WICMAD Developer Guide

Daniel Krasnov

October 17, 2025

Contents

1 Introduction

1.1 What is WICMAD?

WICMAD (Wavelet-ICM DP-GP sampler with wavelet shrinkage and lightweight diagnostics) is an R package that implements a **Dirichlet Process mixture model for functional curves** using:

- **Intrinsic Coregionalization Model (ICM)** Gaussian process residuals
- **Wavelet-domain shrinkage** for sparsity
- **MCMC sampling** with lightweight diagnostics
- **Anomaly detection** capabilities

The model is designed for **multivariate functional data** where each observation is a curve (or set of curves) that can be clustered, with applications in anomaly detection.

1.2 Core Architecture Overview

```
Input Data (Y, t)
    ↓
1. Wavelet Preprocessing
   - Forward transform all curves
   - Store coefficients and mappings
    ↓
2. MCMC Initialization
   - Initialize cluster assignments
   - Set up kernel parameters
   - Initialize wavelet parameters
    ↓
3. MCMC Loop (for each iteration):
   a. Cluster Assignment (DP)
   b. Wavelet Updates (Shrinkage)
   c. Kernel Updates (MH)
   d. ICM Updates (MH)
   e. Diagnostics Collection
    ↓
4. Postprocessing
   - Dahl partition
   - Diagnostic plots
   - Performance metrics
```

Figure 1: Data Flow Architecture

2 Wavelet Processing Module

2.1 Overview

The wavelet processing module (`R/wavelets.R`) handles the transformation of functional data into sparse wavelet representations. This module provides the core functionality for converting time-domain signals into wavelet coefficients, which are then used in the MCMC sampling process for efficient sparse representation and wavelet shrinkage.

2.2 Wavelets

The wavelet system provides forward and inverse discrete wavelet transforms for converting time-domain signals into sparse coefficient representations. The system supports multiple wavelet families and boundary conditions, with efficient precomputation for MCMC sampling.

```
wt_forward_1d <- function(y, wf = "la8", J = NULL, boundary = "periodic")
   {
  P <- length(y); J <- ensure_dyadic_J(P, J)
  w <- waveslim::dwt(y, wf=wf, n.levels=J, boundary=boundary)
  vec <- c(w$d1); idx <- list(d1 = seq_along(w$d1)); off <- length(w$d1)
  if (J >= 2) for (lev in 2:J) {
    nm <- paste0("d", lev); v <- w[[nm]]; vec <- c(vec, v)
    idx[[nm]] <- (off + 1):(off + length(v)); off <- off + length(v)
  }
  s_nm <- paste0("s", J); vec <- c(vec, w[[s_nm]])
  idx[[s_nm]] <- (off + 1):(off + length(w[[s_nm]]))
  list(coeff = as.numeric(vec), map = list(J=J, wf=wf, boundary=boundary,
     P=P, idx=idx))
}
```

```
wt_inverse_1d <- function(coeff_vec, map) {
  J <- map$J
  w <- vector("list", J + 1L)
  names(w) <- c(paste0("d", 1:J), paste0("s", J))
  for (lev in 1:J) {
    nm <- paste0("d", lev); ids <- map$idx[[nm]]
    w[[nm]] <- as.numeric(coeff_vec[ids])
  }
  ids_s <- map$idx[[paste0("s", J)]]; w[[paste0("s", J)]] <-
     as.numeric(coeff_vec[ids_s])
  attr(w, "wavelet") <- map$wf; attr(w, "boundary") <- map$boundary;
     class(w) <- "dwt"
  waveslim::idwt(w)
}
```

```
wt_forward_mat <- function(y_mat, wf="la8", J=NULL, boundary="periodic") {
  M <- ncol(y_mat); out <- vector("list", M)
  for (m in 1:M) out[[m]] <- wt_forward_1d(y_mat[,m], wf, J, boundary)
  out
}
```

```
precompute_wavelets <- function(Y_list, wf, J, boundary) {
  lapply(Y_list, function(mat) wt_forward_mat(mat, wf, J, boundary))
```

```
}
```

```
stack_D_from_precomp <- function(precomp, idx, M, bias_coeff = NULL) {
  ncoeff <- length(precomp[[ idx[1] ]][[1]]$coeff)
  N <- length(idx)
  D_arr <- array(NA_real_, dim=c(ncoeff, N, M))
  for (jj in seq_along(idx)) {
    i <- idx[jj]
    for (m in 1:M) {
      D_arr[, jj, m] <- precomp[[i]][[m]]$coeff
      if (!is.null(bias_coeff)) {
        bc <- bias_coeff[[m]]
        if (!is.null(bc) && length(bc) == ncoeff) {
          D_arr[, jj, m] <- D_arr[, jj, m] - bc
        }
      }
    }
  }
  maps <- precomp[[ idx[1] ]]
  list(D_arr = D_arr, maps = maps)
}
```

```
compute_mu_from_beta <- function(beta_ch, wf, J, boundary, P) {
  M <- length(beta_ch); zeros <- matrix(0, nrow=P, ncol=M)
  tmpl <- wt_forward_mat(zeros, wf, J, boundary); mu <- matrix(0, P, M)
  for (m in 1:M) mu[,m] <- wt_inverse_1d(beta_ch[[m]], tmpl[[m]]$map)
  mu
}
```

3 Kernel System

3.1 Overview

The kernel system (`R/kernels.R`) provides flexible Gaussian process covariance functions for modeling spatial/temporal dependencies in the functional data.

3.2 Core Functions

3.2.1 `make_kernels()`

The `make_kernels()` function creates a collection of kernel functions with their associated priors and proposal distributions.

```
make_kernels <- function(add_bias_variants = TRUE) {
  k_sqexp <- function(t, l_scale) {
    D2 <- dist_rows(t)^2
    exp(-0.5 * D2 / (l_scale^2))
  }
  k_mat32 <- function(t, l_scale) {
    D  <- dist_rows(t); r <- D / l_scale; a <- sqrt(3) * r
    (1 + a) * exp(-a)
  }
```

```
  k_mat52 <- function(t, l_scale) {
    D  <- dist_rows(t); r <- D / l_scale; a <- sqrt(5) * r
    (1 + a + 5 * r^2 / 3) * exp(-a)
  }
  k_periodic <- function(t, l_scale, period) {
    D <- dist_rows(t)
    exp(- 2 * sin(base::pi * D / period)^2 / (l_scale^2))
  }

  base <- list(
    list(
      name="SE", fun=function(t, par) k_sqexp(t, par$l_scale),
      pnames=c("l_scale"),
      prior = function(par) stats::dgamma(par$l_scale, 2, 2, log=TRUE),
      pstar = function() list(l_scale = stats::rgamma(1, 2, 2)),
      prop_sd = list(l_scale=0.20)
    ),
    list(
      name="Mat32", fun=function(t, par) k_mat32(t, par$l_scale),
      pnames=c("l_scale"),
      prior = function(par) stats::dgamma(par$l_scale, 2, 2, log=TRUE),
      pstar = function() list(l_scale = stats::rgamma(1, 2, 2)),
      prop_sd = list(l_scale=0.20)
    ),
    list(
      name="Mat52", fun=function(t, par) k_mat52(t, par$l_scale),
      pnames=c("l_scale"),
      prior = function(par) stats::dgamma(par$l_scale, 2, 2, log=TRUE),
      pstar = function() list(l_scale = stats::rgamma(1, 2, 2)),
      prop_sd = list(l_scale=0.20)
    ),
    list(
      name="Periodic", fun=function(t, par) k_periodic(t, par$l_scale,
         par$period),
      pnames=c("l_scale","period"),
      prior = function(par) stats::dgamma(par$l_scale, 3, 2, log=TRUE) +
         stats::dbeta(par$period, 5, 5, log=TRUE),
      pstar = function() list(l_scale = stats::rgamma(1, 3, 2), period =
         stats::rbeta(1, 5, 5)),
      prop_sd = list(l_scale=0.20, period=0.20)
    )
  )

  if (!add_bias_variants) return(base)

  k_bias <- function(t, s0) { P <- nloc(t); (s0^2) * matrix(1, P, P) }

  make_bias_variant <- function(kcfg) {
    nm <- paste0(kcfg$name, "+Bias")
    fun_bias <- function(t, par) {
      Kbase <- kcfg$fun(t, par)
      Kbase + k_bias(t, par$s0)
    }
    prior_bias <- function(par) {
```

```
      kcfg$prior(par) + dnorm(log(pmax(par$s0, 1e-9)), mean = -3, sd =
          0.75, log = TRUE)
    }
    pstar_bias <- function() { th <- kcfg$pstar(); th$s0 <- exp(rnorm(1,
        -3, 0.75)); th }
    prop_sd_bias <- kcfg$prop_sd; prop_sd_bias$s0 <- 0.25
    list(
      name    = nm,
      fun     = fun_bias,
      pnames  = c(kcfg$pnames, "s0"),
      prior   = prior_bias,
      pstar   = pstar_bias,
      prop_sd= prop_sd_bias
    )
  }

  bias_variants <- lapply(base, make_bias_variant)
  c(base, bias_variants)
}
```

