Package ‘freesurferformats’

February 2, 2024

Type Package

Title Read and Write 'FreeSurfer' Neuroimaging File Formats

Version 0.1.18

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

Description Provides functions to read and write neuroimaging data in various file formats, with a focus on 'FreeSurfer' <`http://freesurfer.net/`> formats. This includes, but is not limited to, the following file formats: 1) MGH/MGZ format files, which can contain multi-dimensional images or other data. Typically they contain time-series of three-dimensional brain scans acquired by magnetic resonance imaging (MRI). They can also contain vertex-wise measures of surface morphometry data. The MGH format is named after the Massachusetts General Hospital, and the MGZ format is a compressed version of the same format. 2) 'FreeSurfer' morphometry data files in binary 'curv' format. These contain vertex-wise surface measures, i.e., one scalar value for each vertex of a brain surface mesh. These are typically values like the cortical thickness or brain surface area at each vertex. 3) Annotation file format. This contains a brain surface parcellation derived from a cortical atlas. 4) Surface file format. Contains a brain surface mesh, given by a list of vertices and a list of faces.

License MIT + file LICENSE

Encoding UTF-8

URL `https://github.com/dfsp-spirit/freesurferformats`

BugReports `https://github.com/dfsp-spirit/freesurferformats/issues`

Imports utils, pkgfilecache (>= 0.1.1), xml2

Suggests knitr, rmarkdown, curl, testthat (>= 2.1.0), oro.nifti (>= 0.9), gifti (>= 0.7.5), cifti (>= 0.4.5)

VignetteBuilder knitr

RoxygenNote 7.2.3

NeedsCompilation no

Author Tim Schäfer [aut, cre] (<`https://orcid.org/0000-0002-3683-8070`>)

Repository CRAN

Date/Publication 2024-02-02 22:40:02 UTC

R topics documented:

`annot.max.region.idx` *Get max region index of an fs.annot instance.*

Description

Get max region index of an fs.annot instance.

Usage

```
annot.max.region.idx(annot)
```

Arguments

`annot`	fs.annot instance

Value

integer, the max region index. They typically start with 0 and are consecutive, but this is not enforced or checked in any way.

Note

This is a helper function to be used with `annot.unique`, see the example there.

`bvsmp` *Create new bvsmp instance encoding morph data for Brainvoyager.*

Description

Create new bvsmp instance encoding morph data for Brainvoyager.

Usage

```
bvsmp(morph_data)
```

Arguments

`morph_data`	numeric vector, the morphometry data to store in the bvsmp instance (one value per mesh vertex).

Value

bvsmp instance, can be used to write Brainvoyager SMP format morphometry files using `write.smp.brainvoyager`. Modify as needed before writing.

Examples

```
morph_data = rnorm(100L, 3.0, 1.0);
mybvsmp = bvsmp(morph_data);
mybvsmp$smp_version;
```

`cdata` *Create CDATA element string from string.*

Description

Create CDATA element string from string.

Usage

```
cdata(string)
```

Arguments

`string`	character string, the input string, freeform text. Must not contain the cdata start and end tags.

Value

character string, the input wrapped in the cdata tags

Note

This returns a string, not an XML node. See `xml_cdata` if you want a node.

`closest.vert.to.point` *Find vertex index closest to given query coordinate using Euclidean distance.*

Description

Find vertex index closest to given query coordinate using Euclidean distance.

Usage

```
closest.vert.to.point(surface, point_coords)
```

Arguments

`surface`	an fs.surface instance or a nx3 numerical matrix representing mesh points.
`point_coords`	nx3 matrix of query coords. If a vector, will be transformed `byrow` to such a matrix.

Value

named list with entries: 'vertex_id' integer vector, the index of the closest vertex, and 'dist': double vector, the Euclidean distance to that vertex.

See Also

Other Euclidean distance util functions: `vertex.euclid.dist()`, `vertexdists.to.point()`

`colortable.from.annot` *Extract color lookup table (LUT) from annotation.*

Description

Extract a colortable lookup table (LUT) from an annotation. Such a LUT can also be read from files like ‘FREESURFER_HOME/FreeSurferColorLUT.txt‘ or saved as a file, check the 'See Also' section below.

Usage

```
colortable.from.annot(annot, compute_colorcode = FALSE)
```

Arguments

`annot`	An annotation, as returned by `read.fs.annot`. If you want to assign specific indices, you can add a column named 'struct_index' to the data.frame `annot$colortable_df`. If there is no such columns, the indices will be created automatically in the order of the regions, starting at zero.
`compute_colorcode`	logical, indicates whether the unique color codes should be computed and added to the returned data.frame as an extra integer column named 'code'. Defaults to FALSE.

Value

the colortable data.frame extracted from the annotation.

See Also

Other atlas functions: `read.fs.annot()`, `read.fs.colortable()`, `write.fs.annot.gii()`, `write.fs.annot()`, `write.fs.colortable()`

Other colorLUT functions: `read.fs.colortable()`, `write.fs.colortable()`

Examples

```
annotfile = system.file("extdata", "lh.aparc.annot.gz",
 package = "freesurferformats", mustWork = TRUE);
annot = read.fs.annot(annotfile);
colortable = colortable.from.annot(annot);
head(colortable);
```

`delete_all_opt_data` *Delete all data in the package cache.*

Description

Delete all data in the package cache.

Usage

```
delete_all_opt_data()
```

Value

integer. The return value of the unlink() call: 0 for success, 1 for failure. See the unlink() documentation for details.

`doapply.transform.mtx` *Apply a spatial transformation matrix to the given coordinates.*

Description

Apply a spatial transformation matrix to the given coordinates.

Usage

```
doapply.transform.mtx(coords, mtx, as_mat = FALSE)
```

Arguments

`coords`	nx3 (cartesian) or nx4 (homogeneous) numerical matrix, the input coordinates. If nx4, left as is for homogeneous notation, if nx3 (cartesian) a 1 will be appended as the 4th position.
`mtx`	a 4x4 numerical transformation matrix
`as_mat`	logical, whether to force the output coords into a matrix (even if the input was a vector/a single coordinate triple).

Value

the coords after applying the transformation. If coords was nx3, nx3 is returned, otherwise nx4.

Examples

```
coords_tf = doapply.transform.mtx(c(1.0, 1.0, 1.0), mni152reg());
coords_tf;
doapply.transform.mtx(coords_tf, solve(mni152reg()));
```

`download_opt_data` *Download optional data for the freesurferformats package.*

Description

Ensure that the optional data is available locally in the package cache. Will try to download the data only if it is not available. This data is not required for the package to work, but it is used in the examples, in the unit tests and also in the example code from the vignette. Downloading it is highly recommended.

Usage

```
download_opt_data()
```

Value

Named list. The list has entries: "available": vector of strings. The names of the files that are available in the local file cache. You can access them using get_optional_data_file(). "missing": vector of strings. The names of the files that this function was unable to retrieve.

`faces.quad.to.tris` *Convert quadrangular faces or polygons to triangular ones.*

Description

Convert quadrangular faces or polygons to triangular ones.

Usage

```
faces.quad.to.tris(quad_faces)
```

Arguments

`quad_faces`	nx4 integer matrix, the indices of the vertices making up the *n* quad faces.

Value

2nx3 integer matrix, the indices of the vertices making up the *2n* tris faces.

Note

This function does no fancy remeshing, it simply splits each quad into two triangles.

See Also

Other mesh functions: `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`faces.tris.to.quad` *Convert tris faces to quad faces by simple merging.*

Description

This is experimental. Note that it can only work if the number of 'tris_faces' is even, as two consecutive tris-faces will be merged into one quad face. We could set the index to NA in that case, but I do not know how FreeSurfer handles this, so we do not guess.

Usage

```
faces.tris.to.quad(tris_faces)
```

Arguments

`tris_faces`	*nx3* integer matrix, the indices of the vertices making up the *n* tris faces.

Value

n/2x4 integer matrix, the indices of the vertices making up the *n* quad faces.

Note

This function does not implement proper remeshing of tri-meshes to quad-meshes. Use a proper mesh library if you need that.

`flip2D` *Flip a 2D matrix.*

Description

Flip a 2D matrix.

Usage

```
flip2D(slice, how = "horizontally")
```

Arguments

`slice`	a 2D matrix
`how`	character string, one of 'vertically' / 'v' or 'horizontally' / 'h'. Note that flipping *horizontally* means that the image will be mirrored along the central *vertical* axis. If 'NULL' is passed, the passed value is returned unaltered.

Value

2D matrix, the flipped matrix.

`flip3D` *Flip a 3D array along an axis.*

Description

Flip the slice of an 3D array horizontally or vertically along an axis. This leads to an output array with identical dimensions.

Usage

```
flip3D(volume, axis = 1L, how = "horizontally")
```

Arguments

`volume`	a 3D image volume
`axis`	positive integer in range 1L..3L or an axis name, the axis to use.
`how`	character string, one of 'horizontally' / 'h' or 'vertically' / 'v'. How to flip the 2D slices. Note that flipping *horizontally* means that the image will be mirrored along the central *vertical* axis.

Value

a 3D image volume, flipped around the axis. The dimensions are identical to the dimensions of the input image.

See Also

Other volume math: `rotate3D()`

`fs.get.morph.file.ext.for.format` *Determine morphometry file extension from format*

Description

Given a morphometry file format, derive the proper file extension.

Usage

```
fs.get.morph.file.ext.for.format(format)
```

Arguments

`format,`	string. One of c("mgh", "mgz", "curv", "gii").

Value

file ext, string. The standard file extension for the format. (May be an empty string for some formats.)

See Also

Other morphometry functions: `fs.get.morph.file.format.from.filename()`, `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`fs.get.morph.file.format.from.filename`

Determine morphometry file format from filename

Description

Given a morphometry file name, derive the proper file format, based on the end of the string. Case is ignored, i.e., cast to lowercase before checks. If the filepath ends with "mgh", returns format "mgh". For suffix "mgz", returns "mgz" format. For all others, returns "curv" format.

Usage

```
fs.get.morph.file.format.from.filename(filepath)
```

Arguments

`filepath,`	string. A path to a file.

Value

format, string. The format, one of c("mgz", "mgh", "curv", "gii", "smp").

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`fs.patch` *Constructor for fs.patch*

Description

Constructor for fs.patch

Usage

```
fs.patch(vertices, faces = NULL)
```

Arguments

`vertices`	numerical *n*x5 matrix (or *n*x7 matrix), see `read.fs.patch` for details. If it has 5 columns, columns 6-7 will be computed automatically from the first 5 columns (from column 1 and 5).
`faces`	numerical *n*x5 matrix, see `read.fs.patch.asc` for details. Can be 'NULL'.

Value

instance of class 'fs.patch'

See Also

Other patch functions: `read.fs.patch.asc()`, `read.fs.patch()`, `write.fs.patch()`

Examples

```
num_vertices = 6L;   # a tiny patch
vertices = matrix(rep(0., num_vertices*5), ncol=5);
vertices[,1] = seq.int(num_vertices);  # 1-based vertex indices
vertices[,2:4] = matrix(rnorm(num_vertices*3, 8, 2), ncol=3);  # vertex coords
vertices[,5] = rep(0L, num_vertices);  # is_border
vertices[3,5] = 1L;  # set a vertex to be a border vertex
patch = fs.patch(vertices);
patch;
```

`fs.surface.to.tmesh3d` *Get an rgl tmesh3d instance from a brain surface mesh.*

Description

Convert `fs.surface` to `tmesh` without the `rgl` package.

Usage

```
fs.surface.to.tmesh3d(surface)
```

Arguments

`surface`	an fs.surface instance, as returned `freesurferformats::read.fs.surface`.

Value

a `tmesh3d` instance representing the surface, see `rgl::tmesh3d` for details. It has classes `mesh3d` and `shape3d`.

`get_opt_data_filepath` *Access a single file from the package cache by its file name.*

Description

Access a single file from the package cache by its file name.

Usage

```
get_opt_data_filepath(filename, mustWork = TRUE)
```

Arguments

`filename,`	string. The filename of the file in the package cache.
`mustWork,`	logical. Whether an error should be created if the file does not exist. If mustWork=FALSE and the file does not exist, the empty string is returned.

Value

string. The full path to the file in the package cache or the empty string if there is no such file available. Use this in your application code to open the file.

`giftixml_add_labeltable_from_annot` *Add a label tabel from an annotation to a GIFTI XML tree.*

Description

Computes the LabelTable XML node for the given annotation and adds it to the XML tree.

Usage

```
giftixml_add_labeltable_from_annot(xmltree, annot)
```

Arguments

`xmltree`	an XML tree from xml2, typically the return value from `gifti_xml`.
`annot`	an fs.annotation, the included data will be used to compute the LabelTable node

Value

XML tree from xml2, the modified tree with the LabelTable added below the root node.

`gifti_writer` *Write data to a gifti file.*

Description

Write data to a gifti file.

Usage

```
gifti_writer(filepath, ...)
```

Arguments

`filepath`	path to the output gifti file
`...`	parameters passed to `gifti_xml`.

References

https://www.nitrc.org/frs/download.php/2871/GIFTI_Surface_Format.pdf

Examples

```
## Not run:
  outfile = tempfile(fileext = '.gii');
  dataarrays = list(rep(3.1, 3L), matrix(seq(6), nrow=2L));
 gifti_writer(outfile, dataarrays, datatype=c('NIFTI_TYPE_FLOAT32', 'NIFTI_TYPE_INT32'));

## End(Not run)
```

`gifti_xml` *Get GIFTI XML representation of data.*

Description

Creates a GIFTI XML tree from your datasets (vectors and matrices). The tree can be further modified to add additional data, or written to a file as is to produce a valid GIFTI file (see `gifti_xml_write`).

Usage

```
gifti_xml(
  data_array,
  intent = "NIFTI_INTENT_SHAPE",
  datatype = "NIFTI_TYPE_FLOAT32",
  encoding = "GZipBase64Binary",
  endian = "LittleEndian",
  transform_matrix = NULL,
  force = FALSE
)
```

Arguments

`data_array`	list of data vectors and/or data matrices.
`intent`	vector of NIFTI intent strings for the data vectors in 'data_array' parameter, see `convert_intent`. Example: 'NIFTI_INTENT_SHAPE'. See https://nifti.nimh.nih.gov/nifti-1/documentation/nifti1fields/nifti1fields_pages/group__NIFTI1__INTENT__CODES.html.
`datatype`	vector of NIFTI datatype strings. Example: 'NIFTI_TYPE_FLOAT32'. Should be suitable for your data.
`encoding`	vector of encoding definition strings. One of 'ASCII', 'Base64Binary', 'GZipBase64Binary'.
`endian`	vector of endian definition strings. One of 'LittleEndian' or 'BigEndian'. See `convert_endian`.
`transform_matrix`	optional, a list of transformation matrices, one for each data_array. If one of the data arrays has none, pass 'NA'. Each transformation matrix in the outer list has to be a 4x4 matrix or given as a named list with entries 'transform_matrix', 'data_space', and 'transformed_space'. Here is an example: `list('transform_matrix'=diag(4), 'data_space'='NIFTI_XFORM_UNKNOWN', 'transformed_space'='NIFTI_XFORM_UNKNOWN')`.

`force`	logical, whether to force writing the data, even if issues like a mismatch of datatype and data values are detected.

Value

xml tree, see xml2 package. One could modify this tree as needed using xml2 functions, e.g., add metadata.

Note

Unless you want to modify the returned tree manually, you should not need to call this function. Use `gifti_writer` instead.

References

See https://www.nitrc.org/frs/download.php/2871/GIFTI_Surface_Format.pdf

See Also

The example for `gifti_xml_write` shows how to modify the tree.

