## Sampling Design: Script overview

This chapter is designed for users with a basic understanding of R programming, and provides a comprehensive guide to implementing a soil sampling design for soil monitoring and mapping.

💡**Note**: Ensure you have the tinytex, knitr, rmarkdown, and rstudioapi R packages installed to successfully run and render R Markdown documents before running this script.  
Additionally, install tinytex::install\_tinytex() by running tinytex::install\_tinytex() to enable LaTeX support for PDF generation.

### Setting Up the Environment and Loading Libraries

First, set the working directory to the location of your R script. This ensures that all paths in the script are relative to the script’s location.

setwd(dirname(rstudioapi::getActiveDocumentContext()$path))  
setwd("../") # Move wd down to the main folder  
getwd()

## [1] "D:/FAO/github/SoilFER/sampling design"

### Install required libraries

The script uses several R libraries, each serving a specific purpose. Below is a quick summary of the key functions of these libraries:

Summary of required R libraries and Their purposes

| Package | Purpose |
| --- | --- |
| sp | Handling spatial data and performing geographic operations. |
| terra | Working with raster and vector spatial data, including transformations and analyses. |
| raster | An older but still widely used library for handling raster data. The terra R package has been designed as its replacement. |
| sf | Modern and efficient library for handling vector spatial data. |
| sgsR | Tools for spatial grid sampling and geostatistics. |
| entropy | Estimating entropy and related measures. |
| tripack | Generating triangulations and handling Voronoi tessellations. |
| tibble | An enhanced version of data frames with better usability. |
| manipulate | Creating interactive visualizations. |
| dplyr | Data manipulation and transformation. |
| synoptReg | Clustering and Principal Component Analysis (PCA) for spatial data. |
| doSNOW | Parallel computing for speeding up tasks. |
| Rfast | Tools for faster computation of statistical methods. |
| fields | Tools for spatial data interpolation and visualization. |
| ggplot2 | Data visualization using the grammar of graphics. |
| rassta | Raster spatial sampling and analysis tools. |

Some libraries might need to be installed from GitHub before loading. Uncomment and run the lines below to install the remotes and then the synoptReg R packages. If you have not faced any issues loading all these libraries at once, there is no need to run the commented lines.

# Install synoptReg package from github  
 #install.packages("remotes") # Install remotes if not installed  
 #remotes::install\_github("lemuscanovas/synoptReg")  
  
packages <- c("sp", "terra", "raster", "sf", "sgsR", "entropy",   
 "tripack", "tibble", "manipulate", "dplyr", "synoptReg",   
 "doSNOW", "Rfast", "fields", "ggplot2", "rassta")  
  
invisible(lapply(packages, library, character.only = TRUE))  
  
rm(packages)

An alternative and beginner-friendly way of install R packages using RStudio is through the **Packages** tab in RStudio. First of all, locate this tab in the bottom-right pane of RStudio and click on it. Then, click the *Install* button at the top of the tab. In the dialog box that appears, type the name of the package (e.g., terra) in the text field. Make sure that *Install dependencies* box is checked, and finally, click *Install* to download and install the package.

### Define variables, and parameters

Several variables and parameters must be defined to establish a consistent framework for the script.

The ISO.code variable is used to define the unique [Alpha-3 ISO code](https://www.iso.org/obp/ui/#search) of the country of interest. This standard ensures consistency when naming variables and file paths throughout the script.

ISO.code <- "ZMB"

💡**Note**: In this example, **“ZMB”** is used as the ISO code for Zambia.

The SoilFER project focuses on three primary land-use types: croplands (80%), grasslands (15%), and forests (5%). In this example, the landuse variable specifies *croplands* as the focus of analysis.

landuse <- "crops"

This section sets the file paths for accessing data such as raster files, shapefiles, and other resources. Additionally, it ensures the creation of a results folder for storing processed files related to the specified land use. Organizing file paths ensures that all resources and results are systematically stored and accessed.

# Path to data folders  
raster.path <- "data/rasters/"  
shp.path <- "data/shapes/"  
other.path <- "data/other/"  
landuse\_dir <- paste0("data/results/", landuse, "/")  
   
# check if landuse directory exists if not create it  
if (!file.exists(landuse\_dir)){dir.create(landuse\_dir)}  
  
results.path <- landuse\_dir

The epsg variable specifies the coordinate reference system used for spatial data. EPSG codes are globally recognized standards for geospatial datasets. If you do not have this information, you can search [here](https://epsg.io/).

epsg <- "EPSG:3857"

Users can calculate the total number of sampling sites based on predefined values. For example,out of 300 total sites (i.e., samples), 70% are selected for analysis (share), as this percentage aligns with the proportion of croplands. The final number of samples is stored in nsamples. Adjusting the number of sites ensures that only a subset of the data is analyzed.

nsamples <- 300  
share <- 0.80  
nsamples <- nsamples \* share

Primary Sampling Units (PSUs) and Secondary Sampling Units (SSUs) are defined here, along with their sizes and the number of target/alternative SSUs. PSUs are large units of 2 km x 2 km used as a sampling framework, while SSUs are smaller units (1 hectare or 100 m x 100 m) nested within PSUs. Tertiary Sampling Units (TSUs) represent individual sampling points within SSUs.

# Define the number of PSUs to sample  
n.psu <- round(nsamples/8)  
   
# Define PSU and SSUs sizes   
psu\_size <- 2000 # (default = 2km x 2 km)  
ssu\_size <- 100 # (default = 100m x 100m = 1 ha)  
   
# Define number of target and alternative SSUs at each PSU  
num\_primary\_ssus <- 4  
num\_alternative\_ssus <- 3  
   
# Define number of TSUs at each SSU  
number\_TSUs <- 3

The iterations variable defines how many times the clustering algorithm should run to optimize results. Increasing the number of iterations improves the accuracy of cluster assignment but may increase computation time. 10 is enough.

iterations <- 10

The percent\_crop variable specifies the minimum percentage of crop coverage required for a PSU to be included in the sampling design. This filter ensures that sampling efforts focus on areas with significant crop presence, improving relevance and efficiency.

# Define the minimum crop percent in selected PSUs  
percent\_crop <- 10

### Functions

**Covariate Space Coverage (CSC)** focuses on ensuring sampling points are well-distritbuted in terms of enviromental covariates, and useful for scenarios where the region has heterogeneous environmental conditions.

The CSIS function is the backbone of clustering sampling units in the covariate space. It ensures optimal distribution of sampling units based on covariate data while considering any fixed legacy data. Here’s a step-by-step explanation:

**Step-by-step Explanation**:

* **Input Parameters**:
  + fixed: A dataset of preselected (fixed) sampling points.
  + nsup: Number of additional (new) sampling points to be selected.
  + nstarts: Number of random starting points for the clustering algorithm.
  + mygrd: A grid of covariate data for the region of interest.
* **Workflow**:
  + **Initialization**:
    - Extract the fixed points (fixed) and identify their indices (units).
    - Exclude the fixed points from the grid of covariates (mygrd\_minfx).
  + **Clustering**:
    - Randomly initialize nsup new sampling points (centers\_sup).
    - Combine the fixed points and new centers to form the initial cluster centers (centers).
  + **Iterative Optimization**:
    - Compute the distance between each point in mygrd and the cluster centers.
    - Assign each point in mygrd to the nearest cluster center.
    - Update the cluster centers by calculating the mean of the points assigned to each cluster.
    - Check for convergence: Stop if the cluster centers do not change significantly.
  + **Evaluation**:
    - Calculate the Mean Squared Shortest Distance (MSSSD) to evaluate the clustering quality.
    - Save the best clustering result among all nstarts attempts.
* **Output**:
  + centers: The optimized cluster centers, including both fixed and new points.
  + cluster: The cluster assignment for each point in the grid.

💡**Note**: This function is particularly useful for creating spatially balanced sampling designs by considering existing sampling points and covariate distributions.

