Combined analyses with RPadrino, ipmr, and other databases

Contents

Combining PADRINO data with your own IPMs

In many cases, we may wish to combine data that we've collected and not yet published with data from PADRINO. We can do this using `ipmr` to create IPM objects, and then appending them to an list created with `pdb_make_ipm()`. We'll create two additional deterministic IPMs, and then attach them to a list created from PADRINO.

Creating our own IPMs

The goal of this case study is to show how to combine data, and not necessarily how to use `ipmr`. `ipmr` is extensively documented on the project's website and in the publication describing the package. Therefore, the next few chunks of code assume you've already consulted these resources and will have a reasonable understanding of what's going on. If you have not already consulted these, please do so now.

Our first "homemade" IPM will be a general IPM. For now, we are only going to construct `proto_ipm` objects for each one of these homemade IPMs. Once we have our PADRINO IPMs selected, we'll splice everything together and generate actual IPM objects.

```
# Loading RPadrino automatically loads ipmr, so we don't need to load both.
library(RPadrino)

# Set up the initial population conditions and parameters.
# These are hypothetical values and don't correspond to any particular
# species.

data_list <- list(
  g_int     = 5.781,
  g_slope   = 0.988,
  g_sd      = 20.55699,
  s_int     = -0.352,
  s_slope   = 0.122,
  s_slope_2 = -0.000213,
  r_r_int   = -11.46,
  r_r_slope = 0.0835,
  r_s_int   = 2.6204,
  r_s_slope = 0.01256,
  r_d_mu    = 5.6655,
  r_d_sd    = 2.0734,
  e_p       = 0.15,
  g_i       = 0.5067,
  sb_surv   = 0.2
)

# Lower bound, upper bound, and number of meshpoints.
L <- 1.02
U <- 624
n <- 500

# Initialize a population vector. The continuous state will have 500 meshpoints,
# and we'll pretend there's a seedbank.

init_pop_vec   <- runif(500)
init_seed_bank <- 20

my_general_ipm <- init_ipm(sim_gen = "general", di_dd = "di", det_stoch = "det") %>%
  define_kernel(
    name          = "P",
    formula       = s * g * d_ht,
    family        = "CC",
    g             = dnorm(ht_2, g_mu, g_sd),
    g_mu          = g_int + g_slope * ht_1,
    s             = plogis(s_int + s_slope * ht_1 + s_slope_2 * ht_1^2),
    data_list     = data_list,
    states        = list(c('ht')),
    uses_par_sets = FALSE,
    evict_cor     = TRUE,
    evict_fun     = truncated_distributions('norm',
                                            'g')
  ) %>%
  define_kernel(
    name          = "go_discrete",
    formula       = r_r * r_s * d_ht,
    family        = 'CD',
    r_r           = plogis(r_r_int + r_r_slope * ht_1),
    r_s           = exp(r_s_int + r_s_slope * ht_1),
    data_list     = data_list,
```

```r
    states         = list(c('ht', "b")),
    uses_par_sets = FALSE
  ) %>%
  define_kernel(
    name           = "stay_discrete",
    family         = "DD",
    formula        = sb_surv * (1 - g_i),
    data_list      = data_list,
    states         = list(c("b")),
    uses_par_sets = FALSE
  ) %>%
  define_kernel(
    name           = 'leave_discrete',
    formula        = e_p * g_i * r_d * d_ht,
    r_d            = dnorm(ht_2, r_d_mu, r_d_sd),
    family         = 'DC',
    data_list      = data_list,
    states         = list(c('ht', "b")),
    uses_par_sets = FALSE,
    evict_cor      = TRUE,
    evict_fun      = truncated_distributions('norm',
                                            'r_d')
  ) %>%
  define_impl(
    list(
      P              = list(int_rule    = "midpoint",
                            state_start = "ht",
                            state_end   = "ht"),
      go_discrete    = list(int_rule    = "midpoint",
                            state_start = "ht",
                            state_end   = "b"),
      leave_discrete = list(int_rule    = "midpoint",
                            state_start = "b",
                            state_end   = "ht"),
      stay_discrete  = list(int_rule    = "midpoint",
                            state_start = "b",
                            state_end   = "b")
    )
  ) %>%
  define_domains(
    ht = c(L, U, n)
  ) %>%
  define_pop_state(
    pop_vectors = list(
      n_ht = init_pop_vec,
      n_b  = init_seed_bank
    )
  )
```

