USAID
FROM THE AMERICAN PEOPLE

SANPLAN: THE SANITATION PLANNING TOOL

OPERATING MANUAL

Prepared by: The Aquaya Institute
LAST UPDATED: MAY 2023

TABLE OF CONTENTS

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. SDG 6.2 calls for universal access to adequate and equitable sanitation and hygiene by 2030 and to end open defecation, paying special attention to the needs of women and girls and those in vulnerable situations. 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 (https://www.sanplan.app/). 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 (https://getgeodb.com/). Future product managers have the option to change the source of the Carto server if they prefer not to use GeoDB.

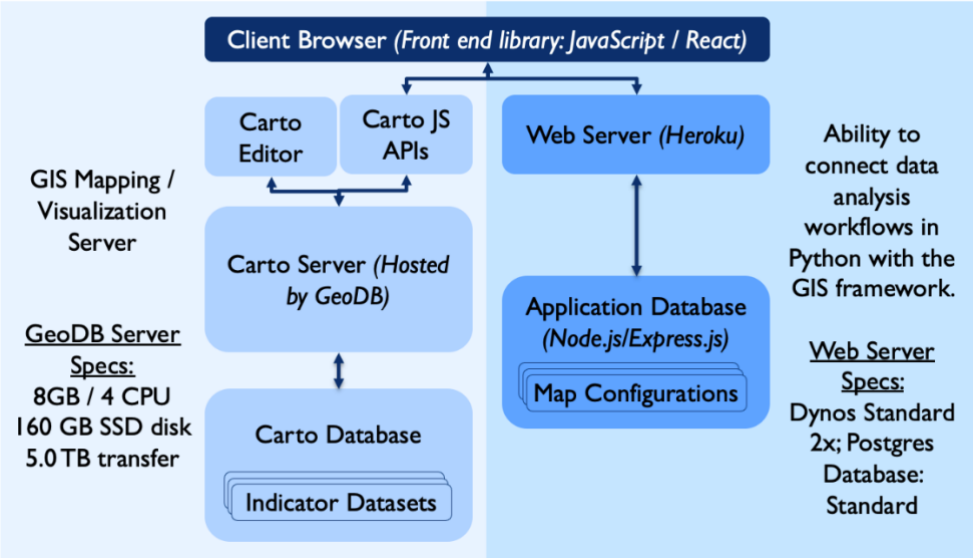

FIGURE 1. SANPLAN APPLICATION ARCHITECTURE AND TECHNOLOGY STACK. JS = "JAVASCRIPT", API = "APPLICATION PROGRAMMING INTERFACE".

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.

The codebase was developed from the Create React App (https://create-react-app.dev/) framework.

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 (e.g., toggles the visibility of a data layer, filters the map, adjusts the zoom, etc.). 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.