# Define Covariate Space Coverage function  
# Clustering CSC function with fixed legacy data  
CSIS <- function(fixed, nsup, nstarts, mygrd) {  
 n\_fix <- nrow(fixed)  
 p <- ncol(mygrd)  
 units <- fixed$units  
 mygrd\_minfx <- mygrd[-units, ]  
 MSSSD\_cur <- NA  
 for (s in 1:nstarts) {  
 units <- sample(nrow(mygrd\_minfx), nsup)  
 centers\_sup <- mygrd\_minfx[units, ]  
 centers <- rbind(fixed[, names(mygrd)], centers\_sup)  
 repeat {  
 D <- rdist(x1 = centers, x2 = mygrd)  
 cluster <- apply(X = D, MARGIN = 2, FUN = which.min) %>% as.factor(.)  
 centers\_cur <- centers  
 for (i in 1:p) {  
 centers[, i] <- tapply(mygrd[, i], INDEX = cluster, FUN = mean)  
 }  
   
 #restore fixed centers  
 centers[1:n\_fix, ] <- centers\_cur[1:n\_fix, ]  
 #check convergence  
 sumd <- diag(rdist(x1 = centers, x2 = centers\_cur)) %>% sum(.)  
 if (sumd < 1E-12) {  
 D <- rdist(x1 = centers, x2 = mygrd)  
 Dmin <- apply(X = D, MARGIN = 2, FUN = min)  
 MSSSD <- mean(Dmin^2)  
 if (s == 1 | MSSSD < MSSSD\_cur) {  
 centers\_best <- centers  
 clusters\_best <- cluster  
 MSSSD\_cur <- MSSSD  
 }  
   
 break  
 }  
   
 }  
   
 print(paste0(s," out of ",nstarts))  
 }  
   
 list(centers = centers\_best, cluster = clusters\_best)  
}

The generate\_tsu\_points\_within\_ssu function focuses on generating TSUs within SSUs. It ensures that TSUs are distributed randomly within the defined SSUs while adhering to specific land-use constraints. This function is crucial for creating a hierarchical sampling structure, where TSUs represent the smallest sampling units.

**Step-by-step Explanation**:

* **Input Parameters**:
  + ssu: A SSU where TSUs will be generated.
  + number\_TSUs: Number of TSUs to be sampled within the SSU.
  + index: Index of the SSU within the PSU grid.
  + ssu\_type: The type of SSU (e.g., Target or Alternative).
  + crops: A raster layer defining the land-use constraints, ensuring TSUs fall within the desired land-use area (e.g., cropland).
* **Workflow**:
  + **SSU Masking**:
    - Convert the selected SSU to a spatial vector (SpatVector) format for masking.
    - Clip the land-use raster (crops) to the boundaries of the SSU, creating a localized raster for sampling.
  + **TSU Sampling**:
    - Use the sample\_srs function to randomly generate TSU points within the clipped raster area. This ensures TSUs are distributed across the SSU while respecting land-use constraints.
    - If the number of sampled points is less than the desired number (number\_TSUs), repeat the sampling process using the spatSample function until the required number of TSUs is obtained.
  + **Metadata Assignment**:
    - Assign metadata to the sampled TSUs, including:
      * **PSU\_ID**: The PSU ID containing the SSU.
      * **SSU\_ID**: The current SSU ID.
      * **TSU\_ID**: A sequential identifier for each TSU within the SSU.
      * **SSU\_Type**: The type of SSU (e.g., Target or Alternative).
      * **TSU\_Name**: A unique name for each TSU, combining PSU, SSU, and TSU identifiers.
* **Output**:
  + Returns a dataset containing the randomly sampled TSU points with metadata.

💡**Note**: This function ensures spatial consistency by sampling TSUs only within specified land-use areas, making it ideal for hierarchical sampling designs.

generate\_tsu\_points\_within\_ssu <- function(ssu, number\_TSUs, index, ssu\_type, crops) {  
 # Convert SSU to SpatVector for masking  
 ssu\_vect <- ssu\_grid\_sf[index, ]  
 # Clip the spatRaster to the SSU to focus the sampling within the SSU boundaries  
 clipped\_lu <- crop(crops, ssu\_vect)  
 # Generate random points within the clipped spatRaster using sample\_srs  
 sampled\_points <- sample\_srs(clipped\_lu, nSamp = number\_TSUs) # Ensure this is compatible or modify  
 # Make sure the TSUs falls within landuse areas  
 while (nrow(sampled\_points) < number\_TSUs) {  
 sampled\_points <- spatSample(clipped\_lu, size = number\_TSUs)  
 }  
   
 # Add metadata to the sampled points  
 sampled\_points$PSU\_ID <- selected\_psu$ID  
 sampled\_points$SSU\_ID <- index  
 sampled\_points$TSU\_ID <- seq\_len(nrow(sampled\_points))  
 sampled\_points$SSU\_Type <- ssu\_type  
 sampled\_points$TSU\_Name <- paste0(sampled\_points$PSU\_ID,".",sampled\_points$SSU\_ID,".",seq\_len(nrow(sampled\_points)))  
 return(sampled\_points)  
}

### Country boundaries and legacy data

In this step, file paths for country boundaries and legacy soil data are defined using the file.path function. These paths are used to locate and load the corresponding shapefiles. After that, the shapefiles are loaded (st\_read) and converted to an sf object (st\_as\_sf) and transformed (st\_transform) to the desired and pre-defined CRS using the EPSG code provided in earlier steps using the sf package.

# Define location of country boundaries  
country\_boundaries <- file.path(paste0(shp.path,"roi\_epsg\_3857.shp"))  
# Define location of soil legacy data  
legacy <- file.path(paste0(shp.path,"zmb\_legacy\_v2\_clipped\_epsg\_3857.shp"))  
  
# Load and transform the country boundaries  
country\_boundaries <- sf::st\_read(country\_boundaries, quiet=TRUE)  
  
if(sf::st\_crs(country\_boundaries)$epsg!=epsg){  
 country\_boundaries <- sf::st\_transform(country\_boundaries, crs = epsg)  
}

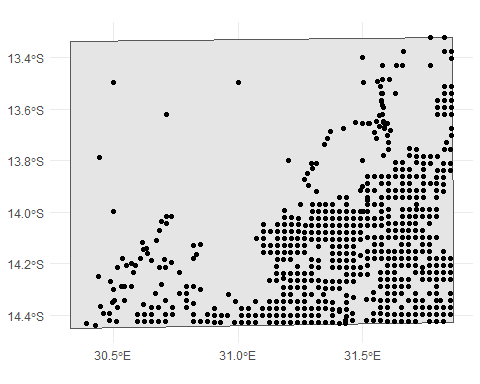
Legacy soil data is loaded if the file exists (i.e., if the country has it.). If the data is in a different CRS, it is transformed to match the CRS used for the country boundaries.

# Load legacy data (if it exists)  
if(file.exists(legacy)){  
 legacy <- sf::st\_read(legacy, quiet=TRUE)  
 # Transform coordinates to the common projection system  
 if(sf::st\_crs(legacy)$epsg!=epsg){  
 legacy <- sf::st\_transform(legacy, crs=epsg)  
 }  
   
 } else {  
 # If legacy data does not exist delete the object "legacy"  
 rm(legacy)  
 }

💡**Note**: Always verify that all geospatial datasets use the same CRS before performing any spatial operations. Misaligned CRSs can lead to incorrect analyses.

Finally, the ggplot2 package is used to visualize the country boundaries and overlay the legacy soil data. This ensures the data was loaded and transformed correctly.

ggplot(data = country\_boundaries) +   
 geom\_sf() +   
 geom\_sf(data = legacy, aes(geometry = geometry)) +  
 theme\_minimal()



### Environmental covariates - Optional

The optional environmental covariates include information on non-protected areas, high slope mask, geology, and geomorphology. These inputs are optional and do not affect the selection of sampling units.

Each explanatory dataset is specified as a file path. This approach ensures clear organization of data sources. The variables geo.classes and geomorph.classes are defined to identify fields within the geology and geomorphology datasets that store their respective types/classes.

# Non-protected areas (raster)  
npa <- file.path(paste0(shp.path,"zmb\_national\_parks\_zari\_clipped\_epsg\_3857.shp"))  
# High slopes (binary raster with 1=<50% and NA = >=50%) - QGIS Calculator   
slope <- file.path(paste0(raster.path,"zmb\_clipped\_slope\_mask\_epsg\_3857.tif"))   
# Geology  
geo <- file.path(paste0(shp.path,"zmb\_geology\_clipped\_epsg\_3857.shp"))  
# Field for geologic classes in the shape  
geo.classes <- "GEO"  
# Geomorphology  
geomorph <- file.path(paste0(raster.path,"zmb\_clipped\_Geomorphon.tif"))  
# Field for geomorphology classes in the shape  
geomorph.classes <- "CODR"

After define the location of those files, it is time to load them.

Non-protected areas (NPAs) can be represented as either shapefiles or raster files. The provided code processes shapefiles directly. If the dataset is in raster format, some lines must be commented or uncommented accordingly to handle raster operations. The primary goal is to calculate the difference between the country boundary and the protected areas, isolating non-protected areas for use in subsequent steps.

