GF-iSSA Code

Rafael Arce Guillen

2023-07-12

Loading the needed libraries and data

First we load all the libraries needed for this script.

```
# Loading libraries
library(dplyr)
library(tidyverse)
library(lubridate)
library(raster)
library(rgdal)
library(sf)
library(rgeos)
library(sp)
library(INLA) #Version 23.06.29
library(deldir)
library(maptools)
library(spatstat)
library(inlabru) #Version 2.8.0.9009
library(ggthemes)
```

The data consist of 25 animal tracks and a landscape. Here we will show how to fit our model to a single animal track. These data were simulated with a GF with spatial range ϕ_s and standard deviation σ equal to 40 and 2 respectively. In order to obtain all simulated scenarios from the main manuscript, please use the simulation code (see DATA AVAILABILITY section).

```
# Setting the working directory
setwd("C:/Users/Rafael Arce Guillen/Desktop/GF-iSSA submission/Major revision")

# Loading data (observed locations (my_data) and landscape for each scenario).
# This can be find in the supplemental material.
load("Simulation/First scenario/tracks/paper_tracks40_4.RData")

# Specifying coordinate reference system
coor_rs <- landscape@crs

# Selecting the corresponding track (for i = 1,...,25))
i <- 15

df1 <- my_data[[i]]
```

Data preparation

We define a modified function from the "MoveHMM" package in order to calculate the turning angles:

```
# Modified function from MoveHMM package (in order to calculate the turning angles)
turnAngle <- function(loc1, loc2, loc_matrix, LLangle) {
  # NA angle if zero step length
  if (all(loc1 == loc2) | all(loc2 == loc_matrix)) {
    return(NA)
  }

  if (LLangle) {
    angle <- (bearing(loc1, loc2) - bearing(loc2, loc_matrix)) / 180 * pi
  } else {
    v <- c(loc2[1] - loc1[1], loc2[2] - loc1[2])
    w <- matrix(
      c(
        loc_matrix[, 1] - loc2[1],
        loc_matrix[, 2] - loc2[2]
      ),
      ncol = 2, byrow = FALSE
    )
    angle <- atan2(w[, 2], w[, 1]) - atan2(v[2], v[1])
  }

  for (i in 1:length(angle)) {
    while (angle[i] <= -pi) {
      angle[i] <- angle[i] + 2 * pi
    }
    while (angle[i] >= pi) {
      angle[i] <- angle[i] - 2 * pi
    }
  }

  return(angle)
}
```

First, we need an "sp" object (observed_sp) of the observed locations containing the corresponding step lengths and turning angles. In addition, we need the domains of availability (samplers_sp), which joined form the effective study area (boundary_sp).

```
# Calculating the observed step lengths
xyset <- cbind(df1$x, df1$y)

step_lengths <- sqrt(rowSums((xyset[-1, ] - xyset[-nrow(xyset), ])^2))

# Replacing zero step lengths by a very small value
step_lengths <- ifelse(step_lengths == 0, 1e-6, step_lengths)

# Specifying the radius of the domains of availability
max_sl <- 1.25 * max(step_lengths)

# Defining observed locations
```

```r
observed <- cbind(df1$x, df1$y)[3:nrow(df1), ]

# Defining a list of the time points
time_list <- as.list(3:nrow(df1))

# Creating an sp object of the observed locations including the SL and TA
observed_sp <- SpatialPointsDataFrame(
  coords = observed,
  data = data.frame(
    case = 1,
    time = 1:nrow(observed),
    sl = step_lengths[2:length(step_lengths)],
    log_sl = log(step_lengths[2:length(step_lengths)]),
    cos_ta = cos(sapply(time_list, function(t) {
      result <- turnAngle(cbind(df1$x[t - 2], df1$y[t - 2]),
        cbind(df1$x[t - 1], df1$y[t - 1]),
        cbind(df1$x[t], df1$y[t]),
        LLangle = F
      )
      return(result)
    }))
  ),
  proj4string = coor_rs
)

# Defining samplers (domains of availability for each time point)
samplers <- st_buffer(st_as_sf(observed_sp), max_sl)

samplers_sp <- as_Spatial(samplers)

# Creating the effective study area
boundary <- st_union(samplers)

boundary_sp <- as_Spatial(boundary)
```