Our next IPM will be a simple one:

```r
# Another hypothetical model. These parameters also do not correspond to any
# species.
```

```
my_data_list = list(s_int     = -2.2,
                    s_slope   = 0.25,
                    g_int     = 0.2,
                    g_slope   = 0.99,
                    sd_g      = 0.7,
                    r_r_int   = 0.003,
                    r_r_slope = 0.015,
                    r_s_int   = 0.45,
                    r_s_slope = 0.075,
                    mu_fd     = 2,
                    sd_fd     = 0.3)

my_simple_ipm <- init_ipm(sim_gen = "simple",
                          di_dd   = "di",
                          det_stoch = "det") %>%
  define_kernel(
    name      = "P_simple",
    family    = "CC",
    formula   = s * G,
    s         = plogis(s_int + s_slope * dbh_1),
    G         = dnorm(dbh_2, mu_g, sd_g),
    mu_g      = g_int + g_slope * dbh_1,
    data_list = my_data_list,
    states    = list(c('dbh')),
    evict_cor = TRUE,
    evict_fun = truncated_distributions(fun    = 'norm',
                                        target = 'G')
  ) %>%
  define_kernel(
    name      = 'F_simple',
    formula   = r_r * r_s * r_d,
    family    = 'CC',
    r_r       = plogis(r_r_int + r_r_slope * dbh_1),
    r_s       = exp(r_s_int + r_s_slope * dbh_1),
    r_d       = dnorm(dbh_2, mu_fd, sd_fd),
    data_list = my_data_list,
    states    = list(c('dbh')),
    evict_cor = TRUE,
    evict_fun = truncated_distributions(fun    = 'norm',
                                        target = 'r_d')
  ) %>%
  define_impl(
    make_impl_args_list(
      kernel_names = c("P_simple", "F_simple"),
      int_rule     = rep("midpoint", 2),
      state_start  = rep("dbh", 2),
      state_end    = rep("dbh", 2)
    )
  ) %>%
  define_domains(
    dbh = c(0,
            50,
            100
    )
  ) %>%
  define_pop_state(
    n_dbh = runif(100)
  )

my_ipm_list = list(ipm_1 = my_general_ipm, ipm_2 = my_simple_ipm)
```

Combining user-defined and PADRINO-defined IPMs

Next, we'll create a list of `proto_ipm` objects from PADRINO, and then put everything together. For simplicity, we'll select a small number of plant species. The `pdb` object is contained within the `RPadrino` package. It is not a complete version of PADRINO. We'll use the complete data set in the next section, accessed with `pdb_download()`.

```
data(pdb)

id_index <- c(
  paste0( "aaaa", c(34, 55)),
  paste0("aaa", c(310, 312, 339, 341, 353, 388))
)

small_db <- pdb_subset(pdb, id_index)
```

Next, we need to create a list that holds both the PADRINO IPMs and the ones we created above. After that, we can call `pdb_make_ipm()` on the combined data set, and voila! We have our database IPMs and our own homemade ones.

```
proto_list <- c(pdb_make_proto_ipm(small_db), my_ipm_list)
## 'ipm_id' aaa310 has the following notes that require your attention:
## aaa310: 'Geo and time info retrieved from COMPADRE (v.X.X.X.4)'
## 'ipm_id' aaa388 has the following notes that require your attention:
## aaa388: 'Same data as AAA388. State variable Height (Cm)'
```

Great! In that single step, we combined PADRINO IPMs with our own IPMs. Because these are all in the `proto_ipm` format, we don't need to think about technical differences between each type - we can use the exact same toolbox for analyzing both! Let's build the IPM objects and calculate deterministic per-capita growth rates!

```
ipm_list   <- pdb_make_ipm(proto_list)
lambdas    <- lambda(ipm_list)
```

We could now proceed with any further analyses just as we did in the previous case study. Since those types of analyses are already covered by the previous case study, we'll move on to combining PADRINO data with information from other databases.