3.3 Kernels

Below are the available kernel functions:

Kernel	Formula
Squared Exponential	$k(x,x') = \sigma^2 \exp(-\frac{\|x-x'\|^2}{2\ell^2})$
Matérn 3/2	$k(x,x') = \sigma^2(1+\sqrt{3}r)\exp(-\sqrt{3}r)$
Matérn 5/2	$k(x,x') = \sigma^2(1+\sqrt{5}r+\frac{5r^2}{3})\exp(-\sqrt{5}r)$
Periodic	$k(x,x') = \sigma^2 \exp(-\frac{2\sin^2(\pi\|x-x'\|/p)}{\ell^2})$

Table 1: Available Kernel Functions

In code the kernel functions are: `k_sqexp()` for squared exponential, `k_mat32()` for Matérn 3/2, `k_mat52()` for Matérn 5/2, `k_periodic()` for periodic patterns, `k_bias()` for bias terms, and `make_bias_variant()` for creating bias variants of existing kernels.

```
k_sqexp <- function(t, l_scale) {
  D2 <- dist_rows(t)^2
  exp(-0.5 * D2 / (l_scale^2))
}
```

```
k_mat32 <- function(t, l_scale) {
  D  <- dist_rows(t); r <- D / l_scale; a <- sqrt(3) * r
  (1 + a) * exp(-a)
}
```

```
k_mat52 <- function(t, l_scale) {
  D  <- dist_rows(t); r <- D / l_scale; a <- sqrt(5) * r
  (1 + a + 5 * r^2 / 3) * exp(-a)
}
```

```
k_periodic <- function(t, l_scale, period) {
  D <- dist_rows(t)
  exp(- 2 * sin(base::pi * D / period)^2 / (l_scale^2))
}
```

```
k_bias <- function(t, s0) { P <- nloc(t); (s0^2) * matrix(1, P, P) }
```

The `make_bias_variant()` function creates bias variants of existing kernels by adding a bias term to handle non-zero mean functions. The `prior_bias` function combines the original kernel's prior with a log-normal prior for the bias scale parameter `s0`, using a mean of -3 and standard deviation of 0.75 on the log scale. The `pstar_bias` function samples from the prior by first sampling the original kernel parameters and then adding a bias scale parameter sampled from the log-normal distribution.

```
make_bias_variant <- function(kcfg) {
  nm <- paste0(kcfg$name, "+Bias")
  fun_bias <- function(t, par) {
    Kbase <- kcfg$fun(t, par)
    Kbase + k_bias(t, par$s0)
  }
  prior_bias <- function(par) {
    kcfg$prior(par) + dnorm(log(pmax(par$s0, 1e-9)), mean = -3, sd =
       0.75, log = TRUE)
  }
  pstar_bias <- function() { th <- kcfg$pstar(); th$s0 <- exp(rnorm(1,
     -3, 0.75)); th }
  prop_sd_bias <- kcfg$prop_sd; prop_sd_bias$s0 <- 0.25
  list(
    name   = nm,
    fun    = fun_bias,
    pnames = c(kcfg$pnames, "s0"),
    prior  = prior_bias,
    pstar  = pstar_bias,
    prop_sd= prop_sd_bias
  )
}
```

4 Intrinsic Coregionalization Model (ICM) Section

4.1 Overview

The ICM system (`R/icm_cache.R`) models multivariate Gaussian process residuals by separating:

(a) a **shared input kernel** $K_x(t, t')$ that governs temporal/spatial smoothness, and

(b) a **coregionalization matrix** B that governs inter-channel correlations.

This yields a covariance of the form $B \otimes K_x$, which we evaluate efficiently by eigendecomposing K_x and working with P small $M \times M$ blocks instead of a single $PM \times PM$ matrix.

4.2 Generative Model and Covariance Structure

Let $Y(t) \in \mathbb{R}^M$ be the M-variate output at t, and collect observations at P input locations into the $P \times M$ matrix $\mathbf{Y} = [Y(t_1), \ldots, Y(t_P)]^\top$. Conditional on kernel/ICM parameters,

$$\mathrm{vec}(\mathbf{Y}) \sim \mathcal{N}\Big(\mathrm{vec}(\mu),\, \Sigma\Big), \quad \Sigma \;=\; \tau_B\, B \otimes K_x \;+\; D_\eta \otimes I_P,$$

with $D_\eta = \mathrm{diag}(\eta_1, \ldots, \eta_M)$. If a bias term is used on the input side, the kernel becomes $K_x^{(+)} = K_x + s_0^2\, \mathbf{1}\mathbf{1}^\top$; the PSD property is preserved and the same machinery applies.

4.3 Coregionalization Matrix Decomposition

To ensure positive semidefiniteness and interpretability, the coregionalization matrix B is parameterized as:

$$B = LL^\top, \qquad L \in \mathbb{R}^{M \times M}_{\text{lower-triangular}}.$$

This guarantees $B \succeq 0$ automatically. In the implementation, B is further normalized to have unit trace:

$$B_{\text{shape}} = \frac{M\, LL^\top}{\mathrm{tr}(LL^\top)}.$$

The rescaling decouples the relative correlation structure (contained in B_{shape}) from the overall scale parameter τ_B, improving numerical stability and mixing. The Cholesky-like matrix L is updated in MCMC, while τ_B scales the shared variability across all channels.