INLA SPDE: GF-iSSA

We then need to place a mesh over the effective study area. A finer mesh results in a better approximation of the GF but comes with a higher computational cost. Once the mesh is defined, we can create an object of the integration points (ips) containing the corresponding integration points (corresponding nodes of the mesh) at each time point, their weights, the step lengths and the turning angles. Given this object is computationally expensive to compute, We recommend, once the mesh resolution is defined, to store this object.

```r
# Defining the inner mesh
mesh_inner <- inla.mesh.2d(
  boundary = boundary_sp,
  max.edge = 1,
  cutoff = 1,
  crs = coor_rs
)

# Defining the mesh outer extension
```

```r
boundary_outer <- inla.nonconvex.hull(mesh_inner$loc,
  convex = max(step_lengths) * 1.25
)

# Defining the final mesh (with outer extension)
mesh <- inla.mesh.2d(
  boundary = list(boundary_sp, boundary_outer),
  max.edge = c(1, 5),
  crs = coor_rs
)


# Defining integration points, weights, TA and SL
compute_ips <- function() {
  integration_mesh <- mesh
  int_points <- ipoints(
    samplers = samplers_sp[unlist(time_list) - 2, ],
    domain = integration_mesh,
    group = "time"
  )

  calc_info <- function(t) {
    time_coords <- int_points[int_points$time == t - 2, ]@coords[, 1:2]

    x_t_1 <- cbind(df1$x, df1$y)[t - 1, ]
    x_t_2 <- cbind(df1$x, df1$y)[t - 2, ]

    my_sl <- pointDistance(
      x_t_1,
      time_coords,
      lonlat = FALSE
    )

    data.frame(
      sl = my_sl,
      log_sl = log(my_sl),
      cos_ta = cos(turnAngle(as.numeric(x_t_2), as.numeric(x_t_1), time_coords,
        LLangle = FALSE
      ))
    )
  }

  result <-
    cbind(
      int_points,
      do.call("rbind", lapply(time_list, calc_info))
    )
  return(result)
}

ips <- compute_ips()
```

After this, we need to specify the penalized complexity priors for the hyperparameters of the GF. Once this

is done, we only need to prepare the spatial layers for analysis before fitting the model. For this, the raster layers need to be converted into a "SpatialPixelsDataFrame" object.

```
# Creating SPDE Model including the prior specification for the hyperparameters of the GF
range_true <- 40
sd_true <- sqrt(4)

gf_model <- inla.spde2.pcmatern(
  mesh = mesh,
  alpha = 2,
  prior.range = c(range_true / 4, 0.05),
  prior.sigma = c(4 * sd_true, 0.05)
)

# Preparing spatial covariates for the analysis

# Transforming raster layers to SpatialPixelsDataFrame object
gcov <- list(
  x1 = as(landscape$x1, "SpatialPixelsDataFrame"),
  x2 = as(landscape$x2, "SpatialPixelsDataFrame"),
  cen = as(landscape$cen, "SpatialPixelsDataFrame")
)

x1 <- gcov$x1
x2 <- gcov$x2
cen <- gcov$cen
```

