Q&A log from 13 March 2026
Automating compositional dissimilarity and biodiversity turnover mapping with dissmapr

Will this session be available somewhere to come back to see the details?

- The recording will be uploaded here early next week: https://b-cubed.eu/videos?categoryId=2. You can also watch for a short news item that will include links to the slides and the Q&A, similar to previous sessions: (see here: https://b-cubed.eu/news).

Can you remind me what we need to get installed, please? R, R-studio, and it was something else. Thank you.

- rtools44

Can you share the link to the repository, please?

- Yes, access it from this link: https://b-cubed-eu.github.io/dissmapr. Additional info here: https://docs.b-cubed.eu/guides/dissimilarity-cube

So, sites will turn into the space component of the cube?

- If there are coordinates, yes.

In which way does this model help with citizen data bias?

- It doesn't. There's another package that is looking at that.

R script for using the Dissmapr

```
#
================================================================
==============
# Compositional Dissimilarity and Biodiversity Turnover Mapping with `dissmapr`
# B-Cubed training: transforming biodiversity observations into insights
# Training date: 13 March 2026
#
# Prepared by:
# Sandra MacFadyen
# Stellenbosch University / BioGIS
# Email: macfadyen@sun.ac.za | sandra@biogis.co.za
#
# Package website: https://b-cubed-eu.github.io/dissmapr/
# B-Cubed project: https://b-cubed.eu/
```

```
#
==============================================================================

#!/usr/bin/env Rscript

# ---------------------------------------------------------------------------
# PURPOSE
# ---------------------------------------------------------------------------
# This teaching script walks through a complete first-pass `dissmapr` workflow
# using the bundled butterfly example data.
#
# The emphasis is on understanding:
# - what each step does
# - which objects are created
# - what you should inspect before moving on
# - where you can substitute their own data later
#
# IMPORTANT:
# Run this script section by section in RStudio. Do not run the whole file at
# once on your first pass.
# ---------------------------------------------------------------------------

# ---------------------------------------------------------------------------
# WHAT THIS SCRIPT COVERS
# ---------------------------------------------------------------------------
#  1. Install and load packages
#  2. Load example occurrence data
#  3. Convert records to site-by-species format
#  4. Build a common analysis grid
#  5. Aggregate point records into grid cells
#  6. Visualise sampling effort and richness
#  7. Extract environmental covariates
#  8. Assemble the site-by-environment table
#  9. Reduce collinearity among predictors
# 10. Explore zeta diversity patterns
# 11. Fit an order-2 zeta MSGDM model
# 12. Predict current biodiversity turnover
# 13. Delineate current bioregions
# 14. Bonus: illustrate possible future turnover and bioregion change
# ---------------------------------------------------------------------------

# ---------------------------------------------------------------------------
# BEFORE YOU START
# ---------------------------------------------------------------------------
# Recommended workflow for you:
# 1. Open this file in RStudio.
# 2. Set the working directory to a folder where you have write access.
# 3. Run one section at a time.
# 4. Read the comments before running the code.
# 5. After each checkpoint, inspect the saved object.
# 6. If something fails, restart from the most recent checkpoint rather than
#    rerunning everything.
#
# Suggested inspection tools in RStudio:
# - View(object)
# - dim(object)
# - str(object)
# - names(object)
# - head(object)
# ---------------------------------------------------------------------------

#
=============================================================
==============
# 0. Housekeeping and helper functions
#
=============================================================
==============

# This block creates folders where outputs will be written.
# Using folders inside the current working directory makes the script easier to
# rerun at home on a different computer.

# # Choose a project folder that exists on your own computer.
# # In class you can set this manually. At home, you can use getwd() or a
# # dedicated RStudio Project folder.
mydir = getwd()

# # OR set your own working directory
# mydir = 'D:/Courses_Workshops/b-cubed_2026/13Mar26_dissmapr' # Change this
to your local working directory
setwd(mydir)

# Create folder paths for the main kinds of output we will save.
# - out_dir: top-level folder for this training workflow
# - data_dir: downloaded environmental data
# - plot_dir: saved figures
# - check_dir: saved R objects that can be reloaded later
out_dir   = file.path(mydir, "dissmapr_training")
data_dir  = file.path(out_dir, "extdata")
plot_dir  = file.path(out_dir, "plots")
check_dir = file.path(out_dir, "checkpoints")

# Create the folders if they do not already exist.
# recursive = TRUE means R can create parent folders as needed.
```

```
dir.create(out_dir,   showWarnings = FALSE, recursive = TRUE)
dir.create(data_dir,  showWarnings = FALSE, recursive = TRUE)
dir.create(plot_dir,  showWarnings = FALSE, recursive = TRUE)
dir.create(check_dir, showWarnings = FALSE, recursive = TRUE)

# Keep character data as character vectors unless we explicitly choose factors.
options(stringsAsFactors = FALSE)

# announce() prints a visible divider in the Console so you can see when a
# new section begins.
announce = function(text) {
  message("\n", paste(rep("=", 78), collapse = ""))
  message(text)
  message(paste(rep("=", 78), collapse = ""), "\n")
}

# checkpoint_save() writes an R object to disk so we can restart from here later
# without rerunning earlier steps.
checkpoint_save = function(object, filename) {
  saveRDS(object, file = filename)
  message("Saved checkpoint: ", normalizePath(filename, winslash = "/", mustWork =
FALSE))
}

# checkpoint_load() reads a saved checkpoint back into R.
# This is not used immediately below, but is useful when rerunning from home.
checkpoint_load = function(filename) {
  readRDS(filename)
}

# save_current_plot() saves the most recent ggplot figure.
# Saving plots helps you compare results when trying different settings.
save_current_plot = function(filename, width = 9, height = 6, dpi = 300) {
  ggplot2::ggsave(filename = filename, width = width, height = height, dpi = dpi)
  message("Saved plot: ", normalizePath(filename, winslash = "/", mustWork =
FALSE))
}

# Print a visible message at the start so users know the script has begun.
announce("Starting teaching-oriented `dissmapr` workflow")
message("Project directory: ", normalizePath(mydir, winslash = "/", mustWork =
FALSE))
message("Outputs will be saved to: ", normalizePath(out_dir, winslash = "/",
mustWork = FALSE))

#
=================================================================
==============
# 1. Install and load packages
#
==================================================================
==============

