Package 'freesurferformats'

October 20, 2026

Type Package

Title Read and Write 'FreeSurfer' Neuroimaging File Formats

Version 1.1.2

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' formats. This includes, but is not limited to, the following file formats: 1) MGH/MGZ/NIFTI 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

Copyright Tim Schäfer (package code); the optional data files that can be downloaded at runtime are owned by their respective copyright holders, see 'inst/COPYRIGHTS'.

Encoding UTF-8

URL https://github.com/dfsp-spirit/freesurferformats, https://dfsp-spirit.github.io/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

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Tim Schäfer [aut, cre] (ORCID: <https://orcid.org/0000-0002-3683-8070>),
Van Essen Lab [cph] (Copyright holder of the Conte69 template and HCP group-average data files that can OPTIONALLY be downloaded via download_opt_data().),
Montreal Neurological Institute (MNI) [cph] (Copyright holder of the MNI152/avg152T1 template that can OPTIONALLY be downloaded via download_opt_data().)

Repository CRAN

Date/Publication 2026-10-20 13:50:09 UTC

Contents

analyzeheader.for.data . . . 7
analyzeheader.template . . . 8
annot.max.region.idx . . . 9
as.fs.tracts . . . 9
as.list.fs.tracts . . . 10
atlas.from.lut.and.csv . . . 11
bvsmp . . . 12
cdata . . . 13
cifti.axis.brain.models . . . 13
cifti.axis.from.template . . . 15
cifti.axis.labels . . . 16
cifti.axis.parcels . . . 17
cifti.axis.parcels.from.annot . . . 18
cifti.axis.scalars . . . 19
cifti.axis.series . . . 20
cifti.brain.model.surface . . . 21
cifti.brain.model.volume . . . 22
cifti.dim.labels . . . 23
cifti.file.type.for.axes . . . 24
cifti.grayordinates . . . 25
cifti.header.from.axes . . . 26
cifti.label.table . . . 27
cifti.parcel . . . 28
cifti.parcels . . . 29
cifti.series.info . . . 30
cifti.structure.data . . . 31
cifti.structures . . . 33
cifti.volume . . . 34
closest.vert.to.point . . . 35
colortable.from.annot . . . 36
delete_all_opt_data . . . 37
doapply.transform.mtx . . . 37
download_opt_data . . . 38
dti.track.bbox . . . 38
dti.track.count . . . 39

dti.track.iterator . . . 40
faces.quad.to.tris . . . 41
faces.tris.to.quad . . . 42
flip2D . . . 43
flip3D . . . 43
fs.get.morph.file.ext.for.format . . . 44
fs.get.morph.file.format.from.filename . . . 44
fs.patch . . . 45
fs.surface.to.tmesh3d . . . 46
fs.tracts . . . 46
fs.tracts.coords . . . 47
fs.tracts.lengths . . . 48
get_opt_data_filepath . . . 49
giftixml_add_labeltable_from_annot . . . 49
gifti_writer . . . 50
gifti_xml . . . 50
gifti_xml_add_global_metadata . . . 52
gifti_xml_write . . . 53
invert.fs.transform . . . 53
is.analyze.file . . . 54
is.bvsmp . . . 55
is.fs.annot . . . 56
is.fs.label . . . 56
is.fs.surface . . . 57
is.fs.tracts . . . 57
is.fs.transform . . . 58
is.fs.volume . . . 59
is.mghheader . . . 59
length.fs.tracts . . . 60
list_opt_data . . . 60
mghheader.centervoxelRAS.from.firstvoxelRAS . . . 61
mghheader.crs.orientation . . . 61
mghheader.is.conformed . . . 62
mghheader.is.ras.valid . . . 62
mghheader.primary.slice.direction . . . 63
mghheader.ras2vox . . . 64
mghheader.ras2vox.tkreg . . . 65
mghheader.scanner2tkreg . . . 66
mghheader.tkreg2scanner . . . 67
mghheader.update.from.vox2ras . . . 68
mghheader.vox2ras . . . 68
mghheader.vox2ras.tkreg . . . 69
mghheader.vox2vox . . . 70
mni152reg . . . 70
ni1header.for.data . . . 71
ni1header.template . . . 72
ni2header.for.data . . . 73
ni2header.template . . . 73

nifti.datadim.from.dimfield . . . 74
nifti.datadim.to.dimfield . . . 75
nifti.file.uses.fshack . . . 75
nifti.file.version . . . 76
nifti.header.check . . . 76
nifti2.extension . . . 77
nifti2.extension.text . . . 78
nifti2.get.extension . . . 78
nii1header.for.mgh . . . 79
print.dti.track.iterator . . . 80
print.fs.annot . . . 80
print.fs.cifti . . . 81
print.fs.cifti.data . . . 81
print.fs.connectome . . . 82
print.fs.label . . . 83
print.fs.patch . . . 83
print.fs.surface . . . 84
print.fs.tracts . . . 84
print.fs.transform . . . 85
print.fs.volume . . . 86
ras.to.surfaceras . . . 86
ras.to.talairachras . . . 87
read.analyze.data . . . 87
read.analyze.header . . . 88
read.cifti . . . 90
read.cifti.header . . . 91
read.cifti.rows . . . 93
read.dti.bval . . . 95
read.dti.bvec . . . 95
read.dti.grad . . . 96
read.dti.gradients . . . 97
read.dti.tck . . . 98
read.dti.tck.header . . . 100
read.dti.trk . . . 100
read.dti.trk.header . . . 102
read.dti.tsf . . . 103
read.dti.tsf.header . . . 104
read.fs.annot . . . 105
read.fs.annot.gii . . . 106
read.fs.colortable . . . 107
read.fs.connectome.cifti . . . 108
read.fs.curv . . . 110
read.fs.gca . . . 111
read.fs.label . . . 111
read.fs.label.gii . . . 112
read.fs.label.native . . . 113
read.fs.mgh . . . 114
read.fs.morph . . . 116

read.fs.morph.asc . . . 117
read.fs.morph.bvsmp . . . 118
read.fs.morph.cifti . . . 118
read.fs.morph.gii . . . 119
read.fs.morph.ni1 . . . 120
read.fs.morph.ni2 . . . 121
read.fs.morph.nii . . . 121
read.fs.morph.txt . . . 122
read.fs.parcellation.cifti . . . 122
read.fs.patch . . . 124
read.fs.patch.asc . . . 124
read.fs.series.cifti . . . 125
read.fs.surface . . . 126
read.fs.surface.asc . . . 127
read.fs.surface.bvsrf . . . 128
read.fs.surface.byu . . . 129
read.fs.surface.geo . . . 129
read.fs.surface.gii . . . 130
read.fs.surface.ico . . . 131
read.fs.surface.mz3 . . . 132
read.fs.surface.obj . . . 132
read.fs.surface.off . . . 133
read.fs.surface.ply . . . 134
read.fs.surface.stl . . . 135
read.fs.surface.stl.bin . . . 135
read.fs.surface.vtk . . . 136
read.fs.tracts.vtk . . . 137
read.fs.transform . . . 138
read.fs.transform.dat . . . 139
read.fs.transform.fslmat . . . 140
read.fs.transform.itk . . . 141
read.fs.transform.lta . . . 142
read.fs.transform.xfm . . . 143
read.fs.volume . . . 144
read.fs.volume.analyze . . . 146
read.fs.volume.nii . . . 148
read.fs.volume.nrrd . . . 151
read.fs.weight . . . 152
read.mesh.brainvoyager . . . 153
read.nifti1.data . . . 154
read.nifti1.header . . . 155
read.nifti2.data . . . 155
read.nifti2.header . . . 156
read.nrrd.header . . . 156
read.smp.brainvoyager . . . 157
readable.files . . . 158
read_nisurface . . . 159
read_nisurfacefile . . . 160

read_nisurfacefile.fsascii . . . 161
read_nisurfacefile.fsnative . . . 161
read_nisurfacefile.gifti . . . 162
rotate2D . . . 162
rotate3D . . . 163
sm0to1 . . . 163
sm1to0 . . . 164
summary.fs.transform . . . 165
surfaceras.to.ras . . . 166
surfaceras.to.talairach . . . 167
talairachras.to.ras . . . 167
transform2lps . . . 168
transform2ras . . . 169
transform2voxel . . . 170
transform2world . . . 171
vertex.euclid.dist . . . 172
vertexdists.to.point . . . 173
write.analyze . . . 173
write.atlas.to.lut.and.csv . . . 175
write.cifti . . . 176
write.dti.bval . . . 177
write.dti.bvec . . . 178
write.dti.grad . . . 179
write.dti.tck . . . 180
write.dti.trk . . . 181
write.dti.tsf . . . 182
write.fs.annot . . . 184
write.fs.annot.gii . . . 185
write.fs.colortable . . . 186
write.fs.connectome.cifti . . . 187
write.fs.curv . . . 189
write.fs.label . . . 190
write.fs.label.gii . . . 191
write.fs.mgh . . . 192
write.fs.morph . . . 193
write.fs.morph.asc . . . 194
write.fs.morph.cifti . . . 195
write.fs.morph.gii . . . 196
write.fs.morph.ni1 . . . 197
write.fs.morph.ni2 . . . 198
write.fs.morph.smp . . . 199
write.fs.morph.txt . . . 199
write.fs.parcellated.cifti . . . 200
write.fs.parcellation.cifti . . . 202
write.fs.patch . . . 203
write.fs.series.cifti . . . 204
write.fs.surface . . . 205
write.fs.surface.asc . . . 206

write.fs.surface.bvsrf . . . 207
write.fs.surface.byu . . . 208
write.fs.surface.gii . . . 209
write.fs.surface.mz3 . . . 211
write.fs.surface.obj . . . 212
write.fs.surface.off . . . 213
write.fs.surface.ply . . . 214
write.fs.surface.ply2 . . . 215
write.fs.surface.stl . . . 216
write.fs.surface.vtk . . . 218
write.fs.transform . . . 219
write.fs.transform.dat . . . 221
write.fs.transform.fslmat . . . 222
write.fs.transform.itk . . . 223
write.fs.transform.lta . . . 224
write.fs.transform.xfm . . . 225
write.fs.volume . . . 226
write.fs.weight . . . 227
write.fs.weight.asc . . . 228
write.nifti1 . . . 229
write.nifti2 . . . 230
write.smp.brainvoyager . . . 231
xml_node_gifti_coordtransform . . . 231
[.fs.tracts . . . 232
[[.fs.tracts . . . 233

Index **234**

`analyzeheader.for.data` *Create ANALYZE 7.5 header suitable for given data.*

Description

Create ANALYZE 7.5 header suitable for given data.

Usage

```
analyzeheader.for.data(analyzedata, ...)
```

Arguments

`analyzedata`	array of numeric (integer or double) data, can have up to 7 dimensions.
`...`	extra parameters passed on to `analyzeheader.template`, ignored in this function.

Value

an ANALYZE 7.5 header (see analyzeheader.template) in which the datatype, bitpix, dim, cal_min and cal_max fields have been set to values suitable for the given data. Feel free to change the other fields, e.g. the voxel sizes in pix_dim or the orient code.

Note

ANALYZE 7.5 has much fewer data types than NIFTI: integers are stored as 32 bit signed integers and floating point data as 32 bit floats, which is what this function selects (the same choice that ni1header.for.data makes for NIFTI v1). To store 16 bit integers, which is the classic ANALYZE data type, set datatype to 4L and bitpix to 16L in the returned header after calling this function.

Examples

```
analyzeheader <- analyzeheader.for.data(array(1:24, dim = c(4, 3, 2)))
analyzeheader$datatype
```

---

analyzeheader.template

Create a template ANALYZE 7.5 header. You will have to adapt it for your use case.

Description

This function returns a valid ANALYZE 7.5 header with all fields present, filled with the default values of the format. You will most likely have to adapt at least the dim, datatype, bitpix and pix_dim fields to your data, which is what analyzeheader.for.data does for you.

Usage

```
analyzeheader.template()
```

Value

named list, the ANALYZE 7.5 header. The fields endian, magic, header_format and spm_origin are not part of the ANALYZE 7.5 header: endian and magic describe how the file is stored, header_format tells which variant of the 348 byte header this is ('analyze' or 'nifti1_pair'), and spm_origin is the interpretation of the originator field that the SPM software uses.

Note

The fields spm_origin and magic are derived when a file is read, they are documented in read.analyze.header.

See Also

analyzeheader.for.data, read.analyze.header, write.analyze

Examples

```
analyzeheader <- analyzeheader.template()
analyzeheader$dim <- c(3L, 4L, 3L, 2L, 1L, 1L, 1L, 1L)
analyzeheader$pix_dim <- c(1., 1., 1., 1., 0., 0., 0., 0.)
```

---

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.

---

as.fs.tracts *Convert a collection of tracts to an fs.tracts instance.*

Description

Accepts an `fs.tracts` instance, a list of numeric matrices with 3 columns, or a single such matrix (which is then treated as one tract), and returns an `fs.tracts` instance. This is what the writers use to accept several input types.

Usage

```
as.fs.tracts(tracts, kind = "tck")
```

Arguments

tracts	the input, see the description.
kind	character string, 'tck' or 'trk', used for the returned instance.

Value

an `fs.tracts` instance.

Examples

```
tracts <- as.fs.tracts(list(matrix(c(0, 0, 0, 1, 1, 1), ncol = 3, byrow = TRUE)));
length(tracts);
```

`as.list.fs.tracts` *Convert an fs.tracts instance to a plain list of tracts.*

Description

Converts to the classic representation as a `list` with one entry per tract. This is convenient for interacting with code that expects a plain list, but note that it materializes one R object per tract and is therefore much more expensive in both memory and time. Prefer `[[` for accessing individual tracts.

Usage

```
## S3 method for class 'fs.tracts'
as.list(x, ...)
```

Arguments

`x`	an `fs.tracts` instance.
`...`	ignored.

