific initial state data, including links to datasets stored on GeoDB |
| madagascar | Country-specific initial state data, including links to datasets stored on GeoDB |
| mali | Country-specific initial state data, including links to datasets stored on GeoDB |
| mozambique | Country-specific initial state data, including links to datasets stored on GeoDB |
| nepal | Country-specific initial state data, including links to datasets stored on GeoDB |
| niger | Country-specific initial state data, including links to datasets stored on GeoDB |
| nigeria | Country-specific initial state data, including links to datasets stored on GeoDB |
| rwanda | Country-specific initial state data, including links to datasets stored on GeoDB |
| s\_sudan | Country-specific initial state data, including links to datasets stored on GeoDB |
| senegal | Country-specific initial state data, including links to datasets stored on GeoDB |
| sudan | Country-specific initial state data, including links to datasets stored on GeoDB |
| tanzania | Country-specific initial state data, including links to datasets stored on GeoDB |
| uganda | Country-specific initial state data, including links to datasets stored on GeoDB |
| CountryTemplate | | | The template used to enter country-specific initial state data, not linked to anything on the site |
| MapState | | | MapState stores all of the initial values for the map page, and stores and communicates updates to these values based on user clicks. The "state" is the memory of each component, remember when to re-render according to user interactions. |
| theme | theme | | | File that sets the global color palette and typologies |
| App.css | | | Style (spacing, font, and size) of the legend |
| index.css | | | Additional style for application body |
| views | About | | | The content found on the "About" page |
| Datasets | | | The content found on the "Datasets Overview" page |
| FAQ | | | The content found on the "FAQ" page |
| Home | | | The container that brings together the pieces of the homepage - see src > components > Home |
| MapDetail | | | The placement of the Map.js within a container |
| Privacy | | | The content found on the "Privacy policy" page, opned via a link within the footer |
| App | | | | | The overall framework of the website - this brings together the navigation bar and the views |
| index | | | | | The container for App.js - this renders the code as a react application |
| .env.local | | | | | Data stored locally (included in .gitignore), here it contains our sensitive GeoDB API information |
| .gitignore | | | | | Dictates the files withing the codebase file tree to ignore when pushing to GitHub. It is common to ignore large data files, node modules, and local environment files. |
| package.json | | | | | Auto-created list of the packages (and version numbers) used in the application. Updating or installing packages will automatically update this file. |
| README.md | | | | | Instructions for new web developers to manage the application |
| static.json | | | | | Auto-created file - Should not need to edit |
| yarn.lock | | | | | Auto-created file listing the installed packages and the associated installed dependencies - DO NOT EDIT |