Package: haze (via r-universe)

November 3, 2024

Type Package

Title Smoothing of per-Vertex Data on Triangular Meshes

Version 0.2.0

Maintainer Tim Schäfer <ts+code@rcmd.org>

Description Smoothing of per-vertex data on triangular meshes, sub mesh creation based on vertex indices, per-vertex data interpolation based on k-d trees.

License MIT + file LICENSE

Encoding UTF-8

URL https://github.com/dfsp-spirit/haze

BugReports https://github.com/dfsp-spirit/haze/issues

SystemRequirements C++11

Imports freesurferformats (>= 0.1.17), Rcpp (>= 1.0.6), Rvcg (>= 0.20.2)

Suggests knitr, rmarkdown, testthat (>= 3.0.0), rgl, fsbrain (>= 0.5.3)

VignetteBuilder knitr

RoxygenNote 7.1.2

Config/testthat/edition 3

LinkingTo Rcpp

Repository https://dfsp-spirit.r-universe.dev

RemoteUrl https://github.com/dfsp-spirit/haze

RemoteRef HEAD

RemoteSha dfccb658d0772f7e6d755ab0e4a29354592cddd0

Contents

haze 2
linear_interpolate_kdtree 2
mesh.adj 3
mesh.neigh.pre 4
nn_interpolate_kdtree 4
pervertexdata.smoothnn 5
pervertexdata.smoothnn.adj 6
read.vv 7
submesh.vertex 8

Index **9**

haze — *haze: Smoothing of per-vertex data on triangular meshes*

Description

Smoothing of per-vertex data on triangular meshes

linear_interpolate_kdtree — *Interpolate per-vertex data at the query points. Or map per-vertex data between subjects.*

Description

This method uses inverse distance weight interpolation within a triangle. First, the face of the `mesh` that the `query_coordinate` falls into is determined. Then results in 3 vertices with respective per-vertex data, and a query coordinate. We then compute the distance to all 3 vertices, and perform inverse distance weight interpolation with a beta setting defined by parameter `iwd_beta`.

Usage

```
linear_interpolate_kdtree(
  query_coordinates,
  mesh,
  pervertex_data,
  iwd_beta = 2,
  ...
)
```

Arguments

`query_coordinates`
: nx3 numerical matrix of x,y,z coordinates. These are typically the vertex positions of a second (spherical!) mesh for that you need per-vertex data (e.g., the `fsaverage6` mesh).

`mesh`
: fs.surface instance, see `read.fs.surface` or `subject.surface` to get one, or turn an `rgl` `tmesh` into one with `tmesh3d.to.fs.surface`.

`pervertex_data`
: numerical vector, the continuous per-vertex data for the vertices of the mesh.

`iwd_beta`
: scalar double, the `beta` parameter for the inverse distance weight interpolation with the triangle. See details.

`...`
: ignore, passed on to internal function `interp_tris`.

Value

named list with entries: 'interp_values', the numerical vector of interpolated data at the query_coordinates. 'nearest_vertex_in_face' the nearest vertex in the face that the respective query coordinate falls into, 'nearest_face' the index of the nearest face that the respective query coordinate falls into.

Note

The mesh must be spherical, and the `query_coordinates` must also be located on the mesh sphere.

`mesh.adj` *Compute vertex neighborhoods for a mesh.*

Description

Compute vertex neighborhoods for a mesh.

Usage

```
mesh.adj(surface, k = 1L)
```

Arguments

`surface`
: a mesh, represented as an `fs.surface` instance from the `freesurferformats` package or a `tmesh3d` instance from `rgl`, or a character string representing the path of a mesh to load with `freesurferformats::read.fs.surface`.

`k`
: scalar positive integer, the k value for the k-ring neighborhood. For k=1, this function computes the adjacency list representation of the graph (where the neighbors include the vertex itself).

Value

list of integer vectors, the neighborhood data

Examples

```
## Not run:
mesh = rgl::tetrahedron3d();
mesh_adj = mesh.adj(mesh, k = 1L);

## End(Not run)
```

mesh.neigh.pre *Return pre-computed neighborhood data for specific meshes.*

Description

Return pre-computed neighborhood data for specific meshes.