4.4 Kronecker Form and the Eigen Trick

We eigendecompose the input kernel,

$$K_x = U_x\, \Lambda_x\, U_x^\top, \qquad \Lambda_x = \mathrm{diag}(\lambda_1, \ldots, \lambda_P),$$

and rotate residuals along the input dimension:

$$\widetilde{\mathbf{Y}}_{\text{res}} \;=\; U_x^\top (\mathbf{Y} - \mu).$$

In this basis the joint covariance decouples across input eigencomponents:

$$\Sigma \;=\; \bigoplus_{j=1}^{P} \Sigma_j, \qquad \Sigma_j \;=\; \tau_B\, \lambda_j\, B \;+\; D_\eta \in \mathbb{R}^{M \times M}.$$

The Gaussian log-likelihood becomes

$$\log p(\mathbf{Y} \mid \theta) = -\tfrac{1}{2}\Big(PM \log 2\pi \;+\; \sum_{j=1}^{P} \log|\Sigma_j| \;+\; \sum_{j=1}^{P} \widetilde{y}_j^\top \Sigma_j^{-1} \widetilde{y}_j\Big),$$

where $\widetilde{y}_j^\top$ is the jth row of $\widetilde{\mathbf{Y}}_{\text{res}}$.

4.5 Code Walkthrough

4.5.1 .build_icm_cache(): eigens, blocks, and Cholesky

```
# R/icm_cache.R
.build_icm_cache <- function(t, kern_cfg, kp, L, eta, tau_B, cache =
    NULL) {
  # --- 1) Input kernel Kx and its eigendecomposition ---
  # Build Kx from the selected kernel config and parameters kp
  Kx <- kern_cfg$fun(t, kp)                          # P x P PSD matrix
  eig <- eigen(Kx, symmetric = TRUE)
  Ux  <- eig$vectors                                 # = U_x
  lam <- pmax(eig$values, 1e-12)                     # = diag entries of
     Lambda_x (clipped)

  # --- 2) Coregionalization matrix B with unit-trace normalization ---
  Bshape <- tcrossprod(L)                            # L L^T  (M x M, PSD)
  trB <- sum(diag(Bshape))
  if (trB > 0) Bshape <- Bshape * (nrow(Bshape) / trB)  # enforce tr(B)=M

  # --- 3) Assemble Sigma_j = tau_B * lambda_j * B + D_eta and precompute
     Cholesky ---
  M    <- length(eta)
  Deta <- diag(eta, M, M)
  chol_list   <- vector("list", length(lam))
  logdet_sum  <- 0
  for (j in seq_along(lam)) {
    Sj <- tau_B * lam[j] * Bshape + Deta             # M x M block
    # Robust Cholesky with jitter escalation:
    ok <- FALSE; Lj <- NULL; jitter <- 1e-8
    for (tries in 1:6) {
      out <- try(chol(Sj + diag(jitter, M)), silent = TRUE)
      if (!inherits(out, "try-error")) { Lj <- out; ok <- TRUE; break }
      jitter <- jitter * 10
    }
    if (!ok) stop("Cholesky failed in Sigma_j block. Check
       priors/proposals.")
    chol_list[[j]] <- Lj
    logdet_sum <- logdet_sum + 2 * sum(log(diag(Lj)))  # accum
       log|Sigma_j|
  }

  # --- 4) Store cache used by fast likelihood ---
  cache <- list(Ux_x = Ux, lam_x = lam, Bshape = Bshape,
                chol_list = chol_list, logdet_sum = logdet_sum,
                tau = tau_B, eta = eta,
                key_kx = paste(kern_cfg$name, paste(unlist(kp),
                   collapse="|"), sep="::"),
                key_B  = paste(round(L,8), collapse="|"))
  cache
}
```

Notes.

- `kern_cfg$fun(t,kp)` builds K_x (SE/Matérn/Periodic/ +Bias).

- `Ux_x` and `lam_x` store U_x and $\{\lambda_j\}$.
- `Bshape` holds $B = LL^\top$ rescaled to unit trace.
- Each block Σ_j is Cholesky-factorized once and cached.

4.5.2 `fast_icm_loglik_curve()`: quadratic form and log-det

```
# R/icm_cache.R
fast_icm_loglik_curve <- function(y_resid, cache) {
  # y_resid: P x M residuals for one curve (Y - mu)
  Ux <- cache$Ux_x
  Ytil <- t(Ux) %*% y_resid                    # rotate along input:
     U_x^T * (Y - mu)

  quad <- 0
  for (j in seq_along(cache$chol_list)) {
    Lj <- cache$chol_list[[j]]                 # Cholesky of Sigma_j
    v  <- as.numeric(Ytil[j, ])                # 1 x M row -> length-M
       vector
    # Solve Sigma_j^{-1/2} v via two triangular solves:
    w  <- backsolve(Lj, forwardsolve(t(Lj), v, upper.tri=TRUE,
       transpose=TRUE))
    quad <- quad + sum(w * w)                  # v^T Sigma_j^{-1} v =
       ||w||^2
  }
  # PM*log(2*pi) + sum_j log|Sigma_j| + sum_j v_j^T Sigma_j^{-1} v_j
  const <- nrow(y_resid) * ncol(y_resid) * log(2*pi)
  -0.5 * (const + cache$logdet_sum + quad)
}
```

Notes.

- The input rotation $U_x^\top$ makes the covariance block-diagonal.
- Each block contribution uses cached `Lj` to avoid re-factorization.
- The total log-likelihood is the sum over $j = 1{:}P$ blocks.

5 Wavelet Block with Shrinkage Priors

5.1 Overview

The wavelet block system (`R/wavelet_block.R`) models cluster-level mean functions in the wavelet domain using hierarchical shrinkage priors. Each cluster's mean coefficients are assumed sparse, reflecting that most wavelet coefficients of smooth functions are near zero. The Besov prior formalism provides a probabilistic representation of this sparsity via a spike-and-slab mixture, which adaptively shrinks coefficients toward zero while retaining sharp local features.

5.2 Hierarchical Prior and Shrinkage Structure

Let $\beta_{m,j,k}$ denote the wavelet coefficient for output channel m at level j and position k. The model introduces a binary inclusion indicator $\gamma_{m,j,k}$ and defines:

$$\beta_{m,j,k} \mid \gamma_{m,j,k}, \sigma_m^2, g_j \sim \begin{cases} \mathcal{N}(0,\, \sigma_m^2), & \gamma_{m,j,k} = 0, \\ \mathcal{N}(0,\, \sigma_m^2(1+g_j)), & \gamma_{m,j,k} = 1, \end{cases} \qquad \gamma_{m,j,k} \sim \text{Bernoulli}(\pi_j),$$

with level-wise hyperpriors $\pi_j \sim \text{Beta}(a_\pi, b_\pi)$ and $g_j \sim \text{Gamma}(a_g, b_g)$. The marginal prior

$$p(\beta_{m,j,k}) = (1-\pi_j)\,\mathcal{N}(0, \sigma_m^2) + \pi_j\,\mathcal{N}(0, \sigma_m^2(1+g_j))$$

induces adaptive sparsity: small coefficients are likely to be set to zero, while significant local features survive the shrinkage.

5.3 Posterior Updates (Closed Forms)

Let n denote the number of curves in the cluster (columns in D), and $\bar{d}_{m,j,k}$ the sample mean across them.