# WHAT THIS SECTION DOES
# ---------------------
# Installs and loads the packages used in this workflow.
#
# WHY THIS MATTERS
# ----------------
# Many workshop issues are package-installation issues. If a package fails here,
# fix that before continuing.
#
# WHAT YOU SHOULD CHECK
# -------------------------
# - every package loads without an error
# - warnings are noted but not confused with errors
# - `dissmapr` loads successfully before moving on

announce("1. Installing and loading packages")

# This character vector lists the CRAN packages used in the script.
# You can compare this list with their own project needs later.
required_pkgs = c(
  "dplyr", "tidyr", "sf", "terra", "zetadiv", "ggplot2","units","prodlim",
  "mclust", "purrr", "viridisLite", "remotes", "knitr", "scam", "remotes"
)

# install_if_missing() checks whether each package is already installed.
# It only installs the ones that are missing, which makes reruns faster.
install_if_missing = function(pkgs) {
  missing = pkgs[!vapply(pkgs, requireNamespace, logical(1), quietly = TRUE)]
  if (length(missing) > 0) {
    message("Installing missing packages: ", paste(missing, collapse = ", "))
    install.packages(missing)
  } else {
    message("All listed CRAN packages already available.")
  }
}

# Run the helper to install any missing CRAN packages.
install_if_missing(required_pkgs)

# Install `dissmapr` from GitHub if needed.
# Leave this active if you are running the workshop code for the first time at home.
# During the training session, `dependencies = FALSE` saves time by avoiding a long
# dependency install. If you get errors about missing packages, change it to
# `dependencies = TRUE` so R also installs the required package dependencies.
if (!requireNamespace("dissmapr", quietly = TRUE)) {
  remotes::install_github("b-cubed-eu/dissmapr", dependencies = FALSE)
}

# Load the packages into the current R session.
# If a library() call fails, stop and resolve that before continuing.
library(dissmapr)
library(dplyr)
library(tidyr)
library(sf)
library(terra)
library(zetadiv)
library(ggplot2)
library(mclust)
library(purrr)

# Save sessionInfo() to a text file.
# This is very helpful for troubleshooting package or version problems later.
# writeLines(capture.output(sessionInfo()), con = file.path(out_dir, "sessionInfo.txt"))

#
=================================================================
=============
# 2. Load example occurrence data
#
=================================================================
=============

# WHAT THIS SECTION DOES
# ----------------------
# Loads the bundled butterfly dataset and standardises it with
# `get_occurrence_data()`.
#
# CONCEPTUAL GOAL
# ---------------
# At this stage the data are still occurrence records: one row per record,
# not one row per grid cell.
#
# WHAT YOU SHOULD LEARN HERE
# ------------------------------
# - what an occurrence table looks like before gridding
# - where coordinates and taxonomy are stored
# - why data formatting is a common bottleneck
#
# TRY YOUR OWN DATA
# -----------------
# Later you can replace the bundled example with your own CSV if it has:
```

```
# - longitude and latitude columns
# - a species/taxon column
# - one row per occurrence record

announce("2. Loading and standardising occurrence records")

# Load the example RData object shipped with the package.
# The object is loaded into the current environment.
load(
  system.file("extdata", "gbif_butterflies_csv.RData", package = "dissmapr"),
  envir = knitr::knit_global()
)

# Convert the raw occurrence table into a standardised format that `dissmapr`
# can use more reliably.
bfly_data = get_occurrence_data(
  data        = gbif_butterflies_csv,
  source_type = "data_frame"
)

# Example alternative for home use:
# This shows how you could load their own CSV file instead.
# bfly_data = get_occurrence_data(
#   data        = "path/to/your_occurrences.csv",
#   source_type = "local_csv",
#   sep         = ","
# )

# Inspect the imported data.
# Here we print dimensions and a subset of columns to check that the import
# worked as expected.
message("Occurrence data dimensions:")
dim(bfly_data)
str(bfly_data[, c(51, 52, 22, 23, 1, 14, 16, 17, 30)])
head(bfly_data[, c(51, 52, 22, 23, 1, 14, 16, 17, 30)])

#
============================================================================
# 3. Convert occurrence records to site-by-species format
#
============================================================================

# WHAT THIS SECTION DOES
# ---------------------
# `format_df()` converts the occurrence table into objects that are easier to
# use in community-level analysis.
```

```r
#
# OUTPUTS
# -------
# - `site_obs`: site-observation table
# - `site_spp`: site-by-species table
#
# WHAT YOU SHOULD CHECK
# -------------------------
# - site_obs exists and looks sensible
# - site_spp has coordinates plus many species columns
# - species names are not garbled or duplicated by formatting issues

announce("3. Building site-by-species tables")

# format_df() reshapes the long occurrence data into analysis-ready tables.
# - species_col points to the taxon name column
# - value_col uses the presence/absence values already in the data
# - format = "long" tells the function the source data are in long format
bfly_result = format_df(
  data        = bfly_data,
  species_col = "verbatimScientificName",
  value_col   = "pa",
  extra_cols  = NULL,
  format      = "long"
)

# Show the structure of the returned list so you can see what objects were
# created.
str(bfly_result, max.level = 1)

# Extract the two most important outputs into separate objects with short names.
site_obs = bfly_result$site_obs
site_spp = bfly_result$site_spp

# Inspect the site-level observation table.
message("site_obs dimensions:")
dim(site_obs)
head(site_obs)

# Inspect the site-by-species table.
message("site_spp dimensions:")
dim(site_spp)
head(site_spp[, 1:6])

# Count how many species columns are present.
# The first three columns are site / coordinate fields, so we subtract 3.
n_sp = ncol(site_spp) - 3
message("Number of species columns in site_spp: ", n_sp)
```

```
# Print the first few species column names as a quick check.
sp_cols = names(site_spp)[-c(1:3)]
sp_cols[1:10]