Value

list with one entry per tract.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck");
tracts_list <- as.list(tck$tracks);

## End(Not run)
```

`atlas.from.lut.and.csv` *Construct a brain atlas from a colortable (LUT) file and a per-vertex label file.*

Description

Construct a brain atlas (surface annotation) from a FreeSurfer ASCII colortable lookup table (LUT) file and a text file that assigns a label (struct index) to each vertex of a brain surface. This is useful when an atlas is distributed as two text files: a colortable (LUT) plus a per-vertex label assignment, for example the cortical atlases bundled with the Python package 'yabplot' (files like 'aparc_LUT.txt' and 'aparc_conte69.csv').

Usage

```
atlas.from.lut.and.csv(
  lut_file,
  csv_file = NULL,
  label_indices = NULL,
  num_vertices = NULL,
  unknown_label_name = "unknown",
  unknown_index = 0L,
  include_unknown_in_colortable = TRUE
)
```

Arguments

`lut_file`	string, path to a colortable file in FreeSurfer ASCII LUT format (see `read.fs.colortable`). The file must contain the columns 'struct_index', 'struct_name', 'r', 'g', 'b', 'a', as in the 'FreeSurferColorLUT.txt' files. The struct index is the region identifier used in the label file.
`csv_file`	string, path to a text file that assigns a label to each vertex. The file must contain one integer per line: the struct index (label ID) of the region for that vertex, as defined in the first column of the LUT file. See parameter `unknown_index` for how unlabeled vertices are encoded. If you already have the indices in memory, use parameter `label_indices` instead.
`label_indices`	integer vector, an alternative to parameter 'csv_file'. The per-vertex struct indices, one per vertex. If given, 'csv_file' is ignored.
`num_vertices`	integer, optional. The expected number of vertices. If given, it must match the number of label indices read from 'csv_file' or 'label_indices'.
`unknown_label_name`	string, the name to use for unlabeled vertices (e.g., the medial wall). Defaults to 'unknown'.
`unknown_index`	integer, the struct index that is used in the label file to mark unlabeled vertices. Defaults to 0.

include_unknown_in_colortable
: logical, whether to add a region for unlabeled vertices to the colortable of the returned annotation, in case the LUT file does not already contain a region with the 'unknown_index'. Defaults to TRUE. This ensures that vertices without a valid label (e.g., the medial wall) get a proper region name and color in the returned annotation.

Value

an 'fs.annot' instance with class 'fs.annot', see `read.fs.annot` for the structure. Can be written to a FreeSurfer annotation file with `write.fs.annot`.

See Also

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

Examples

```
## Not run:
# Example for the yabplot cortical atlas files (LUT + per-vertex indices):
annot <- atlas.from.lut.and.csv("aparc_LUT.txt", "aparc_conte69.csv")
write.fs.annot("lh.aparc.annot", fs.annot = annot)

## End(Not run)
```

---

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.

`cifti.axis.brain.models`
Create a CIFTI-2 axis for brain models.

Description

A brain model axis describes which surface vertices and volume voxels a matrix dimension of a CIFTI-2 file contains, in the order in which they appear in the matrix. This is the mapping of a dense file (`.dscalar`, `.dtseries`, `.dlabel`, `.dconn`), and of the files that mix dense and parcellated data.

The index ranges of the models are computed from the index lists, so the models cover the dimension without gaps, which the format requires. A structure may appear in several models (e.g. as a surface and as a volume part in a grayordinates file), and the order of the models is the order of the matrix indices.

Usage

```
cifti.axis.brain.models(models, surfaces = NULL, volume = NULL)
```

Arguments

models	list of brain model entries, as created by cifti.brain.model.surface and cifti.brain.model.volume.
surfaces	named integer vector or NULL, the number of vertices of the complete surfaces the file refers to, named by brain structure (e.g. c(CORTEX_LEFT = 32492, CORTEX_RIGHT = 32492)). This is written as the optional Surface elements of the axis. Connectome Workbench does not write them, and the surface size is then taken from the brain model entries only, so this can be left at NULL unless the file has to state the surface sizes explicitly.
volume	a volume, see cifti.volume, or NULL. Required if any of the models is a volume model, because the voxel indices of a model can not be interpreted without the voxel grid and the transformation matrix.

Value

a named list with the entries 'type', 'brain_models', 'surfaces' and 'volumes', an axis to be passed to cifti.header.from.axes or write.cifti.

See Also

Other cifti functions: cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
axis <- cifti.axis.brain.models(list(
  cifti.brain.model.surface("lh", 10L),
  cifti.brain.model.surface("rh", 12L)))
axis$type
```

`cifti.axis.from.template` *Create a CIFTI-2 axis from a template file.*

Description

Read the axes of an existing CIFTI-2 file, so that a new file can be written with the same mapping. This is the recommended way to write a file for real data: the mapping of an HCP grayordinates file (which vertices are in the file, and which volume voxels) can not be invented, it has to come from the file the data was derived from.

Usage

```
cifti.axis.from.template(x, dim = NULL)
```

Arguments

`x`	character string (the path of a CIFTI-2 file), an `fs.cifti` object (see `read.cifti.header`) or an `fs.cifti.data` object (see `read.cifti`).
`dim`	integer or `NULL`, the matrix dimension to get the axis for. If `NULL` (the default), the axes of all matrix dimensions are returned as a list.

Value

the axis of the requested dimension (a named list, see `cifti.axis.brain.models`), or a list of axes (one per dimension; note that a file whose dimensions share a single XML mapping returns the same axis for both, as two entries).

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
axes <- cifti.axis.from.template(cifti_file)
length(axes)
axes[[2L]]$type
```

`cifti.axis.labels` *Create a CIFTI-2 axis for label maps.*

Description

See `cifti.axis.scalars`; a labels axis is the same thing with integer label keys instead of scalar values, and it stores the label table that defines the keys.

Usage

```
cifti.axis.labels(names = NULL, label_tables = NULL, metadata = NULL)
```

Arguments

`names`	character vector or `NULL`, the map names. The number of maps is the number of names, so this has to be given (use empty strings for unnamed maps).
`label_tables`	list of label tables or `NULL`, one per map. Each label table is a data.frame with the columns 'key' (integer), 'red', 'green', 'blue', 'alpha' (numeric in the range 0 to 1) and 'label' (character string), i.e. the format that `cifti.label.table` returns. `NULL` entries are allowed for maps without a label table.
`metadata`	list of metadata lists or `NULL`, one per map. Each metadata list is a named list of character strings, as returned for a map by `read.cifti.header`. This is used for the palette information that Connectome Workbench stores per map.

Value

a named list with the entries 'type' and 'named_maps', an axis to be passed to `cifti.header.from.axes` or `write.cifti`.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
label_table <- data.frame(key = 0:1, red = c(1, 0), green = c(1, 0.5), blue = c(1, 1),
                          alpha = c(0, 1), label = c("???", "PARCEL_A"))
axis <- cifti.axis.labels("parcellation", label_tables = list(label_table))
axis$named_maps[[1]]$labels$label
```

`cifti.axis.parcels` *Create a CIFTI-2 axis for parcels.*

Description

A parcel axis describes the parcels of a parcellated matrix dimension (the mapping of `.pscalar`, `.ptseries`, `.pconn` and the second dimension of `.pdconn`). As for brain models, the index of a parcel is its position in the list.

Usage

```
cifti.axis.parcels(parcels, surfaces = NULL, volume = NULL)
```

Arguments

`parcels`	list of parcels, as created by `cifti.parcel`.
`surfaces`	named integer vector or `NULL`, the number of vertices of the complete surfaces the parcels refer to, see `cifti.axis.brain.models`. Unlike Connectome Workbench, which writes them, this is optional here as well.
`volume`	a volume, see `cifti.volume`, for parcels that consist of volume voxels, or `NULL`.

Value

a named list with the entries 'type', 'parcels', 'surfaces' and 'volumes', an axis to be passed to `cifti.header.from.axes` or `write.cifti`.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
axis <- cifti.axis.parcels(list(
  cifti.parcel("PARCEL_A", list(CORTEX_LEFT = 0:2)),
  cifti.parcel("PARCEL_B", list(CORTEX_LEFT = 3:5))))
length(axis$parcels)
```

`cifti.axis.parcels.from.annot` *Create a CIFTI-2 parcels axis from brain surface annotations.*

Description

A parcellated CIFTI-2 file stores no label table, it describes each parcel by its name and by the vertices (per brain structure) and volume voxels it contains. This function builds such a parcels axis from brain surface annotations (see `read.fs.annot`), which assign a region label to every vertex of a hemisphere, so that data that was computed per region of an atlas (a FreeSurfer parcellation, a Schaefer atlas, ...) can be written to a parcellated file.

The vertices of a parcel are collected by the *name* of the region, not by its label key: the hemispheres of an atlas usually use the same names for the same region, but the names often carry a hemisphere marker (e.g. `L_superiorfrontal` in one hemisphere and `R_superiorfrontal` in the other, or `7Networks_LH_Vis_1` and `7Networks_RH_Vis_1`), and a parcel of a parcellated CIFTI-2 file is a region that spans the structures it occurs in. The markers are removed before the names are compared, see `cifti.region.name.without.hemisphere` for the exact rules. Vertices whose label is not in the label table of the annotation (the medial wall, which is usually the 'unknown' region with key 0) form a parcel like any other, named `default_label_name`: a parcellated file has no 'no value' case, and dropping the medial wall silently would change the data.

Usage

```
cifti.axis.parcels.from.annot(
  annots,
  structure = NULL,
  parcel_names = NULL,
  default_label_name = "unknown"
)
```

Arguments

`annots`	an `fs.annot` instance (see `read.fs.annot`), or a named list of them, with the brain structures as the names (e.g. `list(lh = lh_annot, rh = rh_annot)`).
`structure`	character string or `NULL`, the brain structure of the annotation, needed if `annots` is a single `fs.annot` instance instead of a named list.
`parcel_names`	character vector or `NULL`, the names of the parcels to write, in that order. This selects and orders the parcels of the axis; without it all regions of the parcellation are used, in the order in which the label table of the annotation lists them (the atlas order, not the order in which the vertices happen to be stored in the mesh). Renaming is not supported, since the name is what the hemispheres of a region are matched by.
`default_label_name`	character string, the name to use for vertices whose label is not in the label table of the annotation.

Value

an axis of type 'CIFTI_INDEX_TYPE_PARCELS', see `cifti.axis.parcels`, to be passed to `write.cifti` or to one of the writers that accept it (e.g. `write.fs.parcellated.cifti`).

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
lh_annot_file <- system.file("extdata", "lh.aparc.annot.gz", package = "freesurferformats")
lh_annot <- read.fs.annot(lh_annot_file)
axis <- cifti.axis.parcels.from.annot(lh_annot, structure = "lh")
length(axis$parcels)
axis$parcels[[1L]]$name
```

---

cifti.axis.scalars *Create a CIFTI-2 axis for scalar or label maps.*

Description

A scalars axis holds a set of maps (e.g. the myelin map and the cortical thickness of a subject in one `.dscalar` file), a labels axis holds a set of label maps (one per parcellation in a `.dlabel` file). Both are described by their map names; a labels axis can additionally have a label table per map, which maps the integer label keys in the data to names and colors.

Usage

```
cifti.axis.scalars(names = NULL, metadata = NULL)
```

Arguments

`names`	character vector or `NULL`, the map names. The number of maps is the number of names, so this has to be given (use empty strings for unnamed maps).
`metadata`	list of metadata lists or `NULL`, one per map. Each metadata list is a named list of character strings, as returned for a map by `read.cifti.header`. This is used for the palette information that Connectome Workbench stores per map.

Value

a named list with the entries 'type' and 'named_maps', an axis to be passed to `cifti.header.from.axes` or `write.cifti`.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
axis <- cifti.axis.scalars(c("thickness", "area"))
length(axis$named_maps)
```

---

cifti.axis.series *Create a CIFTI-2 axis for a series.*

Description

A series axis describes a dimension that holds an ordered sequence of samples, usually the time points of a `.dtseries` or `.ptseries` file. The value of series index `i` (0-based) is `(start + i * step) * 10^exponent`, in the unit that `unit` names.

Usage

```
cifti.axis.series(
  number_of_series_points,
  start = 0,
  step = 1,
  exponent = 0L,
  unit = "SECOND"
)
```

Arguments

`number_of_series_points`	integer, the number of samples, which must match the size of that matrix dimension of the data.
`start`	numeric, the value of the first sample.
`step`	numeric, the difference between consecutive samples.

exponent	integer, the power of ten the values are in (e.g. -3 for milliseconds).
unit	character string, one of 'SECOND', 'HERTZ', 'METER' or 'RADIAN'.

Value

a named list with the entries 'type' and 'series', an axis to be passed to cifti.header.from.axes or write.cifti.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
axis <- cifti.axis.series(10L, start = 0, step = 0.72, unit = "SECOND")
axis$series$number_of_series_points
```

---

cifti.brain.model.surface

Create a CIFTI-2 brain model entry for a surface.

Description

A surface brain model entry describes a set of surface vertices that one matrix dimension of a CIFTI-2 file contains, e.g. one hemisphere. The vertex indices are 0-based and refer to the vertices of the surface mesh that the file is defined on. If they are NULL, the model covers all vertices of the surface, which is only allowed if their number matches the size of the index range (the reader validates this, and the writer computes the range from the index list).

Usage

```
cifti.brain.model.surface(
  structure,
  surface_number_of_vertices,
  vertices = NULL
)
```

Arguments

structure	character string, the brain structure, see cifti.structure.canonical for the accepted spellings.
surface_number_of_vertices	integer, the number of vertices of the complete surface (not the number of vertices in this model: a grayordinates file of the HCP leaves out the medial wall, so its models usually cover fewer vertices than the surface has).
vertices	integer vector or NULL, the 0-based vertex indices in this model.

Value

a named list, one brain model entry, to be passed to cifti.axis.brain.models.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
model <- cifti.brain.model.surface("lh", 10L, vertices = 0:4)
model$index_count
```

---

cifti.brain.model.volume

Create a CIFTI-2 brain model entry for volume voxels.

Description

A volume brain model entry describes the voxels of one subcortical structure, as 0-based IJK indices into the voxel grid of the volume that is part of the same axis (see cifti.volume). If the indices are NULL, the model covers all voxels of that grid.

Usage

```
cifti.brain.model.volume(structure, voxel_indices_ijk = NULL)
```

Arguments

structure	character string, the brain structure, see cifti.structure.canonical for the accepted spellings.
voxel_indices_ijk	n x 3 integer matrix or NULL, the 0-based voxel indices, one voxel per row. A vector of length 3*n is accepted as well and interpreted as the rows.

Value

a named list, one brain model entry, to be passed to cifti.axis.brain.models.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
voxels <- matrix(c(0L, 0L, 2L, 1L, 0L, 2L), ncol = 3L, byrow = TRUE)
model <- cifti.brain.model.volume("CEREBELLUM", voxels)
model$index_count
```

---

cifti.dim.labels *Get axis labels for a matrix dimension of a CIFTI-2 file.*

Description

Get a label for each index of one matrix dimension of a CIFTI-2 file, e.g. the map names of a .dscalar, the parcel names of a .pconn or identifiers for the grayordinates of a dense file. This is what read.cifti uses to name the dimensions of the data array.

Usage

```
cifti.dim.labels(cii, dim = 0L)
```

Arguments

cii	an fs.cifti instance, see read.cifti.header.
dim	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See read.cifti.header for details on the dimensions.

Value

character vector of length `cii$matrix$dim_sizes[dim + 1]`, the labels, or `NULL` if no labels are known for this kind of dimension.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.ptseries.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cifti.dim.labels(cii, dim = 1L)
```

---

cifti.file.type.for.axes

Get the standard CIFTI-2 file type for a set of axes.

Description

Look up the file type that the format defines for the combination of matrix index types of the axes, see `cifti.file.types`.

Usage

```
cifti.file.type.for.axes(axes)
```

Arguments

`axes`	list of axes, see `cifti.header.from.axes`.

Value

a one row data.frame, see `cifti.file.types`, or a row with `intent_code = 3000` ('ConnUnknown') and empty file name extension for a combination that the standard does not define.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
axis_brain <- cifti.axis.brain.models(list(cifti.brain.model.surface("lh", 10L)))
type <- cifti.file.type.for.axes(list(cifti.axis.scalars("m"), axis_brain))
type$extension
```

---

cifti.grayordinates *Get the brainordinate table of a CIFTI-2 file.*

Description

Get the mapping from the matrix indices of one dimension to the individual surface vertices and volume voxels a CIFTI-2 file contains. This is the per-index version of the table returned by cifti.structures, which reports index ranges instead of single indices. It is useful to find out which vertex or voxel a data row belongs to, or to find the data index of a vertex or voxel.