# Non-protected areas (if it exists)  
if(file.exists(npa)){  
 npa <- sf::st\_read(npa, quiet = FALSE) # Import Protected area  
 npa <- sf::st\_union(npa)  
 npa <- sf::st\_difference(country\_boundaries, npa)  
 #npa <- rast(npa)   
 if(sf::st\_crs(npa)$epsg !=epsg){  
 npa <- sf::st\_transform(npa, crs = epsg)  
 #npa <- project(npa, epsg, method="near")  
 }  
   
 } else {  
 # If non-protected areas data does not exist delete the object "npa"  
 rm(npa)  
 }

The slope mask represents areas with a slope less than 50%. This dataset is typically generated using GIS software such as QGIS. After loading, the mask values are standardized to 0 or 1. This step ensures that only areas meeting the slope criteria are included in subsequent analyses. If slope data is missing, the variable is removed.

# Slopes < 50% raster (if it exists)  
if(file.exists(slope)){  
 slope <- rast(slope)  
 if(crs(slope)!=epsg){  
 slope <- project(slope, epsg, method="near")  
 }  
 slope <- slope/slope  
 } else {  
 # If slope data does not exist delete the object "slope"  
 rm(slope)  
 }

The geology dataset is a shapefile containing geological classes. If the dataset exists, it is loaded, and its CRS is transformed for consistency. The geological classes are then converted to numeric values for compatibility with subsequent processing steps.

# Geology (if it exists)  
if(file.exists(geo)){  
 geo <- sf::st\_read(geo, quiet=TRUE)  
 if(sf::st\_crs(geo)$epsg !=epsg){  
 geo <- sf::st\_transform(geo, crs=epsg)  
 }  
 geo$GEO <- as.numeric(as.factor(geo[[geo.classes]]))  
 } else {  
 # If legacy data does not exist delete the object "geo"  
 rm(geo)  
 }

Geomorphology data can be provided as either a raster or shapefile. The code handles both formats by checking the file extension. If the data is a raster, it is loaded directly. If it is a shapefile, the CRS is transformed, and the geomorphology classes are converted to numeric values. If the file is neither a raster nor a shapefile, a warning is issued.

# Geomorphology (if it exists)  
if (file.exists(geomorph)) {  
 # Check the file extension to determine if it is a raster (.tif) or shapefile (.shp)  
 file\_extension <- tools::file\_ext(geomorph)  
 if (file\_extension == "tif") {  
 # If it's a TIFF file, no processing is needed  
 geomorph <- rast(geomorph)  
 names(geomorph) <- 'GEOMORPH'  
 # No further action required for TIFF files  
 if(crs(geomorph)!=epsg){  
 geomorph <- project(geomorph, epsg, method="near")  
 }  
 } else if (file\_extension == "shp") {  
 # If shapefile, execute the shapefile processing code  
 geomorph <- sf::st\_read(geomorph, quiet = TRUE)  
 # Check and transform CRS if necessary  
 if (sf::st\_crs(geomorph)$epsg != epsg) {  
 geomorph <- sf::st\_transform(geomorph, crs = epsg)  
 }  
 geomorph$GEOMORPH <- as.numeric(as.factor(geomorph[[geomorph.classes]]))  
 } else {  
 # If the file is neither a raster nor a shapefile, handle the error or delete the object  
 warning("File format not recognized. Expected .tif or .shp.")  
 rm(geomorph)  
 }  
 } else {  
 # If the file does not exist, delete the object "geomorph"  
 rm(geomorph)  
 }

### Mandatory environmental covariates

A list of mandatory environmental covariates was created to serve as parameters for selecting sampling units, providing a baseline for countries lacking high-resolution environmental data. If available, countries may use their own datasets as substitutes.

The following steps outline how covariates are loaded, processed, and prepared for analysis. Let’s take a look into them!

The list.files() function is used to locate raster files containing covariate data, and these files are loaded into a SpatRaster object using terra::rast(). We have displayed in a table the covariate names for easy reference and interpretation.

# Load covariate data  
cov.dat <- list.files(raster.path, pattern = "covs\_zam\_clipped.tif$", recursive = TRUE, full.names = TRUE)  
cov.dat <- terra::rast(cov.dat) # SpatRaster from terra  
  
# Convert the variable names into a matrix with 3 columns  
cov.dat.names <- matrix(names(cov.dat), ncol = 3, byrow = TRUE)  
  
# Create a markdown table  
knitr::kable(cov.dat.names, format = "markdown",   
 caption = "Environmental covariates")

Environmental covariates

|  |  |  |
| --- | --- | --- |
| bio1 | bio12 | bio13 |
| bio14 | bio16 | bio17 |
| bio5 | bio6 | ngd10 |
| pet\_penman\_max | pet\_penman\_mean | pet\_penman\_min |
| pet\_penman\_range | sfcWind\_max | sfcWind\_mean |
| sfcWind\_range | fpar\_030405\_500m\_mean | fpar\_030405\_500m\_sd |
| fpar\_060708\_500m\_mean | fpar\_060708\_500m\_sd | fpar\_091011\_500m\_mean |
| fpar\_091011\_500m\_sd | fpar\_120102\_500m\_mean | fpar\_120102\_500m\_sd |
| lstd\_030405\_mean | lstd\_030405\_sd | lstd\_060708\_mean |
| lstd\_060708\_sd | lstd\_091011\_mean | lstd\_091011\_sd |
| lstd\_120102\_mean | lstd\_120102\_sd | ndlst\_030405\_mean |
| ndlst\_030405\_sd | ndlst\_060708\_mean | ndlst\_060708\_sd |
| ndlst\_091011\_mean | ndlst\_091011\_sd | ndlst\_120102\_mean |
| ndlst\_120102\_sd | ndvi\_030405\_250m\_mean | ndvi\_030405\_250m\_sd |
| ndvi\_060708\_250m\_mean | ndvi\_060708\_250m\_sd | ndvi\_091011\_250m\_mean |
| ndvi\_091011\_250m\_sd | ndvi\_120102\_250m\_mean | ndvi\_120102\_250m\_sd |
| snow\_cover | swir\_060708\_500m\_mean | crops |
| flooded\_vegetation | grass | shrub\_and\_scrub |
| trees | dtm\_curvature\_250m | dtm\_downslopecurvature\_250m |
| dtm\_dvm2\_250m | dtm\_dvm\_250m | dtm\_elevation\_250m |
| dtm\_mrn\_250m | dtm\_neg\_openness\_250m | dtm\_pos\_openness\_250m |
| dtm\_slope\_250m | dtm\_tpi\_250m | dtm\_twi\_500m |
| dtm\_upslopecurvature\_250m | dtm\_vbf\_250m | bio1 |

💡**Note**: There is an initial number of 68 environmental covariates.

The newhall dataset contains soil climate variables like temperature and moisture regimes. Subdivisions that are not required (regimeSubdivision1 and regimeSubdivision2) are removed to optimize processing.

# Load soil climate data  
newhall <- list.files(raster.path, pattern = "newhall\_zam\_clipped.tif$", recursive = TRUE, full.names = TRUE)  
newhall <- terra::rast(newhall) # SpatRaster from terra  
  
# Convert the variable names into a matrix with 3 columns  
newhall.names <- matrix(names(newhall), ncol = 3, byrow = TRUE)  
  
# Create a markdown table  
knitr::kable(newhall.names, format = "markdown",   
 caption = "Environmental covariates")  
  
# Delete regimeSubdivisions (don't needed)  
newhall$regimeSubdivision1 <- c()  
newhall$regimeSubdivision2 <- c()

Environmental covariates

|  |  |  |
| --- | --- | --- |
| annualRainfall | waterHoldingCapacity | annualWaterBalance |
| annualPotentialEvapotranspiration | summerWaterBalance | dryDaysAfterSummerSolstice |
| moistDaysAfterWinterSolstice | numCumulativeDaysDry | numCumulativeDaysMoistDry |
| numCumulativeDaysMoist | numCumulativeDaysDryOver5C | numCumulativeDaysMoistDryOver5C |
| numCumulativeDaysMoistOver5C | numConsecutiveDaysMoistInSomeParts | numConsecutiveDaysMoistInSomePartsOver8C |
| temperatureRegime | moistureRegime | regimeSubdivision1 |
| regimeSubdivision2 | annualRainfall | waterHoldingCapacity |

💡**Note**: The final number of soil climate variables is 17.

For subsequent statistical analysis, categorical covariates must be converted into binary layers to ensure compatibility with spatial clustering. Following this, the original covariates, temperatureRegime and moistureRegime were removed from newhall.

# Convert temperatureRegime and moistureRegime to dummy variables (1 = presence; 0 = absence)  
temperatureRegime <- dummies(ca.rast = newhall$temperatureRegime, preval = 1, absval = 0)  
moistureRegime <- dummies(ca.rast = newhall$moistureRegime, preval = 1, absval = 0)  
  
# Delete temperatureRegime and moistureRegime rasters from soil climate data  
newhall$temperatureRegime <- c()  
newhall$moistureRegime <- c()

The processed soil climate data is appended to the main covariate stack (cov.dat). If the dataset uses a different coordinate reference system (CRS), it is reprojected to match epsg.