# # Save checkpoints so we can start from here later.
# checkpoint_save(site_obs, file.path(check_dir, "site_obs.rds"))
# checkpoint_save(site_spp, file.path(check_dir, "site_spp.rds"))

# STOP HERE AND CHECK
# -------------------
# You should confirm that:
# - the first columns are identifiers / coordinates
# - the remaining columns are taxa
# - the values look like presence / absence, not strings or counts by mistake

#
============================================================================
# 4. Define the area of interest and a common grid
#
============================================================================

# WHAT THIS SECTION DOES
# ----------------------
# Builds a regular grid over South Africa. This grid becomes the common spatial
# unit for observations, environmental predictors and model outputs.
#
# WHY GRID SIZE MATTERS
# ---------------------
# Smaller grid cells capture finer spatial detail but increase sparsity and
# runtime. Larger grid cells are simpler and faster but may smooth over real
# ecological variation.
#
# TRY YOUR OWN DATA
# -----------------
# This is one of the most important adaptation points in the workflow.
# You can later replace the South Africa boundary with their own study area
# and test different resolutions.

announce("4. Creating the area of interest and grid template")

# Read the South Africa boundary shapefile bundled with the package.
rsa = sf::st_read(system.file("extdata", "rsa.shp", package = "dissmapr"), quiet =
TRUE)

# Set the grid resolution in decimal degrees.
# This determines the size of the analysis cells used later.
grid_res_deg = 0.5

# Convert the boundary to a terra vector object because later raster steps use
# the terra package.
rsa_vect = terra::vect(rsa)

# Create an empty raster template covering the study area.
# Each raster cell will later represent one analysis grid cell.
grid_template = terra::rast(rsa_vect, resolution = grid_res_deg, crs =
terra::crs(rsa_vect))

# Fill the raster with a placeholder value so it can be masked.
values(grid_template) = 1

# Mask the raster to the South Africa boundary so only cells inside the country
# remain in the study area.
grid_masked = terra::mask(grid_template, rsa_vect)

# Example for home use with a custom boundary:
# These lines show how you could replace the bundled boundary with their
# own study area polygon.
# study_area = sf::st_read("path/to/your_boundary.gpkg")
# study_vect = terra::vect(study_area)
# grid_template = terra::rast(study_vect, resolution = 0.25, crs =
terra::crs(study_vect))
# values(grid_template) = 1
# grid_masked = terra::mask(grid_template, study_vect)

#
==================================================================
==============
# 5. Aggregate point data into grid cells
#
==================================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# `generate_grid()` aggregates point records to the selected grid.
#
# KEY OUTPUTS
# -----------
# - `grid_sf`: spatial grid-cell object
# - `grid_spp`: gridded summaries with richness and effort
# - `grid_spp_pa`: site-by-species presence/absence by cell
# - `grid_r`: raster outputs
#
```

```
# WHAT YOU SHOULD CHECK
# -------------------------
# - number of occupied cells
# - that the grid-based objects were created
# - that downstream objects now represent grid cells, not original records

announce("5. Aggregating occurrences to the analysis grid")

# generate_grid() overlays the site data on the chosen grid and summarises the
# species information for each occupied cell.
# sum_cols tells the function which columns contain species values to aggregate.
grid_list = generate_grid(
  data      = site_spp,
  x_col     = "x",
  y_col     = "y",
  grid_size = grid_res_deg,
  sum_cols  = 4:ncol(site_spp),
  crs_epsg  = 4326
)

# Examine the main components returned by generate_grid().
str(grid_list, max.level = 1)

# Extract the outputs into shorter object names for convenience.
grid_r      = grid_list$grid_r$grid_id
grid_sf     = grid_list$grid_sf
grid_spp    = grid_list$grid_spp
grid_spp_pa = grid_list$grid_spp_pa

# Print dimensions to check how many cells and variables were created.
message("grid_sf dimensions:")
dim(grid_sf)
message("grid_spp dimensions:")
dim(grid_spp)
message("grid_spp_pa dimensions:")
dim(grid_spp_pa)

# # Save checkpoints for the gridded biodiversity objects.
# checkpoint_save(grid_list,   file.path(check_dir, "grid_list.rds"))
# checkpoint_save(grid_spp,    file.path(check_dir, "grid_spp.rds"))
# checkpoint_save(grid_spp_pa, file.path(check_dir, "grid_spp_pa.rds"))

#
==================================================================
==============
# 6. Quick visual check of sampling effort
```

```
#
============================================================================
==============

# WHY THIS MATTERS
# ----------------
# Biodiversity records are rarely sampled evenly. It is useful to inspect effort
# before fitting models because some apparent patterns may simply reflect where
# people collected most intensively.

announce("6. Mapping observation density for a quick quality check")

# Build a map showing the number of observations per grid cell.
# sqrt(obs_sum) is used to compress large values so the map is easier to read.
p_effort = ggplot() +
  geom_sf(data = grid_sf, fill = NA, colour = "darkgrey", linewidth = 0.2, alpha = 0.5) +
  geom_point(
    data = grid_spp,
    aes(
      x = centroid_lon,
      y = centroid_lat,
      size = sqrt(obs_sum),
      colour = sqrt(obs_sum)
    ),
    alpha = 0.8
  ) +
  scale_colour_viridis_c(option = "turbo", name = "√ observations") +
  scale_size_continuous(name = "√ observations", guide = "none") +
  geom_sf(data = rsa, fill = NA, colour = "black", linewidth = 0.4) +
  theme_minimal() +
  labs(
    title = "Observation density across South Africa (0.5° grid)",
    x = "Longitude", y = "Latitude"
  )

# Print the figure to the plotting window.
print(p_effort)

# # Save the figure so you have a copy outside the session.
# save_current_plot(file.path(plot_dir, "01_sampling_effort_map.png"))

#
============================================================================
==============
# 7. Quick visual check of effort and richness
#
============================================================================
==============
```

```r
# WHY THIS MATTERS
# ----------------
# Species richness often partly reflects sampling effort. Looking at the two
# side by side helps you think critically about possible sampling bias.

announce("7. Comparing effort and richness")