Note that the same vertex or voxel can only appear once per dimension (the matrix has one entry per brainordinate), but a structure can be split over several brain model entries, and the vertices of a grayordinates file are a subset of the vertices of the surface it refers to.

Usage

```
cifti.grayordinates(cii, dim = 0L)
```

Arguments

cii	an fs.cifti instance, see read.cifti.header.
dim	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See read.cifti.header for details on the dimensions.

Value

a data.frame with one row per index of the matrix dimension and the columns 'index' (integer, 0-based matrix index), 'structure' (character string, the brain structure as spelled in the file), 'structure_short' (without the CIFTI_STRUCTURE_ prefix), 'model_type' ('SURFACE' or 'VOXELS'), 'vertex_index' (integer, 0-based index of the vertex in the surface for surface models, NA for volume models) and 'i', 'j', 'k' (integer, 0-based voxel indices for volume models, NA for surface models).

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
grayordinates <- cifti.grayordinates(cii, dim = 1L)
head(grayordinates)
```

---

cifti.header.from.axes

Create the CIFTI-2 XML metadata for a set of axes.

Description

Build the CIFTI-2 XML document (the part of a CIFTI-2 file that describes what the data matrix contains) from a set of axes. This is the inverse of the XML parsing that read.cifti.header performs: pass the result of this function to cifti.parse.xml or to read.cifti.header (after storing it in a file) and you get back the axes you put in.

Usage

```
cifti.header.from.axes(axes, metadata = NULL)
```

Arguments

axes	list of axes, one per matrix dimension, as created by the cifti.axis.*() functions or by cifti.axis.from.template. If the list is named, the names have to be the matrix dimensions ('0' and '1'), otherwise the axes are assigned to the dimensions in the order in which they are given.
metadata	named character vector or named list or NULL, the metadata of the matrix (e.g. the provenance information), as name/value pairs.

Value

character string, the CIFTI-2 XML document.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
axis_series <- cifti.axis.series(4L, start = 0, step = 2.5)
axis_brain <- cifti.axis.brain.models(list(
  cifti.brain.model.surface("lh", 10L),
  cifti.brain.model.surface("rh", 12L)))
xml <- cifti.header.from.axes(list(axis_series, axis_brain))
cat(substr(xml, 1, 200))
```

---

cifti.label.table *Get a label table from a CIFTI-2 label file.*

Description

Get the label table of one label map (a .dlabel file can contain several maps). The label table maps integer label keys to names and RGBA colors, and is the CIFTI analogue of a FreeSurfer color lookup table.

Usage

```
cifti.label.table(cii, dim = 0L, map = 1L)
```

Arguments

cii	an fs.cifti instance, see read.cifti.header.
dim	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See read.cifti.header for details on the dimensions.
map	integer, the number of the label map to get the label table for (1-based). A CIFTI label file can contain several maps, use cii$matrix$indices_maps or the map_name field of the result to find out which one you need.

Value

a data.frame with one row per label and the columns 'key' (integer, the label key used in the data), 'red', 'green', 'blue', 'alpha' (numeric, in range 0 to 1) and 'label' (character string, the label name), plus 'x', 'y', 'z' (numeric, the optional label coordinates; NA if not present in the file). Returns NULL if the specified matrix dimension does not contain label maps, or if the specified map has no label table.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dlabel.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cifti.label.table(cii, dim = 0L, map = 1L)
```

---

cifti.parcel *Create a CIFTI-2 parcel.*

Description

Create one parcel, i.e. a named set of surface vertices (per structure) and volume voxels. The name is the only label a parcellated file has: the position in the list of parcels is the parcel index, and the reader returns the names.

Usage

```
cifti.parcel(name, vertices = NULL, voxel_indices_ijk = NULL)
```

Arguments

name	character string, the parcel name.
vertices	named list or NULL, the 0-based vertex indices per brain structure, e.g. list(CORTEX_LEFT = c(0, 1, 2), CORTEX_RIGHT = c(4, 5)). Structures without vertices are left out.
voxel_indices_ijk	n x 3 integer matrix or NULL, the 0-based voxel indices of the parcel, one voxel per row.

Value

a named list, one parcel, to be passed to cifti.axis.parcels.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
parcel <- cifti.parcel("PARCEL_A", list(CORTEX_LEFT = 0:2, CORTEX_RIGHT = 4:5))
parcel$name
```

---

cifti.parcels *Get the parcel table of a CIFTI-2 file.*

Description

Get the parcels for one matrix dimension of a parcellated CIFTI-2 file. A parcel is a named set of surface vertices and/or volume voxels; its index is its position in the list (CIFTI-2 parcels have no index attribute). The vertex lists themselves are in the field 'vertices' of the fs.cifti object, see read.cifti.header.

Usage

```
cifti.parcels(cii, dim = 0L)
```

Arguments

cii	an fs.cifti instance, see read.cifti.header.
dim	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See read.cifti.header for details on the dimensions.

Value

a data.frame with one row per parcel and the columns 'index' (integer, 0-based parcel index), 'name' (character string, the parcel name), 'num_vertices' (integer, the total number of surface vertices in this parcel) and 'num_voxels' (integer, the number of volume voxels in this parcel). The result is ordered by parcel index.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.ptseries.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cifti.parcels(cii, dim = 1L)
```

---

cifti.series.info *Get the series information of a CIFTI-2 file.*

Description

Get the description of the series dimension of a CIFTI-2 file, e.g. the time points of a `.dtseries` or `.ptseries` file. The time (or other unit) of the series index `i` is `(start + i * step) * 10^exponent`, with `i` starting at 0.

Usage

```
cifti.series.info(cii, dim = 0L)
```

Arguments

`cii`	an `fs.cifti` instance, see `read.cifti.header`.
`dim`	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See `read.cifti.header` for details on the dimensions.

Value

a named list with the entries 'number_of_series_points' (integer), 'start' (numeric), 'step' (numeric), 'exponent' (integer) and 'unit' (character string, one of 'SECOND', 'HERTZ', 'METER', 'RADIAN'), or `NULL` if the specified matrix dimension is not a series. To check whether a dimension is a series, inspect the field 'type' of the corresponding entry of `cii$matrix$indices_maps`.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dtseries.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cifti.series.info(cii, dim = 0L)
```

---

cifti.structure.data *Extract the data of one brain structure from a CIFTI-2 file.*

Description

Get the data values of a single brain structure (e.g. one hemisphere) from a CIFTI-2 file, reconstructed for the full surface. The result has one row per vertex of the surface (in the order of the surface, which is the order the vertices have in the surface mesh files) and one column per index of the other matrix dimension of the file. Vertices of the surface that have no value in the file are reported as NA: grayordinates files have a reduced mesh (the medial wall vertices are missing), and returning the values for the complete surface is what makes such files usable together with the standard surface meshes of a subject.

Volume structures (a brain model of type 'CIFTI_MODEL_TYPE_VOXELS', which subcortical structures use) cannot be expanded like this, because the voxels a structure consists of are not a rectangular block of a volume and are not ordered in any meaningful way. For those, the data values are returned together with the voxel indices and the affine transformation that maps them to coordinates, see the Value section.

Usage

```
cifti.structure.data(x, structure = NULL, dim = NULL)
```

Arguments

x	an fs.cifti.data object, see read.cifti. An fs.cifti metadata object or the path of a CIFTI-2 file are accepted as well, in which case the data are read from the file.

`structure` character string or NULL, the brain structure to extract the data for. The usual spellings are accepted ('lh', 'CORTEX_LEFT', 'CIFTI_STRUCTURE_CORTEX_LEFT'), see `cifti.structure.canonical`. If it is NULL, the data of all structures are returned.

`dim` integer or NULL, the CIFTI matrix dimension that holds the brain structure (the brainordinate dimension). For most files there is only one such dimension and this can be left at NULL. Files in which both dimensions are brain models (.dconn and .pdconn) require an explicit value, since they contain the same structures in both dimensions.

Value

If 'structure' is a single structure: a named list with the entries 'structure' (character string, the canonical name of the structure, e.g. 'CIFTI_STRUCTURE_CORTEX_LEFT'), 'structure_short' (the name without the prefix, e.g. 'CORTEX_LEFT'), 'model_type' (character string, one of 'SURFACE', 'VOXELS' or 'SURFACE_AND_VOXELS'), 'surface' and 'volume'. If 'structure' is NULL: a named list of such lists, one per structure, named by the short structure name.

For a selection that contains a surface model, the 'surface' entry holds a numeric (or integer) array with one row per vertex of the full surface and the remaining dimensions of the data, with NA for vertices that the file does not contain. Its dimensions beyond the first are named like in `read.cifti`. If the selection contains no surface model, 'surface' is NULL.

For a selection that contains a volume model, the 'volume' entry holds a named list with the entries 'values' (an array like 'surface', but with one row per voxel of the structure instead of per surface vertex), 'voxel_indices_ijk' (an n x 3 integer matrix of 0-based voxel indices, in the same order as the rows of 'values'), 'dimensions' (integer vector of length 3, the dimensions of the volume the voxel indices refer to) and 'transformation_matrix' and 'meter_exponent' (the 4x4 row-major matrix from the file that maps the voxel indices to coordinates in units of `10^meter_exponent`, and the exponent). If the selection contains no volume model, 'volume' is NULL.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
cii <- read.cifti(cifti_file)
lh_data <- cifti.structure.data(cii, "lh")
dim(lh_data$surface)
# vertices beyond the end of the mesh in the file are NA:
sum(is.na(lh_data$surface[, 1]))
```

```
# Volume structures are returned with their voxel indices:
vol_file <- system.file("extdata", "cifti", "tiny_volume.dscalar.nii",
                        package = "freesurferformats")
vol_cii <- read.cifti(vol_file)
cerebellum <- cifti.structure.data(vol_cii, "CEREBELLUM")
cerebellum$volume$voxel_indices_ijk
```

`cifti.structures` *Get the brain model table of a CIFTI-2 file.*

Description

Get the brain model entries for one matrix dimension of a CIFTI-2 file, i.e. the mapping from matrix indices to surface vertices or volume voxels. Note that a brain structure can appear in several brain model entries: a grayordinates file can contain a surface part and a volume part for the same structure, and a structure can be split into several index ranges. Always use the index ranges from this table to map data values, never the structure names.

Usage

```
cifti.structures(cii, dim = 0L)
```

Arguments

`cii`	an `fs.cifti` instance, see `read.cifti.header`.
`dim`	integer, the matrix dimension to get the brain models for. CIFTI-2 files have two dimensions, so this is usually 0 (Workbench calls it the ROW dimension) or 1 (the COLUMN dimension). See `read.cifti.header` for details on the dimensions.

Value

a data.frame with one row per brain model entry and the columns 'structure' (character string, the brain structure in the spelling used in the file), 'structure_short' (character string, the normalized name without the `CIFTI_STRUCTURE_` prefix, e.g. 'CORTEX_LEFT'), 'model_type' ('SURFACE' or 'VOXELS'), 'index_offset' (integer, 0-based index of the first matrix entry covered by this brain model), 'index_count' (integer, the number of matrix entries covered) and 'surface_number_of_vertices' (integer, the number of vertices of the complete surface, for surface models; `NA` for volume models). Use `index_offset` and `index_count` to extract the data values for a brain model from the data matrix.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`,

cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cifti.structures(cii, dim = 1L)
```

---

cifti.volume *Create a CIFTI-2 axis for a volume.*

Description

A volume axis describes the voxel grid that the volume brain models of a CIFTI-2 file refer to: its dimensions and the 4x4 transformation matrix that maps the (0-based) IJK voxel indices to coordinates. The matrix is stored row by row in the file, and the coordinates it produces are in units of `10^meter_exponent` (which is -3, i.e. millimeters, for the files that Connectome Workbench writes).

Usage

```
cifti.volume(dimensions, transformation_matrix, meter_exponent = -3L)
```

Arguments

`dimensions`	integer vector of length 3, the dimensions of the voxel grid.
`transformation_matrix`	4x4 numeric matrix, the transformation from (0-based) voxel indices to coordinates. There is no default: a CIFTI-2 file does not store the voxel size or the position of a volume anywhere else, so a wrong or invented matrix silently puts the voxels of a structure in the wrong place. Pass the matrix that the template file has (see `cifti.axis.from.template`) if you do not know it.
`meter_exponent`	integer, the exponent of the unit of the coordinates.