Extending analyses with other databases

The possibilities for extending PADRINO with other databases are numerous, so we will only cover two here. We'll use range maps from BIEN and augment PADRINO with data from COMPADRE to examine if distance from the edge of a species' range influences the population's per-capita growth rate. This analysis won't be the most complete, and is intended to demonstrate the steps for combining data, not to make a scientific point. With that in mind, let's dive in!

Required packages

BIEN allows users to download range maps programmatically from their database using the BIEN R package. You can install that from CRAN using the chunk below. We'll also use `Rcompadre`, `mgcv`, `ggplot2`, `sf`, and `dplyr` to work with the data, so you'll need to install those as well.

```
install.packages(c("BIEN", "ggplot2", "sf", "dplyr", "Rcompadre", "mgcv"))
```

After that, we have to load them:

```
library(BIEN)
library(ggplot2)
library(sf)
library(dplyr)
library(Rcompadre)
library(RPadrino)
library(mgcv)
```

Data identification

BIEN allows us to programmatically query the database and retrieve all species names for which there is a range map. We'll load that, then load COMPADRE and PADRINO, and see how much overlap there is.

```
bien_rng_spps <- BIEN_ranges_list()
pdb           <- pdb_download(save = FALSE)
cdb           <- cdb_fetch("compadre")
```

```
## This is COMPADRE version 6.21.8.0 (release date Aug_20_2021)
## See user agreement at https://compadre-db.org/Help/UserAgreement
## See how to cite at https://compadre-db.org/Help/HowToCite
```

```
# Insert an underscore to make sure name format matches BIEN and PADRINO

cdb_spp <- gsub(" ", "_", cdb$SpeciesAccepted)

pdb_spp <- pdb$Metadata$species_accepted
```

Nice! We have 480 overlapping species between COMPADRE/PADRINO and BIEN's range maps.

```
all_spp <- unique(c(cdb_spp, pdb_spp))
pos_spp <- all_spp[all_spp %in% bien_rng_spps$species]

pdb_rng_spp <- unique(pdb_spp[pdb_spp %in% pos_spp])
cdb_rng_spp <- unique(cdb_spp[cdb_spp %in% pos_spp])
```

Subsetting

We probably shouldn't use all of these, as those calculations would take quite some time for a tutorial, so we'll select a subset. We'll take the species for which the demographic data are from North America. For PADRINO, we need to find their `ipm_id`s, and then pass those into `pdb_subset()`. For COMPADRE, we can just use `dplyr` verbs as if we were working with a `data.frame`.

```
pdb_ids     <- pdb$Metadata$ipm_id[pdb$Metadata$species_accepted %in% pdb_rng_spp &
                                   pdb$Metadata$continent == "n_america" &
                                   pdb$Metadata$treatment == "Unmanipulated"]
```

```
use_pdb     <- pdb_subset(pdb, pdb_ids)

cdb_rng_spp_f <- gsub("_", " ", cdb_rng_spp)

use_cdb <- filter(cdb,
                  SpeciesAccepted %in% cdb_rng_spp_f &
                  Continent == "N America" &
                  MatrixTreatment == "Unmanipulated")
```

Check data quality

PADRINO data is validated before it is uploaded to ensure the IPM behaves as the published version behaves. There are additional checks you might want to perform on your own, and those depend on the subsequent anaylsis. Case study 1 shows an example of this where a singular kernel created some very strange results. However, there aren't built-in functions in `RPadrino` yet to assist with this. Therefore, it is usually a good idea to check the original publications just to be sure there aren't caveats to the model that the authors have raised. We can find the citations using `pdb_citation()` and `pdb_report()`. `pdb_citation()` returns a character vector of citations in APA style, whereas `pdb_report()` generates an RMarkdown report based on the information in the database.

```
cites <- pdb_citations(use_pdb)

pdb_report(use_pdb)
```