Model Fitting

We can now fit the model with the bru() function from the "inlabru" package. For added speed, the number of threads used can be increased according to the users resources.

```
# Defining the model components of the GF-iSSA
comp <- coordinates ~
  -1 + x1(x1, model = "linear") +
  x2(x2, model = "linear") +
  cen(cen, model = "linear") +
  sl(sl, model = "linear") +
  log_sl(log_sl, model = "linear") +
  sl_radial_term(-log(sl), model = "const") +
  cos_ta(cos_ta, model = "linear") +
  mySmooth(coordinates, model = gf_model) +
  random_intercept(time,
    model = "iid",
    hyper = list(theta = list(initial = log(1e-6), fixed = T))
  )

# Defining the integration scheme
lik <- like(coordinates ~ .,
  family = "cp",
  data = observed_sp,
  ips = ips,
```

```
  E = 1e6 # Convert the integration units to m^2 instead of km^2
)

# Fitting the GF-iSSA model
GF_iSSA <- bru(comp,
  lik,
  options = list(
    num.threads = 7,
    verbose = TRUE,
    bru_verbose = 3,
    control.predictor = list(compute = FALSE),
    control.compute = list(
      hyperpar = TRUE,
      return.marginals = FALSE,
      mlik = FALSE
    ),
    control.inla = list(int.strategy = "eb"),
    safe = TRUE
  )
)
```

```
##
##  *** inla.core.safe:  rerun to try to solve negative eigenvalue(s) in the Hessian
```

```
# Specifying the linear predictor for the NHPP
comp_nhpp <- coordinates ~
  -1 + x1(x1, model = "linear") +
  x2(x2, model = "linear") +
  cen(cen, model = "linear") +
  sl(sl, model = "linear") +
  log_sl(log_sl, model = "linear") +
  sl_radial_term(-log(sl), model = "const") +
  cos_ta(cos_ta, model = "linear") +
  random_intercept(time,
    model = "iid",
    hyper = list(theta = list(initial = log(1e-6), fixed = T))
  )

# Fitting the NHPP model
NHPP <- bru(comp_nhpp,
  lik,
  options = list(
    num.threads = 7,
    verbose = TRUE,
    bru_verbose = 3,
    control.predictor = list(compute = FALSE),
    control.compute = list(
      hyperpar = TRUE,
      return.marginals = FALSE,
      mlik = FALSE
    ),
    control.inla = list(int.strategy = "eb"),
    safe = TRUE
```

```
  )
)
```

Summary of the fixed effects and hyperparameters

```
# Summary of the fixed effects
GF_iSSA$summary.fixed
```

```
##               mean         sd 0.025quant    0.5quant  0.975quant        mode
## x1      1.62680744 0.12240259  1.3869028  1.62680744  1.86671212  1.62680744
## x2      0.99358571 0.12150821  0.7554340  0.99358571  1.23173742  0.99358571
## cen    -0.03677265 0.05675964 -0.1480195 -0.03677265  0.07447421 -0.03677265
## sl     -1.98114108 0.09301701 -2.1634511 -1.98114108 -1.79883109 -1.98114108
## log_sl  3.03743566 0.16876434  2.7066636  3.03743566  3.36820769  3.03743566
## cos_ta  0.94999553 0.05283104  0.8464486  0.94999553  1.05354246  0.94999553
##        kld
## x1       0
## x2       0
## cen      0
## sl       0
## log_sl   0
## cos_ta   0
```

```
NHPP$summary.fixed
```

```
##               mean         sd  0.025quant    0.5quant  0.975quant        mode
## x1      1.51227567 0.10419948  1.30804844  1.51227567  1.71650289  1.51227567
## x2      0.96247691 0.10985560  0.74716388  0.96247691  1.17778994  0.96247691
## cen    -0.01926142 0.02239437 -0.06315358 -0.01926142  0.02463075 -0.01926142
## sl     -2.03650097 0.09285201 -2.21848755 -2.03650097 -1.85451438 -2.03650097
## log_sl  3.02501566 0.16871690  2.69433661  3.02501566  3.35569470  3.02501566
## cos_ta  0.87826459 0.05150626  0.77731418  0.87826459  0.97921500  0.87826459
##        kld
## x1       0
## x2       0
## cen      0
## sl       0
## log_sl   0
## cos_ta   0
```

```
# Summary of the hyperparameters
GF_iSSA$summary.hyperpar
```

```
##                          mean         sd 0.025quant  0.5quant 0.975quant
## Range for mySmooth 47.308816 16.8892111   25.28399 44.710846  91.837356
## Stdev for mySmooth  2.120008  0.8600594    1.04266  1.975215   4.418832
##                          mode
## Range for mySmooth 36.798560
## Stdev for mySmooth  1.575506
```