Value

a named list with the entries 'dimensions', 'meter_exponent' and 'transformation_matrix', to be passed to `cifti.axis.brain.models` or `cifti.axis.parcels`.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
volume <- cifti.volume(c(4L, 4L, 4L), diag(c(2, 2, 2, 1)))
volume$dimensions
```

---

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: `atlas.from.lut.and.csv()`, `read.fs.annot()`, `read.fs.colortable()`, `write.atlas.to.lut.and.csv()`, `write.fs.annot()`, `write.fs.annot.gii()`, `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. The downloaded data files are owned by their respective copyright holders and are subject to their respective licenses; see the file 'inst/COPYRIGHTS' for attribution and license details.

Usage

```
download_opt_data(
  dl_from = c("rcmd.org", "github_fsf_release"),
  scheme = c("https", "http")
)
```

Arguments

dl_from	string, the source to download from. Either "rcmd.org" (the default) or "github_fsf_release" to use the GitHub release at https://github.com/dfsp-spirit/freesurferformats/releases/tag/nitestdata-v1.
scheme	string, the URL scheme to use. Either "https" (the default) or "http". Both the rcmd.org server and GitHub support HTTPS, so this should rarely need to be changed.

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.

dti.track.bbox *Compute the bounding box of all tract coordinates in a file.*

Description

Streams through the tract file and returns the bounding box of all coordinates, without keeping the coordinates in memory. This is useful to determine the axis limits for plotting a large tractogram.

Usage

```
dti.track.bbox(filepath, chunk_values = 4e+06)
```

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.
chunk_values	integer, the number of payload values to read per chunk. Advanced tuning parameter, see read.dti.tck.

Value

numeric vector of length 6, c(xmin, xmax, ymin, ymax, zmin, zmax), or NULL if the file contains no coordinates. The coordinates are in the coordinate system used by the file, see the note in read.dti.trk.

Examples

```
## Not run:
bbox <- dti.track.bbox("brain.tck");

## End(Not run)
```

---

dti.track.count *Count the tracts in a DTI tract file.*

Description

Counts the tracts in a TRK, TCK or TSF file without reading their coordinates, so this works on arbitrarily large tractograms with a small and constant amount of memory. This is the equivalent of tckinfo <file> -count in MRtrix.

Usage

```
dti.track.count(filepath, chunk_values = 4e+06)
```

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.
chunk_values	integer, the number of payload values to read per chunk. Advanced tuning parameter, see read.dti.tck.

Value

integer, the number of tracts in the file. Note that this can differ from the count entry in the file header, which the MRtrix documentation explicitly describes as unreliable, and which does not count empty tracts.

Examples

```
## Not run:
dti.track.count("brain.tck");

## End(Not run)
```

`dti.track.iterator` *Create an iterator over the tracts of a DTI tract file.*

Description

Reads a TRK, TCK or TSF file tract by tract, so that a tractogram of any size can be processed with a constant amount of memory. This is the low-level interface behind `dti.track.count` and friends, use it when neither reading a subset (`max_tracks`, `skip_tracks`, `bbox` in `read.dti.tck`) nor one of the aggregate functions fits your use case.

The returned object is an environment with the following entries:

- `next.track()`: returns the next tract (a n x 3 matrix for TCK, a numeric vector of per-point values for TSF, and a list with the entries `coords`, `num_points`, `scalars` and `properties` for TRK, just like `tracks[[i]]` does for the readers), or `NULL` when there are no more tracts.
- `close()`: closes the underlying file connection. It is safe to call this more than once, and it is also called automatically when the iterator is garbage collected.
- `tracks.read`: the number of tracts returned so far.
- `filepath`, `format`: the file and its detected format.

Usage

```
dti.track.iterator(
  filepath,
  skip_tracks = 0L,
  bbox = NULL,
  chunk_values = 4e+06
)
```

Arguments

`filepath`	character string, path to the `TCK` or `TSF` file to read. Gzip-compressed files (i.e., `.tck.gz`) are supported and the compression is detected from the file content, not the file name.
`skip_tracks`	integer, the number of tracks to skip before reading any. Skipped tracks are never held in memory.

bbox — numeric vector of length 6 or NULL. If given, only tracks that have at least one point inside the box are read, the box is given as `c(xmin, xmax, ymin, ymax, zmin, zmax)` in the coordinate system of the stored data. This has no effect for TSF files, which store no coordinates. The number of matching tracks cannot be known in advance, so for very large files this should be combined with a `max_tracks` value that is known to be safe.

chunk_values — integer, the number of payload values that are read per chunk. This is an advanced tuning parameter that does not change the result, only the peak memory usage (at most one chunk is buffered at a time) and the granularity of the file reads. The default of 4e6 values corresponds to about 32 MB of doubles. Lower it on a machine with very little free memory.

Value

the iterator environment, see the description.

Examples

```
## Not run:
itr <- dti.track.iterator("brain.tck");
total_points <- 0;
while (!is.null(track <- itr$next.track())) {
  total_points <- total_points + nrow(track);
}
itr$close();

## End(Not run)
```

---

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()`, `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_nisurface()`, `read_nisurfacefile()`, `write.fs.surface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`

---

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

`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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

---

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

---

fs.tracts *Create an fs.tracts instance from a compact tract representation.*

Description

Creates the compact container used by the DTI tract readers and writers. Reading a track file with `read.dti.tck`, `read.dti.trk` or `read.dti.tsf` returns instances of this class, and this function is the way to build one from your own data, e.g., to write a tractogram that was assembled or edited in R with `write.dti.tck` or `write.dti.trk`.

Usage

```
fs.tracts(coords, lengths, scalars = NULL, properties = NULL, kind = "tck")
```

Arguments

coords	numeric matrix with 3 columns, the concatenated coordinates of all tracts.
lengths	integer vector, the number of points of each tract. Must sum up to nrow(coords).
scalars	numeric matrix or NULL. Per-point data, with one row per point (i.e., nrow(scalars) == nrow(coords)).
properties	numeric matrix or NULL. Per-tract data, with one row per tract.
kind	character string, either 'tck' or 'trk'. Determines what [[returns for a single tract.

Value

an fs.tracts instance.

Examples

```
# Two tracts, the first with two points and the second with one.
coords <- matrix(c(0, 0, 0, 1, 1, 1, 5, 5, 5), ncol = 3, byrow = TRUE);
tracts <- fs.tracts(coords, lengths = c(2L, 1L));
length(tracts);
tracts[[1]];
```

---

fs.tracts.coords *Get the concatenated coordinates of fs.tracts instances.*

Description

Returns all tract coordinates as a single N x 3 matrix, with the tracts concatenated along the rows. This is the compact representation used internally, and the fastest way to access all coordinates, e.g., for plotting or for computing a bounding box.

Usage

```
fs.tracts.coords(tracts)
```

Arguments

tracts	an fs.tracts instance, as returned in the tracks entry of read.dti.tck or read.dti.trk.

Value

numeric matrix with 3 columns and one row per point of all tracts.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck");
coords <- fs.tracts.coords(tck$tracks);
bbox <- apply(coords, 2, range);

## End(Not run)
```

`fs.tracts.lengths` *Get the number of points of each tract.*

Description

Returns one integer per tract, the number of points it consists of. The coordinates of tract `i` are the rows `(cumsum(c(1, lengths))[i]):(cumsum(lengths)[i])` of `fs.tracts.coords()`.

Usage

```
fs.tracts.lengths(tracts)
```

Arguments

`tracts`	an `fs.tracts` instance, as returned in the `tracks` entry of `read.dti.tck` or `read.dti.trk`.

Value

integer vector with one entry per tract.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck");
lengths <- fs.tracts.lengths(tck$tracks);
mean(lengths);

## End(Not run)
```

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', 'GZip-Base64Binary'.
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)
```

invert.fs.transform *Invert a transformation.*

Description

Compute the transformation that undoes the given one, i.e. the one that maps the target coordinates back to the source coordinates. The source and target of the result are swapped, along with the coordinate spaces, and the matrix is inverted. A transformation is never stored inverted, use this function when you need the reverse mapping.

Usage

```
invert.fs.transform(tf)
```

Arguments

tf	an fs.transform instance.

Value

an fs.transform instance that maps from the target of tf back to its source.

Note

The format-specific fields of tf (e.g. the parsed header and the volume info of an LTA file, or the intensity of a register.dat file) describe the file that tf was read from and are therefore copied to the result unchanged. They are metadata, not part of the mapping.

See Also

Other header coordinate space: is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
tf <- read.fs.transform(system.file("extdata", "talairach.xfm",
  package = "freesurferformats", mustWork = TRUE
))
tf_back <- invert.fs.transform(tf)
tf_back$space_in # the spaces are swapped
max(abs(tf_back$matrix %*% tf$matrix - diag(4))) # the identity
```

---

is.analyze.file *Check whether a file is an ANALYZE 7.5 file.*

Description

Check whether a file is an ANALYZE 7.5 file.

Usage

```
is.analyze.file(filepath)
```

Arguments

filepath	character string, the path to a .hdr file (or to a file whose name ends with the base name of one, see analyze.pair.files).

Value

logical, whether the 348 byte header is an ANALYZE 7.5 header. This is the case when the header contains neither of the two NIFTI v1 magic strings, i.e. when the magic is empty. Files that use the ni1 magic are NIFTI v1 pair files, see read.nifti1.header, and files with the n+1 magic are single file NIFTI v1 files.

Note

Note that this function returns TRUE for any two-file header that does not carry a NIFTI magic, which is what "ANALYZE 7.5" means in practice: these files were written by ANALYZE itself, by SPM, by AFNI or by FreeSurfer. Only the fields of the ANALYZE specification are defined for them, but SPM stores extra information in fields that ANALYZE leaves unused, see read.fs.volume.analyze.

A file path that does not describe an existing pair file (for example a single file NIFTI image, or a file that does not exist) returns FALSE rather than an error.

Examples

```
hdrfile <- system.file("extdata", "analyze", "tiny_u8.hdr",
  package = "freesurferformats", mustWork = TRUE
)
is.analyze.file(hdrfile)
```

---

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.tracts *Check whether an object is an fs.tracts instance.*

Description

Check whether an object is an fs.tracts instance.

Usage

```
is.fs.tracts(x)
```

Arguments

x	any R object.

Value

logical, TRUE if x is an `fs.tracts` instance.

Examples

```
is.fs.tracts("not tracts");
```

`is.fs.transform` *Check whether an object is an fs.transform instance.*

Description

Check whether an object is an fs.transform instance.

Usage

```
is.fs.transform(x)
```

Arguments

`x`	any object.

Value

logical, whether `x` is an `fs.transform` instance.

See Also

Other header coordinate space: `invert.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

Examples

```
tf <- read.fs.transform(system.file("extdata", "talairach.xfm",
  package = "freesurferformats", mustWork = TRUE
))
is.fs.transform(tf)
is.fs.transform("no transform")
```

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.

length.fs.tracts *Number of tracts in an fs.tracts instance.*

Description

Number of tracts in an fs.tracts instance.

Usage

```
## S3 method for class 'fs.tracts'
length(x)
```

Arguments

x	an fs.tracts instance.

Value

integer, the number of tracts.

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: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

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: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(),

sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

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, pair = 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.
pair	logical, whether the header should describe a NIFTI v1 pair (a .hdr header file plus a separate .img data file) instead of a single file. See ni1header.template.

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(pair = FALSE)
```

Arguments

`pair`	logical, whether the template describes a NIFTI v1 *pair* file (a `.hdr` header file plus a separate `.img` data file, the variant that is also used by the ANALYZE 7.5 format and that FSL writes) instead of a single file (`.nii`, the header and the data in one file). The two variants differ in the `magic` field and in the `vox_offset` field, which is 352 for a single file (the data starts behind the header) and 0 for a pair (the data starts at the first byte of the `.img` file). Both variants are written by `write.nifti1`.

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.

The 'magic' field of a NIFTI v2 file must be the string 'n+2'. Versions of this package before 1.1.0 wrote the NIFTI v1 magic 'n+1' here, which violates the standard and makes other software (nibabel, Connectome Workbench) refuse the file; write.nifti2 writes the full 8 byte magic of the standard.

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

---

nifti2.extension *Create a NIFTI v2 header extension.*

Description

NIFTI v2 files can store arbitrary data in header extensions, in the area between the header and the voxel data. This function creates such an extension. Pass the result to write.nifti2, which writes it to the file, and later read it back with read.nifti2.header plus nifti2.get.extension.

Usage

```
nifti2.extension(ecode, content = NULL)
```

Arguments

ecode	integer, the extension code. This identifies the type of the payload, but the code is not interpreted in any way. The CIFTI2 XML metadata uses code 32L.
content	raw vector, character string, or NULL. The payload of the extension, i.e., the data to store. A character string is converted to UTF-8 bytes (several strings are joined with newline characters). Use NULL or omit for an empty payload.

Value

named list with the entries 'ecode' and 'content', representing one NIFTI v2 header extension.

See Also

nifti2.get.extension, nifti2.extension.text

Other nifti2 extensions: nifti2.extension.content(), nifti2.extension.size(), nifti2.extension.text(), nifti2.get.extension()

nifti2.extension.text *Get the payload of a NIFTI v2 header extension as text.*

Description

Convert the payload of a header extension to a character string. The trailing NUL bytes, which are only used to pad the extension to a multiple of 16 bytes, are removed. Note that embedded NUL bytes cannot be represented in an R string and are removed as well, so this function is only meaningful for text payloads (like the CIFTI2 XML metadata).

Usage

```
nifti2.extension.text(extension)
```

Arguments

extension	a NIFTI v2 header extension, a named list with entries 'ecode' and 'content', see nifti2.extension.

Value

character string, the payload of the extension. The string is marked as UTF-8 encoded, which is the encoding required by the NIFTI standard.

See Also

Other nifti2 extensions: nifti2.extension(), nifti2.extension.content(), nifti2.extension.size(), nifti2.get.extension()

nifti2.get.extension *Get a NIFTI v2 header extension by code.*

Description

Retrieve one header extension from a NIFTI v2 header, by its extension code. See nifti2.extension for details on header extensions.

Usage

```
nifti2.get.extension(niiheader, ecode)
```

Arguments

niiheader	named list, a NIFTI v2 header as returned by read.nifti2.header or created by ni2header.template.
ecode	integer, the extension code to search for. The CIFTI2 XML metadata uses code 32L.

Value

the extension (a named list with entries 'ecode' and 'content', see nifti2.extension), or NULL if the header contains no extension with this code.

See Also

nifti2.extension.text

Other nifti2 extensions: nifti2.extension(), nifti2.extension.content(), nifti2.extension.size(), nifti2.extension.text()

---

nii1header.for.mgh *Create a NIFTI v1 header from the header information contained in an fs.volume instance.*

Description

Create a NIFTI v1 header from the header information contained in an fs.volume instance.

Usage

```
nii1header.for.mgh(mgh, endian = "little", pair = FALSE)
```

Arguments

mgh	an fs.volume instance, or a string. If a string, it is interpreted as a filepath to a volume file (NIFTI, MGH or MGZ) that will be loaded with read.fs.volume.
endian	character string, the endianness to use. Either 'little' or 'big'. Defaults to 'little'.
pair	logical, whether the header should describe a NIFTI v1 *pair* (a .hdr header file with the voxel data in a separate .img file, the variant of the format that FSL and 3D Slicer work with) instead of a single file (.nii). See ni1header.template.

Value

a NIFTI v1 header structure (see ni1header.template). Note that the header may or may not contain full RAS information, depending on whether the source fs.volume contained such information or not. If the MGH header does not have valid RAS information, the qform and sform codes will be set to 0 (unknown).

Note

This is intended to be used with write.nifti1, which allows users to convert MGH/MGZ data to NIFTI files.

See Also

Other nifti1 writers: write.fs.morph.ni1(), write.nifti1()

print.dti.track.iterator *Print a tract file iterator.*

Description

Print a tract file iterator.

Usage

```
## S3 method for class 'dti.track.iterator'
print(x, ...)
```

Arguments

x	a dti.track.iterator instance.
...	ignored.

Value

the iterator, invisibly.

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.cifti *Print an fs.cifti instance.*

Description

Print an fs.cifti instance.

Usage

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

Arguments

x	an fs.cifti instance, see read.cifti.header.
...	ignored.

Value

the instance, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

print.fs.cifti.data *Print an fs.cifti.data instance.*

Description

Print an fs.cifti.data instance.

Usage

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

Arguments

x	an fs.cifti.data instance, see read.cifti.
...	ignored.

Value

the instance, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

---

print.fs.connectome *Print an fs.connectome instance.*

Description

Print an fs.connectome instance.

Usage

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

Arguments

x	an fs.connectome instance, see read.fs.connectome.cifti.
...	ignored.

Value

the instance, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes() cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

---

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.tracts *Print an fs.tracts instance.*

Description

Print an fs.tracts instance.

Usage

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

Arguments

x	an fs.tracts instance.
...	ignored.

Value

the instance, invisibly.

print.fs.transform *Print an fs.transform instance.*

Description

Print an fs.transform instance.