# Create a two-layer raster object with transformed effort and richness.
# Square-root transformation makes the maps visually easier to compare.
effRich_r = sqrt(grid_list$grid_r[[c("obs_sum", "spp_rich")]])

# Save default plotting layout
# old_par = par(mfrow = c(1, 1), mar = c(1, 1, 1, 5))
# Save current plotting parameters
old_par = par(no.readonly = TRUE)

# Set up a 1x2 plotting layout for base R maps.
par(mfrow = c(1, 2), mar = c(1, 1, 1, 2))

# Loop over the two raster layers and plot them one after the other.
for (i in 1:2) {
  plot(
    effRich_r[[i]],
    col = viridisLite::turbo(100),
    colNA = NA,
    axes = FALSE,
    main = c("Sampling effort (√ observation count)",
             "Species richness (√ unique species count)")[i],
    cex.main = 0.8
  )
  plot(terra::vect(rsa), add = TRUE, border = "black", lwd = 0.4)
}

# Reset plotting parameters so later plots are not affected.
par(old_par)

#
==================================================================
==============
# 8. Extract environmental covariates for grid cells
#
==================================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# `get_enviro_data()` retrieves environmental raster layers and extracts values
# to the grid-cell centroids.
#
# IN THIS SCRIPT
# --------------
# The predictors are WorldClim bioclimatic variables.
#
# WHAT YOU SHOULD CHECK
# -------------------------
# - `enviro_list` contains both raster and table outputs
# - `env_df` has one row per grid cell
# - environmental variables are numeric and mostly not missing
#
# TRY YOUR OWN DATA
# -----------------
# Later you can substitute:
# - other climate sources
# - topography
# - soils
# - land cover
# - remote sensing summaries
# The important thing is that predictor values match the same sites/grid cells
# used for the biodiversity data.

announce("8. Extracting environmental covariates")

# Download or load environmental predictors and extract values for the gridded
# biodiversity data.
# buffer_km sets the extraction neighbourhood, and source = "geodata" uses the
# geodata-backed climate source available through the function.
enviro_list = get_enviro_data(
  data      = grid_spp,
  buffer_km = 10,
  source    = "geodata",
  var       = "bio",
  res       = 5,
  grid_r    = grid_r,
  path      = data_dir,
  sp_cols   = 7:ncol(grid_spp),
  ext_cols  = c("obs_sum", "spp_rich")
)

# Print the structure of the returned object.
str(enviro_list, max.level = 1)

# Extract the raster and table components into shorter objects.
env_r  = enviro_list$env_rast
env_df = enviro_list$env_df
```

```
# Inspect the table version of the environmental data.
message("Environmental table dimensions:")
dim(env_df)
head(env_df)

# # Save a checkpoint so this expensive step does not need to be repeated.
# checkpoint_save(enviro_list, file.path(check_dir, "enviro_list.rds"))

#
==================================================================
==============
# 9. Inspect the environmental table
#
==================================================================
==============

# WHY THIS MATTERS
# ----------------
# It is easier to detect a problem in one predictor now than later when a model
# fails or behaves strangely.

announce("9. Inspecting extracted environmental data")

# At this stage env_df still uses the original variable names (e.g. bio01).
# We can use bio01 as a quick map of mean annual temperature.

dim(env_df)
head(env_df[, 1:6])

# Make a quick point map of one environmental variable to check that extraction
# has worked and values show a sensible spatial gradient.
p_temp = ggplot() +
  geom_sf(data = grid_sf, fill = NA, colour = "darkgrey", alpha = 0.4) +
  geom_point(
    data = env_df,
    aes(x = centroid_lon, y = centroid_lat, colour = bio01),
    shape = 15,
    size = 3
  ) +
  scale_colour_viridis_c(option = "turbo") +
  geom_sf(data = rsa, fill = NA, colour = "black") +
  theme_minimal() +
  labs(
    title = "Grid-cell mean annual temperature",
    x = "Longitude", y = "Latitude"
  )

# Print and save the environmental check map.
```

```
print(p_temp)
# save_current_plot(file.path(plot_dir, "02_mean_annual_temperature_map.png"))

#
================================================================
==============
# 10. Assemble the final site-by-environment table
#
================================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# Builds the site-level predictor table used later in zeta modelling.

announce("10. Building the combined site-by-environment table")

# Reorder columns so the site ID, coordinates and key summary variables appear
# first, followed by all environmental predictors.
grid_env = env_df %>%
  dplyr::select(
    grid_id, centroid_lon, centroid_lat,
    obs_sum, spp_rich, dplyr::everything()
  )

# Inspect the combined table structure.
str(grid_env, max.level = 1)
head(grid_env)

# # Save a checkpoint of the site-by-environment table.
# checkpoint_save(grid_env, file.path(check_dir, "grid_env_preproj.rds"))

#
================================================================
==============
# 11. Add projected coordinates
#
================================================================
==============

# WHY THIS MATTERS
# ----------------
# Geographic coordinates are useful for mapping, but distance-based analyses are
# easier to interpret in projected coordinates measured in metres.
#
# TRY YOUR OWN DATA
# -----------------
# For a different study region, choose a projection appropriate to that region.
```

```
announce("11. Reprojecting centroids for distance-based analysis")

# Convert the data frame to an sf point object using longitude and latitude.
centroids_sf = sf::st_as_sf(
  grid_env,
  coords = c("centroid_lon", "centroid_lat"),
  crs = 4326,
  remove = FALSE
)

# Reproject points to an equal-area coordinate reference system.
centroids_aea = sf::st_transform(centroids_sf, 9822)

# Extract projected x/y coordinates and append them to the original table.
# These projected coordinates are used later for distance-based zeta functions.
grid_env = cbind(
  grid_env,
  sf::st_coordinates(centroids_aea) |>
    as.data.frame() |>
    setNames(c("x_aea", "y_aea"))
)

# Inspect the old and new coordinate columns side by side.
head(grid_env[, c("grid_id", "centroid_lon", "centroid_lat", "x_aea", "y_aea")])

# # Save the projected version of the table.
# checkpoint_save(grid_env, file.path(check_dir, "grid_env.rds"))

#
============================================================================
==============
# 12. Remove strongly correlated predictors
#
============================================================================
==============