Examples

```
## Not run:
  my_data_sets = list(rep(3.1, 3L), matrix(seq(6)+0.1, nrow=2L));
 transforms = list(NA, list('transform_matrix'=diag(4), 'data_space'='NIFTI_XFORM_UNKNOWN',
   'transformed_space'='NIFTI_XFORM_UNKNOWN'));
 xmltree = gifti_xml(my_data_sets, datatype='NIFTI_TYPE_FLOAT32', transform_matrix=transforms);
  # Verify that the tree is a valid GIFTI file:
  gifti_xsd = "https://www.nitrc.org/frs/download.php/158/gifti.xsd";
  xml2::xml_validate(xmltree, xml2::read_xml(gifti_xsd));

## End(Not run)
```

---

gifti_xml_add_global_metadata

Add metadata to GIFTI XML tree.

Description

Add metadata to GIFTI XML tree.

Usage

```
gifti_xml_add_global_metadata(xmltree, metadata_named_list, as_cdata = TRUE)
```

Arguments

`xmltree`	XML tree from xml2
`metadata_named_list`	named list, the metadata entries
`as_cdata`	logical, whether to wrap the value in cdata tags

Value

the modified tree.

Note

Assumes that there already exists a global MetaData node. Also not that this is not supposed to be used for adding metadata to datarrays.

Examples

```
## Not run:
  xmltree = gifti_xml(list(rep(3.1, 3L), matrix(seq(6)+0.1, nrow=2L)));
 newtree = gifti_xml_add_global_metadata(xmltree, list("User"="Me", "Weather"="Great"));
  gifti_xsd = "https://www.nitrc.org/frs/download.php/158/gifti.xsd";
  xml2::xml_validate(newtree, xml2::read_xml(gifti_xsd));

## End(Not run)
```

`gifti_xml_write` *Write XML tree to a gifti file.*

Description

Write XML tree to a gifti file.

Usage

```
gifti_xml_write(filepath, xmltree, options = c("as_xml", "format"))
```

Arguments

`filepath`	path to the output gifti file
`xmltree`	XML tree from xml2
`options`	output options passed to `write_xml`.

References

https://www.nitrc.org/frs/download.php/2871/GIFTI_Surface_Format.pdf

Examples

```
## Not run:
  outfile = tempfile(fileext = '.gii');
  my_data_sets = list(rep(3.1, 3L), matrix(seq(6)+0.1, nrow=2L));
  xmltree = gifti_xml(my_data_sets, datatype='NIFTI_TYPE_FLOAT32');
  # Here we add global metadata:
  xmltree = gifti_xml_add_global_metadata(xmltree, list("User"="Me", "Day"="Monday"));
  # Validating your XML never hurts
  gifti_xsd = "https://www.nitrc.org/frs/download.php/158/gifti.xsd";
  xml2::xml_validate(xmltree, xml2::read_xml(gifti_xsd));
  gifti_xml_write(outfile, xmltree);  # Write your custom tree to a file.

## End(Not run)
```

`is.bvsmp` *Check whether object is a bvsmp instance.*

Description

Check whether object is a bvsmp instance.

Usage

```
is.bvsmp(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is an bvsmp instane (that is, has "bvsmp" amongst its classes) and FALSE otherwise.

`is.fs.annot` *Check whether object is an fs.annot*

Description

Check whether object is an fs.annot

Usage

```
is.fs.annot(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is a brain surface annotation (that is, has "fs.annot" amongst its classes) and FALSE otherwise.

`is.fs.label` *Check whether object is an fs.label*

Description

Check whether object is an fs.label

Usage

```
is.fs.label(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is a brain surface label (that is, has ‘fs.label‘ amongst its classes) and FALSE otherwise.

`is.fs.surface` *Check whether object is an fs.surface*

Description

Check whether object is an fs.surface

Usage

```
is.fs.surface(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is a brain surface (that is, has "fs.surface" amongst its classes) and FALSE otherwise.

`is.fs.volume` *Check whether object is an fs.volume*

Description

Check whether object is an fs.volume

Usage

```
is.fs.volume(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is a brain volume (that is, has "fs.volume" amongst its classes) and FALSE otherwise.

`is.mghheader` *Check whether object is an mghheader*

Description

Check whether object is an mghheader

Usage

```
is.mghheader(x)
```

Arguments

`x`	any ‘R‘ object

Value

TRUE if its argument is an MGH header (that is, has "mghheader" amongst its classes) and FALSE otherwise.

`list_opt_data` *Get file names available in package cache.*

Description

Get file names of optional data files which are available in the local package cache. You can access these files with get_optional_data_file().

Usage

```
list_opt_data()
```

Value

vector of strings. The file names available, relative to the package cache.

`mghheader.centervoxelRAS.from.firstvoxelRAS` *Compute RAS coords of center voxel.*

Description

Compute RAS coords of center voxel.

Usage

```
mghheader.centervoxelRAS.from.firstvoxelRAS(header, first_voxel_RAS)
```

Arguments

`header`	Header of the mgh datastructure, as returned by `read.fs.mgh`. The ‘c_r‘, ‘c_a‘ and ‘c_s‘ values in the header do not matter of course, they are what is computed by this function.
`first_voxel_RAS`	numerical vector of length 3, the RAS coordinate of the first voxel in the volume. The first voxel is the voxel with ‘CRS=1,1,1‘ in R, or ‘CRS=0,0,0‘ in C/FreeSurfer. This value is also known as *P0 RAS*.

Value

numerical vector of length 3, the RAS coordinate of the center voxel. Also known as *CRAS* or *center RAS*.

`mghheader.crs.orientation` *Compute MGH volume orientation string.*

Description

Compute MGH volume orientation string.

Usage

```
mghheader.crs.orientation(header)
```

Arguments

`header`	Header of the mgh datastructure, as returned by `read.fs.mgh`.

Value

character string of length 3, one uppercase letter per axis. Each of the three position is a letter from the alphabet: ‘LRISAP¿. The meaning is ‘L‘ for left, ‘R‘ for right, ‘I‘ for inferior, ‘S‘ for superior, ‘P‘ for posterior, ‘A‘ for anterior. If the direction cannot be computed, all three characters are ‘¿ for unknown. Of course, each axis (‘L/R‘, ‘I/S‘, ‘A/P‘) is only represented once in the string.

`mghheader.is.conformed` *Determine whether an MGH volume is conformed.*

Description

In the FreeSurfer sense, *conformed* means that the volume is in coronal primary slice direction, has dimensions 256x256x256 and a voxel size of 1 mm in all 3 directions. The slice direction can only be determined if the header contains RAS information, if it does not, the volume is not conformed.

Usage

```
mghheader.is.conformed(header)
```

Arguments

`header`	Header of the mgh datastructure, as returned by `read.fs.mgh`.

Value

logical, whether the volume is *conformed*.

`mghheader.is.ras.valid` *Check whether header contains valid ras information*

Description

Check whether header contains valid ras information

Usage

```
mghheader.is.ras.valid(header)
```

Arguments

`header`	mgh header or 'fs.volume' instance with header

Value

logical, whether header contains valid ras information (according to the 'ras_good_flag').

See Also

Other header coordinate space: `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                           package = "freesurferformats",
                           mustWork = TRUE);
vdh = read.fs.mgh(brain_image, with_header = TRUE);
mghheader.is.ras.valid(vdh$header);
```

`mghheader.primary.slice.direction` *Compute MGH primary slice direction*

Description

Compute MGH primary slice direction

Usage

```
mghheader.primary.slice.direction(header)
```

Arguments

`header`	Header of the mgh datastructure, as returned by `read.fs.mgh`.

Value

character string, the slice direction. One of 'sagittal', 'coronal', 'axial' or 'unknown'.

`mghheader.ras2vox` *Compute ras2vox matrix from basic MGH header fields.*

Description

This is also known as the 'scanner' or 'native' ras2vox. It is the inverse of the respective vox2ras, see `mghheader.vox2ras`.

Usage

```
mghheader.ras2vox(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

`sm1to0`

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg` `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                          package = "freesurferformats",
                          mustWork = TRUE);
vdh = read.fs.mgh(brain_image, with_header = TRUE);
mghheader.ras2vox(vdh$header);
```

`mghheader.ras2vox.tkreg` *Compute ras2vox-tkreg matrix from basic MGH header fields.*

Description

This is also known as the 'tkreg' ras2vox. It is the inverse of the respective vox2ras, see `mghheader.vox2ras.tkreg`.

Usage

```
mghheader.ras2vox.tkreg(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

`sm1to0`

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
    brain_image = system.file("extdata", "brain.mgz",
                              package = "freesurferformats",
                              mustWork = TRUE);
    vdh = read.fs.mgh(brain_image, with_header = TRUE);
    mghheader.ras2vox.tkreg(vdh$header);
```

`mghheader.scanner2tkreg` *Compute scanner-RAS 2 tkreg-RAS matrix from basic MGH header fields.*

Description

This is also known as the 'scanner2tkreg' matrix. Note that this is a RAS-to-RAS matrix. It is the inverse of the 'tkreg2scanner' matrix, see `mghheader.tkreg2scanner`.

Usage

```
mghheader.scanner2tkreg(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
    brain_image = system.file("extdata", "brain.mgz",
                              package = "freesurferformats",
                              mustWork = TRUE);
    vdh = read.fs.mgh(brain_image, with_header = TRUE);
    mghheader.scanner2tkreg(vdh$header);
```

`mghheader.tkreg2scanner` *Compute tkreg-RAS to scanner-RAS matrix from basic MGH header fields.*

Description

This is also known as the 'tkreg2scanner' matrix. Note that this is a RAS-to-RAS matrix. It is the inverse of the 'scanner2tkreg' matrix, see `mghheader.scanner2tkreg`.

Usage

```
mghheader.tkreg2scanner(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                               package = "freesurferformats",
                               mustWork = TRUE);
vdh = read.fs.mgh(brain_image, with_header = TRUE);
mghheader.tkreg2scanner(vdh$header);
```

`mghheader.update.from.vox2ras` *Update mghheader fields from vox2ras matrix.*

Description

Update mghheader fields from vox2ras matrix.

Usage

```
mghheader.update.from.vox2ras(header, vox2ras)
```

Arguments

`header`	Header of the mgh datastructure, as returned by `read.fs.mgh`.
`vox2ras`	4x4 numerical matrix, the vox2ras transformation matrix.

Value

a named list representing the header

`mghheader.vox2ras` *Compute vox2ras matrix from basic MGH header fields.*

Description

This is also known as the 'scanner' or 'native' vox2ras. It is the inverse of the respective ras2vox, see `mghheader.ras2vox`.

Usage

```
mghheader.vox2ras(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

`sm0to1`

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `read.fs.transform.dat`
`read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                          package = "freesurferformats",
                          mustWork = TRUE);
vdh = read.fs.mgh(brain_image, with_header = TRUE);
mghheader.vox2ras(vdh$header);
```

`mghheader.vox2ras.tkreg`

Compute vox2ras-tkreg matrix from basic MGH header fields.

Description

This is also known as the 'tkreg' vox2ras. It is the inverse of the respective ras2vox, see `mghheader.ras2vox.tkreg`.

Usage

```
mghheader.vox2ras.tkreg(header)
```

Arguments

`header`	the MGH header

Value

4x4 numerical matrix, the transformation matrix

See Also

`sm0to1`

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                          package = "freesurferformats",
                          mustWork = TRUE);
vdh = read.fs.mgh(brain_image, with_header = TRUE);
mghheader.vox2ras.tkreg(vdh$header);
```

`mghheader.vox2vox` *Compute vox2vox matrix between two volumes.*

Description

Compute vox2vox matrix between two volumes.

Usage

```
mghheader.vox2vox(header_from, header_to)
```

Arguments

`header_from`	the MGH header of the source volume
`header_to`	the MGH header of the target volume

Value

4x4 numerical matrix, the transformation matrix

`mni152reg` *Get fsaverage (MNI305) to MNI152 transformation matrix.*

Description

The uses the 4x4 matrix from the FreeSurfer CoordinateSystems documentation.

Usage

```
mni152reg()
```

Note

There are better ways to achieve this transformation than using this matrix, see Wu et al., 'Accurate nonlinear mapping between MNI volumetric and FreeSurfer surface coordinate system', Hum Brain Mapp. 2018 Sep; 39(9): 3793–3808. doi: 10.1002/hbm.24213. The mentioned method is available in R from the 'regfusionr' package (GitHub only atom, not on CRAN).

Examples

```
coords_tf = doapply.transform.mtx(c(10.0, -20.0, 35.0), mni152reg());
coords_tf; #  10.695, -18.409, 36.137
doapply.transform.mtx(coords_tf, solve(mni152reg()));
```

`ni1header.for.data` *Create NIFTI v1 header suitable for given data.*

Description

Create NIFTI v1 header suitable for given data.

Usage

```
ni1header.for.data(niidata, allow_fshack = FALSE)
```

Arguments

`niidata`	array of numeric (integer or double) data, can have up to 7 dimensions.
`allow_fshack`	logical, whether to allow data in which the first dimension is larger than 32767, and use the FreeSurfer NIFTI v1 hack to support his. The hack will be used only if needed. WARNING: Files written with the hack do not conform to the NIFTI v1 standard and will not be read correctly by most software. All FreeSurfer tools and the Python 'nibabel' module support it.

Value

a NIFTI v1 header (see `ni1header.template`) in which the datatype, bitpix, dim and dim_raw fields have been set to values suitable for the given data. Feel free to change the other fields.

`ni1header.template` *Create a template NIFTI v1 header. You will have to adapt it for your use case.*

Description

Create a template NIFTI v1 header. You will have to adapt it for your use case.

Usage

```
ni1header.template()
```

Value

named list, the NIFTI v1 header. All fields are present and filled with values of a proper type. Whether or not they make sense is up to you, but you will most likely have to adapt at least the following fields to your data: ‘dim_raw‘, ‘datatype‘, ‘bitpix‘.

Note

Commonly used data type settings are: for signed integers datatype = ‘8L‘ and bitpix = ‘32L‘; for floats datatype = ‘16L‘ and bitpix = ‘32L‘. See the NIFTI v1 standard for more options. You may want to call `ni1header.for.data` instead of this function.

See Also

`ni1header.for.data`

`ni2header.for.data` *Create NIFTI v2 header suitable for given data.*

Description

Create NIFTI v2 header suitable for given data.

Usage

```
ni2header.for.data(niidata)
```

Arguments

`niidata`	array of numeric (integer or double) data, can have up to 7 dimensions.

Value

a NIFTI v2 header (see `ni2header.template`) in which the datatype, bitpix, dim and dim_raw fields have been set to values suitable for the given data. Feel free to change the other fields.

`ni2header.template` *Create a template NIFTI v2 header. You will have to adapt it for your use case.*

Description

Create a template NIFTI v2 header. You will have to adapt it for your use case.

Usage

```
ni2header.template()
```

Value

named list, the NIFTI v2 header. All fields are present and filled with values of a proper type. Whether or not they make sense is up to you, but you will most likely have to adapt at least the following fields to your data: ‘dim_raw‘, ‘datatype‘, ‘bitpix‘.

Note

Commonly used data type settings are: for signed integers datatype = ‘8L‘ and bitpix = ‘32L‘; for floats datatype = ‘16L‘ and bitpix = ‘32L‘. See the NIFTI v2 standard for more options. You may want to call `ni2header.for.data` instead of this function.

See Also

`ni2header.for.data`

`nifti.datadim.from.dimfield` *Compute data dimensions from the ’dim’ field of the NIFTI (v1 or v2) header.*

Description

Compute data dimensions from the ’dim’ field of the NIFTI (v1 or v2) header.

Usage

```
nifti.datadim.from.dimfield(dimfield)
```

Arguments

`dimfield`	integer vector of length 8, the ‘dim‘ field of a NIFTI v1 or v2 header, as returned by `read.nifti2.header` or `read.nifti1.header`.

Value

integer vector of length <= 7. The lengths of the used data dimensions. The 'dim' field always has length 8, and the first entry is the number of actually used dimensions. The return value is constructed by stripping the first field and returning the used fields.

See Also

Other NIFTI helper functions: `nifti.datadim.to.dimfield()`

Examples

