Explanation of LGCP INLA code

Synne

2025-06-20

Required libraries:

```
library(geometry)
library(Matrix)
library(INLA)
library(FNN)
```

`jitter_points`

The `jitter_points` function adds a small amount of random noise to each coordinate in a set of points to avoid numerical issues such as degenerate simplices in mesh generation.

```
jitter_points <- function(pts, eps = 1e-4) {
  pts + matrix(runif(length(pts), -eps, eps), ncol = ncol(pts))
}
```

`build_mesh`

The `build_mesh` function has as purpose to construct a Delaunay triangulation (mesh) over a d-dimensional domain by discretizing it into points and connecting them into simplices.

```
build_mesh <- function(d, m, bounds) {
  grid_axes <- lapply(bounds, function(lim) seq(lim[1], lim[2], length.out = m))
  grid_points <- expand.grid(grid_axes)
  pts <- jitter_points(as.matrix(grid_points))
  simplices <- delaunayn(pts, options = "QJ")
  list(points = pts, simplices = simplices)
}
```

Parameters:

- `d`: number of dimensions
- `m`: number of equally spaced grid points per dimension
- `bounds`: A list of length `d`, where each element is a vector of the form `c(min, max)` defining the range for that dimension

```
grid_axes <- lapply(bounds, function(lim) seq(lim[1], lim[2], length.out = m))
```

Step-by-step explanation of `build_mesh`: Creates a sequence of `m` evenly spaced points between the lower and upper bounds in each dimension.

```
grid_points <- expand.grid(grid_axes)
```

Creates a full grid of coordinates.

```
pts <- jitter_points(as.matrix(grid_points))
```

Adds a small random jitter.

```
 simplices <- delaunayn(pts, options = "QJ")
```

Creates a Delaunay triangulation of the jittered points.

`compute_fem_matrices()`

`compute_fem_matrices()` assembles the global matrices `C` and `G` using the finite element method over a mesh of simplices in arbitrary dimension `d`.

```
compute_fem_matrices <- function(points, simplices) {
  n <- nrow(points)
  d <- ncol(points)
  C <- Matrix(0, n, n, sparse = TRUE)
  G <- Matrix(0, n, n, sparse = TRUE)
  for (i in 1:nrow(simplices)) {
    idx <- simplices[i, ]
    verts <- points[idx, , drop = FALSE]
    T <- t(verts[-1, , drop = FALSE] - matrix(verts[1, ], d, d, byrow = TRUE))
    detT <- det(T)
    if (abs(detT) < 1e-12) next
    vol <- abs(detT) / factorial(d)
    Ghat <- cbind(-1, diag(d))
    grads <- solve(t(T), Ghat)
    GK <- vol * t(grads) %*% grads
    MK <- matrix(1, d+1, d+1); diag(MK) <- 2
    MK <- MK * vol / ((d + 1) * (d + 2))
    C[idx, idx] <- C[idx, idx] + MK
    G[idx, idx] <- G[idx, idx] + GK
  }
  list(C = C, G = G)
}
```

Parameters:

- `points`: A `n x d` matrix of mesh point coordinates
- `simplices`: A matrix where each row contains the indices of the `d+1`vertices forming a simplex

```
  n <- nrow(points)
  d <- ncol(points)
  C <- Matrix(0, n, n, sparse = TRUE)
  G <- Matrix(0, n, n, sparse = TRUE)
```

Step-by- step explanation: Initializes empty `n x n` sparse matrices for `C`and `G`.

```
for (i in 1:nrow(simplices)) {
    idx <- simplices[i, ]
```

Loop over all simplices. `idx` gives the indices of the `d+1` vertices for simplex `i`.

```
verts <- points[idx, , drop = FALSE]
```

Retrieves the actual coordinates of the simplex vertices.

```
T <- t(verts[-1, , drop = FALSE] - matrix(verts[1, ], d, d, byrow = TRUE))
```

Constructs the Jacobian matrix of the affine transformation from the reference simplex to this one. `verts[-1, ] - verts[1, ]` gives edge vectors from the first vertex to the others.

```
detT <- det(T)
if (abs(detT) < 1e-12) next
```

If the determinant is near zero, the simplex is numerically degenerate (almost flat) and is skipped.

```
vol <- abs(detT) / factorial(d)
```