Usage

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

Arguments

x	an fs.transform instance.
...	ignored, only present for consistency with the generic.

Value

the transform x, invisibly.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
tf <- read.fs.transform(system.file("extdata", "talairach.lta",
  package = "freesurferformats", mustWork = TRUE
))
print(tf)
```

`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/transfo
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.analyze.data *Read ANALYZE 7.5 data from file.*

Description

Read ANALYZE 7.5 data from file.

Usage

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

Arguments

filepath	character string, the path to the .hdr file. The base name without the extension is accepted as well, see analyze.pair.files.
header	optional ANALYZE 7.5 header as returned by read.analyze.header. It will be read from the file automatically if left at NULL.
drop_empty_dims	logical, whether to drop empty dimensions in the returned data array, see drop.

Value

the data in the .img file of the pair, as an array (or a vector for 1D data). Note that the fields of the header are not applied in any way: the data are returned exactly as they are stored in the file, in the storage order of the format (the first dimension varies fastest, which is also the order used by R and by the MGH/MGZ format), and the funused1 field that SPM uses as a scale factor is not applied (see read.fs.volume.analyze if you want that).

See Also

read.analyze.header

Examples

```
hdrfile <- system.file("extdata", "analyze", "tiny_u8.hdr",
  package = "freesurferformats", mustWork = TRUE
)
data <- read.analyze.data(hdrfile)
dim(data)
```

---

read.analyze.header *Read ANALYZE 7.5 header from file.*

Description

Read the header of an ANALYZE 7.5 file, i.e. of a file in the two-file format that stores the voxel data in a separate .img file.

Usage

```
read.analyze.header(filepath)
```

Arguments

filepath	character string, the path to the .hdr file. The base name without the extension is accepted as well, see analyze.pair.files.

Value

named list with the ANALYZE 7.5 header fields, in the field naming of the format specification. See analyzeheader.template for a description of all of them. In addition to the fields of the specification, the following entries are present:

`endian` character string, the endianness of the file, `'little'` or `'big'`. It is detected from the `sizeof_hdr` field.

`magic` character string, the 4 bytes at offset 344 (which are the `smin` field of the ANALYZE specification), interpreted as a string. It is empty for ANALYZE files, see is.analyze.file.

`header_format` character string, always `'analyze'` here. The other variant of the two-file format is reported as `'nifti1_pair'`, see read.fs.volume.analyze.

`spm_origin` integer vector of length 3, the interpretation that SPM gives to the `originator` field: the voxel coordinates of the image origin, as stored in the first 6 bytes of the field. It is `NULL` if all of them are zero, which means that the field does not store an origin.

`originator_bytes` raw vector of length 10, the unmodified content of the `originator` field. The `originator` entry is the same content with the zero bytes removed and interpreted as a string, which loses information for the files in which SPM stores the image origin there.

`filepath_header`, `filepath_image` character strings, the paths of the two files of the pair.

Note

ANALYZE 7.5 does not define the meaning of the world coordinate system of an image: the header stores the voxel sizes but neither the direction of the voxel axes nor the position of the image in space. This function therefore reports the fields as they are stored and does not derive a transformation matrix. See read.fs.volume.analyze for the ways to get a matrix anyway, and for what is known and unknown about them.

See Also

read.analyze.data, analyzeheader.template

Examples

```
hdrfile <- system.file("extdata", "analyze", "tiny_u8.hdr",
  package = "freesurferformats", mustWork = TRUE
)
analyzeheader <- read.analyze.header(hdrfile)
analyzeheader$datatype
analyzeheader$pix_dim
```

`read.cifti` *Read a CIFTI-2 file.*

Description

Read a CIFTI-2 file, i.e. its XML metadata (see `read.cifti.header`) and its data values. The data are returned as an array in the order in which they are stored in the file: the first array dimension is CIFTI matrix dimension 0 (which Connectome Workbench calls the ROW dimension), the second one is matrix dimension 1 (the COLUMN dimension). This is the same order that other CIFTI implementations use for their data arrays.

For a dense file like a `.dscalar`, dimension 0 holds the scalars (maps) and dimension 1 the brainordinates (surface vertices and volume voxels), so the data matrix has one row per map and one column per grayordinate. For a `.dtseries`, dimension 0 holds the series, so the matrix has one row per series point. The functions `cifti.series.info` and `cifti.map.for.dim` tell you what a dimension is; never conclude it from the file name. Use `cifti.structure.data` to extract the data of a brain structure, which is the more convenient interface for most purposes and the one that the `read.fs.*.cifti` functions are built on.

Reading a file twice (once for the header, once for the data) is not necessary to extract a single structure, but it is what this function does; use it if you need more than one structure or the raw matrix.

Usage

```
read.cifti(filepath, rows = NULL, columns = NULL)
```

Arguments

`filepath`	character string, the path to a CIFTI-2 file. Note that a CIFTI-2 file is a NIFTI-2 file, but its data are not a 3D volume; gzipped CIFTI files do not exist (the format forbids compression so that random access remains possible).
`rows`	integer vector or `NULL`, the indices of matrix dimension 0 to read. Indices are 1-based, like everywhere else in R, and they refer to the rows of the returned array, i.e. to the first CIFTI matrix dimension. Use `NULL` (the default) to read all of them. Note that this selection is applied after reading the whole matrix: use `read.cifti.rows` if the file is too large for that.
`columns`	integer vector or `NULL`, the indices of matrix dimension 1 to read, i.e. the columns of the returned array. This is the matrix dimension that holds the brainordinates of a dense file (the grayordinates of a `.dscalar`, `.dtseries` or `.dlabel`), and it is also one of the two dimensions that hold them in a connectome file like a `.dconn`. Selecting columns avoids reading the rest of the file, which is the only practical way to read a small part of a huge file like a `.dconn` (9 to 38 GB): for a 2-dimensional matrix, one contiguous block of the file holds all values of one column. Note that `rows` does not have this property, selecting rows still reads all values of the file.

Value

a named list with the entries 'header' (an `fs.cifti` object, see `read.cifti.header`) and 'data' (the numeric or integer array). The array dimensions are named with the axis labels of the corresponding matrix dimensions, see `cifti.dim.labels`. The object has class 'fs.cifti.data'.

Note

The generic readers `read.fs.morph` and `read.fs.volume` dispatch on the file name and would therefore match a CIFTI-2 file (which is a NIFTI-2 file), so they detect this case and stop with a pointer to this function and the other CIFTI readers instead of returning the values of the matrix in an order that means nothing.

See Also

Other cifti functions: `cifti.axis.brain.models()`, `cifti.axis.from.template()`, `cifti.axis.labels()`, `cifti.axis.parcels()`, `cifti.axis.parcels.from.annot()`, `cifti.axis.scalars()`, `cifti.axis.series()`, `cifti.brain.model.surface()`, `cifti.brain.model.volume()`, `cifti.dim.labels()`, `cifti.file.type.for.axes()`, `cifti.grayordinates()`, `cifti.header.from.axes()`, `cifti.label.table()`, `cifti.parcel()`, `cifti.parcels()`, `cifti.series.info()`, `cifti.structure.data()`, `cifti.structures()`, `cifti.volume()`, `print.fs.cifti()`, `print.fs.cifti.data()`, `print.fs.connectome()`, `read.cifti.header()`, `read.cifti.rows()`, `read.fs.connectome.cifti()`, `write.cifti()`, `write.fs.connectome.cifti()`, `write.fs.morph.cifti()`, `write.fs.parcellated.cifti()`, `write.fs.parcellation.cifti()`, `write.fs.series.cifti()`

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
cii <- read.cifti(cifti_file)
dim(cii$data)
# The first 3 grayordinates of the first map:
cii$data[1, 1:3]
```

`read.cifti.header` *Read CIFTI-2 header (XML metadata).*

Description

Read the metadata of a CIFTI version 2 file, i.e. the XML document that describes everything except the actual data values: the matrix dimensions and their meaning, the brain models (surface vertices and volume voxels), the parcels, the series information, and the label tables. CIFTI-2 files are NIFTI-2 files that store this XML in a NIFTI v2 header extension with the extension code 32. Use `read.cifti` to read the data values as well (not implemented yet), or the accessor functions `cifti.structures`, `cifti.parcels`, `cifti.series.info` and `cifti.label.table` to inspect the result.

Usage

```
read.cifti.header(filepath)
```

Arguments

filepath	character string, the path to a CIFTI-2 file (usually one of .dscalar.nii, .dtseries.nii, .dlabel.nii, .dconn.nii, .pscalar.nii, .ptseries.nii, .pconn.nii, .dpconn.nii or .pdconn.nii). Note that this is not a NIFTI file, despite the .nii part. Gzipped CIFTI files are not supported, because the CIFTI-2 format forbids compression.

Value

an fs.cifti object, a named list with the entries: 'filepath' (the file path), 'niiheader' (the NIFTI-2 header as returned by read.nifti2.header), 'version' (the CIFTI version, always '2' for files that can be read), and 'matrix', a named list with the entries 'metadata' (the matrix-level metadata, a named list of character strings, in file order; the names may repeat) and 'indices_maps' (a list of the MatrixIndicesMap elements, see below), plus 'dim_sizes' (the integer sizes of the matrix dimensions; these are stored in entries 5, 6, ... of the dim field of the NIFTI-2 header, and R index vectors are 1-based, so the 6th entry of dim holds matrix dimension 0).

Each element of indices_maps is a named list with the entries: 'dims' (integer vector, the 0-based matrix dimensions this mapping applies to; it has several entries for files like a .dconn, where one mapping describes both dimensions), 'applies_to' (the same as a character string, as found in the file), 'type' (character string, one of 'CIFTI_INDEX_TYPE_BRAIN_MODELS', 'CIFTI_INDEX_TYPE_PARCELS', 'CIFTI_INDEX_TYPE_SERIES', 'CIFTI_INDEX_TYPE_SCALARS' or 'CIFTI_INDEX_TYPE_LABELS'), 'size' (integer vector, the size of the dimension(s) from 'dims'), 'series' (a list with entries 'number_of_series_points', 'start', 'step', 'exponent' and 'unit', for series mappings; NULL otherwise), 'surfaces' (a list of lists with entries 'brain_structure' and 'surface_number_of_vertices'), 'volumes' (a list of lists with entries 'dimensions' (integer vector of length 3), 'meter_exponent' and 'transformation_matrix' (4x4 numeric matrix, row-major as in the file, mapping 0-based voxel indices to coordinates in units of 10^meter_exponent)), 'brain_models' (a list of lists with entries 'index_offset' (0-based), 'index_count', 'model_type', 'brain_structure', 'surface_number_of_vertices' (surfaces only, NA otherwise), 'vertex_indices' (0-based integer vector, or NULL if all vertices of the surface are used) and 'voxel_indices_ijk' (an n x 3 integer matrix of 0-based voxel indices, or NULL)), 'parcels' (a list of lists with entries 'index' (0-based position in the list), 'name', 'vertices' (named list of 0-based vertex index vectors, named by the canonical brain structure name, e.g. 'CORTEX_LEFT') and 'voxel_indices_ijk'), and 'named_maps' (a list of lists with entries 'name', 'metadata' and 'labels'; 'labels' is a data.frame with the columns 'key', 'red', 'green', 'blue', 'alpha', 'label', 'x', 'y' and 'z', see cifti.label.table).

Note

This function performs structural validation of the XML and stops with a descriptive error if the file is not a CIFTI-2 file, if the XML is inconsistent (e.g., index ranges that do not cover the data, or a number of parcels that does not match the matrix dimension), or if a file in the outdated CIFTI-1 format is passed (which has to be converted first, see the error message).

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
cii <- read.cifti.header(cifti_file)
cii
cii$matrix$indices_maps[[1]]$type
cifti.structures(cii, dim = 1L)
```

`read.cifti.rows` *Read selected matrix rows of a CIFTI-2 file without loading the whole matrix.*

Description

Read a few indices of CIFTI matrix dimension 0 (the rows of the data matrix) from a large file, without ever holding the values of the rows that are not requested in memory. `read.cifti` supports the same selection, but it reads the whole matrix first and then drops the unrequested rows, which is impossible for a file that does not fit into memory: the matrix of an HCP subject (91,282 grayordinates in both dimensions) is 33 GB. This function streams through the file in chunks and keeps only the requested rows, so its memory usage is the size of the result plus one chunk (32 MB by default), regardless of the size of the file.

Note which direction of a CIFTI-2 file is the cheap one: the values of a *cell of a row* are stored with a stride (`dim[5]` values lie between the values of one row), while the values of a column are contiguous. Reading rows therefore has to touch every value of the file once (it is a single sequential pass, not a seek per value, but the I/O is the size of the file), while selecting `columns` reads only what was asked for. Use `read.cifti` with the `columns` parameter if the rows you need are the ones that a column selection can give you, e.g. because the matrix is symmetric (which a `.dconn` is), and use this function for the cases that need rows: for a `.dtseries`, one row is one time point of all grayordinates, so reading the first few time points of a 1.7 GB file with this function needs a few KB instead of the whole file.

Usage

```
read.cifti.rows(filepath, rows, columns = NULL, chunk_values = 4000000L)
```

Arguments

filepath	character string, the path to a CIFTI-2 file. Note that a CIFTI-2 file is a NIFTI-2 file, but its data are not a 3D volume; gzipped CIFTI files do not exist (the format forbids compression so that random access remains possible).
rows	integer vector, the indices of matrix dimension 0 to read. At least one index has to be given; the indices are 1-based, see read.cifti.
columns	integer vector or NULL, the indices of matrix dimension 1 to read, i.e. the columns of the returned array. This is the matrix dimension that holds the brainordinates of a dense file (the grayordinates of a .dscalar, .dtseries or .dlabel), and it is also one of the two dimensions that hold them in a connectome file like a .dconn. Selecting columns avoids reading the rest of the file, which is the only practical way to read a small part of a huge file like a .dconn (9 to 38 GB): for a 2-dimensional matrix, one contiguous block of the file holds all values of one column. Note that rows does not have this property, selecting rows still reads all values of the file.
chunk_values	integer, the number of data values that are read from the file per chunk. This does not change the result, only the peak memory usage and the I/O granularity, so it is rarely needed: the default of 4 millions values corresponds to about 16 MB. The chunk size is rounded up to a whole number of matrix columns, and the result is the same for every chunk size.

Value

a named list with the entries 'header' and 'data', see read.cifti. The 'data' entry holds the requested rows, with the requested columns if columns was given.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
cifti_file <- system.file("extdata", "cifti", "tiny.dtseries.nii", package = "freesurferformats")
# The first two time points of all grayordinates:
first_frames <- read.cifti.rows(cifti_file, rows = 1:2)
dim(first_frames$data)

# A few time points and a few grayordinates:
subset <- read.cifti.rows(cifti_file, rows = 2, columns = 1:3)
subset$data
```

`read.dti.bval` *Read b-values from a FSL-style bvals file.*

Description

Reads the b-values of a diffusion MRI dataset, i.e. the diffusion weighting of each volume in units of s/mm^2. This implements the *bvals* part of the FSL format: a plain text file with one value per volume, usually in a single row. The HCP variant, which stores one value per line, is read as well.