# WHY THIS MATTERS
# ----------------
# Highly correlated predictors can make model interpretation unstable.
# Predictor reduction is also an ecological decision, not just a technical one.

announce("12. Reducing collinearity among predictors")

# Rename env_df columns to the clearer variable labels defined earlier.
# The first three columns are site ID and coordinates, then climate variables,
# then observation and richness summaries.
```

```
names(env_df) = c("grid_id", "centroid_lon", "centroid_lat", names_env, "obs_sum",
"spp_rich")

# Remove predictors with pairwise correlations above the selected threshold.
# plot = TRUE will usually show a diagnostic plot to help inspect the result.
env_vars_reduced = rm_correlated(
  data      = env_df[, 4:23],
  cols      = NULL,
  threshold = 0.70,
  plot      = TRUE
)

# Compare the number of variables before and after filtering.
c(
  original = ncol(env_df[, 4:23]),
  reduced  = ncol(env_vars_reduced)
)

# # Save the reduced predictor set.
# checkpoint_save(env_vars_reduced, file.path(check_dir, "env_vars_reduced.rds"))

# STOP HERE AND CHECK
# -------------------
# You should look at which variables remain and ask:
# - Are the retained predictors ecologically interpretable?
# - Would I keep the same threshold for my own dataset?

#
=================================================================
==============
# 13. Expected zeta decline
#
=================================================================
==============

# CONCEPTUAL GOAL
# ---------------
# Zeta diversity measures how many species are shared among multiple sites.
# As the zeta order increases, the number of species shared by all sites usually
# declines.

announce("13. Calculating expected zeta decline")

# Calculate the expected zeta decline curve from order 1 to 10.
# We exclude the first metadata columns and keep only species presence/absence.
zeta_decline_ex = Zeta.decline.ex(
  grid_spp_pa[, -(1:7)],
  orders = 1:10
)

# # Save the result for later plotting or inspection.
# checkpoint_save(zeta_decline_ex, file.path(check_dir, "zeta_decline_ex.rds"))

#
=================================================================
==============
# 14. Empirical zeta decline
#
=================================================================
==============

# WHY THIS MATTERS
# ----------------
# The empirical curve gives a more realistic picture of shared-species decline
# in the observed data because it uses Monte Carlo sampling.

announce("14. Estimating empirical zeta decline")

# Estimate empirical zeta decline using Monte Carlo sampling and projected
# coordinates.
# normalize = "Jaccard" scales shared-species values to a similarity measure.
zeta_mc_utm = Zeta.decline.mc(
  grid_spp_pa[, -(1:7)],
  grid_env[, c("x_aea", "y_aea")],
  orders    = 1:10,
  sam       = 100,
  NON       = TRUE,
  normalize = "Jaccard"
)

# # Save the empirical zeta result.
# checkpoint_save(zeta_mc_utm, file.path(check_dir, "zeta_mc_utm.rds"))

#
=================================================================
==============
# 15. Distance decay of zeta diversity
#
=================================================================
==============

# WHAT YOU SHOULD LOOK FOR
# ----------------------------
# - declining similarity with increasing distance
# - differences among zeta orders
# - uncertainty around the fitted relationships
```

```
announce("15. Exploring distance decay across zeta orders")

# Close any open graphics device if needed.
# This helps avoid plotting issues in some R sessions.
if (dev.cur() > 1) dev.off()
par(old_par)

# Fit distance-decay relationships across several zeta orders.
# sam = 1000 increases the number of sampled combinations and usually gives a
# smoother result, but may take longer to run.
zeta_decays = Zeta.ddecays(
  grid_env[, c("x_aea", "y_aea")],
  grid_spp_pa[, -(1:7)],
  sam           = 100,
  orders        = 2:6,
  plot          = TRUE,
  confint.level = 0.95
)

# # Save the distance-decay outputs.
# checkpoint_save(zeta_decays, file.path(check_dir, "zeta_decays.rds"))

#
==============================================================
==============
# 16. Fit an order-2 zeta MSGDM model
#
==============================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# Fits the main explanatory model linking compositional turnover to environment
# and distance.
#
# WHY ORDER 2?
# ------------
# Order-2 zeta is the closest analogue to pairwise turnover and is a useful
# starting point for teaching and interpretation.
#
# NOTE
# ----
# The pkgdown site also includes helper functions such as `run_ispline_models()`,
# `plot_ispline_lines()` and `plot_ispline_boxplots()` for multi-order ispline
# workflows. You can explore these later after mastering the single-model
# version below.
```

```
announce("16. Fitting the zeta order-2 MSGDM model")

# Set a random seed so that workshop participants obtain reproducible sampling-
# based results.
set.seed(264)

# Fit a multi-site generalised dissimilarity model (MSGDM) for zeta order 2.
# - env_vars_reduced supplies the environmental predictors
# - centroid coordinates are used for geographic distance
# - reg.type = "ispline" allows flexible nonlinear relationships
zeta2 = Zeta.msgdm(
  grid_spp_pa[, -(1:7)],
  env_vars_reduced,
  grid_env[, c("centroid_lon", "centroid_lat")],
  sam           = 100,
  order         = 2,
  distance.type = "Euclidean",
  normalize     = "Jaccard",
  reg.type      = "ispline"
)

# Extract fitted I-splines from the model.
splines = Return.ispline(zeta2, env_vars_reduced, distance = TRUE)

# Plot the spline shapes to help interpret predictor effects.
Plot.ispline(splines, distance = TRUE)

# # Save the fitted model and spline objects.
# checkpoint_save(zeta2,    file.path(check_dir, "zeta2.rds"))
# checkpoint_save(splines,  file.path(check_dir, "zeta2_splines.rds"))

# STOP HERE AND CHECK
# -------------------
# You should inspect the spline plots and ask:
# - which predictors appear strongest?
# - are responses linear, threshold-like or saturating?
# - does distance appear more important than some environmental predictors?

#
============================================================
==============
# 17. Inspect model fit
#
============================================================
==============

announce("17. Reviewing fitted model performance")
```

```
# Calculate a simple deviance-explained measure from the fitted model summary.
model_dev_explained = with(summary(zeta2$model), 1 - deviance / null.deviance)
model_dev_explained

# Print the full model summary for closer inspection.
summary(zeta2$model)