We'll also want to check COMPADRE for some common data issues using the `cdb_flag()` function. This is documented much more thoroughly in the `Rcompadre` package website. For simplicity, we'll just use ones which do not raise any flags, as fixing issues with COMPADRE data is beyond the scope of this case study. Furthermore, we'll subset out the mean matrices, as we want to work with individual transitions.

```
cdb_f <- cdb_flag(use_cdb)

use_cdb <- filter(cdb_f, !check_NA_A & !check_NA_U & !check_NA_F & !check_NA_C&
                         !check_zero_U & !check_singular_U & check_component_sum &
                         check_ergodic & check_irreducible & check_primitive &
                         check_surv_gte_1 & MatrixComposite == "Individual")
```

Data transformation

Next, we need to do a bit of data wrangling. We only need the `ipm_id` and species names for plotting and analyzing, so we'll just grab those from the metadata table. We're going to create an `sf` object for this data using the coordinates stored in the `"lat"` and `"lon"` columns of the metadata. `sf` provides a standardized interface for dealing with multiple types of spatial data, and also plays nicely with `dplyr`, which makes managing data much easier. The `st_as_sf()` function handles the conversion for us.

```
temp_coords <- use_pdb$Metadata %>%
  select(ipm_id, species_accepted, lat, lon)

temp_db <- use_cdb@data %>%
  select(MatrixID, SpeciesAccepted, Lat, Lon) %>%
  setNames(names(temp_coords)) %>%
  rbind(temp_coords) %>%
  mutate(lat = lat,
         lon = lon) %>%
  .[complete.cases(.), ]


# Create an 'sf' object with the combined coordinates from COMPADRE and PADRINO

study_coords <- st_as_sf(temp_db,
                         coords = c("lon", "lat"),
                         crs = "WGS84") %>%
  arrange(species_accepted) %>%
  .[!duplicated(.$species_accepted), ]
```

Querying BIEN

Now that we have our final species list, we're going to download the range maps for each species using the `BIEN_ranges_load_species()` function, and then convert that into an `sf` object which will make subsequent analysis and plotting easier.

After converting the range maps to an `sf` object, we also need to create a different version of the polygons that are a set of lines representing the edges. This will allow us to quickly calculate the distance between our study points and the edge of the range. `st_cast()` handles this conversion for us.

> NB: while the data frame of species names we downloaded from BIEN before uses an "_" in species names, the `BIEN_ranges_load_species()` function expects names without underscores!

```
study_coords$species_accepted <- gsub("_", " ", study_coords$species_accepted)

rng_maps <- BIEN_ranges_load_species(study_coords$species_accepted) %>%
  st_as_sf()

line_maps     <- st_cast(rng_maps, "MULTILINESTRING")

# Put the "_"'s back in so that we can match all names later on.
study_coords$species_accepted <- gsub(" ", "_", study_coords$species_accepted)
```

Compute distance from edges

Ok, we're finally ready to compute the distance from each study site to the range edge. We're going to use the `st_distance()` function for this. This finds the minimum distance between the first and second arguments and computes a matrix for all possible combinations. It will ignore the fact that sometimes the closest edge is an ocean (which our species cannot grow in). However, working out how to improve that calculation is a problem for another day!

We start by extracting a distance matrix and taking the diagonal. The diagonal represents the shortest distance between our species study site and the edge of the polygon of its range map (NB: This only works because we sorted each object alphabetically ahead of time!). Next, we add in the species name information and set the data frame's names to something useful. Finally, we'll convert the distances to kilometers.

