Package 'DiscreteMorseR'

July 16, 2026

Type Package

Title Discrete Morse Theory for 3D Meshes Derived from Point Clouds

Version 1.0.0

Date 2026-07-07

Description Ultra-fast computation of discrete Morse (Marston Morse) gradient vector fields and critical simplices (0-simplices: vertices, 1-simplices: edges, 2-simplices: faces) on 3D triangular meshes from point clouds. Provides C++ backend with parallel processing for large-scale topological analysis, including connected component analysis and visualization tools. Perfect for Light Detection and Ranging ('LiDAR') data, computational topology, and geometric analysis applications. The implementation follows Forman (1998) <doi:10.1007/PL00009307> for discrete Morse theory, with extensions for 3D mesh processing.

Language en-US

License MIT + file LICENSE

URL https://github.com/DijoG/DiscreteMorseR

BugReports https://github.com/DijoG/DiscreteMorseR/issues

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.10), dplyr (>= 1.0.0), purrr (>= 0.3.0), stringr (>= 1.4.0), data.table (>= 1.12.0), gtools (>= 3.8.0), readr (>= 1.3.0), future (>= 1.18.0), furrr (>= 0.2.0)

Suggests clustermq (>= 0.8.0), ggplot2 (>= 3.3.0), testthat (>= 3.0.0), knitr (>= 1.30), rmarkdown (>= 2.0), tictoc

LinkingTo Rcpp, RcppArmadillo

SystemRequirements C++17

Encoding UTF-8

RoxygenNote 7.3.1

Config/testthat/edition 3

NeedsCompilation yes

Author Gergo Dioszegi [aut, cre, cph] (ORCID:
<https://orcid.org/0009-0003-3454-9093>)

Maintainer Gergo Dioszegi <dijogergo@gmail.com>

Repository CRAN

Date/Publication 2026-07-16 12:50:40 UTC

Contents

`compute_lowerSTAR_parallel` *Compute lower star in parallel*

Description

Compute lower star in parallel

Usage

```
compute_lowerSTAR_parallel(
  vertex,
  edge,
  face,
  output_dir = NULL,
  cores = NULL,
  batch_size = NULL
)
```

Arguments

`vertex`	Vertex data
`edge`	Edge data
`face`	Face data
`output_dir`	Output directory
`cores`	Number of cores (default: available cores-1)
`batch_size`	Number of vertices per batch

Value

A list of lower star sets, one for each vertex. Each element contains:

`id`	Vertex ID
`lexi_label`	Lexicographically sorted simplex label
`lexi_id`	Lexicographically sorted simplex IDs

Examples

```
# Small example for automatic testing
# Create synthetic mesh
set.seed(456)
n_verts <- 50

# Generate random vertices
vertices <- matrix(rnorm(n_verts * 3), ncol = 3)
colnames(vertices) <- c("X", "Y", "Z")

# Create edges (connect each vertex to 3-5 random others)
edges_list <- list()
for (i in 1:n_verts) {
  n_edges <- sample(3:5, 1)
  neighbors <- sample(setdiff(1:n_verts, i), min(n_edges, n_verts-1))
  for (j in neighbors) {
    edges_list[[length(edges_list) + 1]] <- c(min(i, j), max(i, j))
  }
}
edges <- unique(do.call(rbind, edges_list))
colnames(edges) <- c("i1", "i2")

# Create faces (triangulate using edge connections)
faces_list <- list()
for (i in 1:n_verts) {
  connected <- unique(c(
    edges[edges[,1] == i, 2],
    edges[edges[,2] == i, 1]
  ))
  if (length(connected) >= 3) {
    for (j in 1:(min(length(connected), 4) - 1)) {
      for (k in (j+1):(min(length(connected), 4))) {
        if (length(faces_list) < n_verts * 2) {
          faces_list[[length(faces_list) + 1]] <- c(i, connected[j], connected[k])
        }
      }
    }
  }
}
faces <- unique(do.call(rbind, faces_list))
colnames(faces) <- c("i1", "i2", "i3")

# Create mesh object
mesh <- list(
```

```
  vertices = vertices,
  faces = faces,
  edges = data.frame(edges)
)
attr(mesh, "input_truth") <- 1:n_verts

# Compute Morse complex sequentially (for CRAN checks)
result_seq <- compute_MORSE_complex(mesh, parallel = FALSE)

# Results
str(result_seq)


# Parallel computation (if clustermq is available)
if (requireNamespace("clustermq", quietly = TRUE)) {
  # Compute in parallel with 2 cores
  result_par <- compute_MORSE_complex(
    mesh,
    output_dir = tempdir(),
    parallel = TRUE,
    cores = 2
  )

  cat("Parallel results:\n")
  cat("  Critical simplices:", length(result_par$critical), "\n")
  cat("  Gradient pairs:", length(result_par$vector_field), "\n")

  # Verify results match
  if (length(result_seq$critical) == length(result_par$critical) &&
      length(result_seq$vector_field) == length(result_par$vector_field)) {
    cat("[OK] Sequential and parallel results match!\n")
  }
}
```