# Merge covs and climate data  
cov.dat <- c(cov.dat, newhall,temperatureRegime,moistureRegime)  
   
# Project covariates if necessary  
if(crs(cov.dat)!=epsg){  
 cov.dat <- terra::project(cov.dat, epsg, method="near")  
}

The loaded geology data is converted into raster layers and dummy variables to ensure it can be included in spatial clustering and analysis.

# Transform Geology if exists  
if(exists("geo")){  
 geo <- rasterize(as(geo,"SpatVector"), cov.dat,field="GEO")  
 geo <- dummies(ca.rast = geo$GEO, preval = 1, absval = 0)  
 } else {  
 # If Geology data does not exist delete the object "geo"  
 rm(geo)  
 }  
  
# Merge covs and geology if exists)  
if(exists("geo")){  
 cov.dat <- c(cov.dat, geo)  
}

The same process is applied to geomorphology data, ensuring compatibility with the other covariates.

# Transform geomorphology if exists and   
# Check if the 'geomorph' object exists  
if (exists("geomorph")) {  
 # Check if 'geomorph' is already a raster  
 if (!inherits(geomorph, "SpatRaster")) {  
 # If 'geomorph' is not a raster, convert it to a raster  
 geomorph <- rasterize(as(geomorph, "SpatVector"), cov.dat, field = "GEOMORPH")  
 }  
 # Apply the dummies function, assuming geomorph is now a raster  
 geomorph <- dummies(ca.rast = geomorph$GEOMORPH, preval = 1, absval = 0)  
}  
  
# Merge covs and geomorph (if exists)  
if (exists("geomorph")) {  
 # Ensure both rasters have the same resolution and extent  
 if (!identical(ext(cov.dat), ext(geomorph))) {  
 geomorph <- extend(geomorph, cov.dat)  
 }  
 # Optionally, if they have different resolutions, resample geomorph to match cov.dat  
 if (!all(res(cov.dat) == res(geomorph))) {  
 geomorph <- resample(geomorph, cov.dat, method = "near")  
 }  
 # Merge cov.dat and geomorph  
 cov.dat <- c(cov.dat, geomorph)  
}

After stacking, intermediate datasets like newhall, geomorph, and geo are removed using the rm() function. This is done to free up memory and avoid unnecessary resource usage. By stacking all the relevant layers into cov.dat and cleaning up temporary datasets, the workflow remains efficient, organized, and ready for subsequent analyses like Principal Component Analysis (PCA) or sampling unit selection.

# Remove newhall SpatRaster and clean memory  
rm(newhall)  
rm(geomorph)  
rm(geo)  
gc()

## used (Mb) gc trigger (Mb) max used (Mb)  
## Ncells 3393292 181.3 6783468 362.3 4282678 228.8  
## Vcells 4779481 36.5 10146329 77.5 6445985 49.2

💡**Note**: Stacking is the process of combining multiple spatial data layers (raster datasets) into a single multi-layer object. In the context of geospatial analysis, this is especially useful when you have multiple variables (e.g., environmental covariates, soil properties, or climate data) that need to be analyzed together as a single dataset.

Covariates are cropped to align with the administrative boundary of the target region. The processed dataset is saved as a raster stack for later use.

# Crop covariates on administrative boundary  
cov.dat <- crop(cov.dat, country\_boundaries, mask=TRUE, overwrite=TRUE)  
writeRaster(cov.dat, paste0(raster.path,"cov\_dat\_stack.tif"),overwrite=TRUE)  
  
# Load cov.dat again, in case of issues  
#cov.dat <- rast(paste0(raster.path,"cov\_dat\_stack.tif"))

PCA is used to reduce the dimensionality of the dataset, retaining only the most important components while removing redundancy. Components explaining 99% of the variance are kept.

# Simplify raster information with PCA   
pca <- synoptReg::raster\_pca(cov.dat) # Faster than with terra::princomp  
  
# Get SpatRaster layers from the pca object  
cov.dat <- pca$PCA  
summary(cov.dat)  
  
# Subset rasters to main PC (var. explained >= 0.99)  
n\_comps <- first(which(pca$summaryPCA[3,] > 0.99))  
cov.dat <- pca$PCA[[1:n\_comps]]

The processed PCA is exported for future analysis or backup in case of issues.

# Save PCA rasters to results folder  
writeRaster(cov.dat,paste0(results.path,"PCA\_projected.tif"),overwrite=TRUE)  
  
# Remove pca stack  
rm(pca)  
  
# Load pretreated PCA rasters  
#cov.dat <- rast(paste0(results.path,"PCA\_projected.tif"))

### Organizing the Sampling Universe

After incorporating all environmental covariates and performing a Principal Component Analysis (PCA), additional steps are taken to refine the sampling universe by removing protected areas, filtering areas with slopes up to 50%, resampling land use data, and filtering legacy data.

This step defines the file path for the **land-use** raster, ensuring that only areas corresponding to the target land-use (e.g., croplands) are considered in the analysis. The land-use raster acts as a key layer for filtering and refining the sampling universe.

landuse <- file.path(paste0(raster.path,"cropland\_clipped\_zmb\_v1\_epsg\_3857.tif"))

The land-use is a binary raster where *1* represents land-use of interest and *NA* represents all others. The operation crops/crops divides each cell value by itself meaning for cell where the value is not NA (e.g., land-use cells with valid data), the division results in *1*.

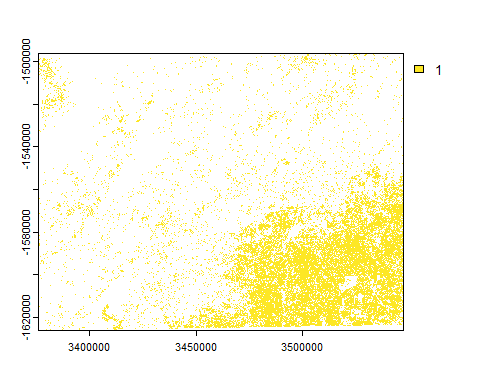
In this example, the current resolution is approximately 20 x 21 m (x, y).

crops <- rast(landuse)  
crops <- crops/crops

##   
|---------|---------|---------|---------|  
=========================================

names(crops) <- "lu"  
crops

## class : SpatRaster   
## dimensions : 6126, 8296, 1 (nrow, ncol, nlyr)  
## resolution : 20.61473, 21.17437 (x, y)  
## extent : 3375972, 3546991, -1626000, -1496286 (xmin, xmax, ymin, ymax)  
## coord. ref. : WGS 84 / Pseudo-Mercator (EPSG:3857)   
## source : spat\_4c3c36e0527b\_19516.tif   
## varname : cropland\_clipped\_zmb\_v1\_epsg\_3857   
## name : lu   
## min value : 1   
## max value : 1



Protected areas are excluded from the analysis to ensure that sampling units are not located within legally protected zones. If the npa variable exists, the mask() function is applied to exclude these regions from the crops raster. Once processed, npa is removed to free memory.

If npa is a raster instead of a shapefile, uncomment the appropriate lines in the code and comment out the ones for shapefiles.

if(exists("npa")){  
 # If npa is a shapefile  
 crops <- mask(crops, npa)  
 # If npa is a raster. Resample to match crops  
 # npa <- resample(npa,crops, method="near")  
 # crops <- crops \* npa  
}

##   
|---------|---------|---------|---------|  
=========================================

rm(npa)

💡**Note**: Use the correct operation for your npa dataset type (shapefile or raster) to ensure compatibility and accurate processing.

Restricting the analysis to areas with slopes of 50% or less ensures that sampling focuses on accessible and agriculturally viable regions. Steep slopes are excluded as they are often unsuitable for farming or inaccessible.

The slope raster is resampled to align with the resolution of the crops raster before being used to filter out areas with slopes above 50%. The filtered raster ensures the focus remains on usable land.

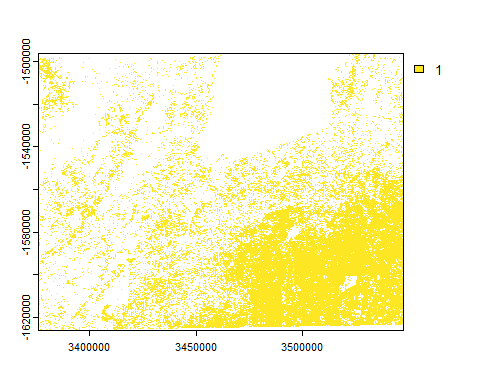
if(exists("slope")){  
 # Resample to match crops  
 slope <- resample(slope, crops, method="near")  
 crops <- crops \* slope  
}  
  
rm(slope)

Land-use is resampled to coarser 100 m resolution, significantly optimizing processing speed and reducing storage requirements. This is achieved by aggregating the 20 m resolution raster data to a 1-hectare resolution (100 m x 100 m). Users can take advantage of multi-core processing to speed up this step.