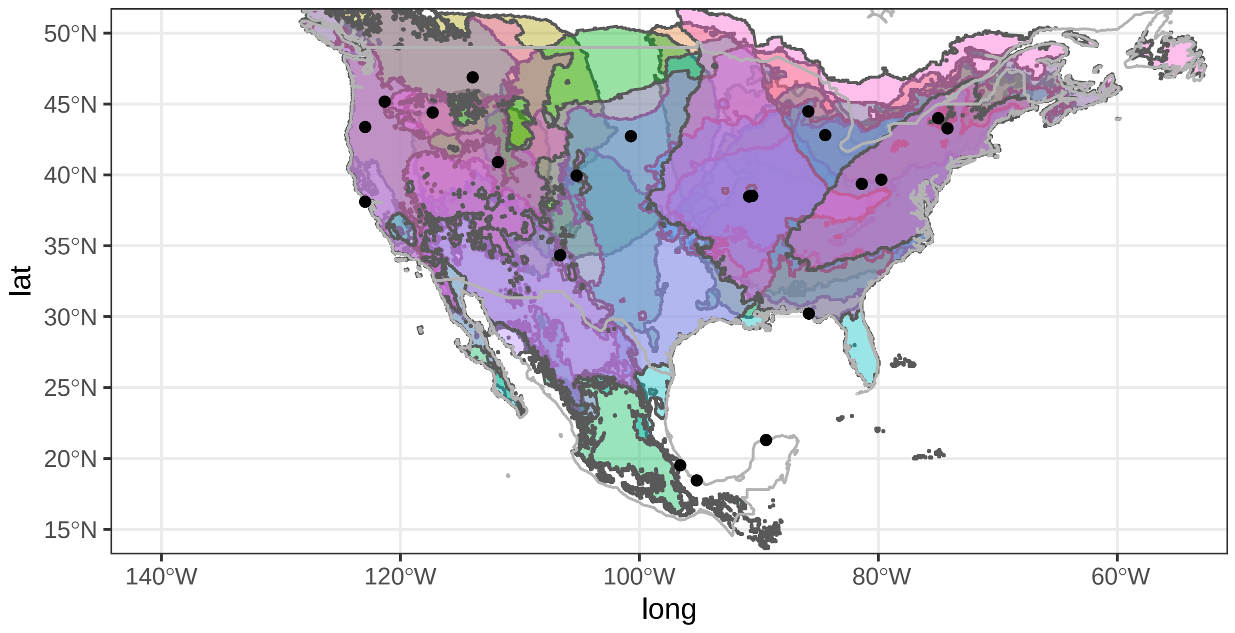
```
# Quickly check to make sure all of our species line up positionally.
# If not, we'd need to make sure they do, otherwise it will be difficult
# to extract the distances from the distance matrix we are about to compute!

stopifnot(all(line_maps$species == study_coords$species_accepted))
```

```r
dist_from_edge <- st_distance(study_coords, line_maps) %>%
  diag() %>%
  data.frame() %>%
  cbind(study_coords$species_accepted, .) %>%
  setNames(c("species", "distance_in_meters"))

dist_from_edge$distance_in_km <-
  round(as.numeric(dist_from_edge$distance_in_meters) / 1e3, 2)
```

Visualize our dataset

We'll plot our range maps with the study sites overlaid on them using `ggplot2`. `ggplot2` has built in `geoms` designed to handle `sf` objects, which will make our lives much easier!

```r
world     <- map_data("world")
n_america <- filter(world, region %in% c("USA", "Canada", "Mexico"))

ggplot(rng_maps) +
  geom_sf(aes(fill = species), alpha = 0.4) +
  geom_polygon(data = n_america, aes(x = long, y = lat, group = group),
               inherit.aes = FALSE,
               color = "grey70",
               fill = NA) +
  geom_sf(data  = study_coords) +
  coord_sf(xlim = c(-140, -55),
           ylim = c(15, 50)) +
  theme_bw() +
  theme(
    legend.position = "none"
  )
```



Already, we can see that our range maps don't perfectly align with the PADRINO population coordinates. We can check and see which study populations are actually contained by their range map like so:

```
covered_ind <- st_covered_by(study_coords, rng_maps, sparse = FALSE) %>%
  diag()
```

```
## although coordinates are longitude/latitude, st_covered_by assumes that they are planar
```

```
# Print studies not covered by BIEN range map
study_coords[!covered_ind , ]
```

```
## Simple feature collection with 6 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -122.9567 ymin: 18.45 xmax: -85.82 ymax: 45.16667
## Geodetic CRS:  WGS 84
## # A tibble: 6 x 3
##   ipm_id species_accepted                      geometry
##   <chr>  <chr>                              <POINT [°]>
## 1 241875 Astragalus_tyghensis   (-121.3167 45.16667)
## 2 242052 Calathea_ovandensis           (-95.2 18.45)
## 3 239708 Euphorbia_telephioides    (-85.82 30.21972)
## 4 244332 Lupinus_tidestromii    (-122.9567 38.10861)
## 5 247007 Mammillaria_gaumeri            (-89.4 21.3)
## 6 241182 Tillandsia_deppeana    (-96.58333 19.51667)
```