`compute_MORSE_complex` *Compute Morse complex from mesh*

Description

Compute Morse complex from mesh

Usage

```
compute_MORSE_complex(
  mesh,
  output_dir = NULL,
  parallel = TRUE,
  cores = 4,
  batch_size = NULL
)
```

Arguments

mesh	Mesh object from get_CCMESH()
output_dir	Optional directory to save results. If provided, writes: - vertices.txt, edges.txt, faces.txt: Mesh simplices - vector_field.txt: Gradient vector field pairs - critical_simplices.txt: Critical simplices - lowerSTAR.txt: Lower star filtration data
parallel	Whether to use parallel processing (default: TRUE)
cores	Number of cores for parallel processing (default: 4)
batch_size	Number of vertices per batch in parallel processing

Value

A list with three components:

vector_field	Character vector of gradient pairs in format "from:to"
critical	Character vector of critical simplices (vertices, edges, faces)
simplices	List containing vertices, edges, and faces data frames

Examples

```
# Create a tetrahedron mesh
vertices <- matrix(c(0,0,0, 1,0,0, 0,1,0, 0,0,1), ncol=3, byrow=TRUE)
colnames(vertices) <- c("X", "Y", "Z")
faces <- matrix(c(1,2,3, 1,2,4, 1,3,4, 2,3,4), ncol=3, byrow=TRUE)
colnames(faces) <- c("i1", "i2", "i3")

# Extract unique edges from faces
all_edges <- rbind(faces[,c(1,2)], faces[,c(1,3)], faces[,c(2,3)])
unique_edges <- unique(t(apply(all_edges, 1, sort)))
edges <- data.frame(i1 = unique_edges[,1], i2 = unique_edges[,2])

# Create mesh object
mesh <- list(vertices = vertices, faces = faces, edges = edges)
attr(mesh, "input_truth") <- 1:nrow(vertices)

# Compute Morse complex (sequential mode for CRAN checks)
result <- compute_MORSE_complex(mesh, parallel = FALSE)

# View results
print(paste("Critical simplices:", length(result$critical)))
print(paste("Gradient pairs:", length(result$vector_field)))
```

---

get_CCMESH *Ultra-fast mesh preparation with connected components*

Description

Ultra-fast mesh preparation with connected components

Usage

```
get_CCMESH(alphahull, select_largest = TRUE)
```

Arguments

alphahull	Alphahull generated by ahull3D::ahull3d()
select_largest	If TRUE, returns only largest connected component (default: TRUE) If FALSE, returns list of all connected components

Value

If select_largest = TRUE: A mesh list with components:

vertices	Nx3 matrix of vertex coordinates (X, Y, Z)
faces	Mx3 matrix of face indices (i1, i2, i3)
edges	Kx2 data frame of edge indices (i1, i2)

with attribute "input_truth" containing propagated labels. If select_largest = FALSE: A list of mesh objects (same structure as above) with attributes "n_components" and "component_sizes".