Usage

```
read.dti.bval(filepath, layout = "auto")
```

Arguments

`filepath`	character string, path to the b-values file, typically ending in `.bval` or `.bvals`, but without an extension in the HCP layout.
`layout`	character string, one of 'auto', 'components' or 'volumes'. `components` means that the file stores the values in a single line (the FSL layout), `volumes` means one value per line (the HCP layout). With 'auto' (the default), the layout is derived from the file content.

Value

numeric vector of length *n*, the b-value of each of the *n* volumes.

See Also

Other dti functions: `read.dti.bvec()`, `read.dti.grad()`, `read.dti.gradients()`, `write.dti.bval()`, `write.dti.bvec()`, `write.dti.grad()`

`read.dti.bvec` *Read b-vectors from a FSL-style bvecs file.*

Description

Reads the diffusion gradient directions of a diffusion MRI dataset, i.e. the *b-vectors* part of the FSL bvecs/bvals format. The file contains unit vectors (or zero vectors for the *b=0* volumes), stored as three rows of *n* values, one row per component, one column per volume.

Usage

```
read.dti.bvec(filepath, layout = "auto")
```

Arguments

`filepath`	character string, path to the b-vectors file, typically ending in `.bvec` or `.bvecs`, but without an extension in the HCP layout.
`layout`	character string, one of 'auto', 'components' or 'volumes'. `components` means that there are 3 lines, one per vector component (the FSL layout, and what the file name `bvecs` implies), `volumes` means one volume per line, i.e. *n* lines with 3 values each (the HCP layout). With 'auto' (the default), the layout is derived from the matrix dimensions and a warning is raised for a square table, which is ambiguous.

Value

a numeric matrix with *n* rows and 3 columns, one row per volume, the columns being the x, y and z component of the gradient direction. Note that these vectors are given with respect to the *image* axes, not to the scanner axes, so they are only meaningful together with the image they belong to: converting them to scanner space requires the transform of that image. Use `read.dti.gradients` to read b-vectors and b-values together, with consistency checks.

Note

The b-vectors are returned exactly as they are stored, i.e. they are *not* renormalized, and missing values (see `read.dti.gradients`) are not replaced here.

See Also

Other dti functions: `read.dti.bval()`, `read.dti.grad()`, `read.dti.gradients()`, `write.dti.bval()`, `write.dti.bvec()`, `write.dti.grad()`

`read.dti.grad` *Read a gradient table in MRtrix3 format.*

Description

Reads a diffusion gradient table in MRtrix format: a plain text file with one line per DWI volume, each line holding the four space-separated values `[ x y z b ]`, with the direction in *scanner* space and the b-value in s/mm^2. This is the format expected by the `-grad` option of the MRtrix3 commands (and produced by `-export_grad_mrtrix`), and it is also the format in which MRtrix3 stores the table in the `dw_scheme` entry of a `.mif` header. A first line consisting of a single integer is accepted as a volume count header.

Usage

```
read.dti.grad(filepath, layout = "auto")
```

Arguments

filepath	character string, path to the file. There is no fixed file name convention, grad.b and encoding.b are common. Gzip-compressed files are supported.
layout	character string, one of 'auto', 'components' or 'volumes'. volumes means one volume per line (4 values per line, the MRtrix layout), components the transposed variant (4 lines). With 'auto' (the default), the layout is derived from the matrix dimensions.

Value

a numeric matrix with *n* rows and 4 columns named 'x', 'y', 'z' and 'b', one row per volume. The directions are in scanner space.

See Also

Other dti functions: read.dti.bval(), read.dti.bvec(), read.dti.gradients(), write.dti.bval(), write.dti.bvec(), write.dti.grad()

---

read.dti.gradients *Read and validate a diffusion MRI gradient table.*

Description

The main entry point for reading diffusion gradients. It reads b-vectors and b-values from either a pair of FSL-style files or from a single MRtrix-style gradient table file, checks that they are consistent, and reports suspicious entries. Use this rather than calling read.dti.bvec and read.dti.bval separately if you intend to use the gradients for anything, because a mismatched or malformed gradient table is silently wrong otherwise.

Usage

```
read.dti.gradients(bvec, bval = NULL, n_volumes = NULL, layout = "auto")
```

Arguments

bvec	character string (path to a b-vectors file), or a numeric matrix with one row per volume (or one column per volume) and 3 columns.
bval	character string (path to a b-values file), or a numeric vector with one value per volume. Can be NULL if bvec identifies both files.
n_volumes	scalar numeric or integer, the number of volumes in the DWI image that the gradients belong to, used to check that the table matches the image. Typically dim(volume$data)[4]. Set to NULL (the default) to skip this check.
layout	character string, passed on to the readers, one of 'auto', 'components' or 'volumes'.

Details

Both parameters accept either a file path or already read data, so the function can also be used to validate gradients that were obtained from somewhere else. If `bval` is `NULL` and `bvec` is a single file path, the file is interpreted as follows. A file whose name contains `bvec` is read as a b-vectors file (since the name is the user's clearest statement about the content, and a b-vectors file with 4 volumes would otherwise look exactly like a 3-volume gradient table). Otherwise, a file that has 4 values per line, or 4 lines, is read as an MRtrix gradient table. Otherwise the file is read as b-vectors and a matching b-values file is looked up next to it, which covers the common dataset layouts such as BIDS (`sub-01_dwi.bvec` with `sub-01_dwi.bval`).

Value

a named list with the entries `bvec` (numeric matrix with *n* rows and 3 columns, one row per volume, components in *image* space) and `bval` (numeric vector of length *n*).

Note

The following problematic cases are reported, but the data is always returned as read, except for the missing values, which are replaced by zeros: (1) `NA`/`NaN` entries, which are interpreted as `b=0` volumes, as MRtrix3 does, provided that they do not leave a volume with a b-value but no direction or vice versa, which is an error; (2) gradient vectors whose norm deviates from 1, which is reported because MRtrix3 rescales the b-value by the squared norm in this situation; (3) volumes with a b-value above the b=0 threshold but a zero direction; (4) volumes with a small positive b-value but a non-zero direction.

See Also

Other dti functions: `read.dti.bval()`, `read.dti.bvec()`, `read.dti.grad()`, `write.dti.bval()`, `write.dti.bvec()`, `write.dti.grad()`

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

Description

Reads streamlines from a TCK file, optionally gzip-compressed.

Usage

```
read.dti.tck(
  filepath,
  max_tracks = Inf,
  skip_tracks = 0L,
  bbox = NULL,
  chunk_values = 4e+06
)
```

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.
max_tracks	numeric, the maximum number of tracks to read. Use Inf (the default) to read all tracks. This allows reading a subset of a very large tractogram without holding all of it in memory.
skip_tracks	integer, the number of tracks to skip before reading any. Skipped tracks are never held in memory.
bbox	numeric vector of length 6 or NULL. If given, only tracks that have at least one point inside the box are read, the box is given as c(xmin, xmax, ymin, ymax, zmin, zmax) in the coordinate system of the stored data. This has no effect for TSF files, which store no coordinates. The number of matching tracks cannot be known in advance, so for very large files this should be combined with a max_tracks value that is known to be safe.
chunk_values	integer, the number of payload values that are read per chunk. This is an advanced tuning parameter that does not change the result, only the peak memory usage (at most one chunk is buffered at a time) and the granularity of the file reads. The default of 4e6 values corresponds to about 32 MB of doubles. Lower it on a machine with very little free memory.

Value

named list with entries 'header' and 'tracks'. The tracks are stored in an is.fs.tracts instance: all coordinates are kept in a single matrix and tracks[[i]] returns the n x 3 coordinate matrix of the i-th track. Use fs.tracts.lengths(tck$tracks) to get the number of points of each track, and as.list(tck$tracks) to convert to a plain list of matrices.

Examples

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

# Read only the first 1000 streamlines of a huge tractogram:
first_tracks <- read.dti.tck(tckf, max_tracks = 1000)

# Read the streamlines passing through a region:
region <- read.dti.tck(tckf, bbox = c(-10, 10, -20, 20, 0, 30))

## End(Not run)
```

read.dti.tck.header *Read the header of a file in MRtrix 'TCK' or 'TSF' format.*

Description

Reads only the ASCII header of the file, without touching the binary payload. This is cheap even for huge tractograms and can be used to inspect a file before deciding whether to read its tracks.

Usage

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

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.

Value

named list, the parsed header. The entries of the 'derived' entry describe the payload (data offset, datatype, endianness, and whether the file is gzip-compressed).

Examples

```
## Not run:
hdr <- read.dti.tck.header("brain.tck");
hdr$count;              # number of streamlines stated in the header
hdr$derived$gzipped;    # TRUE for a .tck.gz file

## 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,
  shift_origin = TRUE,
  max_tracks = Inf,
  skip_tracks = 0L,
  bbox = NULL,
  coords = NULL
)
```

Arguments

filepath	character string, path to file in trk format. Gzip-compressed files are supported as well (the compression is detected from the file content, so a `.trk.gz` file is read like any other TRK file), which is convenient since tractograms are large and are regularly stored compressed. Note that track files cannot be compressed on the fly for other software: the TrackVis tools and MRtrix do not read compressed track files.
shift_origin	logical, whether to apply the half-voxel origin shift when computing the corrected vox2ras matrix. The TRK format stores a matrix that maps to the voxel corner, not the voxel center (as is the NIfTI convention). Set to `TRUE` (the default) to compute the corrected `vox2ras` that maps to voxel centers, as used by TrackVis. Set to `FALSE` if the file was written by DSI Studio, which does not apply this shift. See the notes for details.
max_tracks	numeric, the maximum number of tracks to read. Use `Inf` (the default) to read all tracks. This allows reading a subset of a very large tractogram without holding all of it in memory.
skip_tracks	integer, the number of tracks to skip before reading any. Skipped tracks are never held in memory. Note that the count refers to the tracks that a reader returns, and that empty tracks (tracks without any point, which the TRK format allows) are returned as well, e.g., a file with one regular track, then an empty one, yields the empty track for `skip_tracks = 1`. The only exception is a `bbox` filter, which drops empty tracks, since no point of them can be inside the box.
bbox	numeric vector of length 6 or NULL. If given, only tracks that have at least one point inside the box are read, the box is given as `c(xmin, xmax, ymin, ymax, zmin, zmax)`.
coords	character string or NULL, the coordinate system of the returned track coordinates. One of 'native' (the coordinates exactly as stored in the file, which for TRK files is usually 'voxelmm' space, in which the coordinates refer to voxel corners) or 'ras' (RAS+ mm space in which a coordinate of (0,0,0) is the center of the first voxel, which is what 'nibabel' and DIPY return by default). If `NULL` (the default), the coordinates are returned as stored, but a warning is raised when the file stores a transformation that is not the identity, since the coordinates are then not in RAS space and plotting them directly would produce e.g., a mirrored brain. Pass `'native'` to silence that warning. The `bbox` filter is always interpreted in the same coordinate system as the returned coordinates.

Value

named list, the parsed file data. The naming of the variables follows the spec at `http://trackvis.org/docs/?subsect=fil` The returned header will contain the field `vox2ras` (the raw matrix stored in the TRK file, mapping from mm space to RAS) and, if `shift_origin` is `TRUE`, the additional field `vox2ras_corrected` (the computed matrix mapping from voxel indices to voxel center RAS coordinates). It also contains the entry `coords_space` ('native' or 'ras'), which records the coordinate system the returned tracks are in, so that the result can be passed to `write.dti.trk()` without losing that information. The tracks are stored in an `is.fs.tracts` instance: `tracks[[i]]` returns a list with the entries `coords` (n x 3 matrix), `num_points` (integer), `scalars` (matrix or NULL) and `properties` (numeric vector or NULL) for the i-th track. Use `fs.tracts.lengths(trk$tracks)` to get the number of points of each track, and `as.list(trk$tracks)` to convert to a plain list.

Note

The 4x4 matrix stored in TRK files (labeled `vox_to_ras` in the spec) is actually a transformation from **mm** space to RAS, not from voxel space to RAS. The TRK format was designed by TrackVis with the assumption that voxels are 1 mm^3, and that coordinates refer to voxel corners rather than centers. To obtain the actual vox2ras matrix (voxel center in RAS), the raw matrix must be combined with a voxel-size scaling and a half-voxel offset correction: `vox2ras_corrected = mm2ras %*% mm_correction %*% vox2mm`, where `vox2mm` scales by the inverse voxel size and `mm_correction` shifts by -0.5 mm. Note that DSI Studio does **not** apply this half-voxel shift, so you may need to set `shift_origin=FALSE` for DSI Studio files.

Use `coords = 'ras'` to have the transformation applied to the returned coordinates. That transformation additionally accounts for the `voxel_order` stored in the header, like the reference implementation in 'nibabel' does, see `trackvis.affine.to.rasmm`. Note that the matrix stored in the header alone is **not** sufficient to transform the coordinates: it has to be combined with the voxel sizes, the half-voxel offset and the orientation.

Examples

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

# Coordinates in RAS+ mm, ready for plotting against an MNI template:
trk_ras <- read.dti.trk("~/simple.trk", coords = "ras")

## End(Not run)
```

`read.dti.trk.header` *Read the header of a TrackVis TRK file.*

Description

Reads and parses the 1000 byte header of a TRK file without touching the track data. This is cheap even for huge tractograms and can be used to inspect a file before deciding whether to read its tracks.

Usage

```
read.dti.trk.header(filepath, shift_origin = TRUE)
```

Arguments

filepath	character string, path to a file in TRK format. Gzip-compressed files (typically named .trk.gz) are supported, the compression is detected from the file content, not from the file name.
shift_origin	logical, whether to compute the corrected vox2ras matrix, see read.dti.trk.

Value

named list, the header. It contains all fields of the TRK header as described at http://trackvis.org/docs/?subsect=file plus vox2ras (the matrix stored in the file) and, if shift_origin is TRUE, vox2ras_corrected.

Examples

```
## Not run:
hdr <- read.dti.trk.header("brain.trk");
hdr$n_count;      # number of tracks stated in the header
hdr$voxel_order;

## End(Not run)
```

---

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

Description

Reads per-vertex track scalars from a TSF file, optionally gzip-compressed.

Usage

```
read.dti.tsf(
  filepath,
  max_tracks = Inf,
  skip_tracks = 0L,
  chunk_values = 4e+06
)
```

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.
max_tracks	numeric, the maximum number of tracks to read. Use Inf (the default) to read all tracks. This allows reading a subset of a very large tractogram without holding all of it in memory.
skip_tracks	integer, the number of tracks to skip before reading any. Skipped tracks are never held in memory.
chunk_values	integer, the number of payload values that are read per chunk. This is an advanced tuning parameter that does not change the result, only the peak memory usage (at most one chunk is buffered at a time) and the granularity of the file reads. The default of 4e6 values corresponds to about 32 MB of doubles. Lower it on a machine with very little free memory.