Step 1: Inclusion indicators $\gamma_{m,j,k}$. Under spike $\mathcal{N}(0, \sigma_m^2)$ and slab $\mathcal{N}(0, \sigma_m^2(1+g_j))$,

$$p(\gamma_{m,j,k} = 1 \mid \beta_{m,j,k}) = \frac{\pi_j\, \phi(\beta_{m,j,k}; 0, \sigma_m^2(1+g_j))}{\pi_j\, \phi(\beta_{m,j,k}; 0, \sigma_m^2(1+g_j)) + (1-\pi_j)\, \phi(\beta_{m,j,k}; 0, \sigma_m^2)}.$$

Equivalently, in log-odds form:

$$\log \frac{p(\gamma=1 \mid \beta)}{p(\gamma=0 \mid \beta)} = \log \frac{\pi_j}{1-\pi_j} - \tfrac{1}{2}\log(1+g_j) + \frac{\beta_{m,j,k}^2}{2\sigma_m^2}\frac{g_j}{1+g_j}.$$

This step determines which wavelet coefficients are active (included in the model) versus inactive (set to zero). The log-odds formulation provides numerical stability by avoiding division by small probabilities.

Step 2: Slab inflation g_j. Assume $g_j \sim \text{InvGamma}(\text{shape}_0, \text{rate}_0)$. Let $\mathcal{A}_j$ be the active set at level j (γ=1), with $S_j = \sum_{(m,k)\in\mathcal{A}_j} \beta_{m,j,k}^2/\sigma_m^2$ and $n_j = |\mathcal{A}_j|$. Then

$$g_j \mid \{\beta, \gamma\} \sim \text{InvGamma}\Big(\text{shape}_0 + \tfrac{n_j}{2},\, \text{rate}_0 + \tfrac{S_j}{2}\Big).$$

The shrinkage parameter g_j controls how much active coefficients are shrunk toward zero. Higher values of g_j lead to stronger shrinkage, while lower values allow coefficients to remain closer to their observed values. The posterior updates based on the sum-of-squares of active coefficients, appropriately scaled by the noise variance.

Step 3: Levelwise sparsity π_j. With Beta prior mean $m_j = \kappa_\pi 2^{-c_2 j}$ and concentration τ_π,

$$\pi_j \sim \text{Beta}(a_0, b_0), \quad a_0 = \tau_\pi m_j, \;\; b_0 = \tau_\pi(1-m_j).$$

Given counts n_1, n_0 of active/inactive coefficients:

$$\pi_j \mid \gamma \sim \text{Beta}(a_0 + n_1,\; b_0 + n_0).$$

This step updates the level-specific sparsity probabilities, which control the overall sparsity pattern across resolution levels. The Besov prior structure ensures that finer scales (higher j) have lower sparsity probabilities, encouraging smoothness while allowing for local features. The concentration parameter τ_π controls the strength of this prior.

Step 4: Coefficients $\beta_{m,j,k}$ (active only). With n replicates per coefficient,

$$\beta_{m,j,k} \mid D, \gamma{=}1 \sim \mathcal{N}\big(\mu_j^{\text{post}},\, v_{m,j}^{\text{post}}\big), \quad \mu_j^{\text{post}} = \frac{n}{n + 1/g_j}\bar{d}_{m,j,k}, \;\; v_{m,j}^{\text{post}} = \frac{\sigma_m^2}{n + 1/g_j}.$$

Active coefficients are sampled from their Gaussian posteriors, which balance the observed data (sample mean) with the shrinkage prior (mean zero). The posterior mean is a shrinkage estimator that pulls the sample mean toward zero, with the amount of shrinkage controlled by g_j. The posterior variance reflects both the data uncertainty and the prior precision.

Step 5: Channel noise σ_m^2. With prior $\sigma_m^2 \sim \text{InvGamma}(a_\sigma, b_\sigma\tau_\sigma)$ and residual $R_m = D_m - \beta_m \mathbf{1}_n^\top$,

$$\sigma_m^2 \mid D, \beta, \tau_\sigma \sim \text{InvGamma}\Big(a_\sigma + \tfrac{n_{\text{eff}}}{2},\; b_\sigma\tau_\sigma + \tfrac{1}{2}\|R_m\|_F^2\Big), \quad n_{\text{eff}} = n_{\text{coeff}} \cdot n.$$

The channel-specific noise variances are updated based on the residuals between the observed wavelet coefficients and the current estimates. The hierarchical structure allows information sharing across channels through the global scaling parameter τ_σ.

Step 6: Global scale τ_σ. With prior $\tau_\sigma \sim \text{Gamma}(a_\tau, b_\tau)$,

$$\tau_\sigma \mid \{\sigma_m^2\}_{m=1}^M \sim \text{Gamma}\Big(a_\tau + Ma_\sigma,\; b_\tau + b_\sigma \sum_{m=1}^{M} \tfrac{1}{\sigma_m^2}\Big).$$

The global scaling parameter provides a hierarchical structure for the noise variances, allowing information sharing across channels while maintaining channel-specific flexibility. This step completes the hierarchical update cycle.