These are COMPADRE matrices. Rather than try to figure out what's going on, we'll just drop those out of our analysis.

```r
study_coords <- study_coords[covered_ind, ]
```

Compute lambdas for each type of model

Great! The next step is to generate and then join our lambda values with the distance information. This is a two-step process. First, we'll build our PADRINO IPMs:

```r
# Extract PADRINO IDs - we don't want to give COMPADRE ones to PADRINO machinery!

pdb_ids <- study_coords$ipm_id[study_coords$ipm_id %in% pdb$Metadata$ipm_id]

proto_list <- pdb_make_proto_ipm(
  use_pdb,
  pdb_ids
)
```

```
## 'ipm_id' aaa310 has the following notes that require your attention:
## aaa310: 'Geo and time info retrieved from COMPADRE (v.X.X.X.4)'

## 'ipm_id' aaa329 has the following notes that require your attention:
## aaa329: 'Based on IPM from Rose Ecology 2005; The GPS coordinates were approximated
## to the closest geographic location described in the reference'

## 'ipm_id' aaa385 has the following notes that require your attention:
## aaa385: 'Same data as AAA385. State variable Height (Cm)'
```

```r
ipm_list <- pdb_make_ipm(proto_list)

# We're going to use the mean value of lambda for IPMs with many values.
# We neeed to convert those from a list to a data.frame for modeling, and need
# to keep track of which lambda belongs to which ID. The loop below will correctly
# format this.

lambdas <- lambda(ipm_list, type_lambda = "last")

temp <- data.frame(ipm_id = NA,
                   lambda = NA)

for(i in seq_along(lambdas)) {

  temp_2 <- data.frame(ipm_id = names(lambdas)[i],
                       lambda = lambdas[[i]])

  temp <- rbind(temp, temp_2)

}

temp <- temp[-1, ]
```

Next, we'll get our COMPADRE lambdas, and stick them back in with the PADRINO lambdas.

```r
use_cdb <- filter(use_cdb, MatrixID %in% study_coords$ipm_id)

matAs   <- lapply(use_cdb$mat, function(x) x@matA)
```

```r
use_cdb@data$lambda <- vapply(matAs,
                              function(x) Re(eigen(x)$values[1]),
                              numeric(1L))

cdb_lambda <- use_cdb@data %>%
  select(MatrixID, lambda) %>%
  setNames(c("ipm_id", "lambda"))

all_lambdas <- rbind(temp, cdb_lambda)
```

Prepare lambdas for analysis

Finally, we need to join lambda values with coordinate data set to recover the species names, and then use those to join with the distance from edge object. Once that's done, we can plot everything!