```
nifti.datadim.from.dimfield(c(3, 256, 256, 256, 1, 1, 1, 1));
```

`nifti.datadim.to.dimfield` *Compute NIFTI dim field for data dimension.*

Description

Compute NIFTI dim field for data dimension.

Usage

```
nifti.datadim.to.dimfield(datadim)
```

Arguments

`datadim`	integer vector, the result of calling ‘dim‘ on your data. The length must be <= 7.

Value

NIFTI header ‘dim‘ field, an integer vector of length 8

See Also

Other NIFTI helper functions: `nifti.datadim.from.dimfield()`

Examples

```
nifti.datadim.to.dimfield(c(256, 256, 256));
```

`nifti.file.uses.fshack` *Determine whether a NIFTI file uses the FreeSurfer hack.*

Description

Determine whether a NIFTI file uses the FreeSurfer hack.

Usage

```
nifti.file.uses.fshack(filepath)
```

Arguments

`filepath`	path to a NIFTI v1 file (single file version), which can contain the FreeSurfer hack.

Value

logical, whether the file header contains the FreeSurfer format hack. See `read.nifti1.header` for details. This function detects NIFTI v2 files, but as they cannot contain the hack, it will always return ‘FALSE‘ for them.

Note

Applying this function to files which are not in NIFTI format will result in an error. See `nifti.file.version` to determine whether a file is a NIFTI file.

`nifti.file.version` *Determine NIFTI file version information and whether file is a NIFTI file.*

Description

Determine NIFTI file version information and whether file is a NIFTI file.

Usage

```
nifti.file.version(filepath)
```

Arguments

`filepath`	path to a file in NIFTI v1 or v2 format.

Value

integer, the NIFTI file version. One if ‘1‘ for NIFTI v1 files, ‘2‘ for NIFTI v2 files, or ‘NULL‘ if the file is not a NIFTI file.

`nifti.header.check` *Perform basic sanity checks on NIFTI header data. These are in no way meant to be exhaustive.*

Description

Perform basic sanity checks on NIFTI header data. These are in no way meant to be exhaustive.

Usage

```
nifti.header.check(niiheader, nifti_version = 1L)
```

Arguments

`niiheader`	named list, the NIFTI header.
`nifti_version`	integer, one of 1L or 2L. The NIFTI format version.

Value

logical, whether the check was okay

`print.fs.annot` *Print description of a brain atlas or annotation.*

Description

Print description of a brain atlas or annotation.

Usage

```
## S3 method for class 'fs.annot'
print(x, ...)
```

Arguments

`x`	brain surface annotation or atlas with class ‘fs.annot‘.
`...`	further arguments passed to or from other methods

`print.fs.label` *Print description of a brain surface label.*

Description

Print description of a brain surface label.

Usage

```
## S3 method for class 'fs.label'
print(x, ...)
```

Arguments

`x`	brain surface label with class ‘fs.label‘.
`...`	further arguments passed to or from other methods

`print.fs.patch` *Print description of a brain surface patch.*

Description

Print description of a brain surface patch.

Usage

```
## S3 method for class 'fs.patch'
print(x, ...)
```

Arguments

`x`	brain surface patch with class ‘fs.patch‘.
`...`	further arguments passed to or from other methods

`print.fs.surface` *Print description of a brain surface.*

Description

Print description of a brain surface.

Usage

```
## S3 method for class 'fs.surface'
print(x, ...)
```

Arguments

`x`	brain surface with class ‘fs.surface‘.
`...`	further arguments passed to or from other methods

`print.fs.volume` *Print description of a brain volume.*

Description

Print description of a brain volume.

Usage

```
## S3 method for class 'fs.volume'
print(x, ...)
```

Arguments

`x`	brain volume with class ‘fs.volume‘.
`...`	further arguments passed to or from other methods

`ras.to.surfaceras` *Translate RAS coordinates, as used in volumes by applying vox2ras, to surface RAS.*

Description

Translate RAS coordinates, as used in volumes by applying vox2ras, to surface RAS.

Usage

```
ras.to.surfaceras(header_cras, ras_coords, first_voxel_RAS = c(1, 1, 1))
```

Arguments

`header_cras`	an MGH header instance from which to extract the cras (center RAS), or the cras vector, i.e., a numerical vector of length 3
`ras_coords`	nx3 numerical vector, the input surface RAS coordinates. Could be the vertex coordinates of an 'fs.surface' instance, or the RAS coords from a surface label.
`first_voxel_RAS`	the RAS of the first voxel, see `mghheader.centervoxelRAS.from.firstvoxelRAS` for details. Ignored if 'header_cras' is a vector.

Value

the surface RAS coords for the input RAS coords

Note

The RAS can be computed from Surface RAS by adding the center RAS coordinates, i.e., it is nothing but a translation.

`ras.to.talairachras` *Compute MNI talairach coordinates from RAS coords.*

Description

Compute MNI talairach coordinates from RAS coords.

Usage

```
ras.to.talairachras(ras_coords, talairach, invert_transform = FALSE)
```

Arguments

`ras_coords`	nx3 numerical vector, the input surface RAS coordinates. Could be the vertex coordinates of an 'fs.surface' instance, or the RAS coords from a surface label.
`talairach`	the 4x4 numerical talairach matrix, or a character string which will be interpreted as the path to an xfm file containing the matrix (typically '$SUBJECTS_DIR/$subject/mri/transfor
`invert_transform`	logical, whether to invert the transform. Do not use this, call `link{talairachras.to.ras}` instead.

Value

the Talairach RAS coordinates for the given RAS coordinates

Note

You can use this to compute the Talairach coordinate of a voxel, based on its RAS coordinate.

`read.dti.tck` *Read DTI tracking data from file in MRtrix 'TCK' format.*

Description

Read DTI tracking data from file in MRtrix 'TCK' format.

Usage

```
read.dti.tck(filepath)
```

Arguments

`filepath`	character string, path to the `TCK` file to read.

Value

named list with entries 'header' and 'tracks'. The tracks are organized into a list of matrices. Each n x 3 matrix represents the coordinates for the n points of one track, the values in each row are the xyz coords.

Examples

```
## Not run:
 tckf = "~/simple.tck";
 tck = read.dti.tck(tckf);

## End(Not run)
```

`read.dti.trk` *Read fiber tracks from Diffusion Toolkit in trk format.*

Description

Read fiber tracks from Diffusion Toolkit in trk format.

Usage

```
read.dti.trk(filepath)
```

Arguments

`filepath`	character string, path to file in trk format.

Value

named list, the parsed file data. The naming of the variables follows the spec at `http://trackvis.org/docs/?subsect=fil`

Examples

```
## Not run:
trk = read.dti.trk("~/simple.trk");
trk2 = read.dti.trk("~/standard.trk");
trk3 = read.dti.trk("~/complex_big_endian.trk");

## End(Not run)
```

`read.dti.tsf` *Read DTI tracking per-coord data from file in MRtrix 'TSF' format.*

Description

Read DTI tracking per-coord data from file in MRtrix 'TSF' format.

Usage

```
read.dti.tsf(filepath)
```

Arguments

`filepath`	character string, path to the `TSF` file to read.

Value

named list with entries 'header' and 'scalars'. The scala data are available in 2 representations: 'merged': a vector of all values (requires external knowledge on track borders), and 'scalar_list': organized into a list of vectors. Each vector represents the values for the points of one track.

Note

The data in such a file is one value per track point, the tracks are not part of the file but come in the matching TCK file.

See Also

`read.dti.tck`

Examples

```
## Not run:
 tsff = "~/simple.tsf";
 tsf = read.dti.tsf(tsff);

## End(Not run)
```

`read.fs.annot` *Read file in FreeSurfer annotation format*

Description

Read a data annotation file in FreeSurfer format. Such a file assigns a label and a color to each vertex of a brain surface. The assignment of labels to vertices is based on at atlas or brain parcellation file. Typically the atlas is available for some standard template subject, and the labels are assigned to another subject by registering it to the template. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/label/lh.aparc.annot', which contains the annotation based on the Desikan-Killiany Atlas for the left hemisphere of bert.

Usage

```
read.fs.annot(
  filepath,
  empty_label_name = "empty",
  metadata = list(),
  default_label_name = ""
)
```

Arguments

`filepath`	string. Full path to the input annotation file. Note: gzipped files are supported and gz format is assumed if the filepath ends with ".gz".
`empty_label_name`	character string, a base name to use to rename regions with empty name in the label table. This should not occur, and you can ignore this parameter setting. A warning will be thrown if this ever triggers. Not to be confused with parameter `default_label_name`, see below.
`metadata`	named list of arbitrary metadata to store in the instance.
`default_label_name`	character string, the label name to use for vertices which have a label code that does not occur in the label table. This is typically the case for the 'unknown' region, which often has code `0`. You can set this to avoid empty region label names. The typical setting would be 'unknown', however by default we leave the names as-is, so that annots which are read and then written back to files with this library do not differ.

Value

named list, entries are: "vertices" vector of n vertex indices, starting with 0. "label_codes": vector of n integers, each entry is a color code, i.e., a value from the 5th column in the table structure included in the "colortable" entry (see below). "label_names": the n brain structure names for the vertices, already retrieved from the colortable using the code. "hex_colors_rgb": Vector of hex color for each vertex. The "colortable" is another named list with 3 entries: "num_entries": int, number of brain structures. "struct_names": vector of strings, the brain structure names. "table": numeric matrix with num_entries rows and 5 colums. The 5 columns are: 1 = color red channel, 2=color blue channel, 3=color green channel, 4=color alpha channel, 5=unique color code. "colortable_df": The same information as a dataframe. Contains the extra columns "hex_color_string_rgb" and "hex_color_string_rgba" that hold the color as an RGB(A) hex string, like "#rrggbbaa".

See Also

Other atlas functions: `colortable.from.annot()`, `read.fs.colortable()`, `write.fs.annot.gii()`, `write.fs.annot()`, `write.fs.colortable()`

Examples

```
annot_file = system.file("extdata", "lh.aparc.annot.gz",
                         package = "freesurferformats",
                         mustWork = TRUE);
annot = read.fs.annot(annot_file);
print(annot);
```

`read.fs.annot.gii` *Read an annotation or label in GIFTI format.*

Description

Read an annotation or label in GIFTI format.

Usage

```
read.fs.annot.gii(
  filepath,
  element_index = 1L,
  labels_only = FALSE,
  rgb_column_names = c("Red", "Green", "Blue", "Alpha"),
  key_column_name = "Key",
  empty_label_name = "unknown"
)
```

Arguments

`filepath`	string. Full path to the input label file in GIFTI format.
`element_index`	positive integer, the index of the dataarray to return. Ignored unless the file contains several dataarrays.
`labels_only`	logical, whether to ignore the colortable and region names. The returned annotation will only contain the a vector that contains one integer label per vertex (as entry 'label_codes'), but no region names and colortable information.
`rgb_column_names`	vector of exactly 4 character strings, order is important. The column names for the red, green, blue and alpha channels in the lable table. If a column does not exist, pass NA. If you do not know the column names, just call the function, it will print them. See 'labels_only' if you do not care.
`key_column_name`	character string, the column name for the key column in the lable table. This is the column that holds the label value from the raw vector (see 'labels_only') that links a label value to a row in the label table. Without it, one cannot recostruct the region name and color of an entry. Passing NA has the same effect as setting 'labels_only' to TRUE.
`empty_label_name`	character string, a base name to use to rename regions with empty name in the label table. This should not occur, and you can ignore this parameter setting. A warning will be thrown if this ever triggers. Not to be confused with parameter `default_label_name`, see below.

See Also

Other gifti readers: `read.fs.label.gii()`, `read.fs.morph.gii()`, `read.fs.surface.gii()`

`read.fs.colortable` *Read colortable file in FreeSurfer ASCII LUT format.*

Description

Read a colortable from a text file in FreeSurfer ASCII colortable lookup table (LUT) format. An example file is 'FREESURFER_HOME/FreeSurferColorLUT.txt'.

Usage

```
read.fs.colortable(filepath, compute_colorcode = FALSE)
```

Arguments

`filepath,`	string. Full path to the output colormap file.
`compute_colorcode`	logical, indicates whether the unique color codes should be computed and added to the returned data.frame as an extra integer column named 'code'. Defaults to FALSE.

Value

the data.frame that was read from the LUT file. It contains the following columns that were read from the file: 'struct_index': integer, index of the struct entry. 'struct_name': character string, the label name. 'r': integer in range 0-255, the RGBA color value for the red channel. 'g': same for green channel. 'b': same for blue channel. 'a': same for alpha (transparency) channel. If 'compute_colorcode' is TRUE, it also contains the following columns which were computed from the color values: 'code': integer, unique color identifier computed from the RGBA values.

See Also

Other atlas functions: `colortable.from.annot()`, `read.fs.annot()`, `write.fs.annot.gii()`, `write.fs.annot()`, `write.fs.colortable()`

Other colorLUT functions: `colortable.from.annot()`, `write.fs.colortable()`

Examples

```
lutfile = system.file("extdata", "colorlut.txt", package = "freesurferformats", mustWork = TRUE);
 colortable = read.fs.colortable(lutfile, compute_colorcode=TRUE);
 head(colortable);
```

`read.fs.curv` *Read file in FreeSurfer curv format*

Description

Read vertex-wise brain morphometry data from a file in FreeSurfer 'curv' format. Both the binary and ASCII versions are supported. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.thickness', which contains n values. Each value represents the cortical thickness at the respective vertex in the brain surface mesh of bert.

Usage

```
read.fs.curv(filepath, format = "auto", with_header = FALSE)
```

Arguments

`filepath`	string. Full path to the input curv file. Note: gzipped binary curv files are supported and gz binary format is assumed if the filepath ends with ".gz".
`format`	one of 'auto', 'asc', 'bin', 'nii' or 'txt'. The format to assume. If set to 'auto' (the default), binary format will be used unless the filepath ends with '.asc' or '.txt'. The latter is just one float value per line in a text file.
`with_header`	logical, whether to return named list with 'header' and 'data' parts. Only valid with FreeSurfer binary curv format.

Value

data vector of floats. The brain morphometry data, one value per vertex.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`,
`write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`,
`write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`,
`write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Examples

```
    curvfile = system.file("extdata", "lh.thickness",
                              package = "freesurferformats", mustWork = TRUE);
    ct = read.fs.curv(curvfile);
    cat(sprintf("Read data for %d vertices. Values: min=%f, mean=%f, max=%f.\n",
                              length(ct), min(ct), mean(ct), max(ct)));
```

`read.fs.gca` *Read FreeSurfer GCA file.*

Description

Read FreeSurfer GCA file.

Usage

```
read.fs.gca(filepath)
```

Arguments

`filepath`	character string, path to a file in binary GCA format. Stores array of Gaussian classifiers for probabilistic atlas.

Value

named list, the file fields. The GCA data is in the data field.

Author(s)

This function is based on Matlab code by Bruce Fischl, published under the FreeSurfer Open Source License available at https://surfer.nmr.mgh.harvard.edu/fswiki/FreeSurferSoftwareLicense. The R version was written by Tim Schaefer.

Examples

```
## Not run:
gca_file = file.path(Sys.getenv('FREESURFER_HOME'), 'average', 'face.gca');
gca = read.fs.gca(gca_file);