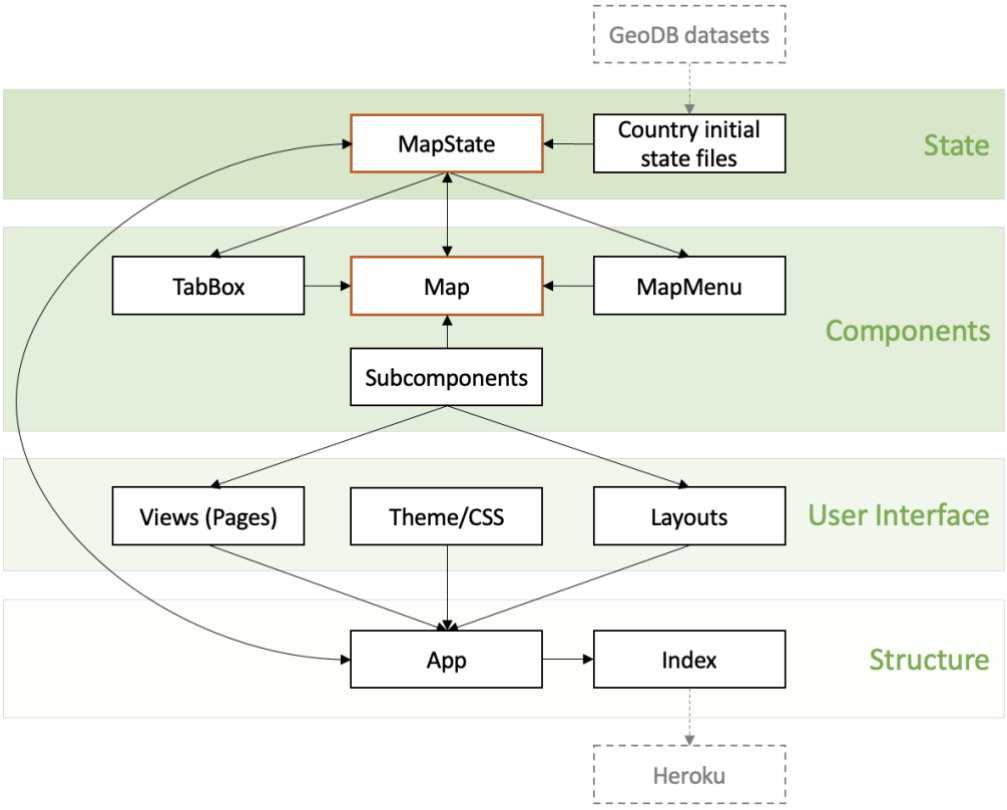

FIGURE 2. SANPLAN INFORMATION FLOW SCHEME.