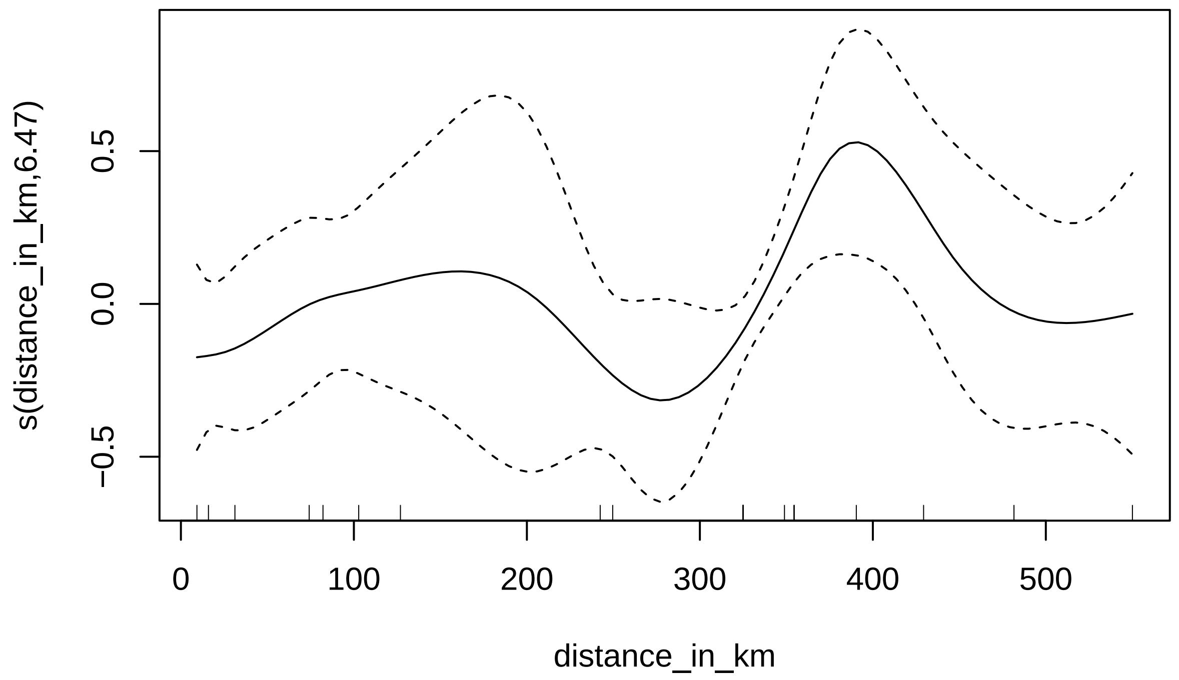
```r
all_lambdas <- left_join(all_lambdas, study_coords, by = "ipm_id") %>%
  select(-geometry)

all_data <- left_join(all_lambdas, dist_from_edge,
                      by = c("species_accepted" = "species"))
```

We're ready to plot and analyze the data. GAMs (Wood 2011) are a great way to spot general trends in data, so we'll use those.