The aggregate() function increases pixel size, where a 5x5 grid of 20 m pixels is combined into one 100 m pixel using the modal value. If we had a 25 m resolution raster, the number would be 4 instead of 5. The cores argument specifies the number of cores to parallelize the process for faster computation.

lu <- aggregate(crops, 5, fun=modal, cores = 4, na.rm=T)  
lu <- lu/lu  
lu

## class : SpatRaster   
## dimensions : 1226, 1660, 1 (nrow, ncol, nlyr)  
## resolution : 103.0736, 105.8718 (x, y)  
## extent : 3375972, 3547074, -1626085, -1496286 (xmin, xmax, ymin, ymax)  
## coord. ref. : WGS 84 / Pseudo-Mercator (EPSG:3857)   
## source(s) : memory  
## name : lu   
## min value : 1   
## max value : 1



Processed land-use data is saved to ensure data persistence and allow for reuse in future analyses or in case of system interruptions. This is an essential step for safeguarding progress and avoiding re-processing.

The writeRaster() function saves the processed land-use raster to disk, and the rast() function reloads it when needed.

writeRaster(lu,paste0(raster.path,"cropland\_zmb\_100m\_v1\_epsg\_3857.tif"),overwrite=TRUE)  
  
lu <- rast(paste0(raster.path,"cropland\_zmb\_100m\_v1\_epsg\_3857.tif"))

Legacy data points are filtered to retain only those that fall within the designated land-use areas. This step ensures that sampling is consistent with the land-use data and improves the relevance of legacy data for the current analysis.

The terra::extract() function checks whether each legacy point falls within the land-use raster, marking valid points with a corresponding value. Points outside the designated areas are removed.

if (exists("legacy")){  
 legacy$INSIDE <- terra::extract(crops, legacy) %>% dplyr::select(lu)  
 legacy <- legacy[!is.na(legacy$INSIDE),] %>%  
 dplyr::select(-"INSIDE")  
}

### Creating Primary Sampling Units

In this section, we generate a grid of Primary Sampling Units (PSUs). PSUs are 2x2 km grid cells that serve as the foundational sampling units.

The st\_make\_grid() function is used to create a grid of 2x2 km square polygons within the boundaries of the study region. The size of the grid cells is determined by the psu\_size variable that we defined right in the beginning of this code, and each grid cell is assigned a unique identifier (ID).

psu\_grid <- st\_make\_grid(country\_boundaries, cellsize = c(psu\_size, psu\_size), square = TRUE)  
psu\_grid <- st\_sf(geometry = psu\_grid)  
psu\_grid$ID <- 1:nrow(psu\_grid)

The next step is to trim the grid to the administrative boundaries of the study region. This ensures that only PSUs within the defined country boundaries are retained. This operation is computationally intensive, especially for large datasets.

psu\_grid <- psu\_grid[country\_boundaries[1],]

The processed PSU grid is saved to a shapefile using the write\_sf() function. Saving the grid ensures that the trimmed version is available for future use without needing to repeat the computationally expensive trimming process. The st\_read() function reloads the saved shapefile as needed.

write\_sf(psu\_grid,paste0(results.path,"../grid2k.shp"),overwrite=TRUE)  
  
psu\_grid <- sf::st\_read(file.path(paste0(results.path,"../grid2k.shp")))

### Select PSUs with land-use above a certain percentage

This section focuses on identifying *Primary Sampling Units (PSUs)* that meet a minimum threshold for crop coverage. It ensures that sampling efforts are concentrated in areas with sufficient agricultural activity, improving the relevance of the collected data.

the land-use values (lu) corresponding to the cells intersecting with the psu\_grid polygons are extracted. These values represent crop coverage data for each PSU grid cell.

extracted\_values <- terra::extract(lu, psu\_grid)

The extracted land-use data is grouped by grid ID (ID), and the percentage of crop coverage is calculated based on the 1-ha reclassification of land-use (400 pixels represent a 2x2 km PSU).

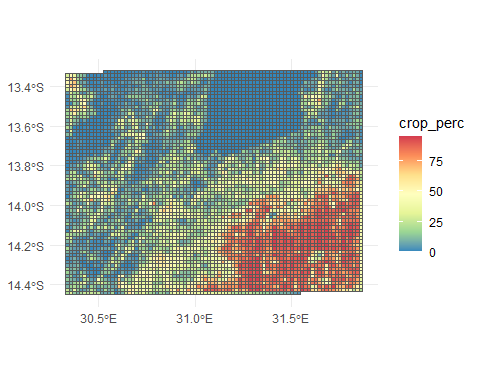
Temporary object extracted\_values are removed to save memory.

crop\_perc <- extracted\_values %>%  
 group\_by(ID) %>%  
 summarize(crop\_perc = sum(lu, na.rm = TRUE)\*100/400)  
  
rm(extracted\_values)

💡**Note**: The formula sum(lu, na.rm = TRUE) \* 100 / 400 ensures that the crop percentage is calculated accurately, with 400 being the total pixels in a 2x2 km PSU.

The calculated crop coverage percentages (crop\_perc) are added to the psu\_grid polygons, creating a new attribute for each grid. This step prepares the data for further filtering and visualization.

psu\_grid$crop\_perc <- crop\_perc$crop\_perc



Like before, the updated psu\_grid with crop coverage percentages is saved to a shapefile for backup and potential reuse in future analyses. If needed, the saved file can be reloaded.

write\_sf(psu\_grid, file.path(paste0(results.path,"/psu\_grid\_counts.shp")), overwrite=TRUE)  
  
psu\_grid <- sf::st\_read(file.path(paste0(results.path,"/psu\_grid\_counts.shp")))

Only PSUs meeting the predefined crop coverage threshold (percent\_crop) are retained. The percent\_crop variable is defined earlier in the script, allowing flexibility in setting the threshold. This filtering step narrows the focus to areas that are agriculturally relevant.

psu\_grid <- psu\_grid[psu\_grid$crop\_perc > percent\_crop,"ID"]

The selected PSUs are rasterized, creating a template where each grid cell represents a specific PSU ID. This raster format is essential for integrating PSUs with covariate data and subsequent spatial analyses.

template <- rast(vect(psu\_grid), res = psu\_size)  
template <- rasterize(vect(psu\_grid), template, field = "ID")

### Rasterising PSUs for Covariate Space Coverage

The covariate stack (cov.dat) is cropped to the sampling universe defined by the PSU grid. The cropped covariates are then resampled to match the rasterized PSU template, ensuring consistent spatial alignment.

cov.dat <- crop(cov.dat, psu\_grid, mask=TRUE, overwrite=TRUE)  
  
PSU.r <- resample(cov.dat, template)

💡**Why this matters**: Cropping and resampling ensure that the spatial resolution and extent of covariates match the PSU grid, facilitating accurate spatial analysis and clustering.

### GAME ON - Covariate Space Coverage - Computing PSUs

Covariate Space Coverage (CSC) sampling is a clustering technique designed to ensure that sampling units are distributed effectively across a multidimensional space of environmental covariates. Covariates are standardized (centered and scaled), so their mean is 0 and standard deviation is 1. This prevents variables with larger ranges (e.g., elevation vs. precipitation) from dominating the clustering process. The algorithm groups raster cells in covariate space into clusters by minimizing the Mean Squared Shortest Scaled Distance (MSSSD). MSSSD is a measure of how well the points in the dataset fit into their assigned clusters. The term “scaled distance” refers to distances measured in the space of standardized covariates, not physically scaled distances.

*Preparing function parameters*

Before running the CSC function, the raster stack information at the PSU aggregated level is converted into a dataframe with coordinates. This transformation allows for easier manipulation and scaling of covariates.

PSU.df <- as.data.frame(PSU.r,xy=T)  
  
covs <- names(cov.dat)

*Scaling covariates*

Environmental covariates are scaled to standardize their values, ensuring all variables contribute equally during clustering. The dataframe mygrd represents the scaled covariates at a resolution of 2 km x 2 km (PSUs).

mygrd <- data.frame(scale(PSU.df[, covs]))

*Performing k-means clustering for CSC*

If legacy data is available, it is utilized to define fixed cluster centers and enhance the sampling design. If not, a traditional k-means clustering approach is used. Thus, legacy data (legacy) is filtered to align with the extent of the PSU grid using the st\_filter function. The st\_coordinates function extracts spatial coordinates from the legacy dataset, creating a data frame (legacy\_df) containing the x and y positions of legacy points.