#
==========================================================================
# 18. Zeta I-spline workflow using `run_ispline_models`
#
==========================================================================
# ---------------------------------------------------------------------------
# Fit zeta models for several orders, then visualize spline responses.
# ---------------------------------------------------------------------------

# Use a fixed seed so you can reproduce the same subsampling result.
set.seed(123)

# Fit I-spline zeta models for orders 2 to 6.
# This gives you both the fitted model objects and one combined table of raw
# predictors plus spline-transformed predictors, labelled by zeta order.
ispline_gdm_tab = dissmapr::run_ispline_models(
  spp_df    = grid_spp_pa[, -(1:7)],
  env_df    = env_vars_reduced,
  xy_df     = grid_env[, c("centroid_lon", "centroid_lat")],
  orders    = 2:6,
  sam       = 100,
  normalize = "Jaccard",
  reg_type  = "ispline"
)

# Check the structure so you can confirm the output looks right.
str(ispline_gdm_tab, max.level = 1)

# Extract the combined I-spline table.
# This is the main object you will use for plotting and interpretation.
ispline_tabs_all = ispline_gdm_tab$ispline_table

# Preview the first few rows so you can inspect the variables.
head(ispline_tabs_all)

# ---------------------------------------------------------------------------
# Plot partial-dependence curves for all covariates
# ---------------------------------------------------------------------------
# Identify all raw variables that have a matching spline term.
# For example, temp_mean_is becomes temp_mean.
raw_vars = sub(
  "_is$",
  "",
  grep("_is$", names(ispline_tabs_all), value = TRUE)
)

# Build one I-spline plot per covariate.
# Each plot shows how the spline response changes across zeta orders.
plots = lapply(raw_vars, function(var) {
  dissmapr::plot_ispline_lines(
    ispline_data = ispline_tabs_all,
    x_var        = var,
    orders       = paste("Order", 2:6),
    cols         = c("green", "cyan", "purple", "blue", "black"),
    shapes       = c(15, 16, 17, 18, 19)
  ) +
    ggplot2::ggtitle(paste("I-Spline Partial Effect of", var))
})

# Combine all covariate plots into one multi-panel figure.
patchwork::wrap_plots(plots, ncol = 2) +
  patchwork::plot_annotation(
    title = "Multi-Panel I-Spline Curves Across Covariates",
    theme = ggplot2::theme(
      plot.title = ggplot2::element_text(size = 16, face = "bold")
    )
  )

# Optional: save the combined multi-panel figure.
# save_current_plot(file.path(plot_dir, "ispline_multi_panel_curves.png"), width = 12,
height = 10)

# ----------------------------------------------------------------------------
# Optional: plot one covariate on its own
# ----------------------------------------------------------------------------

# Use this when you want to discuss one predictor in more detail.
dissmapr::plot_ispline_lines(
  ispline_data = ispline_tabs_all,
  x_var        = "distance",
  orders       = paste("Order", 2:6),
  cols         = c("green", "cyan", "purple", "blue", "black"),
  shapes       = c(15, 16, 17, 18, 19)
) +
  ggplot2::ggtitle("I-Spline Partial Effect of distance")
```

```
# Optional: save the single-variable plot.
# save_current_plot(file.path(plot_dir, "ispline_distance_curve.png"), width = 8, height
= 6)

# ---------------------------------------------------------------------------
# Plot facetted boxplots of all spline terms
# ---------------------------------------------------------------------------

# Plot the distribution of every spline-transformed predictor across zeta orders.
# Each facet is one spline term, which helps you compare their spread by order.
dissmapr::plot_ispline_boxplots(
  ispline_data   = ispline_tabs_all,
  ispline_suffix = "_is",
  order_col      = "zOrder",
  palette        = "viridis",
  direction      = -1,
  ncol           = 3
)

# Optional: save the boxplot figure.
# save_current_plot(file.path(plot_dir, "ispline_boxplots_by_order.png"), width = 12,
height = 9)

# ---------------------------------------------------------------------------
# Optional checkpoint save
# ---------------------------------------------------------------------------

# # Save the fitted output so you can reload it later without rerunning the models.
# checkpoint_save(ispline_gdm_tab, file.path(check_dir, "ispline_gdm_tab.rds"))
# checkpoint_save(ispline_tabs_all, file.path(check_dir, "ispline_tabs_all.rds"))

#
=============================================================
==============
# 19. Predict current zeta diversity across the study region
#
=============================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# `predict_dissim()` generates spatial predictions of compositional turnover.
#
# WHAT YOU SHOULD CHECK
# -------------------------
# - the prediction table has one row per site / cell
# - predicted zeta columns are present
# - the map looks spatially coherent

announce("19. Predicting current zeta diversity")

# Set a seed for reproducibility of any internal sampling steps.
set.seed(123)

# Extract the species column names from the grid presence/absence table.
spp_cols = names(grid_spp_pa[, -(1:7)])

# Predict zeta-based dissimilarity across the study area using the fitted model.
# show_plot = TRUE displays the prediction map directly.
predictors_df = dissmapr::predict_dissim(
  grid_spp     = grid_spp_pa,
  species_cols = spp_cols,
  env_vars     = env_vars_reduced,
  zeta_model   = zeta2,
  grid_xy      = grid_env,
  x_col        = "centroid_lon",
  y_col        = "centroid_lat",
  bndy_fc      = rsa,
  show_plot    = TRUE
)

# Inspect the prediction output.
dim(predictors_df)
names(predictors_df)
head(predictors_df[, 5:11])

# # Save the prediction table.
# checkpoint_save(predictors_df, file.path(check_dir, "predictors_df.rds"))

#
==================================================================
==============
# 20. Delineate current bioregions from predicted turnover
#
==================================================================
==============

# WHAT THIS SECTION DOES
# ----------------------
# `map_bioreg()` clusters the predicted turnover surface into candidate
# bioregions.
#
# WHAT YOU SHOULD LOOK FOR
# ----------------------------
# - whether different clustering methods agree broadly
# - whether regions align with obvious gradients
# - whether the selected k seems ecologically defensible

announce("20. Mapping current bioregions")