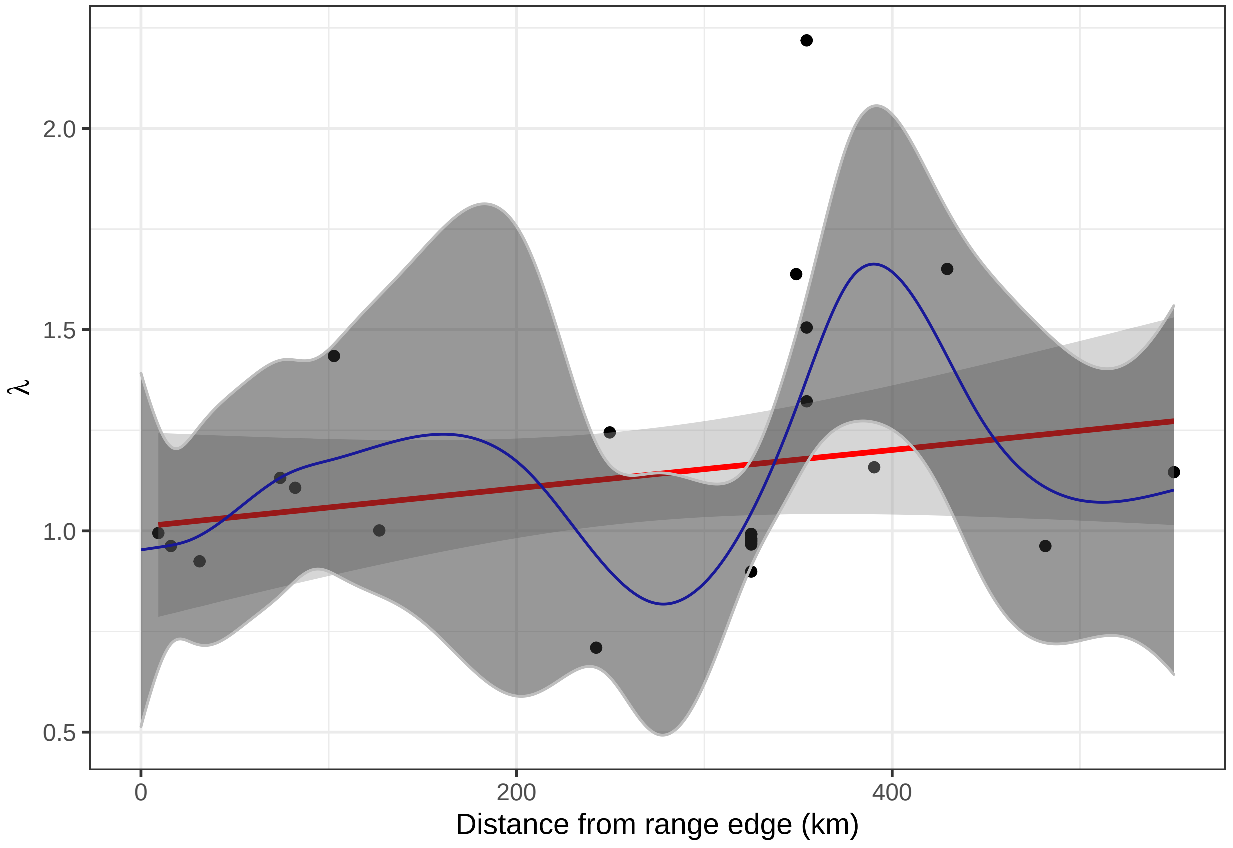
```r
lambda_by_dist <- gam(lambda ~ s(distance_in_km, bs = "cs"),
                      data = all_data,
                      family = Gamma(link = "identity"))

summary(lambda_by_dist)
```

```
##
## Family: Gamma
## Link function: identity
##
## Formula:
## lambda ~ s(distance_in_km, bs = "cs")
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.13374    0.04768   23.78 6.94e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                     edf Ref.df     F p-value
## s(distance_in_km) 6.472      9 1.383   0.115
##
## R-sq.(adj) =  0.215   Deviance explained =   49%
## GCV = 0.058181  Scale est. = 0.047189  n = 27
```

```r
plot(lambda_by_dist)
```



```
preds <- cbind(data.frame(predict(lambda_by_dist,
                                  data.frame(distance_in_km = seq(0, 550, 1)),
                                  type = "response",
                                  se.fit = TRUE)),
               x = seq(0, 550, 1)) %>%
  mutate(upper = fit + se.fit * 1.96,
         lower = fit - se.fit * 1.96)



ggplot(all_data, aes(x = distance_in_km, y = lambda)) +
  geom_point() +
  geom_smooth(method = "glm",
              formula = y ~ x,
              method.args = list(family = Gamma("identity")),
              color = "red") +

  geom_line(data = preds,
            aes(x = x, y = fit),
            inherit.aes = FALSE,
            color = "blue") +
  geom_ribbon(data = preds,
              aes(x = x, ymin = lower, ymax = upper),
              color = "grey",
              alpha = 0.5,
              inherit.aes = FALSE) +
  theme_bw() +
  xlab("Distance from range edge (km)") +
  ylab(expression(lambda))
```



There is a positive trend in range centrality and species performance (red line), and the GAM is likely overfit. There is a lot of residual variance, and we can certainly find better ways to model this phenomenon, but this is a good start for an exploratory analysis. We will leave the further analyses as an exercise to you!

Recap

We have shown how to combine PADRINO data with user-defined IPMs as well as join it with information from other databases. This is hardly a comprehensive overview of PADRINO's applications - there are many other uses and databases one could combine PADRINO with. It is our hope that this and the previous case study provide a general guide to the considerations and steps one needs to take when using this data!

Citations

1. Wood, S.N. (2011) Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. Journal of the Royal Statistical Society (B) 73(1):3-36
2. Maitner, B., Boyle B., Casler N., Condit R., Donoghue J., Duran S.M., *et al.* (2017) The bien r package: A tool to access the Botanical Information and Ecology Network (BIEN) database. Methods in Ecology and Evolution 9(2): 373-379. https://doi.org/10.1111/2041-210X.12861