A loop iterates through each legacy point. Using the Euclidean distance formula, the script calculates the distance between each legacy point and all PSUs in the PSU.df data frame, which contains covariate information. The nearest PSU to each legacy point is identified using which.min, and its index is stored in the units vector. The covariate data corresponding to these fixed points is extracted and scaled (standardized). This step ensures the covariates have equal influence during clustering. The result is stored in the fixed object.

If legacy data exists, the *Covariate Space Coverage Sampling (CSIS)* function is used to integrate these fixed legacy points into the clustering process. This ensures the legacy points remain as fixed centers while optimizing the distribution of new points.

If no legacy data is available, standard kmeans clustering is applied to the scaled covariates (mygrd), focusing solely on environmental variability to determine the cluster centers.

if (exists("legacy")){  
 legacy <- st\_filter(legacy,psu\_grid)  
 legacy\_df <- st\_coordinates(legacy)  
 units <- numeric(nrow(legacy\_df))  
   
 for (i in 1:nrow(legacy\_df)) {  
 distances <- sqrt((PSU.df$x - legacy\_df[i, "X"])^2 + (PSU.df$y - legacy\_df[i, "Y"])^2)  
 units[i] <- which.min(distances)  
 }  
   
 fixed <- unique(data.frame(units, scale(PSU.df[, covs])[units, ]))   
 res <- CSIS(fixed = fixed, nsup = n.psu, nstarts = iterations, mygrd = mygrd)  
 } else {  
 res <- kmeans(mygrd, centers = n.psu, iter.max = 10000, nstart = 100)  
 }

## [1] "1 out of 10"  
## [1] "2 out of 10"  
## [1] "3 out of 10"  
## [1] "4 out of 10"  
## [1] "5 out of 10"  
## [1] "6 out of 10"  
## [1] "7 out of 10"  
## [1] "8 out of 10"  
## [1] "9 out of 10"  
## [1] "10 out of 10"

💡**Note**: Users can follow the progress by observing the printed output. For example, “1 out of 10” indicates that 1 cluster interaction has been done, with 9 more left to process.

This step associates each PSU with its respective cluster, as determined by the clustering algorithm. The cluster assignments from the CSIS function are stored in the PSU dataframe, allowing us to identify the cluster for each grid cell.

PSU.df$cluster <- res$cluster

This step computes the distances from each point in the covariate space to its nearest cluster center. rdist calculates the Euclidean distances between the cluster centers and the scaled covariate data. The apply function identifies the closest cluster center for each PSU, ensuring accurate assignment to clusters.

D <- rdist(x1 = res$centers, x2 = scale(PSU.df[, covs]))  
units <- apply(D, MARGIN = 1, FUN = which.min)

The MSSSD measures how well the selected clusters represent the overall covariate space. dmin calculates the shortest distance from each point to a cluster center. MSSSD averages the squared shortest distances, providing a measure of clustering efficiency.

dmin <- apply(D, MARGIN = 2, min)  
MSSSD <- mean(dmin^2)

Extracts the PSUs that were selected during the clustering process, including their spatial coordinates and covariates. The myCSCsample dataframe holds only the selected PSUs, along with their spatial coordinates (x, y) and associated covariates.

myCSCsample <- PSU.df[units, c("x", "y", covs)]

Legacy PSUs are sampling points that already exist in the dataset, while new PSUs are points added during the clustering process. This chunk identifies and categorizes the selected PSUs into *“legacy”* and *“new”* types. If legacy data exists, it assigns the label legacy to pre-existing sampling points and new to additional points. If no legacy data exists, all selected PSUs are labeled as new.

if (exists("legacy")){  
 myCSCsample$type <- c(rep("legacy", nrow(fixed)), rep("new", length(units)-nrow(fixed)))  
 } else {  
 myCSCsample$type <- "new"  
 }

The selected PSUs (myCSCsample) are converted into a spatial object (sf class) with specified coordinates (x, y) and the defined coordinate reference system (epsg). This transformation allows for geospatial operations and visualization.

myCSCsample <- myCSCsample %>%  
 st\_as\_sf(coords = c("x", "y"), crs = epsg)

Distinguishes between legacy PSUs (existing sampling points) and new PSUs (infill points). Legacy sampling points (type == "legacy") and new points (type == "new") are separated for further analysis and visualization.

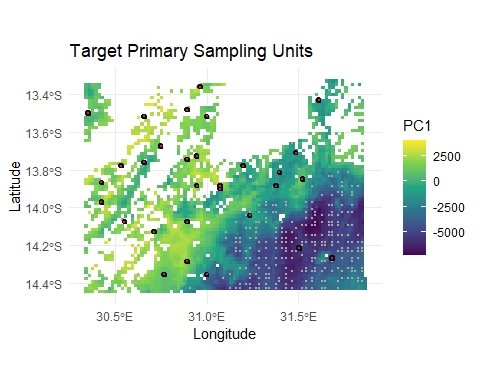
if (exists("legacy")){  
 legacy <- myCSCsample[myCSCsample$type=="legacy",]  
}  
  
new <- myCSCsample[myCSCsample$type=="new",]

Identifies PSUs within the infill data and extracts their unique IDs. The intersection of the PSU grid and infill points ensures only relevant PSUs are selected. The resulting dataframe contains the unique IDs (ID) of these PSUs.

PSUs <- sf::st\_intersection(psu\_grid, new) %>% dplyr::select(ID)

The target.PSUs dataframe holds only the PSUs identified as targets for sampling. It extracts the PSUs targeted for sampling based on the intersection results.

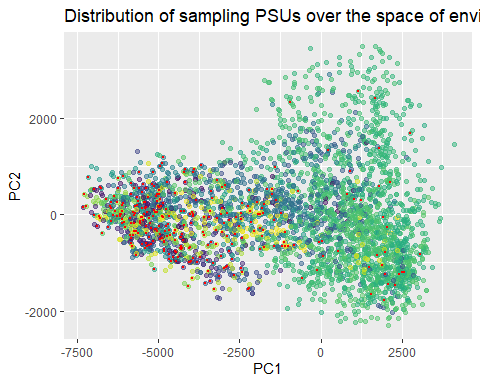
target.PSUs <- psu\_grid[psu\_grid$ID %in% PSUs$ID,] %>% dplyr::select(ID)



Temporary objects (new, dmin, MSSSD) are deleted to free up memory and prevent conflicts.

rm(new,dmin,MSSSD)

Here, you can visualize how the selected PSUs are distributed across the first two principal components (PC1 and PC2) of the environmental covariates.



### Computing SSUs and TSUs

In the previous section, we computed PSUs. Now, the focus shifts to SSUs and TSUs, which add hierarchical levels to the sampling process. SSUs are grid cells within each PSU filtered based on a land-use percentage threshold and randomly selected, and TSUs are randomly points within each SSU providing finer sampling points.

Empty lists are created to store the computed SSUs (selected\_ssus) and TSUs (all\_psus\_tsus). Each PSU will have its own set of SSUs and TSUs added to these lists.

selected\_ssus <- list()  
  
all\_psus\_tsus <- list()

This loop iterates over all selected PSUs generating SSUs and TSUs for each PSU. First of all, a grid of SSUs is created within the PSU boundary (st\_make\_grid), and only SSUs meeting the land-use coverage threshold (percent\_crop) are retained. After that, a specified number of target and alternative SSUs are randomly selected (num\_primary\_ssus and num\_alternative\_ssus). For each SSU, generate TSUs as random points within the SSU boundary being labeled as target or alternative.