5.4 Code Walkthrough

5.4.1 `update_cluster_wavelet_params_besov()`: hierarchical Gibbs updates

```
# R/wavelet_block.R
update_cluster_wavelet_params_besov <- function(
    idx, precomp, M, wpar, sigma2_m, tau_sigma,
    kappa_pi = 0.6, c2 = 1.0, tau_pi = 40,
    g_hyp = NULL,
    a_sig = 2.5, b_sig = 0.02,    # for sigma2_m
    a_tau = 2.0, b_tau = 2.0,     # for tau_sigma
    bias_coeff = NULL
) {
  if (length(idx) == 0) {
    return(list(wpar=wpar, beta_ch=wpar$beta_ch %||%
           lapply(1:M, function(m) 0),
           sigma2_m=sigma2_m, tau_sigma=tau_sigma, maps=NULL))
  }
  stk <- stack_D_from_precomp(precomp, idx, M, bias_coeff=bias_coeff)
  D <- stk$D_arr; maps <- stk$maps
  ncoeff <- dim(D)[1]; N <- dim(D)[2]
  lev_names <- names(maps[[1]]$map$idx)
  det_names <- lev_names[grepl("^d", lev_names)]
  s_name <- lev_names[grepl("^s", lev_names)]
```

```
if (is.null(wpar$pi_level)) wpar$pi_level <- setNames(rep(0.5,
    length(det_names)), det_names)
if (is.null(wpar$g_level))  wpar$g_level  <- setNames(rep(2.0,
    length(det_names)), det_names)
if (is.null(wpar$gamma_ch)) wpar$gamma_ch <- lapply(1:M, function(m)
    rbinom(ncoeff, 1, 0.2))

# 1) gamma updates (sparsity)
for (m in 1:M) {
  Dm <- matrix(D[ , , m, drop=FALSE], nrow=ncoeff, ncol=N)
  gam <- wpar$gamma_ch[[m]]
  for (lev in det_names) {
    ids <- maps[[m]]$map$idx[[lev]]; if (!length(ids)) next
    pi_j <- wpar$pi_level[[lev]]; g_j  <- wpar$g_level[[lev]]
    Dsub <- Dm[ids, , drop=FALSE]
    v_spike <- sigma2_m[m]; v_slab <- (1 + g_j) * sigma2_m[m]
    ll_spike <- -0.5 * rowSums(log(2*pi*v_spike) + (Dsub^2)/v_spike)
    ll_slab  <- -0.5 * rowSums(log(2*pi*v_slab)  + (Dsub^2)/v_slab)
    logit_val <- log(pi_j) + ll_slab - (log(1-pi_j) + ll_spike)
    p1 <- plogis(pmax(pmin(logit_val,35), -35))
    gam[ids] <- rbinom(length(ids), 1, p1)
  }
  if (length(s_name)==1) {
    ids_s <- maps[[m]]$map$idx[[s_name]]
    if (length(ids_s)>0) gam[ids_s] <- 1L
  }
  wpar$gamma_ch[[m]] <- gam
}

# 2) g_level updates
for (lev in det_names) {
  shape0 <- if (!is.null(g_hyp)) g_hyp[lev,"shape"] else 2.0
  rate0  <- if (!is.null(g_hyp)) g_hyp[lev,"rate"]  else 2.0
  ss_over_sigma <- 0; n_sel_total <- 0
  for (m in 1:M) {
    ids <- maps[[m]]$map$idx[[lev]]; if (!length(ids)) next
    Dm <- matrix(D[ , , m, drop=FALSE], nrow=ncoeff, ncol=N)
    sel <- wpar$gamma_ch[[m]][ids] == 1
    if (any(sel)) {
      Did <- Dm[ids[sel], , drop=FALSE]
      ss_over_sigma <- ss_over_sigma + sum(Did^2)/sigma2_m[m]
      n_sel_total <- n_sel_total + nrow(Did)
    }
  }
  shape_post <- shape0 + 0.5*n_sel_total
  rate_post  <- rate0  + 0.5*ss_over_sigma
  wpar$g_level[[lev]] <- invgamma::rinvgamma(1, shape=shape_post,
     rate=rate_post)
}

# 3) pi_level (Besov prior)
for (lev in det_names) {
  jnum <- as.integer(sub("^d","",lev))
  m_j <- max(min(kappa_pi*2^(-c2*jnum), 1-1e-6), 1e-6)
```

```
    a0 <- tau_pi*m_j; b0 <- tau_pi*(1-m_j)
    n1 <- 0; n0 <- 0
    for (m in 1:M) {
      ids <- maps[[m]]$map$idx[[lev]]; if (!length(ids)) next
      gm <- wpar$gamma_ch[[m]][ids]
      n1 <- n1 + sum(gm==1); n0 <- n0 + sum(gm==0)
    }
    wpar$pi_level[[lev]] <- rbeta(1, a0+n1, b0+n0)
  }

  # 4) beta sampling
  beta_ch <- lapply(1:M, function(m) numeric(ncoeff))
  for (m in 1:M) {
    Dm <- matrix(D[ , , m, drop=FALSE], nrow=ncoeff, ncol=N)
    gam <- wpar$gamma_ch[[m]]; b <- numeric(ncoeff)
    for (lev in det_names) {
      ids <- maps[[m]]$map$idx[[lev]]; if (!length(ids)) next
      g_j <- wpar$g_level[[lev]]; n <- N
      Dbar <- rowMeans(Dm[ids, , drop=FALSE])
      shrink <- n/(n + 1/g_j)
      mean_post <- shrink * Dbar
      var_post <- sigma2_m[m]/(n + 1/g_j)
      is_on <- (gam[ids]==1L)
      if (any(is_on)) {
        k <- sum(is_on)
        b[ids[is_on]] <- rnorm(k, mean_post[is_on], sqrt(var_post))
      }
    }
    if (length(s_name)==1) {
      ids_s <- maps[[m]]$map$idx[[s_name]]
      if (length(ids_s)>0) {
        Dbar_s <- rowMeans(Dm[ids_s, , drop=FALSE])
        b[ids_s] <- rnorm(length(ids_s), Dbar_s, sqrt(sigma2_m[m]/N))
      }
    }
    beta_ch[[m]] <- b
  }

  # 5) sigma2_m + 6) tau_sigma
  sigma2_m_new <- sigma2_m; n_eff_m <- ncoeff*N
  for (m in 1:M) {
    Dm <- matrix(D[ , , m, drop=FALSE], nrow=ncoeff, ncol=N)
    resid <- sweep(Dm, 1, beta_ch[[m]], "-")
    ss_m <- sum(resid^2)
    shape_post <- a_sig + 0.5*n_eff_m
    rate_post  <- b_sig*tau_sigma + 0.5*ss_m
    sigma2_m_new[m] <- invgamma::rinvgamma(1, shape=shape_post,
       rate=rate_post)
  }
  a_post <- a_tau + M*a_sig
  b_post <- b_tau + b_sig*sum(1/sigma2_m_new)
  tau_sigma_new <- rgamma(1, shape=a_post, rate=b_post)

  list(wpar=wpar, beta_ch=beta_ch,
```

```
        sigma2_m=sigma2_m_new, tau_sigma=tau_sigma_new, maps=maps)
}
```

Notes.

- The function performs a full Gibbs sweep for all hierarchical parameters.
- Each posterior update corresponds exactly to the analytical forms above.
- Wavelet sparsity is enforced level-wise, consistent with Besov regularity.

6 Main Driver Function

6.1 Overview

The main driver function (`R/driver_wicmad.R`) orchestrates the entire MCMC sampling process. The `wicmad()` function is the primary interface for running the WICMAD model, coordinating all model components including wavelet processing, ICM updates, and cluster assignments.

6.2 Function Structure

The `wicmad()` function is organized into several key phases:

1. **Initialization & Setup**: Parameter validation, data preprocessing, and initial cluster assignments
2. **Main MCMC Loop**: Iterative updates of all model parameters
3. **Diagnostics Collection**: Monitoring and tracking of MCMC performance
4. **Output Generation**: Formatting and returning results

6.3 Initialization and Setup

The function begins with comprehensive parameter validation and data preprocessing:

```
# Parameter validation and guards
if (!is.numeric(n_iter) || n_iter < 2) stop("n_iter must be >= 2.")
if (!is.numeric(thin) || thin < 1) thin <- 1L
n_iter <- as.integer(n_iter); burn <- as.integer(burn); thin <-
    as.integer(thin)

# Data dimensions and preprocessing
N <- length(Y); P <- nrow(Y[[1]]); M <- ncol(Y[[1]]); J <-
    ensure_dyadic_J(P, J)
t <- normalize_t(t, P); t <- scale_t01(t)

# Precompute wavelet transforms for all curves
precomp_all <- precompute_wavelets(Y, wf, J, boundary)
```

The initialization phase ensures numerical stability by scaling time coordinates to [0,1], validates input parameters, and precomputes wavelet transforms to avoid repeated computation during MCMC.

6.4 Main MCMC Loop Structure

The core of the function is the MCMC iteration loop that updates all model parameters:

```
for (iter in 1:n_iter) {
  # 1. Cluster assignment updates
  # 2. Wavelet block updates for each cluster
  # 3. Kernel parameter updates via Metropolis-Hastings
  # 4. ICM parameter updates
  # 5. Diagnostics collection
  # 6. Sample saving (if iter > burn and iter %% thin == 0)
}
```

Each iteration performs a complete update cycle across all model components, ensuring detailed balance and proper MCMC convergence.

6.5 Cluster Assignment Updates

The cluster assignment step uses the Walker/Kalli slice sampling algorithm for the Dirichlet Process. This method efficiently handles the infinite-dimensional nature of the DP while maintaining computational tractability.

```
# Walker/Kalli slice expansion
pi <- stick_to_pi(v)
u  <- sapply(1:N, function(i) runif(1, 0, pi[z[i]]))
u_star <- min(u)
v <- extend_sticks_until(v, alpha, u_star)
pi <- stick_to_pi(v)
K  <- length(v)
while (length(params) < K) {
  params[[length(params) + 1]] <- draw_new_cluster_params(M, P, t,
     kernels, wf, J, boundary)
  k_new <- length(params)
  params <- ensure_complete_cache(params, kernels, k_new, t, M)
}

# Apply cluster assignments
```

The slice sampling algorithm works by first drawing uniform slice variables $u_i \sim \text{Uniform}(0, \pi_{z_i})$ for each curve. The minimum slice value u^* determines how many stick-breaking components need to be generated. The algorithm then extends the stick-breaking representation until all required components are available. For each curve, the assignment function computes the log-likelihood for all valid clusters (those with $\pi_k > u_i$) and samples the assignment probabilistically. The implementation includes caching of mean functions to avoid redundant computation and special handling for revealed curves during the warmup period for semi-supervised learning.