## End(Not run)
```

`read.fs.label` *Read a label file.*

Description

Read a label file.

Usage

```
read.fs.label(filepath, format = "auto", ...)
```

Arguments

`filepath`	string. Full path to the input label file.
`format`	character string, one of 'auto' to detect by file extension, 'asc' for native FreeSurfer ASCII label format, or 'gii' for GIFTI label format.
`...`	extra paramters passed to the respective label function for the format

Note

See `read.fs.label.native` for more details, including important information on loading FreeSurfer volume labels.

See Also

Other label functions: `read.fs.label.gii()`, `read.fs.label.native()`, `write.fs.label()`

Examples

```
labelfile = system.file("extdata", "lh.entorhinal_exvivo.label",
  package = "freesurferformats", mustWork = TRUE);
label = read.fs.label(labelfile);
```

`read.fs.label.gii` *Read a label from a GIFTI label/annotation file.*

Description

Read a label from a GIFTI label/annotation file.

Usage

```
read.fs.label.gii(filepath, label_value = 1L, element_index = 1L)
```

Arguments

`filepath`	string. Full path to the input label file.
`label_value`	integer, the label value of interest to extract from the annotation: the indices of the vertices with this value will be returned. See the note for details.. It is important to set this correctly, otherwise you may accidently load the vertices which are *not* part of the label.
`element_index`	positive integer, the index of the data array to return. Ignored unless the file contains several data arrays.

Value

integer vector, the vertex indices of the label

Note

A GIFTI label is more like a FreeSurfer annotation, as it assigns a label integer (region code) to each vertex of the surface instead of listing only the set of 'positive' vertex indices. If you are not sure about the contents of the label file, it is recommended to read it with `read.fs.annot.gii` instead. The 'read.fs.label.gii' function only extracts one of the regions from the annotation as a label, while `read.fs.annot.gii` reads the whole annotation and gives you access to the label table, which should assign region names to each region, making it clearer which 'label_value' you want.

See Also

Other label functions: `read.fs.label.native()`, `read.fs.label()`, `write.fs.label()`

Other gifti readers: `read.fs.annot.gii()`, `read.fs.morph.gii()`, `read.fs.surface.gii()`

`read.fs.label.native` *Read file in FreeSurfer label format*

Description

Read a mask in FreeSurfer label format. A label defines a list of vertices (of an associated surface or morphometry file) which are part of it. All others are not. You can think of it as binary mask. Label files are ASCII text files, which have 5 columns (vertex index, coord1, coord2, coord3, value), but only the vertex indices are of interest. A label can also contain voxels, in that case the indices are -1 and the coordinates are important.

Usage

```
read.fs.label.native(
  filepath,
  return_one_based_indices = TRUE,
  full = FALSE,
  metadata = list()
)
```

Arguments

`filepath`	string. Full path to the input label file.
`return_one_based_indices`	logical. Whether the indices should be 1-based. Indices are stored zero-based in the file, but R uses 1-based indices. Defaults to TRUE, which means that 1 will be added to all indices read from the file before returning them. Notice that for volume labels, the indices are negative (-1), and the coord fields contain the *positions* of the voxels it tkras space (**not** the voxel *indices* in a volume). If a file contains negative indices, they will NOT be incremented, no matter what this is set to.

`full`	logical, whether to return a full object of class ‘fs.label‘ instead of only a vector containing the vertex indices. If TRUE, a named list with the following two entries is returned: ’one_based_indices’: logical, whether the vertex indices are one-based. ’vertexdata’: a data.frame with the following columns: ’vertex_index’: integer, see parameter ’return_one_based_indices’, ’coord1’, ’coord2’, ’coord3’: float coordinates, ’value’: float, scalar data for the vertex, can mean anything. This parameter defaults to FALSE.
`metadata`	named list of arbitrary metadata to store in the instance, ignored unless the paramter ‘full‘ is TRUE.

Value

vector of integers or ‘fs.label‘ instance (see parameter ‘full‘). The vertex indices from the label file. See the parameter ‘return_one_based_indices‘ for important information regarding the start index.

Note

To load volume/voxel labels, you will have to set the ’full’ parameter to ‘TRUE‘.

See Also

Other label functions: `read.fs.label.gii()`, `read.fs.label()`, `write.fs.label()`

Examples

```
labelfile = system.file("extdata", "lh.entorhinal_exvivo.label",
  package = "freesurferformats", mustWork = TRUE);
label = read.fs.label(labelfile);
```

`read.fs.mgh` *Read file in FreeSurfer MGH or MGZ format*

Description

Read multi-dimensional brain imaging data from a file in FreeSurfer binary MGH or MGZ format. The MGZ format is just a gzipped version of the MGH format. For a subject (MRI image pre-processed with FreeSurfer) named ’bert’, an example file would be ’bert/mri/T1.mgz’, which contains a 3D brain scan of bert.

Usage

```
read.fs.mgh(
  filepath,
  is_gzipped = "AUTO",
  flatten = FALSE,
  with_header = FALSE,
  drop_empty_dims = FALSE
)
```

Arguments

`filepath`	string. Full path to the input MGZ or MGH file.
`is_gzipped`	a logical value or the string 'AUTO'. Whether to treat the input file as gzipped, i.e., MGZ instead of MGH format. Defaults to 'AUTO', which tries to determine this from the last three characters of the 'filepath' parameter. Files with extensions 'mgz' and '.gz' (in arbitrary case) are treated as MGZ format, all other files are treated as MGH. In the special case that 'filepath' has less than three characters, MGH is assumed.
`flatten`	logical. Whether to flatten the return volume to a 1D vector. Useful if you know that this file contains 1D morphometry data.
`with_header`	logical. Whether to return the header as well. If TRUE, return an instance of class 'fs.volume' for data with at least 3 dimensions, a named list with entries "data" and "header". The latter is another named list which contains the header data. These header entries exist: "dtype": int, one of: 0=MRI_UCHAR; 1=MRI_INT; 3=MRI_FLOAT; 4=MRI_SHORT. "voldim": integer vector. The volume (=data) dimensions. E.g., c(256, 256, 256, 1). These header entries may exist: "vox2ras_matrix" (exists if "ras_good_flag" is 1), "mr_params" (exists if "has_mr_params" is 1). See the 'mghheader.*' functions, like `mghheader.vox2ras.tkreg`, to compute more information from the header fields.
`drop_empty_dims`	logical, whether to drop empty dimensions of the returned data

Value

data, multi-dimensional array. The brain imaging data, one value per voxel. The data type and the dimensions depend on the data in the file, they are read from the header. If the parameter flatten is 'TRUE', a numeric vector is returned instead. Note: The return value changes if the parameter with_header is 'TRUE', see parameter description.

See Also

To derive more information from the header, see the 'mghheader.*' functions, like `mghheader.vox2ras.tkreg`.

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(` `read.fs.curv()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                          package = "freesurferformats",
                          mustWork = TRUE);
vd = read.fs.mgh(brain_image);
cat(sprintf("Read voxel data with dimensions %s. Values: min=%d, mean=%f, max=%d.\n",
            paste(dim(vd), collapse = ' '), min(vd), mean(vd), max(vd)));
# Read it again with full header data:
vdh = read.fs.mgh(brain_image, with_header = TRUE);
```

```
# Use the vox2ras matrix from the header to compute RAS coordinates at CRS voxel (0, 0, 0):
 vdh$header$vox2ras_matrix %*% c(0,0,0,1);
```

`read.fs.morph` *Read morphometry data file in any FreeSurfer format.*

Description

Read vertex-wise brain surface data from a file. The file can be in any of the supported formats, and the format will be determined from the file extension.

Usage

```
read.fs.morph(filepath, format = "auto")
```

Arguments

`filepath,`	string. Full path to the input file. The suffix determines the expected format as follows: ".mgz" and ".mgh" will be read with the read.fs.mgh function, all other file extensions will be read with the read.fs.curv function.
`format`	character string, the format to use. One of c("auto", "mgh", "mgz", "curv", "gii"). The default setting "auto" will determine the format from the file extension.

Value

data, vector of floats. The brain morphometry data, one value per vertex.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`,
`write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`,
`write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`,
`write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Examples

```
    curvfile = system.file("extdata", "lh.thickness",
                           package = "freesurferformats", mustWork = TRUE);
    ct = read.fs.morph(curvfile);
    cat(sprintf("Read data for %d vertices. Values: min=%f, mean=%f, max=%f.\n",
                           length(ct), min(ct), mean(ct), max(ct)));


    mghfile = system.file("extdata", "lh.curv.fwhm10.fsaverage.mgz",
                           package = "freesurferformats", mustWork = TRUE);
```

```
curv = read.fs.morph(mghfile);
cat(sprintf("Read data for %d vertices. Values: min=%f, mean=%f, max=%f.\n",
                        length(ct), min(ct), mean(ct), max(ct)));
```

`read.fs.morph.asc` *Read morphometry data from ASCII curv format file*

Description

Read morphometry data from ASCII curv format file

Usage

```
read.fs.morph.asc(filepath)
```

Arguments

`filepath`	path to a file in FreeSurfer ASCII curv format. Such a file contains, on each line, the following fields, separated by spaces: vertex_index, vertex_coord_x, vertex_coord_y, vertex_coord_z, morph_data_value.

Value

numeric vector, the curv data

Note

This format is also known as *dpv* (data-per-vertex) format.

`read.fs.morph.bvsmp` *Read Brainvoyager vertex-wise statistical surface data from SMP file.*

Description

Read Brainvoyager vertex-wise statistical surface data from SMP file.

Usage

```
read.fs.morph.bvsmp(filepath, map_index = 1L)
```

Arguments

`filepath`	character string, path to file in Brainvoyager SMP file format. Alternatively, a ’bvsmp’ instance read with `read.smp.brainvoyager`.
`map_index`	positive integer or character string, the surface value map to load (an SMP file can contain several values per vertex, i.e., several surface maps). If an integer, interpreted as the index of the map. If a character string, as the name of the map.

Value

numeric vector, the values from the respective map.

`read.fs.morph.cifti` *Read surface morphometry data from CIFTI dscalar files.*

Description

Used the 'cifti' package to load the full data from a CIFTI file, then extracts and reconstructs the data for a surface, based on the metadata like vertex counts, indices and offset in the CIFTI file.

Usage

```
read.fs.morph.cifti(
  filepath,
  brain_structure = "CIFTI_STRUCTURE_CORTEX_LEFT",
  data_column = 1L
)
```

Arguments

`filepath`	character string, the full path to a file in CIFTI 2 format, should end with '.dscalar.nii'. Note that this is NOT a NIFTI file, despite the '.nii' part. It uses a CIFTIv2 header though. See the spec for details.
`brain_structure`	character string or integer, the brain structure for which the data should be extracted from the file. Can be a CIFTI brain structure string (one of 'CIFTI_STRUCTURE_CORTEX_LEF or 'CIFTI_STRUCTURE_CORTEX_RIGHT'), or simply one of 'lh', 'rh' (which are used as aliases for the former). If you specify 'both', the concatenated data for 'lh' (first) and 'rh' will be returned, but you will get no information on hemi boundaries. If it is an integer, it will be interpreted as an index into the list of structures within the CIFTI file, use with care.
`data_column`	integer, the data column to return. A CIFTI file can contain several measures in different data columns (e.g., cortical thickness and surface area) in a single file. This specifies which column/measure you want. The columns are not named, so you will need to know this in advance if the file has several measures.

Value

The reconstructed data for the given surface, one value per vertex in the surface. The value for vertices which did not have a value in the CIFTI data is set to ‘NA‘.

Note

This function calls code from the 'cifti' package by John Muschelli: `https://CRAN.R-project.org/package=cifti`.

References

See https://www.nitrc.org/forum/attachment.php?attachid=341&group_id=454&forum_id=1955 for the CIFTI 2 file format spec. See https://www.nitrc.org/projects/cifti/ for more details on CIFTI, including example files.

Examples

```
## Not run:
# Downloaded CIFTI2 example data from https://www.nitrc.org/projects/cifti/
cifti_example_data_dir = "~/data/cifti";
cii_file = file.path(cifti_example_data_dir,
 "Conte69.MyelinAndCorrThickness.32k_fs_LR.dscalar.nii");
sf_lh = freesurferformats::read.fs.surface(file.path(cifti_example_data_dir,
 "Conte69.L.inflated.32k_fs_LR.surf.gii"));
sf_rh = freesurferformats::read.fs.surface(file.path(cifti_example_data_dir,
"Conte69.R.inflated.32k_fs_LR.surf.gii"));
morph_lh = read.fs.morph.cifti(cii_file, 'lh'); # Myelin data
morph_rh = read.fs.morph.cifti(cii_file, 'rh');
morph2_lh = read.fs.morph.cifti(cii_file, 'lh', 2); # Cortical Thickness data
morph2_rh = read.fs.morph.cifti(cii_file, 'rh', 2L);
# fsbrain::vis.fs.surface(sf_lh, per_vertex_data = morph_lh);
# fsbrain::vis.fs.surface(sf_rh, per_vertex_data = morph_rh);
# fsbrain::vis.fs.surface(list('lh'=sf_lh, 'rh'=sf_rh),
# per_vertex_data = list('lh'=morph2_lh, 'rh'=morph2_rh));

## End(Not run)
```

`read.fs.morph.gii` *Read morphometry data file in GIFTI format.*

Description

Read vertex-wise brain surface data from a GIFTI file. The file must be a GIFTI *func* file (not a GIFTI *surf* file containing a mesh, use `read_nisurface` for loading GIFTI surf files).

Usage

```
read.fs.morph.gii(filepath, element_index = 1L)
```

Arguments

`filepath,`	string. Full path to the input GIFTI file.
`element_index`	integer, the element to load in case the GIFTI file containes several datasets (usually time series). Defaults to the first element, 1L.

Value

data, vector of double or integer. The brain morphometry data, one value per vertex. The data type depends on the data type in the file.

Note

This function requires the ‘gifti‘ package, which is an optional dependency, to be installed. It also assumes that the dataset contains a vector or a matrix/array in which all dimensions except for 1 are empty.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Other gifti readers: `read.fs.annot.gii()`, `read.fs.label.gii()`, `read.fs.surface.gii()`

`read.fs.morph.ni1` *Read morphometry data from FreeSurfer NIFTI v1 format files.*

Description

Read morphometry data from FreeSurfer NIFTI v1 format files.

Usage

```
read.fs.morph.ni1(filepath)
```

Arguments

`filepath`	path to a file in FreeSurfer NIFTI v1 format, potentially with the FreeSurfer hack. See `read.nifti1.data` for details.

Value

numeric vector, the morphometry data

Note

This function uses our internal NIFTI reader that supports NIFTI v1 files with the FreeSurfer hack. This function assumes that the data in a file is a 1D vector and flattens it accordingly. It is not suitable to load NIFTI files with arbitrary dimensions.

`read.fs.morph.ni2` *Read morphometry data from FreeSurfer NIFTI v2 format files.*

Description

Read morphometry data from FreeSurfer NIFTI v2 format files.

Usage

```
read.fs.morph.ni2(filepath)
```

Arguments

`filepath`	path to a file in FreeSurfer NIFTI v2 format, potentially with the FreeSurfer hack. See `read.nifti2.data` for details.

Value

numeric vector, the morphometry data

`read.fs.morph.nii` *Read morphometry data from FreeSurfer NIFTI format files, determine NIFTI version automatically.*

Description

Read morphometry data from FreeSurfer NIFTI format files, determine NIFTI version automatically.

Usage

```
read.fs.morph.nii(filepath)
```

Arguments

`filepath`	path to a file in FreeSurfer NIFTI v1 or v2 format, potentially with the FreeSurfer hack for v1. See `read.nifti1.data` and `read.nifti2.data` for details.

Value

numeric vector, the morphometry data

`read.fs.morph.txt` *Read morphometry data from plain text file*

Description

Read morphometry data from plain text file

Usage

```
read.fs.morph.txt(filepath)
```

Arguments

`filepath`	path to a file in plain text format. Such a file contains, on each line, a single float value. This very simply and limited *format* is used by the LGI tool by Lyu et al., and easy to generate in shell scripts.