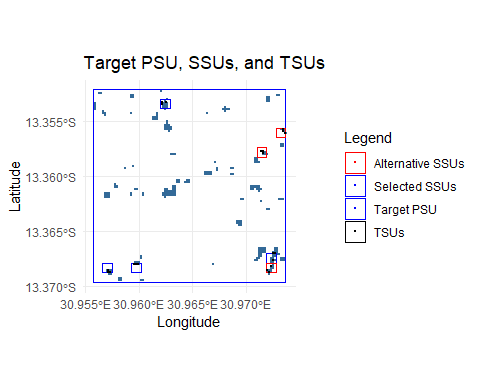
for (psu\_id in 1:nrow(target.PSUs)) {  
   
 # Selects one PSU at a time   
 selected\_psu <- target.PSUs[psu\_id, ]  
   
 # Generate SSUs within the selected PSU  
 ssu\_grid <- st\_make\_grid(selected\_psu, cellsize = c(ssu\_size, ssu\_size), square = TRUE)  
 ssu\_grid\_sf <- st\_sf(geometry = ssu\_grid)  
   
 # Convert ssu\_grid\_sf to SpatVector  
 ssu\_grid\_vect <- vect(ssu\_grid\_sf)  
   
 # Extract values of lu for cells that intersect with ssu\_grid\_vect  
 extracted\_values <- extract(crops, ssu\_grid\_vect, fun=table)  
   
 # Add lu code to the SSUs   
 ssu\_grid\_sf$lu <- (extracted\_values[,2]\*100)/25  
 ssu\_grid\_sf <- ssu\_grid\_sf[ssu\_grid\_sf$lu > percent\_crop, ]  
   
 print(paste(psu\_id, " out of ", nrow(target.PSUs)))  
   
 # Counts the total number of filtered SSUs  
 total\_ssus <- nrow(ssu\_grid\_sf)  
   
 # If the PSU contains enough SSUs, a random sample of SSUs is selected  
 if(total\_ssus >= (num\_primary\_ssus + num\_alternative\_ssus)) {  
 primary\_ssus\_indices <- sample(1:total\_ssus, num\_primary\_ssus, replace = FALSE)  
 available\_for\_alternatives <- setdiff(1:total\_ssus, primary\_ssus\_indices)  
 alternative\_ssus\_indices <- sample(available\_for\_alternatives, num\_alternative\_ssus, replace = FALSE)  
 selected\_ssus[[psu\_id]] <- rbind(ssu\_grid\_sf[primary\_ssus\_indices, ], ssu\_grid\_sf[alternative\_ssus\_indices, ])  
   
 primary\_tsus <- lapply(primary\_ssus\_indices,   
 function(index) {  
 generate\_tsu\_points\_within\_ssu(ssu\_grid\_sf[index, ], number\_TSUs, index, "Target",crops)  
 }  
 )  
 # Generate TSUs for alternative SSUs with naming  
 alternative\_tsus <- lapply(alternative\_ssus\_indices,   
 function(index) {  
 generate\_tsu\_points\_within\_ssu(ssu\_grid\_sf[index, ], number\_TSUs, index, "Alternative",crops)  
 }  
 )  
 # Combine all TSUs of the current PSU into one sf object  
 all\_psus\_tsus[[psu\_id]] <- do.call(rbind, c(primary\_tsus, alternative\_tsus))  
   
 # If there are not enough SSUs, a warning is issued, and the PSU is skipped.  
 } else {  
 warning(paste("PSU", psu\_id, "does not have enough SSUs for selection. Skipping."))  
 }  
}

## [1] "1 out of 30"  
## [1] "2 out of 30"  
## [1] "3 out of 30"  
## [1] "4 out of 30"  
## [1] "5 out of 30"  
## [1] "6 out of 30"  
## [1] "7 out of 30"  
## [1] "8 out of 30"  
## [1] "9 out of 30"  
## [1] "10 out of 30"  
## [1] "11 out of 30"  
## [1] "12 out of 30"  
## [1] "13 out of 30"  
## [1] "14 out of 30"  
## [1] "15 out of 30"  
## [1] "16 out of 30"  
## [1] "17 out of 30"  
## [1] "18 out of 30"  
## [1] "19 out of 30"  
## [1] "20 out of 30"  
## [1] "21 out of 30"  
## [1] "22 out of 30"  
## [1] "23 out of 30"  
## [1] "24 out of 30"  
## [1] "25 out of 30"  
## [1] "26 out of 30"  
## [1] "27 out of 30"  
## [1] "28 out of 30"  
## [1] "29 out of 30"  
## [1] "30 out of 30"

💡**Note**: Users can follow the progress by observing the printed output. For example, "1 out of 30" indicates that processing for the first PSU is complete, and there are 29 more PSUs left.

After all TSUs have been generated, they are combined into a single dataset (all\_tsus). The dataset includes metadata about each TSU’s type (Target or Alternative) and its hierarchical assignment to SSUs and PSUs.

all\_tsus <- do.call(rbind, all\_psus\_tsus)  
all\_tsus$TSU\_Type <- "Target"  
all\_tsus[all\_tsus$TSU\_ID >1,"TSU\_Type"] <- "Alternative"  
all\_tsus$PSU\_Type <- "Target"  
all\_tsus <- all\_tsus %>%  
 dplyr::select("TSU\_Name","PSU\_ID","SSU\_ID","TSU\_ID","PSU\_Type","SSU\_Type","TSU\_Type","geometry")



### Exporting the target sampling units

This section covers the process of exporting target sampling units.

We start it converting the cluster information into a raster format. Each cell corresponds to a cluster, and the coordinate reference system is assigned to ensure spatial consistency.

dfr <- PSU.df[,c("x","y","cluster")]  
dfr$cluster <- as.numeric(dfr$cluster)  
dfr <- rasterFromXYZ(dfr)  
crs(dfr) = epsg

Here, the target PSUs are associated with their corresponding cluster IDs by extracting the cluster data.

PSU\_cluster.id <- unlist(extract(dfr, target.PSUs))  
   
valid.PSU\_clusters <- target.PSUs %>% mutate(  
 cluster = extract(dfr, target.PSUs, fun = mean, na.rm = TRUE)  
 )  
  
all.PSU\_clusters <- psu\_grid %>% mutate(  
 cluster = extract(dfr, psu\_grid, fun = mean, na.rm = TRUE)  
 )  
  
all.PSU\_clusters <- na.omit(all.PSU\_clusters)

Cluster information from the PSUs is joined to the TSUs for identification and analysis.

valid.PSU\_clusters <- valid.PSU\_clusters %>%  
 rename(Replace\_ID= cluster)   
  
all\_tsus <- st\_join(all\_tsus, valid.PSU\_clusters)

Then, an order is assigned to the TSUs within each PSU. The sampling order differentiates between target TSUs (1–4) and alternative TSUs (5–7).

all\_tsus <- all\_tsus %>%  
 group\_by(PSU\_ID) %>%  
 mutate(order = match(SSU\_ID, unique(SSU\_ID))) %>%  
 ungroup()

Each TSU is assigned a unique site ID for traceability. The ID is constructed using the country code, PSU ID, SSU order, and TSU ID.

all\_tsus$site\_id = paste0(ISO.code, all\_tsus$PSU\_ID, "-", all\_tsus$order, "-", all\_tsus$TSU\_ID)

Finally, target PSUs and TSUs, along with the cluster raster, are saved as shapefiles and raster files for further use.

write\_sf(valid.PSU\_clusters,paste0(results.path,"/PSUs\_target.shp"), overwrite=TRUE)  
  
write\_sf(all\_tsus,paste0(results.path,"/TSUs\_target.shp"), overwrite=TRUE)   
  
write\_sf(all.PSU\_clusters,paste0(results.path,"/PSU\_pattern\_cl.shp"), overwrite=TRUE)  
   
writeRaster(dfr, paste0(results.path,"/clusters.tif"), overwrite=TRUE)

### Computing alternative PSUs

This step filters out PSUs already used in valid sampling units, leaving only the remaining PSUs for potential replacements.

remaining.PSU\_clusters <- all.PSU\_clusters %>%  
 filter(!(ID %in% valid.PSU\_clusters$ID))

Distinct cluster values from the valid PSUs are identified to guide replacement sampling.

unique\_cluster <- distinct(valid.PSU\_clusters, Replace\_ID)$Replace\_ID

A vector is initialized to store indices of the replacement PSUs.

sampled\_indices <- integer(0)

The loop iterates through each unique cluster, identifying and sampling potential replacements from the remaining PSUs.

for (clust in unique\_cluster) {  
 candidates\_indices <- which(remaining.PSU\_clusters$cluster == clust)  
   
 if (length(candidates\_indices) > 0) {  
 sampled\_index <- sample(candidates\_indices, size = 1)  
 sampled\_indices <- c(sampled\_indices, sampled\_index)  
 }  
}

The selected replacement indices are used to create a dataset of replacement PSUs.

replacements <- remaining.PSU\_clusters[sampled\_indices, ]

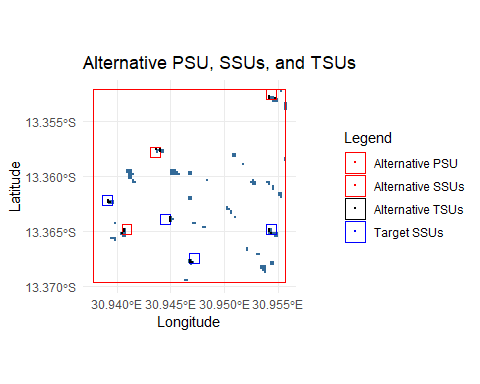
### Computing alternative SSUs and TSUs for alternative PSUs

Alternative SSUs and TSUs are generated for the alternative PSUs using similar methods as for the target PSUs. This process ensures that a backup is available in case PSUs cannot be sampled in field. Therefore, we will not explain the step-by-step again here.