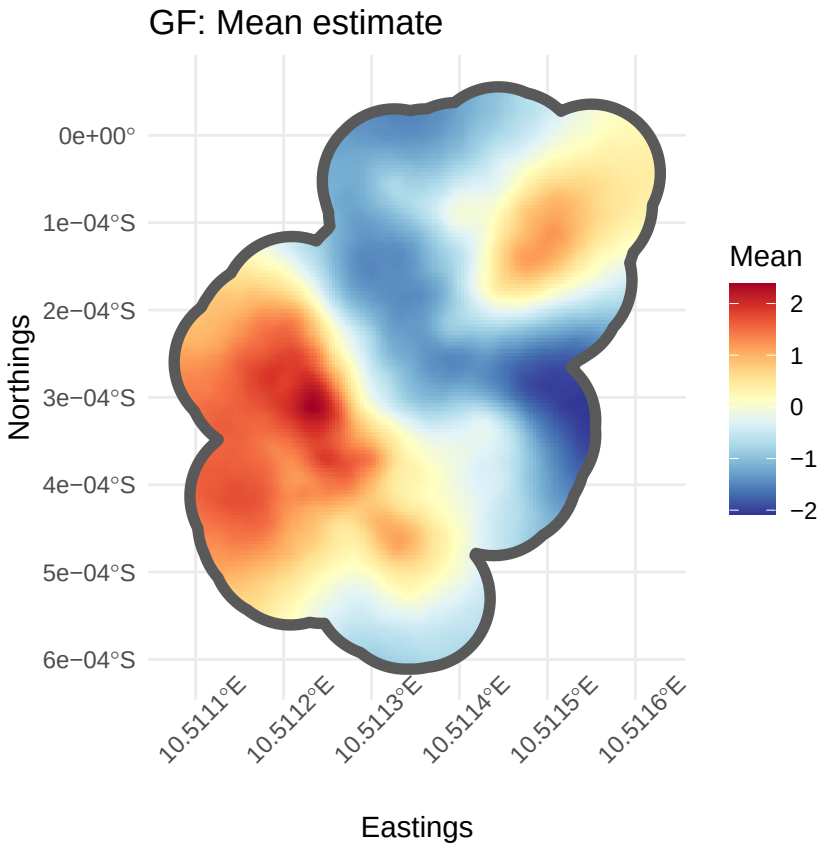
Plotting the estimated GF

```
# Defining palette for plotting
pal <- rev(RColorBrewer::brewer.pal(11, "RdYlBu"))

# Defining the lattice object where the GF will be displayed
df_pixels <- pixels(mesh, mask = boundary_sp, nx = 200, ny = 200)

# Predicting the GF on the lattice object
gf <- predict(GF_iSSA, df_pixels, ~ list(
  field = mySmooth
))

# Plotting the estimated Gaussian Field
ggplot() +
  gg(gf$field, aes(fill = mean), size = 5) +
  geom_sf(data = boundary, alpha = 0.0001, lwd = 2) +
  scale_fill_gradientn(colours = pal) +
  labs(fill = "Mean") +
  theme_minimal() +
  xlab("Eastings") +
  ylab("Northings") +
  ggtitle("GF: Mean estimate") +
  theme(axis.text.x = element_text(angle = 45))
```



Predictive scores: SE and DS

For this minimal example, we sample 20 realizations from the posterior distribution. However, for more accurate results, this number can be increased.

```
# Creating the data frame based on the centroids of the prediction grid
landscape_df <- as.data.frame(extract(landscape, boundary_sp, cellnumbers = T))

# Creating coordinate columns
landscape_df.coords <- cbind(
  landscape_df,
  xyFromCell(landscape, landscape_df[, 1])
)

landscape_df.coords$rsf <- exp(1.5 * landscape_df.coords$x1 +
  landscape_df.coords$x2 -
  0.04 * landscape_df.coords$cen +
  landscape_df.coords$x3)

landscape_df.coords$rsf <- landscape_df.coords$rsf / mean(landscape_df.coords$rsf)

# Transforming it to a SpatialPointsDataFrame
newdata <- SpatialPointsDataFrame(
  coords = cbind(
    landscape_df.coords$x,
    landscape_df.coords$y
  ),
  data = data.frame(rsf = landscape_df.coords$rsf),
  proj4string = landscape@crs
)

# Calculating the estimated normalized RSF of both models
pred_GF_iSSA <- predict(GF_iSSA, newdata,
  formula = ~ exp(x1 + x2 + cen + mySmooth) /
    mean(exp(x1 + x2 + cen + mySmooth)), n.samples = 20
)

pred_NHPP <- predict(NHPP, newdata, formula = ~ exp(x1 + x2 + cen) /
  mean(exp(x1 + x2 + cen)), n.samples = 20)

# True normalized RSF
truth <- newdata$rsf

# Posterior mean
post_E_GF_iSSA <- pred_GF_iSSA$mean
post_E_NHPP <- pred_NHPP$mean

# Posterior variance
post_V_GF_iSSA <- pred_GF_iSSA$sd^2
post_V_NHPP <- pred_NHPP$sd^2

# Calculating predictive scores
SE_score_GF_iSSA <- mean((truth - post_E_GF_iSSA)^2)
DS_score_GF_iSSA <- mean((truth - post_E_GF_iSSA)^2 /
```

```
  post_V_GF_iSSA + log(post_V_GF_iSSA))

SE_score_NHPP <- mean((truth - post_E_NHPP)^2)
DS_score_NHPP <- mean((truth - post_E_NHPP)^2 /
  post_V_NHPP + log(post_V_NHPP))

# Summarizing scores
print(mean(SE_score_GF_iSSA))
```

```
## [1] 1.777048
```

```
print(mean(SE_score_NHPP))
```

```
## [1] 7.224715
```

```
print(mean(DS_score_GF_iSSA))
```

```
## [1] -0.7545236
```

```
print(mean(DS_score_NHPP))
```

```
## [1] 563.1224
```