Value

numeric vector, the curv data

`read.fs.patch` *Read FreeSurfer binary or ASCII patch file.*

Description

A patch is a subset of a surface. Note that the contents of ASCII and binary patch format files is different. A binary format patch contains vertices only, without connection (face) information. ASCII patch files can also contain face data. See the return value description for details.

Usage

```
read.fs.patch(filepath, format = "auto")
```

Arguments

`filepath`	string. Full path to the input patch file. An example file is ‘FREESURFER_HOME/subjects/fsaverage/surf
`format`	one of ’auto’, ’asc’, or ’bin’. The format to assume. If set to ’auto’ (the default), binary format will be used unless the filepath ends with ’.asc’.

Value

named list with 2 entries: "faces": can be NULL, only available if the format is ASCII, see return value of `read.fs.patch.asc`. "vertices": numerical *n*x7 matrix. The columns are named, and appear in the following order: ’vert_index1’: the one-based (R-style) vertex index. ’x’, ’y’, ’z’: float vertex coordinates. ’is_border’: integer, 1 if the vertex lies on the patch border, 0 otherwise (treat as logical). ’raw_vtx’: integer, the raw vtx value encoding index and border. ’vert_index0’: the zero-based (C-style) vertex index.

See Also

Other patch functions: `fs.patch()`, `read.fs.patch.asc()`, `write.fs.patch()`

`read.fs.patch.asc` *Read FreeSurfer ASCII format patch.*

Description

An ASCII format patch is a part of a brain surface mesh, and is a mesh itself. It consists of vertices and faces. The ASCII patch format is very similar to the ASCII surface format. **Note:** The contents of ASCII and binary patch format files is different. The ASCII patch format is not ideal for parsing, and loading such files is currently quite slow.

Usage

```
read.fs.patch.asc(filepath)
```

Arguments

`filepath`	string. Full path to the input patch file in ASCII patch format.

Value

named list. The list has the following named entries: "vertices": see return value of `read.fs.patch`. "faces": numerical *n*x5 matrix. The columns are named, and appear in the following order: 'face_index1': the one-based (R-style) face index. 'vert1_index1', 'vert2_index1', 'vert3_index1': integer vertex indices of the face, they are one-based (R-style). 'face_index0': the zero-based (C-style) face index.

See Also

Other patch functions: `fs.patch()`, `read.fs.patch()`, `write.fs.patch()`

`read.fs.surface` *Read file in FreeSurfer surface format or various mesh formats.*

Description

Read a brain surface mesh consisting of vertex and face data from a file in FreeSurfer binary or ASCII surface format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.white'.

Usage

```
read.fs.surface(filepath, format = "auto")
```

Arguments

`filepath`	string. Full path to the input surface file. Note: gzipped files are supported and gz format is assumed if the filepath ends with ".gz".
`format`	one of 'auto', 'asc', 'vtk', 'ply', 'gii', 'mz3', 'stl', 'byu', 'geo', 'ico', 'tri', 'obj', 'off' or 'bin'. The format to assume. If set to 'auto' (the default), binary format will be used unless the filepath ends with '.asc'.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. This datastructure is known as a is a *face index set*. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
surface_file = system.file("extdata", "lh.tinysurface",
                        package = "freesurferformats", mustWork = TRUE);
mesh = read.fs.surface(surface_file);
cat(sprintf("Read data for %d vertices and %d faces. \n",
                        nrow(mesh$vertices), nrow(mesh$faces)));
```

`read.fs.surface.asc` *Read FreeSurfer ASCII format surface.*

Description

Read FreeSurfer ASCII format surface.

Usage

```
read.fs.surface.asc(filepath, with_values = TRUE, header_numlines = 2L)
```

Arguments

`filepath`	string. Full path to the input surface file in ASCII surface format.
`with_values`	logical, whether to read per-vertex and per-face values.
`header_numlines`	scalar positive integer, the number of header lines.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is also known as *srf* format.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.bvsrf` *Read Brainvoyager srf format (.srf) mesh as surface.*

Description

Read a mesh and associated data like color and normals from a binary file in BrainVoyager SRF mesh format.

Usage

```
read.fs.surface.bvsrf(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in SRF mesh format.

Value

fs.surface instance

References

The srf format spec is at https://support.brainvoyager.com/brainvoyager/automation-development/84-file-formats/344-users-guide-2-3-the-format-of-srf-files.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.byu` *Read mesh in BYU format.*

Description

The BYU or Brigham Young University format is an old ASCII mesh format that is based on fixed character positions in lines (as opposed to whitespace-separated elements). I consider it a bit counter-intuitive.

Usage

```
read.fs.surface.byu(filepath, part = 1L)
```

Arguments

`filepath`	full path of the file in BYU format.
`part`	positive integer, the index of the mesh that should be loaded from the file. Only relevant if the file contains more than one mesh.

Value

an ‘fs.surface‘ instance, aka a mesh

References

See http://www.eg-models.de/formats/Format_Byu.html for a format description.

`read.fs.surface.geo` *Read GEO format mesh as surface.*

Description

This reads meshes from text files in GEO mesh format. This is an ASCII format.

Usage

```
read.fs.surface.geo(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in GEO mesh format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is a fixed width format.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.gii` *Read GIFTI format mesh as surface.*

Description

Read GIFTI format mesh as surface.

Usage

```
read.fs.surface.gii(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in GIFTI format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Other gifti readers: `read.fs.annot.gii()`, `read.fs.label.gii()`, `read.fs.morph.gii()`

`read.fs.surface.ico` *Read ICO format mesh as surface.*

Description

This reads meshes from text files in ICO / TRI mesh format. This format is not to be confused with the the image format used to store tiny icons.

Usage

```
read.fs.surface.ico(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in ICO or TRI mesh format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is a fixed width format.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.mz3` *Read surface mesh in mz3 format, used by Surf-Ice.*

Description

The mz3 format is a binary file format that can store a mesh (vertices and faces), and optionally per-vertex colors or scalars.

Usage

```
read.fs.surface.mz3(filepath)
```

Arguments

`filepath`	full path to surface mesh file in mz3 format.

Value

an ‘fs.surface‘ instance. If the mz3 file contained RGBA per-vertex colors or scalar per-vertex data, these are available in the ’metadata’ property.

References

See https://github.com/neurolabusc/surf-ice for details on the format.

`read.fs.surface.obj` *Read OBJ format mesh as surface.*

Description

This reads meshes from text files in Wavefront OBJ mesh format. This is an ASCII format.

Usage

```
read.fs.surface.obj(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in Wavefront object mesh format. Files with non-standard vertex colors (3 additional float fields after the vertex coordinates in order R, G, B) are supported, and the colors will be returned in the field 'vertex_colors' if present.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is a simple but very common mesh format supported by many applications, well suited for export.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.off` *Read Object File Format (OFF) mesh as surface.*

Description

This reads meshes from text files in OFF mesh format. This is an ASCII format.

Usage

```
read.fs.surface.off(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in OFF mesh format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.ply` *Read Stanford PLY format mesh as surface.*

Description

This reads meshes from text files in PLY format. Note that this does not read arbitrary data from PLY files, i.e., PLY files can store data that is not supported by this function.

Usage

```
read.fs.surface.ply(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in Stanford Triangle (PLY) format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is by far not a complete PLY format reader. It can read PLY mesh files which were written by `write.fs.surface.ply` and Blender. Vertex colors and Blender vertex normals are currently ignored (but files with them are supported in the sense that the mesh data will be read correctly).

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.surface.stl` *Read mesh in STL format, auto-detecting ASCII versus binary format version.*

Description

Read mesh in STL format, auto-detecting ASCII versus binary format version.

Usage

```
read.fs.surface.stl(filepath, digits = 6L, is_ascii = "auto")
```

Arguments

`filepath`	full path to surface mesh file in STL format.
`digits`	the precision (number of digits after decimal separator) to use when determining whether two x,y,z coords define the same vertex. This is used when the polygon soup is turned into an indexed mesh.
`is_ascii`	logical, whether the file is in the ASCII version of the STL format (as opposed to the binary version). Can also be the character string 'auto', in which case the function will try to auto-detect the format.

Value

an 'fs.surface' instance, the mesh.

Note

The mesh is stored in the file as a polygon soup, which is transformed into an index mesh by this function.

`read.fs.surface.stl.bin` *Read surface mesh in STL binary format.*

Description

The STL format is a mesh format that is often used for 3D printing, it stores geometry information. It is known as stereolithography format. A binary and an ASCII version exist. This function reads the binary version.

Usage

```
read.fs.surface.stl.bin(filepath, digits = 6L)
```

Arguments

`filepath`	full path to surface mesh file in STL format.
`digits`	the precision (number of digits after decimal separator) to use when determining whether two x,y,z coords define the same vertex. This is used when the polygon soup is turned into an indexed mesh.

Value

an 'fs.surface' instance.

Note

The STL format does not use indices into a vertex list to define faces, instead it repeats vertex coords in each face ('polygon soup').

References

See https://en.wikipedia.org/wiki/STL_(file_format) for the format spec.

`read.fs.surface.vtk` *Read VTK ASCII format mesh as surface.*

Description

This reads meshes (vtk polygon datasets) from text files in VTK ASCII format. See https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf for format spec. Note that this function does **not** read arbitrary VTK datasets, i.e., it supports only a subset of the possible contents of VTK files (i.e., polygon meshes).

Usage

```
read.fs.surface.vtk(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in VTK ASCII format.

Value

named list. The list has the following named entries: "vertices": nx3 double matrix, where n is the number of vertices. Each row contains the x,y,z coordinates of a single vertex. "faces": nx3 integer matrix. Each row contains the vertex indices of the 3 vertices defining the face. WARNING: The indices are returned starting with index 1 (as used in GNU R). Keep in mind that you need to adjust the index (by substracting 1) to compare with data from other software.

Note

This is by far not a complete VTK format reader.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.fs.transform` *Load transformation matrix from a file.*

Description

Load transformation matrix from a file.

Usage

```
read.fs.transform(filepath, format = "auto")
```

Arguments

`filepath`	character string, the full path to the transform file.
`format`	character string, the file format. Currently 'auto' (guess based on file extension), 'xfm' (for xform format) or 'dat' (for tkregister style, e.g. register.dat) are supported.

Value

named list, the 'matrix field contains a '4x4 numerical matrix, the transformation matrix. Other fields may exist, depending on the parsed format.

Note

Currently this function has been tested with linear transformation files only, all others are unsupported.

See Also

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`

Examples

```
tf_file = system.file("extdata", "talairach.xfm",
                               package = "freesurferformats",
                               mustWork = TRUE);
transform = read.fs.transform(tf_file);
transform$matrix;
```

`read.fs.transform.dat` *Load transformation matrix from a tkregister dat file.*

Description

Load transformation matrix from a tkregister dat file.

Usage

```
read.fs.transform.dat(filepath)
```

Arguments

`filepath`	character string, the full path to the transform file.

Value

4x4 numerical matrix, the transformation matrix

See Also

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
tf_file = system.file("extdata", "register.dat",
                               package = "freesurferformats",
                               mustWork = TRUE);
transform = read.fs.transform.dat(tf_file);
transform$matrix;
```

`read.fs.transform.lta` *Load transformation matrix from a FreeSurfer linear transform array (LTA) file.*

Description

Load transformation matrix from a FreeSurfer linear transform array (LTA) file.

Usage

```
read.fs.transform.lta(filepath)
```

Arguments

`filepath`	character string, the full path to the transform file.

Value

4x4 numerical matrix, the transformation matrix

Note

I found no spec for the LTA file format, only example files, so this function should be used with care. If you have a file that is not parsed correctly, please open an issue and attach it.

See Also

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`, `sm1to0()`

Examples

```
tf_file = system.file("extdata", "talairach.lta",
 package = "freesurferformats", mustWork = TRUE);
transform = read.fs.transform.lta(tf_file);
transform$matrix;
```

`read.fs.transform.xfm` *Load transformation matrix from an XFM file.*

Description

Load transformation matrix from an XFM file.

Usage

```
read.fs.transform.xfm(filepath)
```

Arguments

`filepath`	character string, the full path to the transform file.

Value

4x4 numerical matrix, the transformation matrix

Note

Currently this function has been tested with linear transformation files only, all others are unsupported.

See Also

Other header coordinate space: `mghheader.is.ras.valid`(), `mghheader.ras2vox.tkreg`(), `mghheader.ras2vox`(), `mghheader.scanner2tkreg`(), `mghheader.tkreg2scanner`(), `mghheader.vox2ras.tkreg`(), `mghheader.vox2ras`(), `read.fs.transform.dat`(), `read.fs.transform.lta`(), `read.fs.transform`(), `sm0to1`(), `sm1to0`()

Examples

```
    tf_file = system.file("extdata", "talairach.xfm",
                              package = "freesurferformats",
                              mustWork = TRUE);
    transform = read.fs.transform.xfm(tf_file);
    transform$matrix;
```

`read.fs.volume` *Read volume file in MGH, MGZ or NIFTI format*

Description

Read multi-dimensional brain imaging data from a file.

Usage

```
read.fs.volume(
  filepath,
  format = "auto",
  flatten = FALSE,
  with_header = FALSE,
  drop_empty_dims = FALSE
)
```

Arguments

`filepath`	string. Full path to the input MGZ, MGH or NIFTI file.
`format`	character string, one one of 'auto', 'nii', 'mgh' or 'mgz'. The format to assume. If set to 'auto' (the default), the format will be derived from the file extension.
`flatten`	logical. Whether to flatten the return volume to a 1D vector. Useful if you know that this file contains 1D morphometry data.
`with_header`	logical. Whether to return the header as well. If TRUE, return an instance of class 'fs.volume' for data with at least 3 dimensions, a named list with entries "data" and "header". The latter is another named list which contains the header data. These header entries exist: "dtype": int, one of: 0=MRI_UCHAR; 1=MRI_INT; 3=MRI_FLOAT; 4=MRI_SHORT. "voldim": integer vector. The volume (=data) dimensions. E.g., c(256, 256, 256, 1). These header entries may exist: "vox2ras_matrix" (exists if "ras_good_flag" is 1), "mr_params" (exists if "has_mr_params" is 1). See the 'mghheader.*' functions, like `mghheader.vox2ras.tkreg`, to compute more information from the header fields.
`drop_empty_dims`	logical, whether to drop empty dimensions of the returned data

Value

data, multi-dimensional array. The brain imaging data, one value per voxel. The data type and the dimensions depend on the data in the file, they are read from the header. If the parameter flatten is 'TRUE', a numeric vector is returned instead. Note: The return value changes if the parameter with_header is 'TRUE', see parameter description.

See Also

To derive more information from the header, see the ‘mghheader.*‘ functions, like `mghheader.vox2ras.tkreg`.

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Examples

```
brain_image = system.file("extdata", "brain.mgz",
                           package = "freesurferformats",
                           mustWork = TRUE);
vd = read.fs.volume(brain_image);
cat(sprintf("Read voxel data with dimensions %s. Values: min=%d, mean=%f, max=%d.\n",
             paste(dim(vd), collapse = ' '), min(vd), mean(vd), max(vd)));
# Read it again with full header data:
vdh = read.fs.volume(brain_image, with_header = TRUE);
# Use the vox2ras matrix from the header to compute RAS coordinates at CRS voxel (0, 0, 0):
vox2ras_matrix = mghheader.vox2ras(vdh)
vox2ras_matrix %*% c(0,0,0,1);
```

`read.fs.volume.nii` *Turn a 3D or 4D ‘oro.nifti‘ instance into an ‘fs.volume‘ instance with complete header.*

Description

This is work in progress. This function takes an ‘oro.nifti‘ instance and computes the MGH header fields from the NIFTI header data, allowing for proper orientation of the contained image data (see `mghheader.vox2ras` and related functions). Currently only few datatypes are supported, and the ‘sform‘ header field needs to be present in the NIFTI instance.