The volume of a d-simplex in $\mathbb{R}^d$.

```
Ghat <- cbind(-1, diag(d))
grads <- solve(t(T), Ghat)
```

`Ghat` contains gradients of the barycentric coordinates on the reference simplex. Solving `t(T)` maps these to gradients in physical space.

```
GK <- vol * t(grads) %*% grads
```

This computes the local `G` matrix, the $\nabla\phi_i \cdot \nabla\phi_j$ integrals.

```
MK <- matrix(1, d+1, d+1); diag(MK) <- 2
MK <- MK * vol / ((d + 1) * (d + 2))
```

Constructs the local `M` matrix using closed-form formula for linear basis funcrions over a simplex.

```
C[idx, idx] <- C[idx, idx] + MK
G[idx, idx] <- G[idx, idx] + GK
```

Adds each local matrix into the global matrices at the corresponding vertex indices.

`assemble_precision_matrix()`

Constructs the sparse precision matrix Q of the SPDE-GMRF model

```
assemble_precision_matrix <- function(C, G, tau = 0.1, kappa = 5, jitter = 1e-4) {
  Q <- tau^2 * (kappa^2 * C + G)
  Q <- forceSymmetric(Q)
  diag_vals <- diag(Q)
  diag(Q)[diag_vals < jitter] <- jitter
  return(Q)
}
```

Parameters:

- `C`: matrix from FEM
- `G`: matrix from FEM
- `tau`: Controls the marginal variance of the field
- `kappa`: Controls the spatial scale of the field
- `jitter`: Minimum diagonal value to prevent near-singularity

```
Q <- tau^2 * (kappa^2 * C + G)
```

Step-by-step explanation: Comes from the formula:

$$Q = \tau^2(\kappa^2 C + G)$$

```
Q <- forceSymmetric(Q)
```

This ensures that the matrix is numerically symmetric. Although $\mathbf{Q}$ should be symmetric by theory, small floating-point errors during matrix operations can cause asymmetry. Enforcing symmetry is important for sparse solvers like those used in INLA.

```
diag_vals <- diag(Q)
diag(Q)[diag_vals < jitter] <- jitter
```

This step ensures that diagonal entries in $\mathbf{Q}$ are not too close to zero or negative due to numerical errors. Small values are replaced with a minimal positive value (`jitter = 1e-4` by default) to guarantee positive definiteness and numerical stability.

`simulate_latent_field()`

This function generates a single realization of a latent Gaussian random field from a zero-mean multivariate normal distribution with precision matrix $\mathbf{Q}$. It uses the `inla.qsample()` function from the INLA package to efficiently sample from this sparse Gaussian distribution.

```
simulate_latent_field <- function(Q) {
  as.vector(inla.qsample(n = 1, Q = Q))
}
```

`simulate_lgcp_points_continuous()`

This function simulates a realization of a point pattern from a Log-Gaussian Cox Process (LGCP) in a continuous d-dimensional space. It uses a latent field and a covariate function to modulate the log-intensity surface of the Poisson process.

```
simulate_lgcp_points_continuous <- function(Y, coords, covariate_fn, beta, bounds, scale_intensity = 20
  set.seed(seed)
  d <- ncol(coords)
  volume <- prod(sapply(bounds, function(b) diff(b)))
  eta_vals <- Y + beta * apply(coords, 1, covariate_fn)
  eta_vals <- pmin(pmax(eta_vals, -10), 10)
  lambda_max <- min(scale_intensity * exp(6), 1e5)
  N_max <- rpois(1, lambda_max * volume)
  points <- matrix(runif(N_max * d), ncol = d)
  for (i in 1:d) points[, i] <- bounds[[i]][1] + points[, i] * diff(bounds[[i]])
  nn <- get.knnx(coords, points, k = 1)
  eta_interp <- Y[nn$nn.index] + beta * apply(points, 1, covariate_fn)
  eta_interp <- pmin(pmax(eta_interp, -10), 10)
  lambda_interp <- scale_intensity * exp(eta_interp)
  keep <- runif(N_max) < (lambda_interp / lambda_max)
  points[keep, , drop = FALSE]
}
```

Parameters:

- `Y`: latent Gaussian field values at mesh nodes.
- `coords`: coordinates of the mesh nodes in d dimensions.
- `covariate_fn`: function of location returning covariate value.