DIC score: Suitable for real data

The scores calculated above show the better predictive quality of our model. However, they are not suitable for real data. Therefore, we calculate the DIC score based on the original spatial likelihood from Forester et. al (2009). This is numerically expensive and therefore we restrict its calculation to only the first 10 time points. However, this gives an idea on how the DIC score could be calculated for real data.

```
# Defining function to calculate the DIC
DIC_fun <- function(fit, ips, observed, n, formula_obs, formula_ips, seed = 123) {
  time_list <- as.list(observed$time)
  # Reducing the number of time points (As a proof of concept)
  time_list <- time_list[1:10]

  E_post <- sum(sapply(time_list, function(t) {
    set.seed(seed)
    log_lambda_t <- predict(
      fit,
      newdata = observed[observed$time == t, ],
      formula_obs,
      n.samples = n
    )

    set.seed(seed)
    log_Lambda_t <- predict(
      fit,
      newdata = ips[ips$time == t, ],
```

```
      formula_ips,
      n.samples = n
    )

    log_lik_t <- log_lambda_t$mean - log_Lambda_t$mean

    return(log_lik_t)
  }))

  # Evaluating the posterior mean for each component, and extracting into
  # a list (of length one) of states:
  comp_pred <- predict(fit)
  state <- list(lapply(comp_pred, function(x) x$mean))

  # Finding out which variables are used in the formula:
  used <-
    bru_used(formula_obs,
      labels = names(fit$bru_info$model$effects)
    )

  log_py <- sum(sapply(time_list, function(t) {
    log_lambda_t <- evaluate_model(
      model = fit$bru_info$model,
      state = state,
      data = observed[observed$time == t, ],
      predictor = formula_obs,
      used = used
    )

    log_Lambda_t <- evaluate_model(
      model = fit$bru_info$model,
      state = state,
      data = ips[ips$time == t, ],
      predictor = formula_ips,
      used = used
    )

    log_lik_t <- log_lambda_t - log_Lambda_t

    return(log_lik_t)
  }))


  DIC <- log_py - 2 * (log_py - E_post)
  return(DIC)
}

# Calculating the DIC scores for the GF-iSSA and the NHPP sampling 1000
# realizations from the posterior distribution (uncomment to run)
# DIC_GF_iSSA = DIC_fun(fit = GF_iSSA, ips = ips, observed = observed_sp, n = 1000,
#               formula_obs = ~ x1 + x2 + cen + sl + log_sl + cos_ta + mySmooth,
#               formula_ips = ~ log(sum(weight*exp(x1 + x2 + cen + sl + log_sl +
#                                                  cos_ta + mySmooth))))
```

```
#
# DIC_NHPP = DIC_fun(fit = GF_iSSA, ips = ips, observed = observed_sp, n = 1000,
#                        formula_obs = ~ x1 + x2 + cen + sl + log_sl + cos_ta,
#                        formula_ips = ~ log(sum(weight*exp(x1 + x2 + cen + sl +
#                                                           log_sl + cos_ta ))))
#
# print(DIC_GF_iSSA)
# print(DIC_NHPP)
```