Usage

```
read.fs.volume.nii(
  filepath,
  flatten = FALSE,
  with_header = FALSE,
  drop_empty_dims = FALSE,
  do_rotate = FALSE,
  ...
)
```

Arguments

`filepath`	instance of class ‘nifti‘ from the ‘oro.nifti‘ package, or a path to a NIFTI file as a character string.
`flatten`	logical. Whether to flatten the return volume to a 1D vector. Useful if you know that this file contains 1D morphometry data.
`with_header`	logical. Whether to return the header as well. If TRUE, return an instance of class ‘fs.volume‘ for data with at least 3 dimensions, a named list with entries "data" and "header". The latter is another named list which contains the header data. These header entries exist: "dtype": int, one of: 0=MRI_UCHAR; 1=MRI_INT; 3=MRI_FLOAT; 4=MRI_SHORT. "voldim": integer vector. The volume (=data) dimensions. E.g., c(256, 256, 256, 1). These header entries may exist: "vox2ras_matrix" (exists if "ras_good_flag" is 1), "mr_params" (exists if "has_mr_params" is 1). See the ‘mghheader.*‘ functions, like `mghheader.vox2ras.tkreg`, to compute more information from the header fields.
`drop_empty_dims`	logical, whether to drop empty dimensions of the returned data
`do_rotate`	logical, whether to rotate 3D volumes to compensate for storage order. WIP.
`...`	extra parameters passed to `oro.nifti::readNIfTI`. Leave this alone unless you know what you are doing.

Value

an ‘fs.volume‘ instance. The ‘header‘ fields are computed from the NIFTI header. The ‘data‘ array is rotated into FreeSurfer storage order, but otherwise returned as present in the input NIFTI instance, i.e., no values are changed in any way.

Note

This is not supposed to be used to read 1D morphometry data from NIFTI files generated by FreeSurfer (e.g., by converting ‘lh.thickness‘ to NIFTI using ‘mri_convert‘): the FreeSurfer NIFTI hack is not supported by oro.nifti.

References

See https://nifti.nimh.nih.gov/nifti-1/ for the NIfTI-1 data format spec.

See Also

`oro.nifti::readNIfTI`, `read.fs.mgh`

Examples

```
## Not run:
   base_file = "~/data/subject1_only/subject1/mri/brain";  # missing file ext.
   mgh_file = paste(base_file, '.mgz', sep='');  # the standard MGH/MGZ file
   nii_file = paste(base_file, '.nii', sep='');   # NIFTI file generated with mri_convert
   brain_mgh = read.fs.mgh(mgh_file, with_header = TRUE);
   brain_nii = read.fs.volume.nii(nii_file, with_header = TRUE);
```

```
    all(brain_nii$data == brain_mgh$data);                            # output: TRUE
    all(mghheader.vox2ras(brain_nii) == mghheader.vox2ras(brain_mgh))   # output: TRUE

## End(Not run)
```

`read.fs.weight` *Read file in FreeSurfer weight or w format*

Description

Read morphometry data in weight format (aka ‘w‘ files). A weight format file contains morphometry data for a set of vertices, defined by their index in a surface. This can be only a **subset** of the surface vertices.

Usage

```
read.fs.weight(filepath, format = "auto")
```

Arguments

`filepath`	string. Full path to the input weight file. Weight files typically have the file extension ’.w’, but that is not enforced.
`format`	one of ’auto’, ’asc’, or ’bin’. The format to assume. If set to ’auto’ (the default), binary format will be used unless the filepath ends with ’.asc’.

Value

the indices and weight data, as a named list. Entries: "vertex_indices": vector of *n* vertex indices. They are stored zero-based in the file, but are returned one-based (R-style). "value": double vector of length *n*, the morphometry data for the vertices. The data can be whatever you want.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(` `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`read.mesh.brainvoyager` *Read Brainvoyager srf format (.srf) mesh.*

Description

Read a mesh and associated data like color and normals from a binary file in BrainVoyager SRF mesh format.

Usage

```
read.mesh.brainvoyager(filepath)
```

Arguments

`filepath`	string. Full path to the input surface file in SRF mesh format.

Value

named list of the elements in the file.

References

The srf format spec is at https://support.brainvoyager.com/brainvoyager/automation-development/84-file-formats/344-users-guide-2-3-the-format-of-srf-files.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

`read.nifti1.data` *Read raw NIFTI v1 data from file (which may contain the FreeSurfer hack).*

Description

Read raw NIFTI v1 data from file (which may contain the FreeSurfer hack).

Usage

```
read.nifti1.data(filepath, drop_empty_dims = TRUE, header = NULL)
```

Arguments

`filepath`	path to a NIFTI v1 file (single file version), which can contain the FreeSurfer hack.
`drop_empty_dims`	logical, whether to drop empty dimensions in the loaded data array.
`header`	optional nifti header obtained from `read.nifti1.header`. Will be loaded automatically if left at 'NULL'.

Value

the data in the NIFTI v1 file. Note that the NIFTI v1 header information (scaling, units, etc.) is not applied in any way: the data are returned raw, as read from the file. The information in the header is used to read the data with the proper data type and size.

Note

The FreeSurfer hack is a non-standard way to save long vectors (one dimension greater than 32k entries) in NIFTI v1 files. Files with this hack are produced when converting MGH or MGZ files containing such long vectors with the FreeSurfer 'mri_convert' tool.

`read.nifti1.header` *Read NIFTI v1 header from file (which may contain the FreeSurfer hack).*

Description

Read NIFTI v1 header from file (which may contain the FreeSurfer hack).

Usage

```
read.nifti1.header(filepath)
```

Arguments

`filepath`	path to a NIFTI v1 file (single file version), which can contain the FreeSurfer hack.

Value

named list with NIFTI 1 header fields.

Note

The FreeSurfer hack is a non-standard way to save long vectors (one dimension greater than 32767 entries) in NIFTI v1 files. Files with this hack are produced when converting MGH or MGZ files containing such long vectors with the FreeSurfer 'mri_convert' tool.

`read.nifti2.data` *Read raw data from NIFTI v2 file.*

Description

Read raw data from NIFTI v2 file.

Usage

```
read.nifti2.data(filepath, header = NULL, drop_empty_dims = TRUE)
```

Arguments

`filepath`	path to a NIFTI v2 file.
`header`	optional nifti v2 header obtained from `read.nifti2.header`. Will be loaded automatically if left at ‘NULL‘.
`drop_empty_dims`	logical, whether to drop empty dimensions in the loaded data array.

Value

the data in the NIFTI v2 file. Note that the NIFTI v2 header information (scaling, units, etc.) is not applied in any way: the data are returned raw, as read from the file. The information in the header is used to read the data with the proper data type and size.

`read.nifti2.header` *Read NIFTI v2 header from file.*

Description

Read NIFTI v2 header from file.

Usage

```
read.nifti2.header(filepath)
```

Arguments

`filepath`	path to a NIFTI v2 file.

Value

named list with NIFTI 2 header fields.

---

read.smp.brainvoyager *Read Brainvoyager statistical surface results from SMP file.*

Description

Read Brainvoyager statistical surface results from SMP file.

Usage

```
read.smp.brainvoyager(filepath)
```

Arguments

`filepath`	character string, path to file in Brainvoyager SMP file format

Value

named list of file contents

Note

Currently only SMP file versions 1 to 5 are supported, as these are the only ones for which a spec is available. The version is encoded in the file header.

References

see `https://support.brainvoyager.com/brainvoyager/automation-development/84-file-formats/40-the-format-of-smp-files` for the spec

Examples

```
## Not run:
 # Surface mesh, requires BV demo dataset from website:
 sf = read.fs.surface.bvsrf("~/data/BrainTutorData/CG_LHRH_D65534.srf");
 # Surface map of cortical thickness. Needs to be created in BV.
 smp_file = "~/data/BrainTutorData/CG_LHRH_D65534_Thickness.smp";
 smp = read.smp.brainvoyager(smp_file);
 smp_data = read.fs.morph.bvsmp(smp); # could also pass smp_file.
 fsbrain::vis.fs.surface(sf, per_vertex_data = smp_data);

## End(Not run)
```

`readable.files` *Find files with the given base name and extensions that exist.*

Description

Note that in the current implementation, the case of the filepath and the extension must match.

Usage

```
readable.files(
  filepath,
  precedence = c(".mgh", ".mgz"),
  error_if_none = TRUE,
  return_all = FALSE
)
```

Arguments

`filepath`	character string, path to a file without extension
`precedence`	vector of character strings, the file extensions to check. Must include the dot (if you expect one).
`error_if_none`	logical, whether to raise an error if none of the files exist
`return_all`	logical, whether to return all readable files instead of just the first one

Value

character string, the path to the first existing file (or ‘NULL‘ if none of them exists).

`read_nisurface` *Read a surface, based on the file path without extension.*

Description

Tries to read all files which can be constructed from the base path and the given extensions.

Usage

```
read_nisurface(filepath_noext, extensions = c("", ".asc", ".gii"), ...)
```

Arguments

`filepath_noext`	character string, the full path to the input surface file without file extension.
`extensions`	vector of character strings, the file extensions to try.
`...`	parameters passed on to `read_nisurfacefile`. Allows you to set the ‘methods‘.

Value

an instance of ‘fs.surface‘, read from the file. See `read.fs.surface` for details. If none of the reader methods succeed, an error is raised.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
## Not run:
    surface_filepath_noext =
     paste(get_optional_data_filepath("subjects_dir/subject1/surf/"),
     'lh.white', sep="");
    mesh = read_nisurface(surface_filepath_noext);
    mesh;

## End(Not run)
```

`read_nisurfacefile` *S3 method to read a neuroimaging surface file.*

Description

Tries to read the file with all implemented surface format reader methods. The file must exist. With the default settings, one can read files in the following surface formats: 1) FreeSurfer binary surface format (e.g., ‘surf/lh.white‘). 2) FreeSurfer ASCII surface format (e.g., ‘surf/lh.white,asc‘). 3) GIFTI surface format, only if package ‘gifti‘ is installed. See `gifti::read_gifti` for details. Feel free to implement additional methods. Hint:keep in mind that they should return one-based indices.

Usage

```
read_nisurfacefile(filepath, methods = c("fsnative", "fsascii", "gifti"), ...)
```

Arguments

`filepath`	character string, the full path to the input surface file.
`methods`	list of character strings, the formats to try. Each of these must have a function called `read_nisurface.<method>`, which must return an ‘fs.surface‘ instance on success.
`...`	parameters passed on to the individual methods

Value

an instance of ‘fs.surface‘, read from the file. See `read.fs.surface` for details. If none of the reader methods succeed, an error is raised.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
surface_file = system.file("extdata", "lh.tinysurface",
                          package = "freesurferformats", mustWork = TRUE);
mesh = read_nisurface(surface_file);
mesh;
```

`read_nisurfacefile.fsascii`

Read a FreeSurfer ASCII surface file.

Description

Read a FreeSurfer ASCII surface file.

Usage

```
## S3 method for class 'fsascii'
read_nisurfacefile(filepath, ...)
```

Arguments

`filepath`	character string, the full path to the input surface file.
`...`	parameters passed to `read.fs.surface.asc`.

Value

an instance of ‘fs.surface‘, read from the file. See `read.fs.surface` for details. If none of the reader methods succeed, an error is raised.

`read_nisurfacefile.fsnative` *Read a FreeSurfer ASCII surface file.*

Description

Read a FreeSurfer ASCII surface file.

Usage

```
## S3 method for class 'fsnative'
read_nisurfacefile(filepath, ...)
```

Arguments

`filepath`	character string, the full path to the input surface file.
`...`	parameters passed to `read.fs.surface`.

Value

an instance of ‘fs.surface‘, read from the file. See `read.fs.surface` for details. If none of the reader methods succeed, an error is raised.

`read_nisurfacefile.gifti` *Read a gifti file as a surface.*

Description

Read a gifti file as a surface.

Usage

```
## S3 method for class 'gifti'
read_nisurfacefile(filepath, ...)
```

Arguments

`filepath`	character string, the full path to the input surface file.
`...`	ignored

Value

an instance of ‘fs.surface‘, read from the file. See `read.fs.surface` for details. If none of the reader methods succeed, an error is raised.

`rotate2D` *Rotate a 2D matrix in 90 degree steps.*

Description

Rotate a 2D matrix in 90 degree steps.

Usage

```
rotate2D(slice, degrees = 90)
```

Arguments

`slice`	a 2D matrix
`degrees`	integer, must be a (positive or negative) multiple of 90

Value

2D matrix, the rotated matrix

`rotate3D` *Rotate a 3D array in 90 degree steps.*

Description

Rotate a 3D array in 90 degree steps along an axis. This leads to an array with different dimensions.

Usage

```
rotate3D(volume, axis = 1L, degrees = 90L)
```

Arguments

`volume`	a 3D image volume
`axis`	positive integer in range 1L..3L or an axis name, the axis to use.
`degrees`	integer, must be a (positive or negative) multiple of 90L.

Value

a 3D image volume, rotated around the axis. The dimensions may or may not be different from the input image, depending on the rotation angle.

See Also

Other volume math: `flip3D()`

`sm0to1` *Adapt spatial transformation matrix for 1-based indices.*

Description

Adapt spatial transformation matrix for 1-based indices.

Usage

```
sm0to1(tf_matrix)
```

Arguments

`tf_matrix`	4x4 numerical matrix, the input spatial transformation matrix, suitable for 0-based indices. Typically this is a vox2ras matrix obtained from functions like `mghheader.vox2ras`.

Value

4x4 numerical matrix, adapted spatial transformation matrix, suitable for 1-based indices.

See Also

`sm1to0` for the inverse operation

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm1to0()`

`sm1to0` *Adapt spatial transformation matrix for 0-based indices.*

Description

Adapt spatial transformation matrix for 0-based indices.

Usage

```
sm1to0(tf_matrix)
```

Arguments

`tf_matrix`	4x4 numerical matrix, the input spatial transformation matrix, suitable for 1-based indices.

Value

4x4 numerical matrix, adapted spatial transformation matrix, suitable for 0-based indices.

See Also

`sm0to1` for the inverse operation

Other header coordinate space: `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras.tkreg()`, `mghheader.vox2ras()`, `read.fs.transform.dat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `read.fs.transform()`, `sm0to1()`

`surfaceras.to.ras` *Translate surface RAS coordinates, as used in surface vertices and surface labels, to volume RAS.*

Description

Translate surface RAS coordinates, as used in surface vertices and surface labels, to volume RAS.

Usage

```
surfaceras.to.ras(
  header_cras,
  sras_coords,
  first_voxel_RAS = c(1, 1, 1),
  invert_transform = FALSE
)
```

Arguments

`header_cras`	an MGH header instance from which to extract the cras (center RAS), or the cras vector, i.e., a numerical vector of length 3
`sras_coords`	nx3 numerical vector, the input surface RAS coordinates. Could be the vertex coordinates of an 'fs.surface' instance, or the RAS coords from a surface label. Use the orig surfaces.
`first_voxel_RAS`	the RAS of the first voxel, see `mghheader.centervoxelRAS.from.firstvoxelRAS` for details. Ignored if 'header_cras' is a vector.
`invert_transform`	logical, whether to invert the transform. Do not use this, call `link{ras.to.surfaceras}` instead.

Value

the RAS coords for the input sras_coords

Note

The RAS can be computed from Surface RAS by adding the center RAS coordinates, i.e., it is nothing but a translation.

`surfaceras.to.talairach` *Compute Talairach RAS for surface RAS (e.g., vertex coordinates).*

Description

Compute Talairach RAS for surface RAS (e.g., vertex coordinates).