Value

named list with entries 'header' and 'scalars'. The scalar data are available in 3 representations: 'merged' (a vector of all values), 'lengths' (the number of values per track) and 'scalar_list' (a list of vectors, one per track, which is expensive for large files and should be avoided for whole-brain data in favour of 'merged' and 'lengths').

Examples

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

## End(Not run)
```

---

read.dti.tsf.header *Read the header of a file in MRtrix 'TSF' format.*

Description

Reads only the ASCII header of the file, see read.dti.tck.header.

Usage

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

Arguments

filepath	character string, path to the TCK or TSF file to read. Gzip-compressed files (i.e., .tck.gz) are supported and the compression is detected from the file content, not the file name.

Value

named list, the parsed header.

Examples

```
## Not run:
hdr <- read.dti.tsf.header("brain.tsf");

## 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: `atlas.from.lut.and.csv()`, `colortable.from.annot()`, `read.fs.colortable()`, `write.atlas.to.lut.and.csv()`, `write.fs.annot()`, `write.fs.annot.gii()`, `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: `atlas.from.lut.and.csv()`, `colortable.from.annot()`, `read.fs.annot()`, `write.atlas.to.lut.and.csv()`, `write.fs.annot()`, `write.fs.annot.gii()`, `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.connectome.cifti

Read a CIFTI-2 connectome file.

Description

Read a CIFTI-2 file whose two matrix dimensions both describe brainordinates or parcels, i.e. a dense connectome (`.dconn`), a parcellated connectome (`.pconn`) or one of the mixed types (`.pdconn`, `.dpconn`). The result contains the matrix and the parcels or brainordinates its rows and columns belong to; this is what makes a connectome file usable, since the numbers alone do not say which pair of regions a value describes.

Both values of a connectome are stored in the file (the matrix is not symmetric on disk, and the diagonal is stored as well), so reading a `.pconn` gives a square matrix with the number of parcels as its number of rows and columns. A real `.dconn` (an HCP subject has 91,282 grayordinates, i.e. 8.3 billion values, 33 GB) can not be read into memory at all: use the `rows` and `columns` parameters to read only the part you need, which is a contiguous block of the file for each requested column.

Usage

```
read.fs.connectome.cifti(filepath, rows = NULL, columns = NULL)
```

Arguments

filepath	character string, the path of a CIFTI-2 file, see read.cifti. An fs.cifti object (see read.cifti.header) or an fs.cifti.data object (see read.cifti) are accepted as well.
rows	integer vector or NULL, the indices of matrix dimension 0 to read, see read.cifti.
columns	integer vector or NULL, the indices of matrix dimension 1 to read. This is the way to read part of a file that is too large to read completely, see read.cifti.

Value

a named list with class 'fs.connectome':

- 'data': numeric matrix, the connectome, with the parcels or brainordinates as the dimnames of its rows and columns (see cifti.dim.labels),
- 'parcel_names': character vector, the names of the parcels, or NULL if the file has no parcellated dimension,
- 'parcels': data.frame with one row per parcel (the columns 'index', 'name', 'num_vertices' and 'num_voxels', see cifti.parcels), or NULL,
- 'parcels_dim': integer, the matrix dimension that holds the parcels (0 or 1), or NA if the file has none,
- 'grayordinates': data.frame with one row per brainordinate of the dense dimension (see cifti.grayordinates), or NULL,
- 'grayordinates_dim': integer, the matrix dimension that holds the dense brainordinates (0 or 1), or NA if the file has none,
- 'header': the fs.cifti metadata object, see read.cifti.header.

Note that the parcels and brainordinates describe *all* indices of the dimension they belong to, not only the ones selected with rows or columns.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
pconn_file <- system.file("extdata", "cifti", "tiny.pconn.nii", package = "freesurferformats")
conn <- read.fs.connectome.cifti(pconn_file)
dim(conn$data)
conn$parcel_names
conn$data[1:2, 1:2]
```

`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()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph()`, `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()`, `write.fs.weight.asc()`

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()`, `read.fs.label.native()`, `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(), read.fs.label.gii(), 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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

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.

Note

CIFTI-2 files (which are NIFTI-2 files, see read.cifti) are refused with an error instead of being read as NIFTI morphometry data: their payload is a matrix whose dimensions the CIFTI XML metadata describes, so the values would come back in an order that means nothing. Use read.fs.morph.cifti to read the morphometry data of one brain structure, or read.cifti for the matrix itself.

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(), 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(), write.fs.weight.asc()

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

Read the data from a CIFTI dscalar file (or any other CIFTI-2 file with a single brainordinate dimension, e.g. a '.dtseries') and reconstruct it for one brain structure, based on the metadata like vertex counts, indices and offset in the CIFTI file. The result is a vector with one value per vertex of the surface.

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. An 'fs.cifti' object from read.cifti.header or an 'fs.cifti.data' object from read.cifti may be given instead of a path (this is faster if you need the data of several structures). An object created by the 'cifti' package is still accepted for backwards compatibility.
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 that the result always has the length of the *complete* surface: CIFTI files like the ones used in the HCP leave out the medial wall vertices, and the result reports those as NA.

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

```
cifti_file <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
morph_lh <- read.fs.morph.cifti(cifti_file, "lh")
length(morph_lh)
morph_lh2 <- read.fs.morph.cifti(cifti_file, "lh", 2L)
morph_both <- read.fs.morph.cifti(cifti_file, "both")
```

---

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()`, `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()`, `write.fs.weight.asc()`

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.parcellation.cifti` *Read surface parcellation data from CIFTI dlabel files.*

Description

Read the data from a CIFTI dlabel file (a dense surface parcellation, i.e., an integer label key per vertex) and return the per-vertex label keys for a single brain structure. This is the CIFTI analogue of a FreeSurfer annotation file.

Usage

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

Arguments

filepath	character string, the full path to a file in CIFTI 2 format, should end with '.dlabel.nii'. Note that this is NOT a NIfTI file, despite the '.nii' part; it uses a CIFTI 2 header instead. See the spec for details. An 'fs.cifti' object from read.cifti.header or an 'fs.cifti.data' object from read.cifti may be given instead of a path (this is faster if you need the data of several structures). An object created by the 'cifti' package is still accepted for backwards compatibility.
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 (map) to return. A CIFTI dlabel file can contain several parcellations in different columns in a single file; this selects which one. The columns are not named, so you will need to know this in advance if the file has several maps (you can use the 'with_label_table' option to inspect the map names).
with_label_table	logical, whether to also return the label table (the mapping from label key to region name and RGBA color) for the selected map. If 'TRUE', the return value is a list, see the Value section.

Value

If 'with_label_table' is 'FALSE' (the default), an integer vector with one label key per vertex in the surface. Vertices which did not have a value in the CIFTI data are set to NA; a key of 0 denotes the unknown / medial wall label (see the label table). If 'with_label_table' is 'TRUE', a named list with entries: 'label_keys' (the integer vector), 'label_table' (a data.frame with columns 'Key', 'Red', 'Green', 'Blue', 'Alpha' and 'Label', where the RGBA values are in the range 0 to 1), 'map_name' (the name of the selected map as a character string, or 'NULL' if unnamed), and 'brain_structure' (a character string). If 'brain_structure' is 'both' and 'with_label_table' is 'TRUE', a named list with entries 'lh' and 'rh', each as described above.

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

```
label_file <- system.file("extdata", "cifti", "tiny.dlabel.nii", package = "freesurferformats")
parc_lh <- read.fs.parcellation.cifti(label_file, "lh")
table(parc_lh)
parc_with_table <- read.fs.parcellation.cifti(label_file, "lh", with_label_table = TRUE)
```

```
parc_with_table$label_table
```

`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.series.cifti *Read surface time series data from CIFTI dtseries files.*

Description

Read the data from a CIFTI dtseries file (a dense surface time series) and return the per-vertex time series for a single brain structure as a matrix.

Usage

```
read.fs.series.cifti(filepath, brain_structure = "CIFTI_STRUCTURE_CORTEX_LEFT")
```

Arguments

filepath	character string, the full path to a file in CIFTI 2 format, should end with '.dtseries.nii'. Note that this is NOT a NIfTI file, despite the '.nii' part; it uses a CIFTI 2 header instead. See the spec for details. An 'fs.cifti' object from read.cifti.header or an 'fs.cifti.data' object from read.cifti may be given instead of a path (this is faster if you need the data of several structures). An object created by the 'cifti' package is still accepted for backwards compatibility.
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', a named list with entries 'lh' and 'rh' will be returned. If it is an integer, it will be interpreted as an index into the list of structures within the CIFTI file, use with care.

Value

A numeric matrix with one row per vertex in the surface and one column per time point (series point) in the file. The value for vertices which did not have a value in the CIFTI data is set to NA. If 'brain_structure' is 'both', a named list with entries 'lh' and 'rh', each a matrix as described above.

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

```
cifti_file <- system.file("extdata", "cifti", "tiny.dtseries.nii", package = "freesurferformats")
series_lh <- read.fs.series.cifti(cifti_file, "lh")
dim(series_lh)
series_both <- read.fs.series.cifti(cifti_file, "both")
names(series_both)
```

---

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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

---

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(), 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.mesh.brainvoyager(), read_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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()`, `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.mesh.brainvoyager()`, `read_nisurface()`, `read_nisurfacefile()`, `write.fs.surface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`

---

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(), 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.mesh.brainvoyager(), read_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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(), 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.mesh.brainvoyager(), read_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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()`, `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.mesh.brainvoyager()`, `read_nisurface()`, `read_nisurfacefile()`, `write.fs.surface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`

---

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(), 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.mesh.brainvoyager(), read_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

---

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(), 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.mesh.brainvoyager(), read_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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. The normals of the faces are available in the 'metadata' property, in the entry 'normals' (a matrix with one row per face). Note that the normal vectors stored in the file are ignored, the returned normals are computed from the geometry, see mesh.face.normals.

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 legacy format mesh as surface.*

Description

Reads meshes from files in the VTK legacy format. Both the ASCII and the binary encoding are supported, as are the cell array layouts written by VTK 4.2 and older and by VTK 5.1 and newer, see the notes. See https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf for the format specification. Note that this function does **not** read arbitrary VTK datasets, it supports only the geometry of POLYDATA datasets (meshes and point clouds); attribute data such as normals, texture coordinates or scalars is ignored. Only triangular meshes are supported, files containing other polygons are rejected with an error.

Usage

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

Arguments

filepath	string. Full path to the input surface file in VTK 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. Files that store streamlines instead of a mesh (i.e., that contain a LINES section) are read with `read.fs.tracts.vtk`.

See Also

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

Examples

```
surface_file <- system.file("extdata", "cube.vtk", package = "freesurferformats", mustWork = TRUE)
mesh <- read.fs.surface.vtk(surface_file)
cat(sprintf("Read a mesh with %d vertices and %d faces.\n", nrow(mesh$vertices), nrow(mesh$faces)))
```

---

read.fs.tracts.vtk *Read VTK legacy format streamlines as tracts.*

Description

Reads streamline (tractography) data from files in the VTK legacy format, i.e. from POLYDATA datasets that contain a LINES section. Both the ASCII and the binary encoding are supported, as are the cell array layouts written by VTK 4.2 and older and by VTK 5.1 and newer. This is the format that Paraview, TrackVis and DSI Studio export streamlines in. Attribute data such as scalar values per point is ignored.

Usage

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

Arguments

`filepath`	character string, path to the input file in VTK legacy format.

Value

an `fs.tracts` instance with the streamlines. Use `tracts[[i]]` to get the n x 3 coordinate matrix of a single streamline, and `fs.tracts.lengths` for the number of points of each streamline.

Note

Points that are not part of any streamline are dropped, because the `fs.tracts` data structure stores the coordinates of the streamlines without gaps. A warning is issued if that happens.

Examples

```
# The example file was written by VTK and contains 3 streamlines.
# It also has a point that is not part of any streamline, which is dropped with a warning.
tracts_file <- system.file("extdata", "tracts_v51_binary.vtk",
  package = "freesurferformats", mustWork = TRUE
)
tracts <- suppressWarnings(read.fs.tracts.vtk(tracts_file))
cat(sprintf(
  "Read %d streamlines with %d points.\n",
  length(tracts), nrow(fs.tracts.coords(tracts))
))
```

`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. 'auto' guesses it from the file extension and the file content, and 'xfm' (for xform format), 'dat' (for tkregister style, e.g. register.dat), 'lta' (for FreeSurfer LTA), 'fslmat' (for an FSL/FLIRT matrix) and 'itk' (for an ITK text transform, e.g. a `.tfm` file) can be given explicitly.

Value

an `fs.transform` instance, see `fs.transform`. Its fields include the 'matrix', and the coordinate spaces the matrix maps between (`space_in`, `space_out` and `voxel_base`). Which of them are known depends on the format: an xfm file states neither (both sides are RAS), a register.dat file states both by definition, an FSL matrix maps voxel coordinates (zero-based) to voxel coordinates, and an LTA file states the spaces in its header.

Note

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

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

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

an `fs.transform` instance. A tkregister matrix maps the movable volume (the source) to the target volume, so `space_in` is 'voxel' and `space_out` is 'ras'. It produces RAS coordinates in the tkregister frame of the target volume, which is why `dst` states `frame = 'tkreg'`, see `mghheader.vox2ras.tkreg`. The other entries of the file are kept as the `subject`, `in_plane_resolution`, `between_plane_resolution` and `intensity` fields.

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

Examples

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

---

read.fs.transform.fslmat

Read a transformation matrix from an FSL matrix file.

Description

Read the plain text 4x4 matrix that FSL's `flirt` writes with the `-omat` option, and that FSL, MRtrix3 and FreeSurfer read as the registration between two images.

Usage

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

Arguments

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

Value

an `fs.transform` instance. An FSL matrix maps the voxel coordinates of the image given to `flirt -in` to those of the image given to `flirt -ref`. Both are voxel indices, so `space_in` and `space_out` are 'voxel' and `voxel_base` is 0 (FSL voxel indices are zero-based). The two images are not recorded in the file, so `src` and `dst` are `NULL` and the volumes have to be passed to `transform2world` to interpret the matrix in world coordinates.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
# Write the example LTA as an FSL matrix and read it back: no FSL installation is needed for that.
lta_file <- system.file("extdata", "talairach.lta", package = "freesurferformats", mustWork = TRUE)
mat_file <- tempfile(fileext = ".mat")
write.fs.transform(read.fs.transform(lta_file), mat_file, format = "fslmat")
read.fs.transform(mat_file)$matrix
unlink(mat_file)
```

---

read.fs.transform.itk *Read a transformation matrix from an ITK text transform file.*

Description

Read the plain text file format that ITK and the tools built on it (3D Slicer, ANTs via `ConvertTransformFile`, SimpleITK and the workflows of fMRIPrep/QSIPrep that are based on them) use to store linear transformations, usually with the extension `.tfm` or `.txt`.