- `beta`: coefficient controlling the covariate effect on log-intensity.
- `bounds`: list defining the simulation domain in each dimension.
- `scale_intensity`: multiplier that scales the overall intensity.
- `seed`: random seed for reproducibility.

```
volume <- prod(sapply(bounds, function(b) diff(b)))
```

Step-by-step explanation: Calculates the total volume (or area in 2D) of the simulation domain, which is the Cartesian product of all dimensions in `bounds`.

```
eta_vals <- Y + beta * apply(coords, 1, covariate_fn)
eta_vals <- pmin(pmax(eta_vals, -10), 10)
```

Computes:

$$\eta(s) = Y(s) + \beta \cdot covariate(s)$$

Then clamps η between -10 and 10 for numerical stability.

```
lambda_max <- min(scale_intensity * exp(6), 1e5)
N_max <- rpois(1, lambda_max * volume)
```

Sets an upper bound on the intensity and simulates a maximum number of candidate points from a homogeneous Poisson process with rate λ_{max} over the domain.

```
points <- matrix(runif(N_max * d), ncol = d)
for (i in 1:d) points[, i] <- bounds[[i]][1] + points[, i] * diff(bounds[[i]])
```

Generates `N_max` random points uniformly in the domain. Scales them into the bounds in each dimension.

```
nn <- get.knnx(coords, points, k = 1)
eta_interp <- Y[nn$nn.index] + beta * apply(points, 1, covariate_fn)
eta_interp <- pmin(pmax(eta_interp, -10), 10)
```

Finds the nearest mesh node for each random point. Interpolates η by taking the value at the nearest node and adding the covariate contribution.

```
lambda_interp <- scale_intensity * exp(eta_interp)
keep <- runif(N_max) < (lambda_interp / lambda_max)
points[keep, , drop = FALSE]
```

The log-intensity values $\eta(s)$ are transformed into pointwise intensities using the exponential function:

$$\lambda(s) = \text{scale\_intensity} \cdot e^{\eta(s)}$$

To simulate a realization of the inhomogeneous Poisson process, we employ the thinning method. Each candidate point is retained independently with probability:

$$p(s) = \frac{\lambda(s)}{\lambda_{\max}},$$

where $\lambda_{\max}$ is a fixed upper bound on the intensity. This ensures that the retained points follow the desired LGCP with spatially varying intensity.

`build_projector_matrix`

This function constructs a sparse **projector matrix A** that maps a finite element solution defined at mesh nodes onto arbitrary evaluation points (e.g., observation locations). Each row of **A** represents the barycentric weights used to interpolate values from the mesh nodes to a specific evaluation point.

```
build_projector_matrix <- function(mesh_points, simplices, eval_points) {
  d <- ncol(mesh_points)
  A <- sparseMatrix(i = integer(0), j = integer(0), x = numeric(0),
                    dims = c(nrow(eval_points), nrow(mesh_points)))
  ts <- tsearchn(mesh_points, simplices, eval_points, bary = TRUE)
  valid <- which(!is.na(ts$idx))
  for (i in valid) {
    simplex_id <- ts$idx[i]
    bary <- ts$p[i, ]
    if (any(is.na(bary))) next
    verts <- simplices[simplex_id, ]
    A[i, verts] <- bary
  }
  A
}
```

Parameters:

- `mesh_points`: Matrix of node coordinates from the finite element mesh.
- `simplices`: Matrix where each row defines a simplex by its node indices.
- `eval_points`: Locations at which the latent field should be evaluated or predicted.

```
A <- sparseMatrix(i = integer(0), j = integer(0), x = numeric(0),
                  dims = c(nrow(eval_points), nrow(mesh_points)))
```

Step-by-step explanation: Creates an empty matrix A. Each row corresponds to an evaluation point, each column to a mesh node.

```
ts <- tsearchn(mesh_points, simplices, eval_points, bary = TRUE)
```

For each evaluation point, identifies which simplex it falls into. Computes the barycentric weights within that simplex.

```
for (i in valid) {
  ...
  A[i, verts] <- bary
}
```

For each evaluation point inside a simplex, its row in A is populated with the barycentric weights at the corresponding simplex vertices. In the end it returns a sparse matrix A, where:

$$\hat{u}(s_i) = \sum_{j \in vertices} A_{ij} \cdot u_j$$

.