Usage

```
surfaceras.to.talairach(
  sras_coords,
  talairach,
  header_cras,
  first_voxel_RAS = c(1, 1, 1)
)
```

Arguments

`sras_coords`	nx3 numerical vector, the input surface RAS coordinates. Could be the vertex coordinates of an 'fs.surface' instance, or the RAS coords from a surface label. Use the orig surfaces.
`talairach`	the 4x4 numerical talairach matrix, or a character string which will be interpreted as the path to an xfm file containing the matrix (typically '$SUBJECTS_DIR/$subject/mri/transfor
`header_cras`	an MGH header instance from which to extract the cras (center RAS), or the cras vector, i.e., a numerical vector of length 3
`first_voxel_RAS`	the RAS of the first voxel, see `mghheader.centervoxelRAS.from.firstvoxelRAS` for details. Ignored if 'header_cras' is a vector.

Value

The Talairach RAS coordinates for the vertices of the orig surfaces (or coords in surface RAS space). Based on linear transform.

`talairachras.to.ras` *Compute MNI talairach coordinates from RAS coords.*

Description

Compute MNI talairach coordinates from RAS coords.

Usage

```
talairachras.to.ras(tal_ras_coords, talairach)
```

Arguments

`tal_ras_coords`	coordinate matrix in Talairach RAS space
`talairach`	the 4x4 numerical talairach matrix, or a character string which will be interpreted as the path to an xfm file containing the matrix (typically ‘$SUBJECTS_DIR/$subject/mri/transfor

Value

the Talairach RAS coordinates for the given RAS coordinates. They are based on a linear transform.

Note

You can use this to compute the Talairach coordinate of a voxel, based on its RAS coordinate.

References

see https://en.wikipedia.org/wiki/Talairach_coordinates

`vertex.euclid.dist` *Compute Euclidean distance between two vertices v1 and v2.*

Description

Compute Euclidean distance between two vertices v1 and v2.

Usage

```
vertex.euclid.dist(surface, v1, v2)
```

Arguments

`surface`	an fs.surface instance
`v1`	positive integer, vertex index of v1
`v2`	positive integer, vertex index of v2

Value

double, the Euclidean distance between v1 and v2

See Also

Other Euclidean distance util functions: `closest.vert.to.point()`, `vertexdists.to.point()`

`vertexdists.to.point` *Compute Euclidean distance from all mesh vertices to given point.*

Description

Compute Euclidean distance from all mesh vertices to given point.

Usage

```
vertexdists.to.point(surface, point_coords)
```

Arguments

`surface`	an fs.surface instance
`point_coords`	double vector of length 3, the xyz coords of a single point.

Value

double vector of distances

See Also

Other Euclidean distance util functions: `closest.vert.to.point()`, `vertex.euclid.dist()`

`write.fs.annot` *Write annotation to binary file.*

Description

Write an annotation to a FreeSurfer binary format annotation file in the new format (v2). An annotation (or brain parcellation) assigns each vertex to a label (or region). One of the regions is often called 'unknown' or similar and all vertices which are not relevant for the parcellation are assigned this label.

Usage

```
write.fs.annot(
  filepath,
  num_vertices = NULL,
  colortable = NULL,
  labels_as_colorcodes = NULL,
  labels_as_indices_into_colortable = NULL,
  fs.annot = NULL
)
```

Arguments

`filepath`	string, path to the output file
`num_vertices`	integer, the number of vertices of the surface. Must be given unless parameter ‘fs.annot‘ is not NULL.
`colortable`	dataframe that contains one region per row. Required columns are: ’struct_name’: character string, the region name. ’r’: integer in range 0-255, the RGB color value for the red channel. ’g’: same for the green channel. ’b’: same for the blue channel. ’a’: the alpha (transparency) channel value. Optional columns are: ’code’: the color code. Will be computed if not set. Note that you can pass the dataframe returned by `read.fs.annot` as ’colortable_df’. Only required if ‘labels_as_indices_into_colortable‘ is used.
`labels_as_colorcodes`	vector of *n* integers. The first way to specify the labels. Each integer is a colorcode, that has been computed from the RGBA color values of the regions in the colortable as `r + g*2^8 + b*2^16 + a*2^24`. If you do not already have these color codes, it is way easier to set this to NULL and define the labels as indices into the colortable, see parameter ‘labels_as_indices_into_colortable‘.
`labels_as_indices_into_colortable`	vector of *n* integers, the second way to specify the labels. Each integer is an index into the rows of the colortable. Indices start with 1. This parameter and ‘labels_as_colorcodes‘ are mutually exclusive, but required.
`fs.annot`	instance of class ‘fs.annot‘. If passed, this takes precedence over all other parameters and they should all be NULL (with the exception of ‘filepath‘).

See Also

Other atlas functions: `colortable.from.annot()`, `read.fs.annot()`, `read.fs.colortable()`, `write.fs.annot.gii()`, `write.fs.colortable()`

Examples

```
## Not run:
   # Load annotation
   annot_file = system.file("extdata", "lh.aparc.annot.gz",
                            package = "freesurferformats",
                            mustWork = TRUE);
   annot = read.fs.annot(annot_file);
```

```
    # New method: write the annotation instance:
    write.fs.annot(tempfile(fileext=".annot"), fs.annot=annot);

    # Old method: write it from its parts:
    write.fs.annot(tempfile(fileext=".annot"), length(annot$vertices),
     annot$colortable_df, labels_as_colorcodes=annot$label_codes);

## End(Not run)
```

`write.fs.annot.gii` *Write annotation to GIFTI file.*

Description

Write an annotation to a GIFTI XML file.

Usage

```
write.fs.annot.gii(filepath, annot)
```

Arguments

`filepath`	string, path to the output file.
`annot`	fs.annot instance, an annotation.

Note

This function does not write a GIFTI file that is valid according to the specification: it stores extra color data in the Label nodes, and there is more than one Label in the LabelTable node.

See Also

Other atlas functions: `colortable.from.annot()`, `read.fs.annot()`, `read.fs.colortable()`, `write.fs.annot()`, `write.fs.colortable()`

Other gifti writers: `write.fs.label.gii()`, `write.fs.morph.gii()`, `write.fs.surface.gii()`

Examples

```
## Not run:
    # Load annotation
    annot_file = system.file("extdata", "lh.aparc.annot.gz",
                                package = "freesurferformats",
                                mustWork = TRUE);
    annot = read.fs.annot(annot_file);

    # New method: write the annotation instance:
    write.fs.annot.gii(tempfile(fileext=".annot"), annot);
```

```
## End(Not run)
```

`write.fs.colortable` *Write colortable file in FreeSurfer ASCII LUT format.*

Description

Write the colortable to a text file in FreeSurfer ASCII colortable lookup table (LUT) format. An example file is ‘FREESURFER_HOME/FreeSurferColorLUT.txt‘.

Usage

```
write.fs.colortable(filepath, colortable)
```

Arguments

`filepath,`	string. Full path to the output colormap file.
`colortable`	data.frame, a colortable as read by `read.fs.colortable`. Must contain the following columns: 'struct_name': character string, the label name. 'r': integer in range 0-255, the RGBA color value for the red channel. 'g': same for green channel. 'b': same for blue channel. 'a': same for alpha (transparency) channel. Can contain the following column: 'struct_index': integer, index of the struct entry. If this column does not exist, sequential indices starting at zero are created.

Value

the written dataframe, invisible. Note that this is will contain a column named 'struct_index', no matter whether the input colortable contained it or not.

See Also

Other atlas functions: `colortable.from.annot`(), `read.fs.annot`(), `read.fs.colortable`(), `write.fs.annot.gii`(), `write.fs.annot`()

Other colorLUT functions: `colortable.from.annot`(), `read.fs.colortable`()

`write.fs.curv` *Write file in FreeSurfer curv format*

Description

Write vertex-wise brain surface data to a file in FreeSurfer binary 'curv' format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.thickness', which contains n values. Each value represents the cortical thickness at the respective vertex in the brain surface mesh of bert.

Usage

```
write.fs.curv(filepath, data)
```

Arguments

`filepath,`	string. Full path to the output curv file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`data`	vector of doubles. The brain morphometry data to write, one value per vertex.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`write.fs.label` *Write vertex indices to file in FreeSurfer label format*

Description

Write vertex coordinates and vertex indices defining faces to a file in FreeSurfer binary surface format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/label/lh.cortex'.

Usage

```
write.fs.label(
  filepath,
  vertex_indices,
  vertex_coords = NULL,
  vertex_data = NULL,
  indices_are_one_based = TRUE
)
```

Arguments

`filepath`	string. Full path to the output label file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`vertex_indices`	instance of class ‘fs.label‘ or an integer vector, the label. The vertex indices included in the label. As returned by `read.fs.label`.
`vertex_coords`	an *n* x 3 float matrix of vertex coordinates, where *n* is the number of ’vertex_indices’. Optional, defaults to NULL, which will write placeholder data. The vertex coordinates are not used by any software I know (you should get them from the surface file). Will be used from ‘fs.label‘ instance if given.
`vertex_data`	a numerical vector of length *n*, where *n* is the number of ’vertex_indices’. Optional, defaults to NULL, which will write placeholder data. The vertex data are not used by any software I know (you should get them from a morphometry file). Will be used from ‘fs.label‘ instance if given.
`indices_are_one_based`	logical, whether the given indices are one-based, as is standard in R. Indices are stored zero-based in label files, so if this is TRUE, all indices will be incremented by one before writing them to the file. Defaults to TRUE. If FALSE, it is assumed that they are zero-based and they are written to the file as-is. Will be used from ‘fs.label‘ instance if given.

Value

dataframe, the dataframe that was written to the file (after the header lines).

See Also

Other label functions: `read.fs.label.gii()`, `read.fs.label.native()`, `read.fs.label()`

Examples

```
## Not run: 
    # Write a simple label containing only vertex indices:
    label_vertices = c(1,2,3,4,5,1000,2000,2323,34,34545,42);
    write.fs.label(tempfile(fileext=".label"), label_vertices);

    # Load a full label, write it back to a file:
    labelfile = system.file("extdata", "lh.entorhinal_exvivo.label",
     package = "freesurferformats", mustWork = TRUE);
    label = read.fs.label(labelfile, full=TRUE);
    write.fs.label(tempfile(fileext=".label"), label);

## End(Not run)
```

`write.fs.label.gii` *Write a binary surface label in GIFTI format.*

Description

The data will be written with intent 'NIFTI_INTENT_LABEL' and as datatype 'NIFTI_TYPE_INT32'. The label table will include entries 'positive' (label value 0), and 'negative' (label value 1).

Usage

```
write.fs.label.gii(filepath, vertex_indices, num_vertices_in_surface)
```

Arguments

`filepath`	string, the full path of the output GIFTI file.
`vertex_indices`	integer vector, the vertex indices which are part of the label (positive). All others not listed, up to num_vertices_in_surface, will be set to be negative.
`num_vertices_in_surface`	integer, the total number of vertices in the surface mesh. A GIFTI label is more like a mask/an annotation, so we need to know the number of vertices.

Value

format, string. The format that was used to write the data: "gii".

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Other gifti writers: `write.fs.annot.gii()`, `write.fs.morph.gii()`, `write.fs.surface.gii()`

Examples

```
label = c(1L, 23L, 44L); # the positive vertex indices
outfile = tempfile(fileext=".gii");
write.fs.label.gii(outfile, label, 50L);
```

`write.fs.mgh` *Write file in FreeSurfer MGH or MGZ format*

Description

Write brain data to a file in FreeSurfer binary MGH or MGZ format.

Usage

```
write.fs.mgh(
  filepath,
  data,
  vox2ras_matrix = NULL,
  mr_params = c(0, 0, 0, 0, 0),
  mri_dtype = "auto"
)
```

Arguments

`filepath`	string. Full path to the output curv file. If this ends with ".mgz", the file will be written gzipped (i.e., in MGZ instead of MGH format).
`data`	matrix or array of numerical values. The brain data to write. Must be integers or doubles. (The data type is set automatically to MRI_INT for integers and MRI_FLOAT for doubles in the MGH header).
`vox2ras_matrix`	4x4 matrix. An affine transformation matrix for the RAS transform that maps voxel indices in the volume to coordinates, such that for y(i1,i2,i3) (i.e., a voxel defined by 3 indices in the volume), the xyz coordinates are vox2ras_matrix*[i1 i2 i3 1]. If no matrix is given (or a NULL value), the ras_good flag will be 0 in the file. Defaults to NULL.
`mr_params`	double vector of length four (without fov) or five. The acquisition parameters, in order: tr, flipangle, te, ti, fov. Spelled out: repetition time, flip angle, echo time, inversion time, field-of-view. The unit for the three times is ms, the angle unit is radians. Defaults to c(0., 0., 0., 0., 0.) if omitted. Pass NULL if you do not want to write them at all.
`mri_dtype`	character string representing an MRI data type code or 'auto'. Valid strings are 'MRI_UCHAR' (1 byte unsigned integer), 'MRI_SHORT' (2 byte signed integer), 'MRI_INT' (4 byte signed integer) and 'MRI_FLOAT' (4 byte signed floating point). The default value ‘auto‘ will determine the data type from the type of the ‘data‘ parameter. It will use MRI_INT for integers, so you may be able to save space by manually settings the dtype if the range of your data does not require that. WARNING: If manually specified, no sanitation of any kind is performed. Leave this alone if in doubt.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`write.fs.morph` *Write morphometry data in a format derived from the given file name.*

Description

Given data and a morphometry file name, derive the proper format from the file extension and write the file.

Usage

```
write.fs.morph(filepath, data, format = "auto", ...)
```

Arguments

`filepath,`	string. The full file name. The format to use will be derived from the last characters, the suffix. Supported suffixes are "mgh" for MGH format, "mgz" for MGZ format, "smp" for Brainvoyager SMP format, "nii" or "nii.gz" for NIFTI v1 format, "gii" or "gii.gz" for GIFTI format, everything else will be treated as curv format.
`data,`	numerical vector. The data to write.
`format`	character string, the format to use. One of c("auto", "mgh", "mgz", "curv", "n1", "ni2", "gii"). The default setting "auto" will determine the format from the file extension.
`...`	additional parameters to pass to the respective writer function.

Value

character string. The format that was used to write the data. One of c("auto", "mgh", "mgz", "curv", "ni1", "ni2", "gii").

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.weight.asc()`, `write.fs.weight()`

`write.fs.morph.asc` *Write file in FreeSurfer ASCII curv format*

Description

Write vertex-wise brain surface data to a file in FreeSurfer ascii 'curv' format.

Usage

```
write.fs.morph.asc(filepath, data, coords = NULL)
```

Arguments

`filepath`	string. Full path to the output curv file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`data`	vector of doubles. The brain morphometry data to write, one value per vertex.
`coords`	optional, nx3 matrix of x,y,z coordinates, one row per vertex in 'data'. If ‘NULL‘, all zeroes will be written instead.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format`(), `fs.get.morph.file.format.from.filename`(
`read.fs.curv`(), `read.fs.mgh`(), `read.fs.morph.gii`(), `read.fs.morph`(), `read.fs.volume`(), `read.fs.weight`(), `write.fs.curv`(), `write.fs.label.gii`(), `write.fs.mgh`(), `write.fs.morph.gii`(), `write.fs.morph.ni1`(), `write.fs.morph.ni2`(), `write.fs.morph.smp`(), `write.fs.morph.txt`(), `write.fs.morph`(), `write.fs.weight.asc`(), `write.fs.weight`()

`write.fs.morph.gii` *Write morphometry data in GIFTI format.*

Description

The data will be written with intent 'NIFTI_INTENT_SHAPE' and as datatype 'NIFTI_TYPE_FLOAT32'.

Usage

```
write.fs.morph.gii(filepath, data)
```

Arguments

`filepath`	string, the full path of the output GIFTI file.
`data`	numerical vector, the data to write. Will be coerced to double.