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

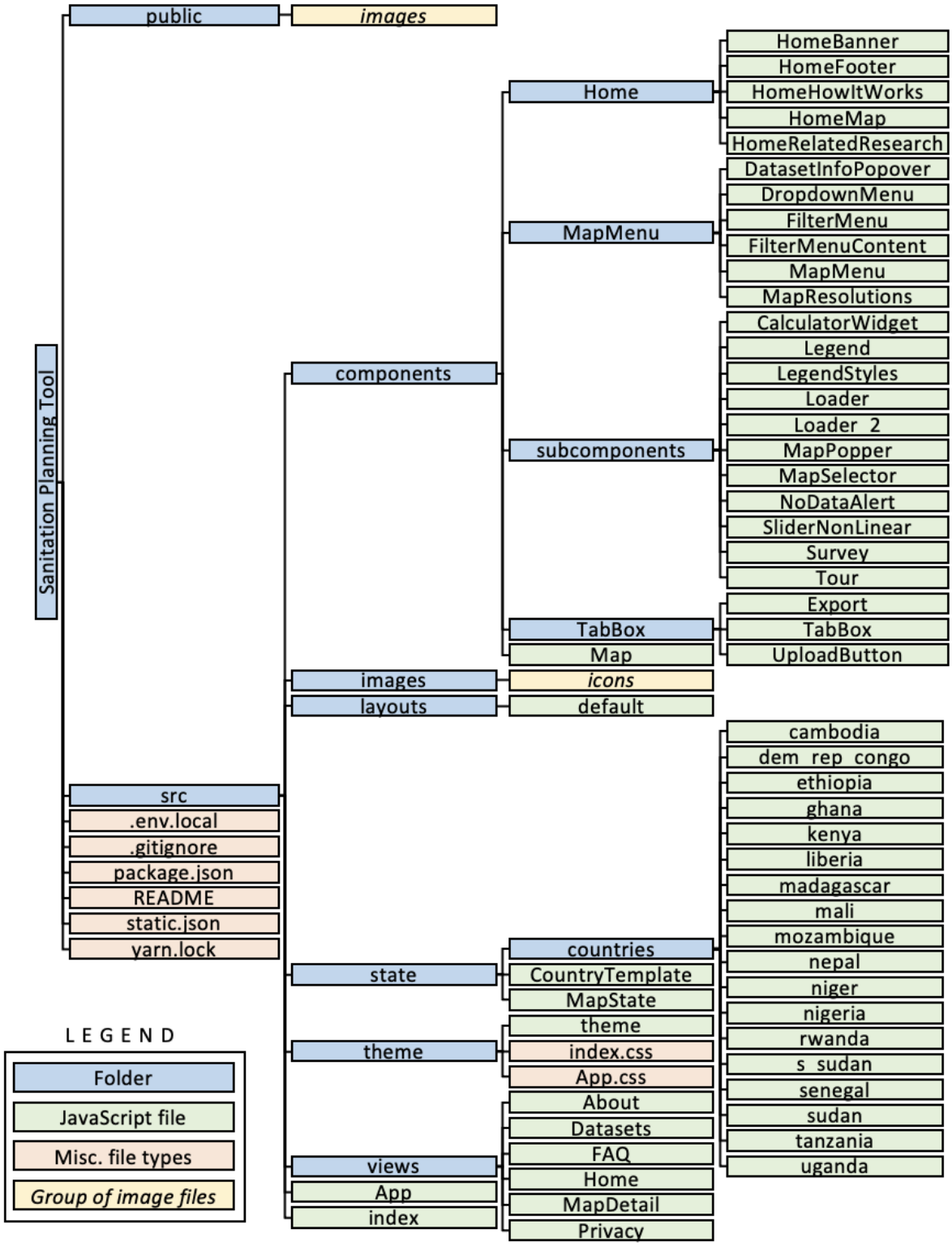

FIGURE 3. THE SANPLAN CODEBASE FILE TREE.

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>
```

7. In the terminal of the code editor (VSCode), install all the dependencies by running "yarn install"
8. Each time to start the development server, run "yarn start" to open a browser tab which will reload as you edit the codebase.
9. 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
Administrative boundary shapefiles (country-specific)	Polygon	Global	OCHA	2022
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." OCHA = "UNITED NATIONS OFFICE FOR THE COORDINATION OF HUMANITARIAN AFFAIRS."

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.

ADDING COUNTRIES AND/OR INDICATOR LAYERS

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. 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.

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")
    a. Create a "global" folder (skip after initial setup)

i. 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.

b. Create a country folder.

The name of the folder should be the three-letter country code as defined by IBAN

i. 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

2. Gather country-level datasets and place in appropriate file tree folder

    a. Source administrative boundary shapefiles from OCHA on HDX. Find the dataset by navigating to HDX 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").

        i. Extract country level (adm0) -> country folder

        ii. 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.

        iii. 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.

    b. Source population raster data from WorldPop -> pop folder

        i. Select the 2020 constrained estimates for the country of interest.

        ii. Ensure the filename follows "countrycode_ppp_2020_constrained.tif"

    c. Source roads shapefiles from Humanitarian Data Exchange -> 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.

        i. Of the file type options, download the lines_shp.zip folder

    d. Source settlements geodatabase data from GRID3 -> 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.

        i. On the website, click the "settlements" tab, locate the desired country and download the associated "settlements extent" geodatabase.

        ii. Rename the geodatabase folder to be "countrycode_GRID3.gdb" (e.g. "gha_GRID3.gdb" for Ghana).

3. Produce custom layers (automated Python/QGIS scripts – Appendix 1, download required packages as needed)

4. If a GRID3 dataset does not exist for the desired country, run settlement script.

    If a GRID3 dataset exists, skip to step 5.

    a. Run Script 0 (0_settlements_noGRID3.py) in QGIS

        i. 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

ii. 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.

5. Run data treatment scripts

a. Run Script 1 (1_towns_construct_GRID3_2023.py) in QGIS

i. 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.

b. Run Script 2 (2_class_dr_dt_construct_2023.py) in QGIS

i. 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.

ii. AFTER RUNNING:

1. Make sure classification layer has exactly 1-4 categories.

c. Run Script 3 (3_zonal_stats_final_2023.py) for either pixel- or sub-country-level in Python

i. 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").

4. 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 districts. 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.

5. 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.

6. 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"]]"*

ii. 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.

d. Repeat previous step (5.c) for the other variable option (pixel or sub-country)

e. Run Script 4 (4_add_admin_2023.py) for either pixel- or sub-country-level in QGIS

i. 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").

4. 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.

5. 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.

6. 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_#".*

f. Repeat previous step (4.e) for the other variable options (pixel = "pixel", settlements = "comms", sub-country = "dist")

g. Run Script 5 (5_comms_join_2023.py) for either pixel- or sub-country-level in QGIS

   i. 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).

6. 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.

7. Update codebase

   a. Copy/paste the content of the country template (Sanitation planning tool > src > state > CountryTemplate.js) into a new javascript file within the MapState folder.

   b. Update and add missing information

i. Name the file the full country name, in lowercase.

ii. Replace "country" in line 1 with the country name.

iii. Add the country name to "name" in line 2, capitalized.

iv. Add the country name to "mapID" in line 3, lowercase.

v. Use Google Maps to approximate the centroid of the country, place the lat/longs in lines 4-5 (as numbers).

vi. Replace "cc" in "carto_tableName" with the country code for each layer.

vii. 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.

viii. 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"

ix. Add layers as needed for additional boundary-levels and repeat steps vi-viii (copy and paste the district-level layer and update accordingly).

x. DO NOT alter:

1. Layer-level:

a. carto_source, carto_layer, carto_style, visible, accessCounter, washCounter, socioCounter, healthCounter,

2. Filter-level:

a. name (except boundary names), unit, type, column_name, subcategory

b. For classification layer, do not alter the value field

c. Add the country name to the list of "maps" in the initial state (sanitation planning tool > src > state > MapState.js).

d. Check Homepage that the country appears on the map selector.

e. Check country dropdown that the name appears.

f. Check map for functionality.

g. Ensure Prettier and ESLint have run prior to committing changes to the codebase (see Best Practices, below).

ROUTINE MAINTENANCE

UPDATE PACKAGES

It is recommended to regularly check that the application is using the latest versions of its dependencies and packages. Package updates are released regularly to introduce bug fixes, security patches, and new features. It is crucial to make updating packages part of your routine maintenance tasks. Regularly repeat the steps outlined in this manual to keep your React application up to date.

1. Open your preferred terminal or command prompt application (e.g., VSCode). Ensure that you are in the root directory of the SanPlan application, where the **package.json** file is located.
2. Check Current Package Versions Before updating your packages, it is a good practice to check the current versions of your dependencies. You can do this by running the following command:

```
yarn outdated
```

   This command will display a list of all packages that have newer versions available. You can see the current version, the latest version, and the recommended version for each package (if specified).

3. To update packages using the interactive mode of Yarn, execute the following command:

```
yarn upgrade-interactive
```

   This command launches an interactive interface that shows all the outdated packages in your project. The packages are color-coded as follows:

   - **Red**: These updates contain breaking changes that might require modifications in your code to maintain compatibility. Reviewing the release notes, changelogs, or migration guides provided by the package maintainers is crucial. Carefully consider the impact of the changes and make the necessary code modifications before proceeding with the update.
   - **Yellow**: These updates introduce new features or improvements that are not breaking changes. However, it is still important to review the release notes to understand the changes and ensure that they won't adversely affect your application. Test your application after the update to verify that it functions as expected.
   - **Green**: These updates are patch releases that include bug fixes and security patches. They generally pose the lowest risk of introducing compatibility issues. It is still a good practice to review the release notes and test your application to ensure stability.

4. Within the interactive interface, you will have options to handle updates for each package:

   - Press the **Spacebar** to select or deselect a package for update.
   - Press the **Enter** key to proceed with the selected updates.

   For red updates (breaking changes), we recommend handling them one by one and ensure that the code remains compatible. Consult the package documentation, seek community support, or contact the package maintainers for guidance if needed.

For yellow and green updates, proceed with updating the recommended packages.

5. After the package updates are complete, verifying that your React application still functions as expected is essential. Run your application and perform thorough testing to ensure all features and functionalities work correctly. To do this, use the following command:

```
yarn start
```

Be sure to test the site's more complex functions and manipulate any features directly connected to the altered packages.

6. Yarn automatically updates the **yarn.lock** and **package.json** files, which track the versions of packages used in your project. Double check these changes are reflected.
7. Commit and push the changes to GitHub when you are satisfied with the updates.

BEST PRACTICES

1. Linting and Code Formatting:

Linting ensures that the code is formatted consistently, automatically adjusting the scripts as needed to match formatting conventions and user-defined settings. The steps below assume VSCode is being used to edit the codebase.

1. Install the ESLint and Prettier extensions in VSCode by navigating to the Extensions tab and searching for "ESLint" and "Prettier."
2. Open the VSCode settings (**settings.json**) by clicking on "Preferences" in the menu bar, selecting "Settings," and choosing "Open Settings (JSON)."
3. Add the following configuration to the **settings.json** file to enable automatic formatting on file save using ESLint and Prettier:

```
{
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "editor.formatOnSave": true
}
```

4. Save the **settings.json** file.
5. Linting checks and code formatting will now run automatically on file save in VSCode using the ESLint and Prettier extensions. Linting issues will be highlighted, and code formatting will be applied based on default configurations unless a unique **.eslintrc.js** file is created.

2. **Monitoring Error Logs and Logging:**

- Monitoring error logs and implementing logging practices in Create React App can be beneficial for identifying and resolving issues. Here's an approach:
    - Set up logging libraries like **winston** or use existing logging solutions like **console.log** or **console.error**.
    - Define appropriate log levels and log statements throughout your codebase to capture relevant information.
    - Configure log output destinations, such as console output or log files.
    - Monitor logs regularly to identify errors or exceptions and take appropriate actions to resolve them.
    - Consider integrating third-party logging services or log management tools for centralized log aggregation and analysis.
    - Ensure that logging is enabled in production environments and properly handled, adhering to security and privacy regulations.

3. **Review Security Practices:**

1. Stay updated with security news and advisories related to React, and your project's dependencies. Ensure you are receiving security alerts from Heroku and GitHub.
    a. At any time, you can navigate to https://status.heroku.com/ to check the current status of Heroku services and view any incident reports or current and past security warnings.

ii. Regularly review and update dependencies to their latest versions, considering security patches and bug fixes (see Updating Packages section, above).

iii. Follow secure coding practices, such as input validation, proper handling of user data, and protection against common security threats like cross-site scripting (XSS) and cross-site request forgery (CSRF). If there are security issues, warnings can often be found in the Chrome Developer Tools console (see Debugging and Error Handling, below).

iv. Implement authentication and authorization mechanisms securely. For example, ensure that the local environment file containing sensitive API key and login information is never loaded to the GitHub repository (it should be listed in your **.gitignore** file).

v. Consider using security tools and libraries specifically designed for React applications, such as security scanners or vulnerability detection tools.

vi. Perform regular security audits and penetration testing on your application to identify and mitigate potential security risks.

8. **Monitoring Application Performance:**

   Use the Lighthouse extension Chrome for performance monitoring.

   1. Build your application for production by running the following command in the terminal:

```
yarn build
```

   2. Deploy the built application to a test or staging environment.
   3. Open the application in your browser and use the Lighthouse extension to analyze its performance metrics, such as page load time, accessibility, etc.
      - To open Lighthouse right-click anywhere on the page and select "Inspect" from the context menu. Alternatively, you can use the keyboard shortcut **Ctrl + Shift + I** (Windows/Linux) or **Command + Option + I** (Mac) to open the Developer Tools and navigate to the Lighthouse tab
   4. Review the Lighthouse reports and identify areas for improvement.
   5. Make necessary optimizations to your code, assets, or configuration based on the performance analysis.

DEBUGGING AND ERROR HANDLING

Use Chrome or Firefox's built-in developer tools to debug and monitor site performance. The instructions below are assuming the use of Google Chrome:

1. Open your web page or application in the Google Chrome browser. You can do this either in a local server built through VSCode (the site that opens after running "yarn start"), or you can use the live website, depending on what version you are investigating.
2. Right-click anywhere on the page and select "Inspect" from the context menu. Alternatively, you can use the keyboard shortcut **Ctrl + Shift + I** (Windows/Linux) or **Command + Option + I** (Mac) to open the Developer Tools.
3. In the Developer Tools panel that opens, you'll see several tabs at the top. Click on the "Sources" tab.
4. In the Sources tab, you'll see a list of files associated with your web page or application (you should be able to navigate to the directory of the codebase and explore the entire file tree). Find the JavaScript file you want to debug. If you are investigating an error thrown by the site, open the console and expand the error to see what file and what script is causing the error. You can click the link to the location of the error and automatically navigate to the correct script and line.
5. Once you've located the JavaScript file, you can set breakpoints by clicking on the line number where you want the execution to pause. A red dot will appear, indicating the breakpoint.

6. Now, interact with your web page or application, triggering the code execution that you want to debug. When the code reaches a breakpoint, it will pause, and you can inspect variables, step through the code, and analyze its behavior.

7. In the right-side panel of the Sources tab, you'll find several debugging options. Here are a few key ones:

    - Step Over (**F10**): Executes the current line of code and moves to the next line. If the current line contains a function call, it will execute the entire function without stepping into it.

    - Step Into (**F11**): Executes the current line of code and steps into the function if the current line contains a function call.

    - Step Out (**Shift + F11**): Steps out of the current function and continues execution until the function returns.

    - Resume Script Execution (**F8**): Continues the script execution until the next breakpoint or until the code finishes executing.

8. While debugging, you can inspect variables by hovering over them to see their values or by adding them to the Watch panel on the right side. You can also modify variable values manually to test different scenarios.

9. If you encounter an error or exception, the Developer Tools will pause at the line of code where the error occurred, and you can inspect the error message and the call stack to identify the issue.

10. Additionally, you can use the Console tab in the Developer Tools to log messages or execute JavaScript code manually. This can be helpful for testing small snippets of code or quickly checking variable values.

11. Experiment with different breakpoints, stepping options, and variable inspections to identify and resolve issues in your JavaScript code.

12. Once you've finished debugging, you can close the Developer Tools panel by clicking on the close button (usually an "X") or by using the keyboard shortcut **Ctrl + Shift + I** (Windows/Linux) or **Command + Option + I** (Mac) again.

13. If you manually implemented changes that successfully debugged the code, you will need to mirror these changes in your codebase and commit these fixes to GitHub. It is recommended to fully debug and test the site using the local server before committing updates to GitHub.

For deeper dives into the performance of the site, use the **React Developer Tools Extension**:

1. Install the React Developer Tools Extension from the Chrome webstore

2. Once installed, in the Developer Tools window, you should see a tab called “React”

3. In the React tab, you can navigate through the component tree, inspect props and state values, and analyze the component hierarchy.

4. You can also make changes to props or state values directly in the React Developer Tools extension to see the impact on your React application in real-time.

5. Experiment with different components, inspect their properties, and observe how they interact with each other using the React Developer Tools.

6. For additional functionalities, refer to React Developer Tools tutorial: https://youtu.be/Cey7BS6dE0M

APPENDIX I: DATA TREATMENT SCRIPTS

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

DEFINING SETTLEMENT BOUNDARIES (NON-AFRICAN COUNTRIES)

Filename: 0_settlements_noGRID3.py

```
##### 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)
#%%
```

Filename: 1_towns_construct_GRID3_2023.py

```
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
```

Filename: 2_class_dr_dt_construct_2023.py

```
#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)
```

Filename: 3_zonal_stats_final_2023.py

```
#!/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=-
3399999952144364200000000000000000000000)
    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=-
3399999952144364200000000000000000000000)
    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_s
urface','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')
#%%
```

Filename: 4_add_admin_2023.py

```
#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)
```

Filename: 5_comms_join_2023.py

```
#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	components	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