Examples

```
# Small example for automatic testing (runs in < 5 seconds)
# Create a synthetic alpha hull object mimicking ahull3D output
set.seed(123)
n_vertices <- 40

# Generate random points on a sphere surface (like alpha hull)
theta <- runif(n_vertices, 0, 2*pi)
phi <- acos(runif(n_vertices, -1, 1))
r <- 2 + runif(n_vertices, -0.5, 0.5)  # Slight radial variation

# Convert to Cartesian coordinates
vertices <- cbind(
  r * sin(phi) * cos(theta),
  r * sin(phi) * sin(theta),
  r * cos(phi)
)
colnames(vertices) <- c("X", "Y", "Z")

# Create a triangulation (simulating alpha hull faces)
# Use a simple approach: connect nearby points
dist_matrix <- as.matrix(dist(vertices))
faces <- matrix(NA, nrow = n_vertices * 2, ncol = 3)
face_count <- 0

for (i in 1:n_vertices) {
  # Find nearest neighbors
  neighbors <- order(dist_matrix[i, ])[2:min(6, n_vertices)]
  for (j in 1:(length(neighbors) - 1)) {
    for (k in (j+1):length(neighbors)) {
```

```
      if (face_count < n_vertices * 2) {
        face_count <- face_count + 1
        faces[face_count, ] <- c(i, neighbors[j], neighbors[k])
      }
    }
  }
}
faces <- unique(faces[!is.na(faces[,1]), ])
colnames(faces) <- c("i1", "i2", "i3")

# Create alpha hull-like object with mesh3d structure
alpha_hull <- list(
  vb = t(cbind(vertices, 1)),  # 4xN matrix with homogeneous coordinates
  it = t(faces),                # 3xM matrix of faces
  normals = t(cbind(vertices / sqrt(rowSums(vertices^2)), 1))  # Approximate normals
)
class(alpha_hull) <- "mesh3d"

# Add attributes that ahull3D would provide
input_truth <- sample(1:20, n_vertices, replace = TRUE)
attr(alpha_hull, "input_truth") <- input_truth

# Create face_truth by propagating vertex labels
face_truth <- matrix(input_truth[faces], nrow = 3, byrow = FALSE)
attr(alpha_hull, "face_truth") <- face_truth
attr(alpha_hull, "alpha") <- 0.5

# Verify structure matches ahull3D output
str(alpha_hull)

# Extract mesh using get_CCMESH
mesh <- get_CCMESH(alpha_hull, select_largest = TRUE)

# Check results
print(paste("Vertices:", nrow(mesh$vertices),
            "Faces:", nrow(mesh$faces),
            "Edges:", nrow(mesh$edges)))
print(paste("Input truth propagated:",
            length(attr(mesh, "input_truth")), "labels"))


# Larger example with more complex structure
set.seed(456)
n_vertices <- 100

# Generate points on multiple surfaces (simulating alpha hull with components)
theta <- runif(n_vertices, 0, 2*pi)
phi <- acos(runif(n_vertices, -1, 1))
r <- 3 + runif(n_vertices, -1, 1)

vertices <- cbind(
  r * sin(phi) * cos(theta),
  r * sin(phi) * sin(theta),
```

```
  r * cos(phi)
)
colnames(vertices) <- c("X", "Y", "Z")

# Create more complex triangulation
dist_matrix <- as.matrix(dist(vertices))
faces <- matrix(NA, nrow = n_vertices * 3, ncol = 3)
face_count <- 0

for (i in 1:n_vertices) {
  neighbors <- order(dist_matrix[i, ])[2:min(8, n_vertices)]
  for (j in 1:(length(neighbors) - 1)) {
    for (k in (j+1):length(neighbors)) {
      if (face_count < n_vertices * 3) {
        face_count <- face_count + 1
        faces[face_count, ] <- c(i, neighbors[j], neighbors[k])
      }
    }
  }
}
faces <- unique(faces[!is.na(faces[,1]), ])
colnames(faces) <- c("i1", "i2", "i3")

alpha_hull <- list(
  vb = t(cbind(vertices, 1)),
  it = t(faces),
  normals = t(cbind(vertices / sqrt(rowSums(vertices^2)), 1))
)
class(alpha_hull) <- "mesh3d"

input_truth <- sample(1:30, n_vertices, replace = TRUE)
attr(alpha_hull, "input_truth") <- input_truth
face_truth <- matrix(input_truth[faces], nrow = 3, byrow = FALSE)
attr(alpha_hull, "face_truth") <- face_truth
attr(alpha_hull, "alpha") <- 0.8

# Extract mesh with all components
mesh <- get_CCMESH(alpha_hull, select_largest = FALSE)
print(paste("Number of connected components:",
            attr(mesh, "n_components")))
print("Component sizes:")
print(attr(mesh, "component_sizes"))
```