After cluster assignments are updated, the stick-breaking parameters are also updated to reflect the new cluster structure. This involves updating the stick-breaking weights v based on the current cluster assignments and then converting them back to cluster probabilities π:

```
# Stick-breaking parameter updates
v  <- update_v_given_z(v, z, alpha)
pi <- stick_to_pi(v)
K  <- length(v)
```

The `update_v_given_z` function updates the stick-breaking weights based on the current cluster assignments, while `stick_to_pi` converts the stick-breaking weights into cluster probabilities. This ensures that the stick-breaking representation remains consistent with the current cluster structure throughout the MCMC sampling.

```
# Stick-breaking weight update function
update_v_given_z <- function(v, z, alpha) {
  K <- length(v); n_k <- tabulate(z, nbins = K); n_tail <-
     rev(cumsum(rev(n_k)))
  for (k in 1:K) {
    a <- 1 + n_k[k];
    b <- alpha + if (k < K) n_tail[k+1] else 0;
    v[k] <- rbeta(1, a, b)
  }
  v
}

# Convert stick weights to cluster probabilities
stick_to_pi <- function(v) {
  K <- length(v); pi <- numeric(K); cum <- 1
  for (k in 1:K) {
    pi[k] <- v[k] * cum;
    cum <- cum * (1 - v[k])
  }
  pi
}
```

The `update_v_given_z` function updates each stick weight v_k by sampling from a Beta distribution with parameters $a = 1 + n_k$ and $b = \alpha + \sum_{j=k+1}^{K} n_j$, where n_k is the number of curves assigned to cluster k. This ensures that the stick weights reflect the current cluster sizes while maintaining the Dirichlet Process structure. The `stick_to_pi` function converts the stick weights to cluster probabilities using the standard stick-breaking construction: $\pi_k = v_k \prod_{j=1}^{k-1}(1 - v_j)$.

6.6 Cluster Assignment Function

The core of the slice sampling implementation is the `assign_one` function, which determines cluster membership for individual curves:

```
assign_one <- function(i) {
  if (length(revealed_idx) && (i %in% revealed_idx) &&
      ( !unpin || (warmup_iters > 0 && iter <= warmup_iters) )) {
    return(1L)  # Force revealed curves to cluster 1 during warmup
  }
  S <- which(pi > u[i]); if (length(S) == 0) S <- 1L
  logw <- rep(-Inf, length(S))
  for (ss in seq_along(S)) {
    k <- S[ss]
    if (is.null(params[[k]]$mu_cached) ||
        is.null(params[[k]]$mu_cached_iter) ||
        params[[k]]$mu_cached_iter != iter) {
      mu_wave <- if (!is.null(params[[k]]$beta_ch) &&
          length(params[[k]]$beta_ch)) {
        compute_mu_from_beta(params[[k]]$beta_ch, wf, J, boundary, P)
      } else matrix(0, P, M)
```

```
      mu_k <- add_bias_to_mu(mu_wave, params[[k]]$bias)
      params[[k]]$mu_cached <- as_num_mat(mu_k)
      params[[k]]$mu_cached_iter <- iter
    }
    ll <- ll_curve_k(k, Y[[i]], params[[k]]$mu_cached)
    logw[ss] <- log(pi[k]) + ll
  }
  w <- exp(logw - max(logw)); w <- w / sum(w)
  S[sample.int(length(S), 1, prob = w)]
}
```

This function implements the core slice sampling logic for individual curve assignment. It first checks if the curve is a revealed curve that should be pinned during warmup. For regular curves, it identifies valid clusters (those with $\pi_k > u_i$), computes log-likelihoods for each valid cluster, and samples the assignment probabilistically. The function includes sophisticated caching of mean functions to avoid redundant computation across iterations.

6.7 Warm-up Period and Revealed Curves

The warm-up mechanism provides a structured approach to semi-supervised learning by controlling how revealed curves are handled during the initial iterations:

```
# Warm-up handling for revealed curves
if (iter == warmup_iters + 1 && length(revealed_idx) && warmup_iters > 0
    && unpin) {
  message("Warm-up ended: released revealed curves for regular DP
      assignments.")
}

# In assign_one function:
if (length(revealed_idx) && (i %in% revealed_idx) &&
    ( !unpin || (warmup_iters > 0 && iter <= warmup_iters) )) {
  return(1L)  # Force revealed curves to cluster 1 during warmup
}
```

The warm-up period serves several important purposes. During the first `warmup_iters` iterations, revealed curves are pinned to cluster 1, allowing the model to establish a stable baseline cluster structure. This prevents the Dirichlet Process from creating too many clusters early in the sampling when the model parameters are still being initialized. After the warm-up period ends, revealed curves are released and can be reassigned to any cluster based on the standard slice sampling mechanism. The `unpin` parameter controls whether revealed curves are ever released (if `unpin=TRUE`) or remain pinned throughout the entire sampling (if `unpin=FALSE`). This design allows for flexible semi-supervised learning scenarios where some curves have known cluster memberships that should either guide the initial learning or remain fixed throughout the analysis.

6.8 Wavelet Block Updates

For each cluster, the function calls the wavelet block update system:

```
# Update wavelet parameters for each cluster
for (k in 1:K) {
  idx_k <- which(z == k)
  if (length(idx_k) > 0) {
```

```
    # Extract wavelet coefficients for cluster k
    D_k <- stack_D_from_precomp(precomp_all, idx_k, M, bias_coeff)

    # Update wavelet parameters with Besov priors
    wpar_k <- update_cluster_wavelet_params_besov(
      idx_k, precomp_all, M, params[[k]]$wpar,
      params[[k]]$sigma2_m, params[[k]]$tau_sigma,
      kappa_pi, c2, tau_pi, ...
    )

    # Reconstruct mean functions
    mu_k <- compute_mu_from_beta(wpar_k$beta_ch, wf, J, boundary, P)
    params[[k]]$mu <- mu_k
  }
}
```

This step implements the core sparsity-inducing updates using the hierarchical Besov prior structure.

6.9 Kernel and ICM Updates

The kernel and ICM updates consist of a comprehensive update sequence for each cluster that includes Carlin-Chib kernel swapping followed by Metropolis-Hastings parameter updates:

```
# Combined kernel and ICM update sequence
for (k in 1:K) {
  idx <- which(z == k)
  if (!length(idx)) next

  # Extract curves for cluster k
  Yk <- lapply(idx, function(ii) Y[[ii]])

  # Step 1: Carlin-Chib kernel swapping
  params <- cc_switch_kernel_eig(k, params, kernels, t, Yk)

  # Step 2: Metropolis-Hastings parameter updates
  params <- mh_update_kernel_eig(k, params, kernels, t, Yk, a_eta, b_eta)
      # Kernel parameters
  params <- mh_update_L_eig(     k, params, kernels, t, Yk, mh_step_L)
          # L matrix
  params <- mh_update_eta_eig(   k, params, kernels, t, Yk, mh_step_eta,
     a_eta, b_eta)  # eta vector
  params <- mh_update_tauB_eig(  k, params, kernels, t, Yk, mh_step_tauB)
      # tau_B scalar
}
```

This combined update sequence ensures that each cluster undergoes both kernel selection and parameter optimization in a coordinated manner. The following sections break down each component of this update process.