Usage

```
mesh.neigh.pre(meshname)
```

Arguments

meshname	a text identifier specifying the mesh you want connectivity data for. Currently supported meshes are listed here. 'lh_fsaverage': the left hemisphere of the FreeSurfer 6 fsaverage template. 'rh_fsaverage': the right hemisphere of the FreeSurfer 6 fsaverage template. 'lh_fsaverage6': the left hemisphere of the FreeSurfer 6 fsaverage6 template. 'rh_fsaverage6': the right hemisphere of the FreeSurfer 6 fsaverage6 template.

Value

list of vectors, the connectivity data as an adjacency list. The outer list has length n, where n is the number of vertices in the graph. The inner lists represent, for each vertex, all of its neighbors.

nn_interpolate_kdtree *Get per-vertex data at vertices closest to the given query coordinates on the mesh.*

Description

Return per-vertex data at the vertices closest to the given query points.

Usage

```
nn_interpolate_kdtree(query_coordinates, mesh, pervertex_data)
```

Arguments

query_coordinates
: nx3 numerical matrix of x,y,z coordinates. These are typically the vertex positions of a second (spherical!) mesh for that you need per-vertex data (e.g., the `fsaverage6` mesh).

mesh
: fs.surface instance, see `read.fs.surface` or `subject.surface` to get one, or turn an `rgl` `tmesh` into one with `tmesh3d.to.fs.surface`.

pervertex_data
: numerical vector, the continuous per-vertex data for the vertices of the mesh.

Value

the per-vertex data for the vertices closest to the query coordinates.

`pervertexdata.smoothnn`

Smooth per-vertex data based on mesh.

Description

Smooth per-vertex data based on mesh.

Usage

```
pervertexdata.smoothnn(surface, pvdata, num_iter, k = 1L, method = "C++")
```

Arguments

surface
: a mesh, represented as an `fs.surface` instance from the `freesurferformats` package or a `tmesh3d` instance from `rgl`, or a character string representing the path of a mesh to load with `freesurferformats::read.fs.surface`.

pvdata
: numerical vector of per-vertex-data for the mesh, one value per vertex. Data values of `NA` will be ignored, allowing you to mask parts of the data. If you pass an `n` x `m` matrix or data.frame, the `n` rows will be treated as (independent) overlays that should be smoothed in parallel. To set the number of cores to use for parallel processing, set the 'mc_cores' option like this: `options("mc.cores"=22L);` before calling this function. Data.frames and matrices with a single row will be converted to vectors, and the return value will be a vector in that case.

num_iter
: positive integer, number of smoothing iterations.

k
: scalar positive integer, the k value for the k-ring neighborhood. For k=1, this function computes the adjacency list representation of the graph (where the neighbors include the vertex itself).

method
: character string, one of 'C++' or 'R'. The C++ version is much faster (about 30 times faster on our test machine), and there is little reason to ever use the R version in production code, so just ignore this.

Value

numerical vector, the smoothed data.

See Also

pervertexdata.smoothnn.adj if you already have pre-computed adjacency data for the mesh. Using that data can increase performance considerably, especially if you need to smooth many data sets.

Examples

```
## Not run:
mesh = rgl::tetrahedron3d();
pvd = rnorm(nrow(mes2$vb), mean = 5.0, sd = 1.0);
pvd_smoothed = pervertexdata.smoothnn(mesh, pvd, num_iter = 30L);

## End(Not run)
```

---

pervertexdata.smoothnn.adj

Smooth per-vertex data using nearest-neighbor smoothing based on mesh adjacency information.

Description

Smooth per-vertex data using nearest-neighbor smoothing based on mesh adjacency information.