`visualize_MORSE_2d` *Fast Morse complex visualization using get_simplexCENTER()*

Description

Fast Morse complex visualization using get_simplexCENTER()

Usage

```
visualize_MORSE_2d(
  morse_complex,
  projection = "XZ",
  point_alpha = 0.6,
  point_size = 1,
  max_points = 30000,
  plot_gradient = TRUE,
  plot_critical = TRUE
)
```

Arguments

morse_complex	Output from compute_MORSE_complex()
projection	Projection plane: "XY", "XZ", or "YZ" (default: "XZ")
point_alpha	Point transparency (default: 0.6)
point_size	Point size (default: 1)
max_points	Maximum points to plot per category (default: 30000)
plot_gradient	Whether to plot gradient arrows (default: TRUE)
plot_critical	Whether to plot critical points (default: TRUE)

Value

A ggplot2 object showing the Morse complex in 2D projection. The plot displays: - Grey arrows: Gradient vector field pairs - Red circles: Critical vertices (0-simplices) - Blue plus signs: Critical edges (1-simplices) - Green diamonds: Critical faces (2-simplices)

Examples

```
# Simple example (runs in < 5 seconds)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  set.seed(123)
  vertices <- matrix(rnorm(30), ncol = 3)
  colnames(vertices) <- c("X", "Y", "Z")

  edges <- data.frame(
    i1 = sample(1:10, 15, replace = TRUE),
    i2 = sample(1:10, 15, replace = TRUE)
  )
  edges <- unique(edges[edges$i1 != edges$i2, ])

  faces <- data.frame(
    i1 = sample(1:10, 10, replace = TRUE),
    i2 = sample(1:10, 10, replace = TRUE),
    i3 = sample(1:10, 10, replace = TRUE)
  )
  faces <- unique(faces[faces$i1 != faces$i2 &
                        faces$i2 != faces$i3 &
```

```
                        faces$i1 != faces$i3, ])

  mesh <- list(vertices = vertices, faces = as.matrix(faces), edges = edges)
  attr(mesh, "input_truth") <- 1:nrow(vertices)

  result <- compute_MORSE_complex(mesh, parallel = FALSE)
  p <- visualize_MORSE_2d(result, plot_gradient = FALSE)
  print(p)
}


# Larger example with gradient arrows
if (requireNamespace("ggplot2", quietly = TRUE)) {
  set.seed(456)
  n_verts <- 80
  vertices <- matrix(rnorm(n_verts * 3), ncol = 3)
  colnames(vertices) <- c("X", "Y", "Z")

  edges <- data.frame(
    i1 = sample(1:n_verts, 40, replace = TRUE),
    i2 = sample(1:n_verts, 40, replace = TRUE)
  )
  edges <- unique(edges[edges$i1 != edges$i2, ])

  faces <- data.frame(
    i1 = sample(1:n_verts, 30, replace = TRUE),
    i2 = sample(1:n_verts, 30, replace = TRUE),
    i3 = sample(1:n_verts, 30, replace = TRUE)
  )
  faces <- unique(faces[faces$i1 != faces$i2 &
                        faces$i2 != faces$i3 &
                        faces$i1 != faces$i3, ])

  mesh <- list(vertices = vertices, faces = as.matrix(faces), edges = edges)
  attr(mesh, "input_truth") <- 1:nrow(vertices)

  result <- compute_MORSE_complex(mesh, parallel = FALSE)
  p <- visualize_MORSE_2d(result, plot_gradient = TRUE)
  print(p)
}
```

Index