# Cluster the prediction surface into candidate bioregions.
# method = "all" applies several clustering methods for comparison.
# k_override = 8 fixes the number of groups to 8 for teaching consistency.
bioreg_current = dissmapr::map_bioreg(
  data        = predictors_df,
  scale_cols  = c("pred_zetaExp", "centroid_lon", "centroid_lat"),
  method      = "all",
  k_override  = 8,
  interpolate = "nn",
  x_col       = "centroid_lon",
  y_col       = "centroid_lat",
  res         = grid_res_deg,
  crs         = "EPSG:4326",
  plot        = TRUE,
  bndy_fc     = rsa
)

# Inspect the returned object to see the available clustering outputs.
str(bioreg_current, max.level = 1)

# # Save the current bioregion result.
# checkpoint_save(bioreg_current, file.path(check_dir, "bioreg_current.rds"))

#
==================================================================
==============
# 21. Final notes
#
==================================================================
==============

announce("21. Main workflow complete")

# Print suggested next steps for you to try after the training session.
message("Suggested next steps for home practice:")
message("1. Reload checkpoints and rerun only the modelling sections.")
message("2. Repeat the workflow with a different grid size.")
message("3. Replace the bundled butterfly data with your own occurrence data.")
message("4. Replace or extend the environmental predictor set.")
message("5. Explore multi-order ispline helpers in the dissmapr articles.")

# Restart example:
# These commented lines show how you can resume from saved checkpoints.
# grid_env          = readRDS(file.path(check_dir, "grid_env.rds"))
# env_vars_reduced  = readRDS(file.path(check_dir, "env_vars_reduced.rds"))
# zeta2             = readRDS(file.path(check_dir, "zeta2.rds"))



#
============================================================
==============
# BONUS SECTION - illustrative scenario workflow
#
============================================================
==============

# IMPORTANT CAUTION FOR YOU
# -----------------------------
# This section is a teaching illustration of how to run scenario-based turnover
# predictions using modified environmental inputs.
#
# It is NOT a formal climate-impact analysis.
# The future values below are stylised changes used to demonstrate workflow
# mechanics. A real forecasting exercise would need appropriately selected
# climate scenarios, explicit assumptions, and careful ecological validation.

announce("Bonus section: illustrative future turnover and bioregion mapping")

# ---------------------------------------------------------------------------
# 1. Predict illustrative future turnover scenarios
# ---------------------------------------------------------------------------

# Reuse the species columns and predictor names from earlier objects.
spp_cols  = names(grid_spp_pa)[-(1:7)]
all_vars  = names(env_vars_reduced)

# Identify groups of variables by name so we can modify them in a structured way.
# grep() searches the variable names using a text pattern.
temp_vars = grep("^temp", all_vars, value = TRUE)
iso_vars  = grep("^iso",  all_vars, value = TRUE)
rain_vars = grep("^rain", all_vars, value = TRUE)
obs_var   = "obs_sum"

# Define the scenario labels used in the teaching example.
horizons   = c("2030", "2040", "2050")

# Define stylised scenario changes used for teaching only.
# These are not official climate projections.
mean_delta = list(
  temp   = c("2030" = +2,   "2040" = +4,   "2050" = +6),
  iso    = c("2030" = +0.5, "2040" = +1.0, "2050" = +1.5),
  rain   = c("2030" = 0.9,  "2040" = 0.8,  "2050" = 0.7),
  effort = c("2030" = 1.3,  "2040" = 1.6,  "2050" = 2.0)
)

# Increase spatial contrast around the mean to create a more visibly different
# teaching scenario.
exagg_factor = c("2030" = 1.5, "2040" = 2.0, "2050" = 2.5)

# Helper function to amplify deviations from the mean.
# This preserves the variable mean while stretching differences among cells.
amplify = function(x, factor) {
  m = mean(x, na.rm = TRUE)
  m + (x - m) * factor
}

# Build a named list of environmental predictor tables.
# It includes the current data plus three altered scenario versions.
env_scenarios = c(
  list(current = env_vars_reduced),
  purrr::map(horizons, function(yr) {
    df = env_vars_reduced
    df[temp_vars] = df[temp_vars] + mean_delta$temp[yr]
    df[iso_vars]  = df[iso_vars]  + mean_delta$iso[yr]
    df[rain_vars] = df[rain_vars] * mean_delta$rain[yr]
    df[[obs_var]] = df[[obs_var]] * mean_delta$effort[yr]
    df[temp_vars] = purrr::map_dfc(df[temp_vars], amplify, factor = exagg_factor[yr])
    df[iso_vars]  = purrr::map_dfc(df[iso_vars],  amplify, factor = exagg_factor[yr])
    df[[obs_var]] = pmin(pmax(df[[obs_var]], 50), 8000)
    df
  }) |> rlang::set_names(horizons)
)

# Inspect the structure of the scenario list.
str(env_scenarios, max.level = 1)

# Predict turnover under each scenario in turn.
# imap() loops over the list and also gives access to each scenario name.
set.seed(123)
scenario_dfs = purrr::imap(env_scenarios, ~ {
  df = dissmapr::predict_dissim(
    grid_spp     = grid_spp_pa,
    species_cols = spp_cols,
    env_vars     = .x,
    zeta_model   = zeta2,
    grid_xy      = grid_env,
    x_col        = "centroid_lon",
    y_col        = "centroid_lat",
    skip_scale   = FALSE,
    show_plot    = FALSE
  )
  df$scenario = .y
  df
})

# Combine the prediction outputs from all scenarios into one long table.
# This gives you a single data frame that is easier to inspect, plot, and pass
# into later bioregionalisation steps.
all_preds = bind_rows(scenario_dfs) |>
  mutate(scenario = factor(scenario, levels = c("current", horizons)))

# Print a quick summary so you can check that the merged table looks sensible
# before using it in maps and clustering.
summary(all_preds)