6.9.1 Carlin-Chib Kernel Swapping

The Carlin-Chib kernel swapping step allows each cluster to switch between different kernel types using Bayesian model selection:

```
# Carlin-Chib kernel swapping for each cluster
for (k in 1:K) {
  if (length(which(z == k)) > 0) {
    Yk <- lapply(which(z == k), function(i) Y[[i]])
    params <- cc_switch_kernel_eig(k, params, kernels, t, Yk)
  }
}
```

The Carlin-Chib implementation computes weights for all available kernels:

```
# Inside cc_switch_kernel_eig function
Mmod <- length(kernels)
p_m <- rep(1 / Mmod, Mmod)  # Uniform prior over kernels
theta_draws <- vector("list", Mmod)

# Draw parameters for all kernels
for (m in 1:Mmod) {
  theta_draws[[m]] <- if (m == params[[k]]$kern_idx)
    params[[k]]$thetas[[m]] else kernels[[m]]$pstar()
}

# Compute log-weights for each kernel
logw <- rep(NA_real_, Mmod)
for (m in 1:Mmod) {
  kc_m <- kernels[[m]]; kp_m <- theta_draws[[m]]
  cache_m <- .build_icm_cache(t, kc_m, kp_m, params[[k]]$L,
                              params[[k]]$eta, params[[k]]$tau_B,
                                 params[[k]]$cache)
  ll_m <- sum_ll_curves(curves, cache_m)
  logw[m] <- log(p_m[m]) + ll_m
}

# Sample new kernel based on weights
w <- exp(logw - max(logw)); w <- w / sum(w)
new_idx <- sample.int(Mmod, 1, prob = w)
params[[k]]$kern_idx <- new_idx
```

6.9.2 Metropolis-Hastings Parameter Updates

After kernel selection, the parameters are updated using Metropolis-Hastings:

```
# Metropolis-Hastings updates for kernel parameters
for (k in 1:K) {
  if (length(which(z == k)) > 0) {
    Yk <- lapply(which(z == k), function(i) Y[[i]])

    # Update kernel parameters (l_scale, period, etc.)
    params <- mh_update_kernel_eig(k, params, kernels, t, Yk, a_eta,
       b_eta)

    # Update ICM parameters
    params <- mh_update_L_eig(k, params, kernels, t, Yk, mh_step_L)
    params <- mh_update_eta_eig(k, params, kernels, t, Yk, a_eta, b_eta)
```

```
    params <- mh_update_tauB_eig(k, params, kernels, t, Yk, mh_step_tauB)
  }
}
```

These updates maintain the intrinsic coregionalization structure while allowing both kernel selection and parameter adaptation to the data.

L Matrix Updates The `mh_update_L_eig` function updates the coregionalization matrix L using Metropolis-Hastings:

```
mh_update_L_eig <- function(k, params, kernels, t, Y_list, mh_step_L) {
  th  <- pack_L(params[[k]]$L)
  thp <- th + rnorm(length(th), 0, mh_step_L)
  Lp  <- unpack_L(thp, nrow(params[[k]]$L))
  curves <- lapply(Y_list, function(Yi) Yi - params[[k]]$mu_cached)
  kc <- kernels[[ params[[k]]$kern_idx ]]; kp <- params[[k]]$thetas[[
     params[[k]]$kern_idx ]]
  cache_cur <- .build_icm_cache(t, kc, kp, params[[k]]$L,
     params[[k]]$eta, params[[k]]$tau_B, params[[k]]$cache)
  ll_cur <- sum_ll_curves(curves, cache_cur)
  cache_prp <- .build_icm_cache(t, kc, kp, Lp, params[[k]]$eta,
     params[[k]]$tau_B, params[[k]]$cache)
  ll_prp <- sum_ll_curves(curves, cache_prp)
  lp_cur <- sum(dnorm(th,  0, 1, log=TRUE)); lp_prp <- sum(dnorm(thp, 0,
     1, log=TRUE))
  a <- (ll_prp + lp_prp) - (ll_cur + lp_cur)
  if (is.finite(a) && log(runif(1)) < a) {
    params[[k]]$L <- Lp; params[[k]]$cache <- cache_prp
    params[[k]]$acc$L["a"] <- params[[k]]$acc$L["a"] + 1
  }
  params[[k]]$acc$L["n"] <- params[[k]]$acc$L["n"] + 1
  params
}
```

This function proposes new values for the coregionalization matrix by adding Gaussian noise to the packed representation, then accepts or rejects based on the Metropolis-Hastings ratio comparing likelihood and prior terms.

Eta Vector Updates The `mh_update_eta_eig` function updates the channel-specific precision parameters:

```
mh_update_eta_eig <- function(k, params, kernels, t, Y_list, mh_step_eta,
   a_eta, b_eta) {
  curves <- lapply(Y_list, function(Yi) Yi - params[[k]]$mu_cached)
  kc <- kernels[[ params[[k]]$kern_idx ]]; kp <- params[[k]]$thetas[[
     params[[k]]$kern_idx ]]
  for (j in seq_along(params[[k]]$eta)) {
    cur <- params[[k]]$eta[j]
    prp <- rlnorm(1, log(cur), mh_step_eta); etap <- params[[k]]$eta;
       etap[j] <- prp
    cache_cur <- .build_icm_cache(t, kc, kp, params[[k]]$L,
       params[[k]]$eta,  params[[k]]$tau_B, params[[k]]$cache)
    cache_prp <- .build_icm_cache(t, kc, kp, params[[k]]$L, etap,
                      params[[k]]$tau_B, params[[k]]$cache)
```

```
    ll_cur <- sum_ll_curves(curves, cache_cur)
    ll_prp <- sum_ll_curves(curves, cache_prp)
    lp_cur <- invgamma::dinvgamma(cur, a_eta, b_eta, log=TRUE)
    lp_prp <- invgamma::dinvgamma(prp, a_eta, b_eta, log=TRUE)
    q_cgpr <- dlnorm(cur, log(prp), mh_step_eta, log=TRUE); q_prgc <-
       dlnorm(prp, log(cur), mh_step_eta, log=TRUE)
    a <- (ll_prp + lp_prp + q_cgpr) - (ll_cur + lp_cur + q_prgc)
    if (is.finite(a) && log(runif(1)) < a) {
      params[[k]]$eta <- etap; params[[k]]$cache <- cache_prp
      params[[k]]$acc$eta[j,"a"] <- params[[k]]$acc$eta[j,"a"] + 1
    }
    params[[k]]$acc$eta[j,"n"] <- params[[k]]$acc$eta[j,"n"] + 1
  }
  params
}
```

This function updates each channel's precision parameter using log-normal proposals and inverse gamma priors, with proper Metropolis-Hastings acceptance accounting for the asymmetric proposal distribution.