`construct_likelihood_data`

This function prepares the combined data vector needed for fitting a LGCP model with INLA. It merges the latent field mesh structure with the observed point locations into a single unified framework.

```
construct_likelihood_data <- function(mesh, observed_points, covariate_fn, alpha_weights) {
  mesh_points <- mesh$points
  n_mesh <- nrow(mesh_points)
  n_obs <- nrow(observed_points)
```

```
  locations <- rbind(mesh_points, observed_points)
  A <- build_projector_matrix(mesh_points, mesh$simplices, locations)
  cov_values <- apply(locations, 1, covariate_fn)
  y <- c(rep(0, n_mesh), rep(1, n_obs))
  weight <- c(alpha_weights, rep(0, n_obs))
  idx <- c(1:n_mesh, rep(NA, n_obs))
  list(y = y, weight = weight, covariate = cov_values, idx = idx, A = A)
}
```

Parameters:

- `mesh`: list containing the finite element mesh.
- `observed_points`: matrix of coordinates representing observed LGCP points.
- `covariate_fn`: function that takes a location vector and returns the value of a covariate at that location.
- `alpha_weights`: numeric vector of integration weights (one per mesh node), representing the relative area or volume associated with each node.

```
mesh_points <- mesh$points
n_mesh <- nrow(mesh_points)
n_obs <- nrow(observed_points)
```

Step-by-Step explanation: Extracts the mesh node coordinates. Counts how many mesh nodes and observed points are present.

```
locations <- rbind(mesh_points, observed_points)
```

Stacks the mesh node coordinates and the observed point coordinates into a single matrix of locations, where the mesh points come first, followed by the observed points.

```
A <- build_projector_matrix(mesh_points, mesh$simplices, locations)
```

Constructs a projector matrix A that interpolates latent field values at both mesh and observation locations.

```
cov_values <- apply(locations, 1, covariate_fn)
```

Applies the user-defined covariate function to all points (mesh + observations).

```
y <- c(rep(0, n_mesh), rep(1, n_obs))
```

Defines a binary outcome vector: - `0` for mesh locations (no observed events), - `1` for actual observed LGCP points.

```
weight <- c(alpha_weights, rep(0, n_obs))
```

Mesh points get exposure weights (e.g., area around node). Observed points have zero weight.

```
idx <- c(1:n_mesh, rep(NA, n_obs))
```

Index vector identifying mesh node positions for the latent field. Observed points have NA since they do not directly link to latent variables.

`run_inla_continuous`

Fits a Poisson log-linear model using INLA, where:

The latent spatial/trait field is modeled using a Gaussian Markov Random Field (GMRF) with precision matrix Q. An optional covariate effect $\beta \cdot x(s)$ is included. The observations are modeled as a point process

over the mesh and observation locations.

```
run_inla_continuous <- function(likelihood_data, Q, estimate_beta = TRUE, beta = 0.0) {
  y <- as.numeric(likelihood_data$y)
  weight <- as.numeric(likelihood_data$weight)
  covariate <- as.numeric(likelihood_data$covariate)
  mesh_idx <- likelihood_data$idx
  A <- likelihood_data$A

  offset <- beta * covariate
  n_mesh <- max(mesh_idx, na.rm = TRUE)
  idx_latent <- 1:n_mesh
  stk <- inla.stack(
    data = list(y = y, E = weight + 1e-10, offset = offset),
    A = list(A, 1),
    effects = list(
      idx = 1:ncol(A),
      covariate = covariate
    ),
    tag = "spatial"
  )

  result <- tryCatch({
    inla(
      formula = y ~ covariate + f(idx, model = "generic0", Cmatrix = Q),
      family = "poisson",
      data = inla.stack.data(stk),
      E = inla.stack.data(stk)$E,
      offset = inla.stack.data(stk)$offset,
      control.predictor = list(A = inla.stack.A(stk), compute = TRUE),
      control.inla = list(strategy = "laplace"),
      verbose = TRUE
    )
  }, error = function(e) {
    cat("INLA failed with error:\n", conditionMessage(e), "\n")
    return(NULL)
  })

  return(result)
}
```

Parameters:

- `likelihood_data`: Output from `construct_likelihood_data()`; includes response, weights, covariates, indices, and projector matrix.
- `Q`: Precision matrix for the latent field (defines spatial structure).
- `estimate_beta`: If `TRUE`, estimate the covariate effect; otherwise use fixed `beta`.
- `beta`: Fixed covariate coefficient (used only if `estimate_beta = FALSE`).

```
y <- as.numeric(likelihood_data$y)
weight <- as.numeric(likelihood_data$weight)
covariate <- as.numeric(likelihood_data$covariate)
```

```
mesh_idx <- likelihood_data$idx
A <- likelihood_data$A
```

Step-by-step explanation:

- `y`: response vector (0 for mesh, 1 for observed points).
- `weight`: exposure weights for mesh points (zero for observations).
- `covariate`: covariate values at each location.
- `idx`: mesh-based latent field indices.
- `A`: projector matrix from mesh nodes to all locations.

```
offset <- beta * covariate
```

Define the offset term.

```
stk <- inla.stack(
  data = list(y = y, E = weight + 1e-10, offset = offset),
  A = list(A, 1),
  effects = list(
    idx = 1:ncol(A),
    covariate = covariate
  ),
  tag = "spatial"
)
```

This creates a structured object containing:

- The data (`y`, exposure `E`, offset),
- Projector matrices for the latent field and covariate,
- Effects: latent field index and covariate.
- The small value `1e-10` is added to `weight` to prevent issues with zero exposure.

```
inla(
  formula = y ~ covariate + f(idx, model = "generic0", Cmatrix = Q),
  ...
)
```

- `formula`: includes covariate and latent field via `f(...)`,
- `generic0` model lets us specify a custom precision matrix `Q`,
- Uses Poisson likelihood with Laplace approximation for posterior inference.

`run_spde_lgcp_pipeline_continuous`

Runs a full simulation-and-estimation pipeline for a LGCP using the SPDE approach in arbitrary dimensions.

```
run_spde_lgcp_pipeline_continuous <- function(
    d = 3, m = 8, covariate_fn = function(x) sum(x), beta = 0.1,
    estimate_beta = TRUE, scale_intensity = 2000
) {
  bounds <- replicate(d, c(0, 1), simplify = FALSE)
  mesh <- build_mesh(d, m, bounds)
  fem <- compute_fem_matrices(mesh$points, mesh$simplices)
  Q <- assemble_precision_matrix(fem$C, fem$G)
  check_matrix_sanity(Q)
  Y <- scale(simulate_latent_field(Q))
  lgcp_points <- simulate_lgcp_points_continuous(Y, mesh$points, covariate_fn, beta, bounds, scale_inte
  alpha_weights <- rep(prod(sapply(bounds, function(b) diff(b))) / nrow(mesh$points), nrow(mesh$points))
  likelihood_data <- construct_likelihood_data(mesh, lgcp_points, covariate_fn, alpha_weights)
```

```
  result <- run_inla_continuous(likelihood_data, Q, estimate_beta)
  list(
    mesh_points = mesh$points,
    latent_field = Y,
    estimated_field = result$summary.random$idx$mean,
    beta_estimate = if (estimate_beta) result$summary.fixed else NULL,
    observed_points = lgcp_points,
    result = result
  )
}
```

Explanation:

1. Define domain bounds in d dimensions.
2. Build the mesh and compute the FEM mass matrix $\mathbf{C}$ and stiffness matrix $\mathbf{G}$.
3. Assemble the SPDE precision matrix $\mathbf{Q}$ from the FEM matrices.
4. Simulate the latent Gaussian field on the mesh nodes using the precision matrix.
5. Generate LGCP points by exponentiating the latent field and applying the covariate effect.
6. Compute integration weights for each mesh node to approximate spatial integrals.
7. Construct likelihood data by combining mesh-based exposure and observed points.
8. Fit the model using INLA, optionally estimating the covariate coefficient β.

Returns: A list with:

- `mesh_points`: Mesh node coordinates.
- `latent_field`: Simulated latent Gaussian values.
- `estimated_field`: Posterior mean estimate of latent field.
- `beta_estimate`: Estimated covariate effect (if applicable).
- `observed_points`: Simulated LGCP observations.
- `result`: Full INLA model output.

Example run:

```
result3d <- run_spde_lgcp_pipeline_continuous( d = 3, m = 5, covariate_fn = function(x) x[1], beta = 3.
```