# Initialize a list to store TSUs for all PSUs  
alt\_psus\_tsus\_sf <- list()  
selected\_ssus\_sf <- list()  
  
for (psu\_id in 1:nrow(replacements)) {  
 # Selects one PSU at a time   
 selected\_psu <- replacements[psu\_id, ]  
   
 # Generate SSUs within the selected PSU  
 ssu\_grid <- st\_make\_grid(selected\_psu, cellsize = c(ssu\_size, ssu\_size), square = TRUE)  
 ssu\_grid\_sf <- st\_sf(geometry = ssu\_grid)  
   
 # Convert ssu\_grid\_sf to SpatVector  
 ssu\_grid\_vect <- vect(ssu\_grid\_sf)  
   
 # Extract values of lu for cells that intersect with ssu\_grid\_vect  
 extracted\_values <- extract(crops, ssu\_grid\_vect, fun=table)  
   
 # Add lu code to the SSUs   
 ssu\_grid\_sf$lu <- (extracted\_values[,2]\*100)/25  
 ssu\_grid\_sf <- ssu\_grid\_sf[ssu\_grid\_sf$lu > percent\_crop, ]  
   
 print(paste(psu\_id, " out of ", nrow(replacements)))  
   
 # Counts the total number of filtered SSUs  
 total\_ssus <- nrow(ssu\_grid\_sf)  
   
 # If the PSU contains enough SSUs, a random sample of SSUs is selected  
 if(total\_ssus >= (num\_primary\_ssus + num\_alternative\_ssus)) {  
 primary\_ssus\_indices <- sample(1:total\_ssus, num\_primary\_ssus, replace = FALSE)  
 available\_for\_alternatives <- setdiff(1:total\_ssus, primary\_ssus\_indices)  
 alternative\_ssus\_indices <- sample(available\_for\_alternatives, num\_alternative\_ssus, replace = FALSE)  
   
 selected\_ssus\_sf[[psu\_id]] <- rbind(ssu\_grid\_sf[primary\_ssus\_indices, ], ssu\_grid\_sf[alternative\_ssus\_indices, ])  
   
 # Generate alternative TSUs for SSUs with naming  
 primary\_tsus <- lapply(primary\_ssus\_indices,   
 function(index) {  
 generate\_tsu\_points\_within\_ssu(ssu\_grid\_sf[index, ], number\_TSUs, index, "Alternative",crops)  
 }  
 )  
   
 # Generate TSUs for alternative SSUs with naming  
 alternative\_tsus <- lapply(alternative\_ssus\_indices,   
 function(index) {  
 generate\_tsu\_points\_within\_ssu(ssu\_grid\_sf[index, ], number\_TSUs, index, "Alternative",crops)  
 }  
 )  
   
 # Combine all TSUs of the current PSU into one sf object  
 alt\_psus\_tsus\_sf[[psu\_id]] <- do.call(rbind, c(primary\_tsus, alternative\_tsus))  
   
 # If there are not enough SSUs, a warning is issued, and the PSU is skipped.  
 } else {  
 warning(paste("PSU", psu\_id, "does not have enough SSUs for selection. Skipping."))  
 }  
}

## [1] "1 out of 30"  
## [1] "2 out of 30"  
## [1] "3 out of 30"  
## [1] "4 out of 30"  
## [1] "5 out of 30"  
## [1] "6 out of 30"  
## [1] "7 out of 30"  
## [1] "8 out of 30"  
## [1] "9 out of 30"  
## [1] "10 out of 30"  
## [1] "11 out of 30"  
## [1] "12 out of 30"  
## [1] "13 out of 30"  
## [1] "14 out of 30"  
## [1] "15 out of 30"  
## [1] "16 out of 30"  
## [1] "17 out of 30"  
## [1] "18 out of 30"  
## [1] "19 out of 30"  
## [1] "20 out of 30"  
## [1] "21 out of 30"  
## [1] "22 out of 30"  
## [1] "23 out of 30"  
## [1] "24 out of 30"  
## [1] "25 out of 30"  
## [1] "26 out of 30"  
## [1] "27 out of 30"  
## [1] "28 out of 30"  
## [1] "29 out of 30"  
## [1] "30 out of 30"

# Combine TSUs from all PSUs into one sf object  
alt\_tsus\_combined\_sf <- do.call(rbind, alt\_psus\_tsus\_sf)  
alt\_tsus\_combined\_sf$TSU\_Type <- "Target"  
alt\_tsus\_combined\_sf[alt\_tsus\_combined\_sf$TSU\_ID >1,"TSU\_Type"] <- "Alternative"  
alt\_tsus\_combined\_sf$PSU\_Type <- "Alternative"  
  
alt\_tsus\_combined\_sf <- alt\_tsus\_combined\_sf %>%  
 dplyr::select("TSU\_Name","PSU\_ID","SSU\_ID","TSU\_ID","PSU\_Type","SSU\_Type","TSU\_Type","geometry")



### Exporting the alternative sampling units

This section covers the process of exporting target sampling units.

Cluster information from the alternative PSUs is joined to the TSUs for identification and analysis.

replacements <- replacements %>%  
 rename(Replace\_ID= cluster)  
  
alt\_tsus\_combined\_sf <- st\_join(alt\_tsus\_combined\_sf, replacements)

Then, an order is assigned to the TSUs within each alternative PSU. The sampling order differentiates between target TSUs (1–4) and alternative TSUs (5–7).

alt\_tsus\_combined\_sf <- alt\_tsus\_combined\_sf %>%  
 group\_by(PSU\_ID) %>%  
 mutate(order = match(SSU\_ID, unique(SSU\_ID))) %>%  
 ungroup()

Each TSU is assigned a unique site ID for traceability. The ID is constructed using the country code, PSU ID, SSU order, and TSU ID.

alt\_tsus\_combined\_sf$site\_id = paste0(ISO.code, alt\_tsus\_combined\_sf$PSU\_ID, "-", alt\_tsus\_combined\_sf$order, "-", alt\_tsus\_combined\_sf$TSU\_ID)

Finally, alternative PSUs and TSUs are saved as shapefiles for further use.

write\_sf(replacements,paste0(results.path,"/PSUs\_replacements.shp"), overwrite=TRUE)  
  
write\_sf(alt\_tsus\_combined\_sf,paste0(results.path,"/TSUs\_replacements.shp"), overwrite=TRUE)

### Final outputs

This section summarizes how to process and export information about PSUs, including counting PSUs per cluster, joining valid and remaining PSUs for availability analysis, and saving the final outputs as shapefiles.

The number of PSUs available is counted in each cluster for the valid PSUs. Here, valid.PSU\_clusters refers to the dataset containing valid PSUs, and the group\_by() function ensures that counts are aggregated by each Replace\_ID (representing unique clusters). The summarise(Count = n()) calculates the total number of PSUs in each cluster.

valid\_counts <- valid.PSU\_clusters %>%  
 group\_by(Replace\_ID) %>%  
 summarise(Count = n())

Similar operation for the remaining (alternative) PSUs, here calculates how many alternative PSUs are available in each cluster.

remaining\_counts <- remaining.PSU\_clusters %>%  
 group\_by(cluster) %>%  
 summarise(Count = n())

The st\_join() function combines the two datasets by matching cluster IDs. The resulting dataset includes both valid and remaining PSU counts, allowing for easy comparison. The shapefile is saved to the results folder (results.path).

availability <- st\_join(valid\_counts, remaining\_counts, by = "cluster", suffix = c("\_valid", "\_remaining"))  
  
write\_sf(availability,paste0(results.path,"/availability.shp"), overwrite=TRUE)

Last but not least, a single file is created to merge the shapefiles for target PSUs (PSUs\_target.shp) and replacement PSUs (PSUs\_replacements.shp).

tar <- sf::st\_read(file.path(paste0(results.path,"PSUs\_target.shp")))  
repl<- sf::st\_read(file.path(paste0(results.path,"PSUs\_replacements.shp")))  
  
crop\_PSU <- rbind(tar,repl)  
  
crop\_PSU <- crop\_PSU %>%  
 group\_by(Replace\_ID) %>%  
 summarize(geometry = st\_union(geometry))  
  
write\_sf(crop\_PSU,paste0(results.path,"/target\_repl\_PSUs.shp"), overwrite=TRUE)