Tau_B Scalar Updates The `mh_update_tauB_eig` function updates the global precision parameter:

```
mh_update_tauB_eig <- function(k, params, kernels, t, Y_list,
   mh_step_tauB) {
  curves <- lapply(Y_list, function(Yi) Yi - params[[k]]$mu_cached)
  kc <- kernels[[ params[[k]]$kern_idx ]]; kp <- params[[k]]$thetas[[
     params[[k]]$kern_idx ]]
  cur <- params[[k]]$tau_B
  prp <- rlnorm(1, log(cur), mh_step_tauB)
  cache_cur <- .build_icm_cache(t, kc, kp, params[[k]]$L,
     params[[k]]$eta, cur, params[[k]]$cache)
  cache_prp <- .build_icm_cache(t, kc, kp, params[[k]]$L,
     params[[k]]$eta, prp, params[[k]]$cache)
  ll_cur <- sum_ll_curves(curves, cache_cur)
  ll_prp <- sum_ll_curves(curves, cache_prp)
  lp_cur <- -log1p(cur); lp_prp <- -log1p(prp)
  q_cgpr <- dlnorm(cur, log(prp), mh_step_tauB, log=TRUE); q_prgc <-
     dlnorm(prp, log(cur), mh_step_tauB, log=TRUE)
  a <- (ll_prp + lp_prp + q_cgpr) - (ll_cur + lp_cur + q_prgc)
  if (is.finite(a) && log(runif(1)) < a) {
    params[[k]]$tau_B <- prp; params[[k]]$cache <- cache_prp
    params[[k]]$acc$tauB["a"] <- params[[k]]$acc$tauB["a"] + 1
  }
  params[[k]]$acc$tauB["n"] <- params[[k]]$acc$tauB["n"] + 1
  params
}
```

This function updates the global precision parameter using log-normal proposals with a log-uniform prior, maintaining the intrinsic coregionalization structure while allowing the model to adapt the overall noise level.

6.10 Escobar-West Update for DP Concentration Parameter

The Escobar-West algorithm updates the Dirichlet Process concentration parameter α using auxiliary variables and a mixture of Gamma distributions:

```
# Escobar-West update for alpha
eta_aux <- rbeta(1, alpha + 1, N)
mix <- (alpha_prior[1] + Kocc - 1) /
  (N * (alpha_prior[2] - log(eta_aux)) + alpha_prior[1] + Kocc - 1)
if (runif(1) < mix) {
  alpha <- rgamma(1, alpha_prior[1] + Kocc, alpha_prior[2] - log(eta_aux))
} else {
  alpha <- rgamma(1, alpha_prior[1] + Kocc - 1, alpha_prior[2] -
     log(eta_aux))
}
```

The Escobar-West algorithm works by introducing an auxiliary variable $\eta_{\text{aux}} \sim \text{Beta}(\alpha + 1, N)$ and computing a mixture weight p based on the prior parameters and current number of occupied clusters K_{occ}. The algorithm then samples α from one of two Gamma distributions: $\text{Gamma}(a_0 + K_{\text{occ}}, b_0 - \log(\eta_{\text{aux}}))$ with probability p, or $\text{Gamma}(a_0 + K_{\text{occ}} - 1, b_0 - \log(\eta_{\text{aux}}))$ with probability $1-p$. This approach maintains the conjugacy of the Gamma prior while allowing the concentration parameter to adapt to the current clustering structure.

7 Postprocessing

7.1 Overview

The postprocessing system (`R/postprocessing.R`) provides methods for extracting meaningful results from MCMC samples. The two main approaches are Dahl's method for finding a representative partition and the maximum a posteriori (MAP) estimate for the most frequent partition.

7.2 Mathematical Foundation

7.2.1 Dahl's Method

Dahl's method finds the partition that minimizes the expected squared loss with respect to the posterior distribution. For MCMC samples $\{z^{(s)}\}_{s=1}^{S}$ where $z^{(s)} = (z_1^{(s)}, \ldots, z_N^{(s)})$ represents the cluster assignments in sample s, the method works as follows:

1. **Posterior Similarity Matrix (PSM):** Compute the posterior probability that observations i and j are in the same cluster:

$$\text{PSM}_{ij} = \frac{1}{S} \sum_{s=1}^{S} \mathbf{1}[z_i^{(s)} = z_j^{(s)}] \tag{1}$$

2. **Dahl's Loss Function:** For each sample s, compute the squared Frobenius norm between the sample's adjacency matrix and the PSM:

$$L^{(s)} = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(\mathbf{1}[z_i^{(s)} = z_j^{(s)}] - \text{PSM}_{ij} \right)^2 \tag{2}$$

3. **Representative Partition:** Select the sample that minimizes the loss:

$$\hat{s} = \arg\min_{s=1,\ldots,S} L^{(s)} \tag{3}$$

The resulting partition $\hat{z} = z^{(\hat{s})}$ minimizes the expected squared loss and provides a representative clustering that best captures the posterior uncertainty structure.

7.2.2 MAP Estimation

The maximum a posteriori (MAP) estimate finds the most frequently observed partition in the MCMC samples:

1. **Canonical Labeling:** Convert each partition $z^{(s)}$ to canonical form using relabeling to ensure consistent representation across samples.

2. **Frequency Counting:** Count the frequency of each unique partition:

$$f(z) = \frac{1}{S}\sum_{s=1}^{S} \mathbf{1}[z^{(s)} \sim z] \tag{4}$$

where $z^{(s)} \sim z$ indicates that $z^{(s)}$ and z represent the same partition (up to label permutation).

3. **MAP Partition:** Select the most frequent partition:

$$\hat{z}_{\text{MAP}} = \arg\max_{z} f(z) \tag{5}$$

7.3 Implementation

7.3.1 `dahl_partition()`

This function finds the representative partition using Dahl's method, which minimizes the expected squared loss with respect to the posterior distribution by computing the posterior similarity matrix and selecting the sample that best matches it.

```
dahl_partition <- function(Z) {
  if (is.null(dim(Z)) || length(dim(Z)) != 2) stop("Z must be an S x N
     matrix of labels.")
  S <- nrow(Z); N <- ncol(Z); PSM <- matrix(0, N, N)
  for (s in 1:S) {
    zs <- Z[s, ]; A <- outer(zs, zs, FUN = "==") * 1L; PSM <- PSM + A
  }
  PSM <- PSM / S; score <- numeric(S)
  for (s in 1:S) {
    zs <- Z[s, ]; A <- outer(zs, zs, FUN = "==") * 1L; score[s] <- sum((A
       - PSM)^2)
  }
  s_hat <- which.min(score); z_hat <- as.integer(factor(Z[s_hat, ]))
  list(z_hat = z_hat, K_hat = length(unique(z_hat)), s_hat = s_hat, PSM =
     PSM, score = score)
}
```

7.3.2 `map_partition()`

This function finds the maximum a posteriori (MAP) partition by identifying the most frequently observed clustering pattern across all MCMC samples, using canonical labeling to handle label permutations.

```
map_partition <- function(Z) {
  if (is.null(dim(Z)) || length(dim(Z)) != 2) stop("Z must be an S x N
     matrix of labels.")
  S <- nrow(Z); N <- ncol(Z); keys <- character(S)
  for (s in 1:S) keys[s] <- paste(.canon_labels(Z[s, ]), collapse = "-")
  tab <- table(keys); key_hat <- names(which.max(tab)); s_hat <-
     which(keys == key_hat)[1]
  z_hat <- .canon_labels(Z[s_hat, ]); K_hat <- length(unique(z_hat))
  list(z_hat = z_hat, K_hat = K_hat, s_hat = s_hat, key_hat = key_hat,
     freq = max(tab))
}
```