Usage

```
pervertexdata.smoothnn.adj(
  mesh_adj,
  pvdata,
  num_iter,
  method = "C++",
  silent = getOption("haze.silent", default = TRUE)
)
```

Arguments

mesh_adj — list of vectors of integers, the adjacency list representation of the mesh. One can use the pre-computed adjacency for some special meshes, see mesh.neigh.pre. Data for vertices should include the vertex itself.

pvdata — numerical vector of per-vertex-data for the mesh, one value per vertex. Data values of NA will be ignored, allowing you to mask parts of the data. If you pass an n x m matrix or data.frame, the n rows will be treated as (independent) overlays that should be smoothed in parallel. To set the number of cores to use for parallel processing, set the 'mc_cores' option like this: `options("mc.cores"=22L);` before calling this function. Data.frames and matrices with a single row will be converted to vectors, and the return value will be a vector in that case.

`num_iter` positive integer, number of smoothing iterations.

`method` character string, one of 'C++' or 'R'. The C++ version is much faster (about 30 times faster on our test machine), and there is little reason to ever use the R version in production code, so just ignore this.

`silent` logical, whether to suppress output messages.

Value

numerical vector, the smoothed data (for vector input). If `pvdata` is a matrix or a data.frame (with more than a single column), the result is also a matrix or data.frame.

See Also

`pervertexdata.smoothnn` if you have a mesh and still need the connectivity to be computed.

Examples

```
## Not run:
mesh = rgl::tetrahedron3d();
mesh_adj = mesh.adj(mesh, k = 1L);
pvd = rnorm(nrow(mesh$vb), mean = 5.0, sd = 1.0);
pvd_smoothed = pervertexdata.smoothnn.adj(mesh_adj, pvd, num_iter = 30L);

## End(Not run)
```

read.vv *Read vv binary file.*

Description

Read matrix-like data from vv files. This is a custom format from the cpp_geodesic repo, designed to allow fast reading of vector-of-vectors data. The format does NOT require that all inner vectors have the same length, so it is NOT limited to matrices. The format is designed for storing graphs as adjacency lists.

Usage

```
read.vv(filepath)
```

Arguments

`filepath` string. Full path to the input vv file.

Value

list of vectors, the data. The vv files may can store double or int, which is encoded in the file header and used accordingly.

`submesh.vertex` *Create a submesh including only the given vertices.*

Description

Create a submesh including only the given vertices.

Usage

```
submesh.vertex(surface_mesh, old_vertex_indices_to_use, ret_mappings = FALSE)
```

Arguments

`surface_mesh` an `fs.surface` instance, the original mesh. See `read.fs.surface` or `subject.surface` to get one. Can also be an rgl tmesh, see `tmesh3d`.

`old_vertex_indices_to_use` integer vector, the vertex indices of the 'surface_mesh' that should be used to construct the new sub mesh.

`ret_mappings` whether to return the vertex mappings. If `TRUE`, the return value becomes a list with entries 'submesh', 'vmap_full_to_submesh', and 'vmap_submesh_to_full'.

Value

the new mesh, made up of the given 'old_vertex_indices_to_use' and all (complete) faces that exist between the query vertices in the source mesh. But see 'ret_mapping' parameter.

Examples

```
## Not run:
if(requireNamespace("fsbrain, quietly=T")) {
sjd = fsbrain::fsaverage.path(T);
sj = "fsaverage";
mesh = fsbrain::subject.surface(sjd, sj, hemi="lh");
lab = fsbrain::subject.label(sjd, sj, "cortex", hemi = "lh");
sm = submesh.vertex(mesh, lab);
fsbrain::vis.fs.surface(mesh); # show the full mesh.
fsbrain::vis.fs.surface(sm);   # show only the cortex.
}
## End(Not run)
```

Index

haze, 2

linear_interpolate_kdtree, 2

mesh.adj, 3
mesh.neigh.pre, 4, *6*

nn_interpolate_kdtree, 4

pervertexdata.smoothnn, 5, *7*
pervertexdata.smoothnn.adj, *6*, 6

read.fs.surface, *3*, *5*, *8*
read.vv, 7

subject.surface, *3*, *5*, *8*
submesh.vertex, 8

tmesh3d, *8*
tmesh3d.to.fs.surface, *3*, *5*