Value

format, string. The format that was used to write the data: "gii".

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Other gifti writers: `write.fs.annot.gii()`, `write.fs.label.gii()`, `write.fs.surface.gii()`

`write.fs.morph.ni1` *Write morphometry data in NIFTI v1 format.*

Description

Write morphometry data in NIFTI v1 format.

Usage

```
write.fs.morph.ni1(filepath, data, ...)
```

Arguments

`filepath`	string, the full path of the output NIFTI file. Should end with '.nii' or '.nii.gz'.
`data`	numerical vector, the data to write. Will be coerced to double.
`...`	extra parameters passed to `write.nifti1`.

Value

format, string. The format that was used to write the data: "ni1".

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Other nifti1 writers: `write.nifti1()`

`write.fs.morph.ni2` *Write morphometry data in NIFTI v2 format.*

Description

Write morphometry data in NIFTI v2 format.

Usage

```
write.fs.morph.ni2(filepath, data, ...)
```

Arguments

`filepath`	string, the full path of the output NIFTI file. Should end with '.nii' or '.nii.gz'.
`data`	numerical vector, the data to write. Will be coerced to double.
`...`	extra parameters passed to `write.nifti2`.

Value

format, string. The format that was used to write the data: "ni2".

Note

Not many software packages support NIFTI v2 format. If possible with your data, you may want to use NIFTI v1 instead.

See Also

`nifti.file.version` can be used to check whether a file is NIFTI v1 or v2 file.

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename()` `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

Other nifti2 writers: `write.nifti2()`

`write.fs.morph.smp` *Write morphometry data in Brainvoyager SMP format.*

Description

Write morphometry data in Brainvoyager SMP format.

Usage

```
write.fs.morph.smp(filepath, data, ...)
```

Arguments

`filepath`	string, the full path of the output SMP file.
`data`	numerical vector, the data to write. Will be coerced to double.
`...`	extra arguments passed to `write.smp.brainvoyager`. Allows yout to save in specific format versions.

Value

format, string. The format that was used to write the data.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(`
`read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`write.fs.morph.txt` *Write curv data to file in simple text format*

Description

Write vertex-wise brain surface data to a file in a simple text format: one value per line.

Usage

```
write.fs.morph.txt(filepath, data)
```

Arguments

`filepath`	string. Full path to the output curv file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`data`	vector of doubles. The brain morphometry data to write, one value per vertex.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(` `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph()`, `write.fs.weight.asc()`, `write.fs.weight()`

`write.fs.patch` *Write a surface patch*

Description

Write a surface patch, i.e. a set of vertices and patch border information, to a binary patch file.

Usage

```
write.fs.patch(filepath, patch)
```

Arguments

`filepath`	string. Full path to the output patch file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`patch`	an instance of class 'fs.patch', see `read.fs.patch`.

Value

the patch, invisible

See Also

Other patch functions: `fs.patch()`, `read.fs.patch.asc()`, `read.fs.patch()`

`write.fs.surface` *Write mesh to file in FreeSurfer binary surface format*

Description

Write vertex coordinates and vertex indices defining faces to a file in FreeSurfer binary surface format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.white'. This function writes the triangle version of the surface file format.

Usage

```
write.fs.surface(filepath, vertex_coords, faces, format = "auto")
```

Arguments

`filepath`	string. Full path to the output curv file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`format`	character string, the format to use. One of 'bin' for FreeSurfer binary surface format, 'asc' for FreeSurfer ASCII format, 'vtk' for VTK ASCII legacy format, 'ply' for Standford PLY format, 'off' for Object File Format, 'obj' for Wavefront object format, 'gii' for GIFTI format, 'mz3' for Surf-Ice MZ3 fomat, 'byu' for Brigham Young University (BYU) mesh format, or 'auto' to derive the format from the file extension given in parameter 'filepath'. With 'auto', a path ending in '.asc' is interpreted as 'asc', a path ending in '.vtk' as vtk, and so on for the other formats. Everything not matching any of these is interpreted as 'bin', i.e., FreeSurfer binary surface format.

Value

character string, the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`

Other mesh export functions: `write.fs.surface.obj()`, `write.fs.surface.off.ply2()`, `write.fs.surface.off()`, `write.fs.surface.ply2()`, `write.fs.surface.ply()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface(tempfile(), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.asc` *Write mesh to file in FreeSurfer ASCII surface format*

Description

Write vertex coordinates and vertex indices defining faces to a file in FreeSurfer ASCII surface format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.white.asc'.

Usage

```
write.fs.surface.asc(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.asc', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.asc(tempfile(fileext=".asc"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.bvsrf`

Write surface to Brainvoyager SRF file.

Description

Write surface to Brainvoyager SRF file.

Usage

```
write.fs.surface.bvsrf(
  filepath,
  vertex_coords,
  faces,
  normals = NULL,
  neighborhoods = NULL
)
```

Arguments

`filepath`	string. Full path to the output curv file. If it ends with ".gz", the file is written in gzipped format. Note that this is not common, and that other software may not handle this transparently.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`normals`	matrix of nx3 vertex normals (x,y,z)
`neighborhoods`	list of integer lists, the indices of the nearest neighbors for each vertex (an adjacency list). The sub list at index n contains the indices of the vertices in the 1-neighborhood of vertex n. The vertex indices in the sub lists must be zero-based.

Note

This function is experimental. Only SRF file format version 4 is supported.

`write.fs.surface.byu` *Write mesh to file in BYU ASCII format.*

Description

Write mesh to file in BYU ASCII format.

Usage

```
write.fs.surface.byu(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.byu', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

Note

This is a fixed field length ASCII format. Keep in mind that the BYU format expects the coordinates to be in the cube -1 to +1 on all three axes.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
```

```
    write.fs.surface.byu(tempfile(fileext=".byu"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.gii` *Write mesh to file in GIFTI surface format*

Description

Write vertex coordinates and vertex indices defining faces to a file in GIFTI surface format. For a subject (MRI image pre-processed with FreeSurfer) named 'bert', an example file would be 'bert/surf/lh.white.asc'.

Usage

```
write.fs.surface.gii(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.asc', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Other gifti writers: `write.fs.annot.gii()`, `write.fs.label.gii()`, `write.fs.morph.gii()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.gii(tempfile(fileext=".gii"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.mz3` *Write mesh to file in mz3 binary format.*

Description

Write mesh to file in mz3 binary format.

Usage

```
write.fs.surface.mz3(filepath, vertex_coords, faces, gzipped = TRUE)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.mz3', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`gzipped`	logical, whether to write a gzip compressed file

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

Note

This format is used by the surf-ice renderer. The format spec is at https://github.com/neurolabusc/surf-ice/tree/master/mz3.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.vtk()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.mz3(tempfile(fileext=".mz3"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.obj` *Write mesh to file in Wavefront object (.obj) format*

Description

The wavefront object format is a simply ASCII format for storing meshes.

Usage

```
write.fs.surface.obj(filepath, vertex_coords, faces, vertex_colors = NULL)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.vtk', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`vertex_colors`	vector of vertex colors. Will be written after the x, y, z coords on vertex lines. WARNING: This is NOT part of the official OBJ standard, and may not work with other software and even break some parsers.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

Note

Do not confuse the Wavefront object file format (.obj) with the OFF format (.off), they are not identical.

See Also

Other mesh export functions: `write.fs.surface.off.ply2()`, `write.fs.surface.off()`, `write.fs.surface.ply2()`, `write.fs.surface.ply()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.obj(tempfile(fileext=".obj"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.off` *Write mesh to file in Object File Format (.off)*

Description

The Object File Format is a simply ASCII format for storing meshes.

Usage

```
write.fs.surface.off(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.off', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

Note

Do not confuse the OFF format (.off) with the Wavefront object file format (.obj), they are not identical.

See Also

Other mesh export functions: `write.fs.surface.obj`(), `write.fs.surface.off.ply2`(), `write.fs.surface.ply2`(), `write.fs.surface.ply`(), `write.fs.surface`()

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.off(tempfile(fileext=".off"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.ply` *Write mesh to file in PLY format (.ply)*

Description

The PLY format is a versatile ASCII format for storing meshes. Also known as Polygon File Format or Stanford Triangle Format.

Usage

```
write.fs.surface.ply(filepath, vertex_coords, faces, vertex_colors = NULL)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.vtk', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.

`faces`	m x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`vertex_colors`	optional, matrix of RGBA vertex colors, number of rows must be the same as for vertex_coords. Color values must be integers in range 0-255. Alternatively, a vector of *n* RGB color strings can be passed.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

References

See http://paulbourke.net/dataformats/ply/ for the PLY format spec.

See Also

Other mesh export functions: `write.fs.surface.obj()`, `write.fs.surface.off.ply2()`, `write.fs.surface.off()`, `write.fs.surface.ply2()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.ply(tempfile(fileext=".ply"), mesh$vertices, mesh$faces);

    # save a version with RGBA vertex colors
    vertex_colors = matrix(rep(82L, 5*4), ncol=4);
    write.fs.surface.ply(tempfile(fileext=".ply"), mesh$vertices,
     mesh$faces, vertex_colors=vertex_colors);

## End(Not run)
```

`write.fs.surface.ply2` *Write mesh to file in PLY2 File Format (.ply2)*

Description

The PLY2 file format is a simply ASCII format for storing meshes. It is very similar to OFF and by far not as flexible as PLY.

Usage

```
write.fs.surface.ply2(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.off', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

See Also

Other mesh export functions: `write.fs.surface.obj()`, `write.fs.surface.off.ply2()`, `write.fs.surface.off()`, `write.fs.surface.ply()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.ply2(tempfile(fileext=".ply2"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.vtk` *Write mesh to file in VTK ASCII format*

Description

Write mesh to file in VTK ASCII format

Usage

```
write.fs.surface.vtk(filepath, vertex_coords, faces)
```

Arguments

`filepath`	string. Full path to the output surface file, should end with '.vtk', but that is not enforced.
`vertex_coords`	n x 3 matrix of doubles. Each row defined the x,y,z coords for a vertex.
`faces`	n x 3 matrix of integers. Each row defined the 3 vertex indices that make up the face. WARNING: Vertex indices should be given in R-style, i.e., the index of the first vertex is 1. However, they will be written in FreeSurfer style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.

Value

string the format that was written. One of "tris" or "quads". Currently only triangular meshes are supported, so always 'tris'.

See Also

Other mesh functions: `faces.quad.to.tris()`, `read.fs.surface.asc()`, `read.fs.surface.bvsrf()`, `read.fs.surface.geo()`, `read.fs.surface.gii()`, `read.fs.surface.ico()`, `read.fs.surface.obj()`, `read.fs.surface.off()`, `read.fs.surface.ply()`, `read.fs.surface.vtk()`, `read.fs.surface()`, `read.mesh.brainvoyager()`, `read_nisurfacefile()`, `read_nisurface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface()`

Examples

```
## Not run:
    # Read a surface from a file:
    surface_file = system.file("extdata", "lh.tinysurface",
     package = "freesurferformats", mustWork = TRUE);
    mesh = read.fs.surface(surface_file);

    # Now save it:
    write.fs.surface.vtk(tempfile(fileext=".vtk"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.weight` *Write file in FreeSurfer weight format*

Description

Write vertex-wise brain data for a set of vertices to file in *weight* format. This format is also known as *paint* format or simply as *w* format.

Usage

```
write.fs.weight(filepath, vertex_indices, values, format = "bin")
```

Arguments

`filepath,`	string. Full path to the output weight file.
`vertex_indices`	vector of integers, the vertex indices. Must be one-based (R-style). This function will substract 1, as they need to be stored zero-based in the file.
`values`	vector of floats. The brain morphometry data to write, one value per vertex.
`format`	character string, one of 'bin' or 'asc'. The weight format type, there is a binary version of the format and an ASCII version.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(` `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight.asc()`

`write.fs.weight.asc` *Write file in FreeSurfer weight ASCII format*

Description

Write vertex-wise brain data for a set of vertices to an ASCII file in *weight* format. This format is also known as *paint* format or simply as *w* format.

Usage

```
write.fs.weight.asc(filepath, vertex_indices, values)
```

Arguments

`filepath,`	string. Full path to the output ASCII weight file.
`vertex_indices`	vector of integers, the vertex indices. Must be one-based (R-style). This function will substract 1, as they need to be stored zero-based in the file.
`values`	vector of floats. The brain morphometry data to write, one value per vertex.

See Also

Other morphometry functions: `fs.get.morph.file.ext.for.format()`, `fs.get.morph.file.format.from.filename(` `read.fs.curv()`, `read.fs.mgh()`, `read.fs.morph.gii()`, `read.fs.morph()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.morph()`, `write.fs.weight()`

`write.nifti1` *Write header and data to a file in NIFTI v1 format.*

Description

Write header and data to a file in NIFTI v1 format.

Usage

```
write.nifti1(filepath, niidata, niiheader = NULL, ...)
```

Arguments

`filepath`	the file to write. The extension should be '.nii' or '.nii.gz'.
`niidata`	array of numeric or integer data, with up to 7 dimensions. Will be written to the file with the datatype and bitpix specified in the 'niiheader' argument. Set to 'NULL' and pass a 'niiheader' to write only the header, and remember to adapt 'magic' in the header.
`niiheader`	an optional NIFTI v1 header that is suitable for the passed 'niidata'. If not given, one will be generated with `ni1header.for.data`.
`...`	additional parameters passed to `ni1header.for.data`. Only used if 'niiheader' is 'NULL'.

See Also

Other nifti1 writers: `write.fs.morph.ni1()`

`write.nifti2` *Write header and data to a file in NIFTI v2 format.*

Description

Write header and data to a file in NIFTI v2 format.

Usage

```
write.nifti2(filepath, niidata, niiheader = NULL)
```

Arguments

`filepath`	the file to write. The extension should be '.nii' or '.nii.gz'.
`niidata`	array of numeric or integer data, with up to 7 dimensions. Will be written to the file with the datatype and bitpix specified in the 'niiheader' argument.
`niiheader`	an optional NIFTI v2 header that is suitable for the passed 'niidata'. If not given, one will be generated with `ni2header.for.data`.

See Also

Other nifti2 writers: `write.fs.morph.ni2()`

`write.smp.brainvoyager`

Write a brainvoyager SMP file.

Description

Write a brainvoyager SMP file, which contains one or more vertex-wise data maps (stats or morphometry data).

Usage

```
write.smp.brainvoyager(filepath, bvsmp, smp_version = 3L)
```

Arguments

`filepath`	character string, the output file
`bvsmp`	bvsmp instance, a named list as returned by `read.smp.brainvoyager`.
`smp_version`	integer, the SMP file format version to use when writing. Versions 2 to 5 are supported, but only versions 2 and 3 have been tested properly. Please report any problems you encounter. When converting between file versions (e.g., loading a v2 file and saving the result as a v5 file), some required fields may be missing, and for those without a default value according to the official spec, you will have to manually add the value you want in the bvsmp object before writing.

See Also

`write.fs.morph.smp`

`xml_node_gifti_coordtransform`

Create XML GIFTI CoordinateSystemTransformMatrix node.

Description

Create XML GIFTI CoordinateSystemTransformMatrix node.

Usage

```
xml_node_gifti_coordtransform(
  transform_matrix,
  data_space = "NIFTI_XFORM_UNKNOWN",
  transformed_space = "NIFTI_XFORM_UNKNOWN",
  as_cdata = TRUE
)
```

Arguments

`transform_matrix`	numerical 4x4 matrix, the transformation matrix from ’data_space’ to ’transformed_space’.
`data_space`	character string, the space used by the data before transformation.
`transformed_space`	character string, the space reached after application of the transformation matrix.
`as_cdata`	logical, whether to wrap text attributes (’data_space’ and ’transformed_space’) in cdata tags.

Value

XML node from xml2

Index