# Plot the predicted zeta[2] surface for each scenario.
# Each tile shows the predicted value at a grid cell, and faceting lets you
# compare the current pattern against future scenarios side by side.
p_future = ggplot(all_preds, aes(centroid_lon, centroid_lat, fill = pred_zetaExp)) +
  geom_tile() +
  facet_wrap(~ scenario, ncol = 2) +
  scale_fill_viridis_c(direction = -1, name = expression(zeta[2])) +
  geom_sf(data = rsa, fill = NA, color = "black", inherit.aes = FALSE) +
  coord_sf() +
  labs(
    x = "Longitude",
    y = "Latitude",
    title = expression("Illustrative predicted " * zeta[2] * " under current and future
scenarios")
  ) +
  theme_minimal() +
  theme(
    strip.text = element_text(face = "bold"),
    panel.grid = element_blank()
  )

# Display the map and save a copy so you can use it in reports or workshop notes.
print(p_future)
# save_current_plot(file.path(plot_dir, "03_future_scenarios_zeta2.png"), width = 10,
height = 8)

# # Save the combined prediction table as a checkpoint.
# # This means you do not need to rebuild it if you restart the workflow later.
# checkpoint_save(all_preds, file.path(check_dir, "all_preds_future_scenarios.rds"))

# ---------------------------------------------------------------------------
# 2. Map future bioregions
# ---------------------------------------------------------------------------

# Split the combined prediction table into a list by scenario.
# The bioregionalisation function expects one data object per scenario.
by_scn = split(all_preds, all_preds$scenario)

# Delineate bioregions for all scenarios using the same settings.
# This lets you compare how cluster boundaries may shift through time.
bioreg_future = dissmapr::map_bioreg(
  data        = by_scn,
  scale_cols  = c("pred_zetaExp", "centroid_lon", "centroid_lat"),
  method      = "all",
  k_override  = 8,
  interpolate = "nn",
  x_col       = "centroid_lon",
  y_col       = "centroid_lat",
  res         = grid_res_deg,
  crs         = "EPSG:4326",
  plot        = TRUE,
  bndy_fc     = rsa
)

# Inspect the output structure so you can confirm what clustering results
# and rasters were returned.
str(bioreg_future, max.level = 1)

# # Save the future bioregion object so you can reload it without recomputing.
# checkpoint_save(bioreg_future, file.path(check_dir, "bioreg_future.rds"))

# ---------------------------------------------------------------------------
# 3. Map sensitivity of current bioregions to clustering method
# ---------------------------------------------------------------------------

# Collect the current-time nearest-neighbour bioregion maps from each clustering
# method into one vector. This lets you measure how sensitive the result is to
# the choice of clustering algorithm.
current_nn = c(
  bioreg_current$nn$current$kmeans_algn_current,
  bioreg_current$nn$current$pam_algn_current,
  bioreg_current$nn$current$hclust_algn_current,
  bioreg_current$nn$current$gmm_algn_current
)

# Calculate agreement and disagreement metrics across clustering methods.
# These layers show where the mapped bioregions are stable or uncertain.
sens_bioregDiff = dissmapr::map_bioregDiff(
  current_nn,
```

```r
  approach = "all"
)

# Resample the sensitivity layers to your masked grid and apply the mask so
# the outputs align neatly with the study area.
mask_sens_bioregDiff = terra::mask(
  terra::resample(sens_bioregDiff, grid_masked, method = "near"),
  grid_masked
)

# Set up a multi-panel plotting layout and readable titles for each metric.
par(mfrow = c(3, 2), mar = c(1, 1, 1, 5))
titles = c(
  "Difference count", "Shannon entropy", "Stability",
  "Transition frequency", "Weighted change index"
)

# Plot each sensitivity metric so you can see where clustering methods agree
# strongly and where the delineation is more uncertain.
for (i in seq_along(titles)) {
  plot(
    mask_sens_bioregDiff[[i]],
    col   = hcl.colors(100, palette = "viridis", rev = TRUE),
    colNA = NA,
    axes  = FALSE,
    main  = titles[i],
    cex.main = 0.8
  )
  plot(terra::vect(rsa), add = TRUE, border = "black", lwd = 0.4)
}

# Restore the previous plotting settings.
par(old_par)

# # Save the masked sensitivity rasters for later interpretation or figure export.
# checkpoint_save(mask_sens_bioregDiff, file.path(check_dir,
"mask_sens_bioregDiff.rds"))

# ---------------------------------------------------------------------------
# 4. Map sensitivity of bioregions to future change
# ---------------------------------------------------------------------------

# Extract the hierarchical clustering result for the current period and each
# future scenario. Here you are holding the clustering method constant so that
# differences mainly reflect projected change through time.
future_hclt = c(
  bioreg_future$nn$current$hclust_current,
  bioreg_future$nn$`2030`$hclust_2030,
  bioreg_future$nn$`2040`$hclust_2040,
  bioreg_future$nn$`2050`$hclust_2050
)

# Calculate change metrics across time slices.
# These layers show where bioregions are projected to remain stable versus shift
# under future scenarios.
future_bioregDiff = dissmapr::map_bioregDiff(future_hclt, approach = "all")

# Align the future-change layers to your masked grid for consistent plotting
# and downstream comparisons.
mask_future_bioregDiff = terra::mask(
  terra::resample(future_bioregDiff, grid_masked, method = "near"),
  grid_masked
)

# Plot the future-change sensitivity metrics using the same titles and colour
# scale as above, so the maps are easy to compare.
par(mfrow = c(3, 2), mar = c(1, 1, 1, 5))
for (i in seq_along(titles)) {
  plot(
    mask_future_bioregDiff[[i]],
    col   = hcl.colors(100, palette = "viridis", rev = TRUE),
    colNA = NA,
    axes  = FALSE,
    main  = titles[i],
    cex.main = 0.8
  )
  plot(terra::vect(rsa), add = TRUE, border = "black", lwd = 0.4)
}

# Restore your original plotting parameters after the multi-panel figure.
par(old_par)

# # Save the future-change sensitivity outputs as another checkpoint.
# checkpoint_save(mask_future_bioregDiff, file.path(check_dir,
"mask_future_bioregDiff.rds"))

# Let you know the script has finished successfully.
announce("Script complete")
message("Main workflow and bonus teaching workflow finished.")
message("For questions after the workshop, contact Sandra MacFadyen:
macfadyen@sun.ac.za | sandra@biogis.co.za")
```