Usage

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

Arguments

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

Value

an `fs.transform` instance. An ITK transform operates on the world coordinates of the images, which in ITK are left-posterior-superior, so `space_in` and `space_out` are 'lps' and `voxel_base` is `NA`. This is not the RAS space that the other formats of this package use, and it is not converted silently: use transform2ras to get a transformation in RAS coordinates. The volumes are not recorded in the file, so `src` and `dst` are `NULL`. The ITK class name (e.g. 'AffineTransform_float_3_3') is stored in the `type` field, and the values of the `FixedParameters` entry in the `fixed_parameters` field.

Note

The format can store many kinds of transforms besides affine ones; this function reads the affine transformations only, i.e. the classes 'AffineTransform_float_3_3', 'AffineTransform_double_3_3', 'MatrixOffsetTransformBase_float_3_3' and 'MatrixOffsetTransformBase_double_3_3'. These are the classes that occur in the output of the pipelines mentioned above, and the only ones for which the interpretation of the parameters could be verified against other implementations. Files that contain several transformations (an ITK 'CompositeTransform') are not supported either, and are reported as such: composing them requires the ordering rules of ITK, which would be a guess without a reference to check against.

The `FixedParameters` entry is the center of rotation, so the matrix that is returned is `y = A(x - c) + t + c`, i.e. it has the center folded in. That is the same thing that the ITK writer of this package stores, and the transformation is not changed by it.

FreeSurfer reads ITK files as well, but two limitations of its version 7.4.1 are worth knowing when the file has to be passed to it: it rejects the 'float' variant of the classes ('readITK: Transform type unknown!'), and its `lta_convert --initk` ignores the `FixedParameters`, so it interprets a file with a non-zero center of rotation differently from ITK itself (which computes `offset = translation + center - matrix * center`, see `ComputeOffset()` in ITK's `itkMatrixOffsetTransformBase.hxx`) and from this package. Both were verified by converting files that encode the same transformation, and both are avoided by the files that `write.fs.transform.itk` writes.

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

Examples

```
xfm_file <- system.file("extdata", "talairach.xfm", package = "freesurferformats", mustWork = TRUE)
tf <- transform2ras(transform2lps(read.fs.transform(xfm_file)))
summary(tf)$space_in
```

---

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

an fs.transform instance. The header of an LTA file states whether the matrix operates on voxel indices (type 0, LINEAR_VOX_TO_VOX) or on RAS world coordinates (type 1, LINEAR_RAS_TO_RAS), and that is used to set space_in, space_out and voxel_base. FreeSurfer voxel indices are zero-based. The file also records both volumes it relates, so src and dst contain the file name, the dimensions, the voxel size and the voxel-to-RAS matrix of each of them. The parsed header and volume info sections are kept in the header and volumes fields.

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: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

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

an fs.transform instance. An xfm file stores a linear transformation between two RAS (world) coordinate spaces, typically the scanner space of a subject and the RAS space of an MNI or Talairach template, so space_in and space_out are both 'ras' and voxel_base is NA. The volumes that the transformation relates are not recorded in the file, so src and dst are NULL.

Note

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

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

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', 'mgz', 'nrrd' or 'analyze'. The format to assume. If set to 'auto' (the default), the format will be derived from the file extension. The value 'analyze' covers the two-file image formats ANALYZE 7.5 and NIFTI v1 pair files, which share the .hdr/.img file extensions, see read.fs.volume.analyze.
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.

Note

CIFTI-2 files (which are NIFTI-2 files, see read.cifti) are refused with an error instead of being read as volumes: their payload is a matrix whose dimensions the CIFTI XML metadata describes, so the values would come back as voxels that mean nothing.

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(), read.fs.morph.gii(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

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.analyze

Read an ANALYZE 7.5 or NIFTI v1 pair file as an fs.volume.

Description

Read a volume from the two-file image formats ANALYZE 7.5 and NIFTI v1: a 348 byte header (<base>.hdr) and the voxel data in a separate file (<base>.img). Read read.analyze.header for the difference between the two variants and for the limitations of the ANALYZE header.

Usage

```
read.fs.volume.analyze(
  filepath,
  flatten = FALSE,
  with_header = FALSE,
  drop_empty_dims = FALSE,
  spm = FALSE
)
```

Arguments

filepath	character string, the path to one of the two files of the pair, or to its base name. See analyze.pair.files.

`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

`spm` logical, whether to interpret the ANALYZE header the way SPM, FreeSurfer and the other tools of the ecosystem do. This concerns the fields that ANALYZE 7.5 leaves undefined but that SPM uses: `funused1` is the scale factor of the data, and `originator` holds the voxel coordinates of the image origin. It makes this function apply the scale factor, and use the origin to derive a transformation matrix when the file has no MATLAB sidecar file (see the geometry section below). Only relevant for ANALYZE files, since the NIFTI v1 variant of the pair stores a proper transformation matrix.

Value

an `fs.volume` instance, i.e. a named list with the entries `data` (the voxel data) and `header` (the image header), or only the data array if `with_header` is `FALSE`. The header is the format specific header (an ANALYZE 7.5 header as returned by `read.analyze.header` for ANALYZE files, a NIFTI v1 header as returned by `read.nifti1.header` for NIFTI pair files), and not an MGH header. In addition, the following entries are present:

`vox2ras_matrix` the 4x4 voxel-to-RAS transformation matrix, computed from the `sform`/`qform` header fields for NIFTI v1 pair files. For ANALYZE files this entry is only present if `spm` is `TRUE`, see the note below.

`vox2ras_source` character string, a description of where the matrix comes from, one of `'sform'`, `'qform'` (for NIFTI v1 pair files), `'mat sidecar'`, `'spm origin'` or `'analyze convention'` (for ANALYZE files, see the geometry section). Only present if `vox2ras_matrix` is present.

`voldim` only present if `flatten` is `TRUE`, exactly as in the other volume readers.

`filepath`, `filepath_header`, `filepath_image` the paths of the header and data files.

The geometry of ANALYZE 7.5 files

The ANALYZE 7.5 header stores the voxel sizes (`pix_dim`) but neither the direction of the voxel axes in world space nor the position of the image in it. This means that the left/right orientation of an ANALYZE image is *not known* from the header, and no implementation of the format can know it: the format is famous for exactly this problem, and the `orient` field that was added to fix it is set by some tools and ignored by others.

There are two ways to get a transformation matrix for such a file, and this function supports both:

The MATLAB sidecar file SPM and FreeSurfer write the transformation matrix into a file named `<base>.mat` next to the `<base>.img` file, as a MATLAB v4 file with a variable named `mat` or `M`. This matrix is real information from the file set, not a convention, so it is used whenever such a file is present, and the `vox2ras_source` of the result is `'mat sidecar'`. The `M` variant does not contain the flip of the first voxel axis that the format's original software applies, so that flip is added (this is also what the reference implementation of the format, nibabel, does). Note that the matrix in the file maps the *1-based* voxel indices that MATLAB uses, and that the matrix returned here maps the 0-based indices that R and this package use: the difference is the sum of the rows of the rotation part, i.e. up to several millimeters, and getting it wrong is exactly the kind of silent error that is hard to notice. FreeSurfer writes only the `M` variable, nibabel writes both. MATLAB files of version 5 and newer (the `.mat` files that recent MATLAB versions write, which are compressed) cannot be read; such a file is reported and ignored, the volume data is still returned.

The SPM convention (`spm = TRUE`) If there is no sidecar file, the SPM and FreeSurfer tools derive the geometry from the fields that ANALYZE leaves unused: the `originator` field holds the voxel coordinates of the image origin, the voxel sizes give the axis lengths, and the axes are assumed to point to the left, the front and the top (`diag(-x, y, z)`, the convention of the format's original software). The same `spm = TRUE` applies the scale factor that SPM stores in `funused1`. Both are heuristics that the file does not state, which is why they are not used by default: a matrix from a convention can be wrong by a mirror image.

Without a sidecar file and with `spm = FALSE` (the default), no matrix is reported at all: the voxel sizes and the orientation code are returned as they are stored, and you can build a matrix from them if the convention of your choice is known to be the right one for your data.

See Also

`read.analyze.header`, `read.analyze.data`, `read.fs.volume`

Examples

```
hdrfile <- system.file("extdata", "analyze", "tiny_u8.hdr",
  package = "freesurferformats", mustWork = TRUE
)
vol <- read.fs.volume.analyze(hdrfile, with_header = TRUE)
dim(vol$data)
vol$header$pix_dim
```

---

read.fs.volume.nii *Read a 3D or 4D NIFTI file into an* `fs.volume` *instance with complete header.*

Description

This function reads a NIFTI v1 or v2 file, or takes a `nifti` instance from the `oro.nifti` package, 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). Files are read with

the NIFTI reader of this package, so the oro.nifti package is only needed if a nifti instance is passed, or if reorient or extra arguments are used. Currently only few datatypes are supported, and the orientation can only be derived if the sform or qform header field is present.

Usage

```
read.fs.volume.nii(
  filepath,
  flatten = FALSE,
  with_header = FALSE,
  drop_empty_dims = FALSE,
  do_rotate = FALSE,
  reorient = 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.
reorient	logical, whether to let oro.nifti::readNIfTI reorient the data array to a standard orientation while reading it from a file. Defaults to FALSE, see the note below. Only relevant if filepath is a path, it is ignored if a nifti instance is passed. Using TRUE requires the oro.nifti package.
...	extra parameters passed to oro.nifti::readNIfTI. Leave this alone unless you know what you are doing. Note that reorient is passed explicitly by this function, so it cannot be set here. Passing any extra parameter makes the file be read by oro.nifti instead of by the NIFTI reader of this package, and thus requires the oro.nifti package.

Value

an fs.volume instance. The header fields are computed from the NIFTI header. The data array is returned in the raw NIFTI file storage order (first dimension fastest), which is also the order

used by the MGH/MGZ format. For a file, the NIFTI data scaling fields `scl_slope`/`scl_inter` of the header are applied to the values while reading (this is what the `oro.nifti` package does for files as well), for an `oro.nifti` instance the values are returned as they are stored in the instance. If the NIFTI file contains `sform` or `qform` geometry information, the returned header contains a `vox2ras_matrix` entry in addition to the MGH header fields, just like the header returned by `read.fs.mgh`.

Note

The data array is *not* reoriented, because the NIFTI geometry is stored in the `sform`/`qform` header fields, and these describe the raw file storage order. Reorientation as performed by `oro.nifti::readNIfTI(reorient = TRUE)` permutes or flips the data array without updating the `sform` fields, so the data array would no longer match the header of the returned `fs.volume` (and thus not the `vox2ras_matrix` used to map voxel indices to coordinates). Using `reorient = TRUE`, or passing a `nifti` instance that was read with `reorient = TRUE`, is therefore discouraged and results in a warning.

Files are read with the NIFTI reader of this package, which reads both NIFTI v1 and NIFTI v2 files and does not require the `oro.nifti` package. Two details of the way `oro.nifti` used to read files are kept, so that the returned values do not change: the NIFTI data scaling fields (`scl_slope`/`scl_inter`) are applied to the values while reading (see the return value section), and voxel sizes that are stored as 0 for a used dimension (or that are not finite) are reported as 1.

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`): such files contain a single dimension, and the volume returned for them is degenerate. Use `read.fs.morph` to read morphometry data.

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.volume.nrrd` *Read volume data from a file in NRRD format.*

Description

Reads a volume from a file in NRRD format (`.nrrd`), i.e. the format that 3D Slicer, ITK/SimpleITK, DTI-TK, DSI Studio and the `teem` tools use. Detached headers (`.nhdr`, whose data are in separate files) and gzip-compressed files (`.nrrd.gz`) are supported as well, and the reader is implemented in this package, so no additional R package or external software is needed. The values are returned in the order in which they are stored in the file, and NRRD stores the first axis fastest, which is the order that R uses for arrays as well, so the returned array is shaped exactly like the volume described by the header.

Usage

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

Arguments

`filepath`	character string, path to the file in NRRD format.
`flatten`	logical, whether to return a numeric vector instead of a multidimensional array.
`with_header`	logical, whether to return an `fs.volume` instance (a named list with the entries `data` and `header`) instead of the data array. The header contains the geometry of the volume in the `vox2ras_matrix` entry, see `read.nrrd.header`.
`drop_empty_dims`	logical, whether to drop dimensions of length 1 from the returned data, e.g. the frame axis of a 3D volume that was stored as a 4D array with one volume. Note that NRRD files state their dimensionality explicitly, so unlike the MGH or NIfTI readers this one only drops dimensions that the file actually has.

Value

a multidimensional array of the dimensions stated in the file header, a numeric vector if `flatten` is `TRUE`, or an `fs.volume` instance if `with_header` is `TRUE`. The data type of the array follows the file: integer for the integer types up to 32 bit, double for the floating point types and for the 64 bit integer types (which have no R equivalent, values above 2^53 lose precision, and a warning is raised in that case).

Note

The data are returned exactly as they are stored in the file: the reader does not reorient, rescale or otherwise change them. Use the `vox2ras_matrix` entry of the header to map voxel indices to RAS+ coordinates, which is the space in which FreeSurfer, NIfTI and the rest of this package work. NRRD files that state the LPS space (which is what the ITK tools write) are handled: the matrix is converted to RAS. For a volume whose file states no space information at all, no matrix is available (the entry is NULL), since any matrix would be a guess.

Diffusion MRI data in NRRD format carry their b-value and their gradient directions in custom header fields (the convention of the `teem` tools, DTI-TK and 3D Slicer). They are parsed into the `dwi` entry of the header by `read.nrrd.header`. The gradients are given in the image frame, and the `measurement_frame` entry of the `dwi` list is the rotation that maps them into the frame in which the measurement was performed; a caller that needs the gradients in that frame has to apply it (multiply the vector by the matrix). The values can be handed to `read.dti.gradients` together with one b-value per volume, see the examples.

Examples

```
nrrd_file <- system.file("extdata", "nrrd", "vol_u8_raw.nrrd",
  package = "freesurferformats", mustWork = TRUE
)
vol <- read.fs.volume.nrrd(nrrd_file)
dim(vol)

# Read the geometry as well:
volh <- read.fs.volume.nrrd(nrrd_file, with_header = TRUE)
volh$header$vox2ras_matrix

## Not run:
# Read a DWI volume and use its gradient table:
hdr <- read.nrrd.header("dwi.nrrd")
gradients <- read.dti.gradients(hdr$dwi$bvec, rep(hdr$dwi$b_value, hdr$dwi$num_gradients))

## 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(), read.fs.morph.gii(), read.fs.volume(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

---

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(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

`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. The header extensions, if the file has any, are returned in the field `extensions`, a list in which each entry is one header extension as created by `nifti2.extension` (i.e., a list with the entries 'ecode' and 'content'). CIFTI2 files store their XML metadata in such an extension, see `nifti2.get.extension`.

See Also

`write.nifti2`

---

read.nrrd.header *Read the header of a file in NRRD format.*

Description

Reads and parses the ASCII header of a NRRD file without touching the volume data. This is cheap, and it can be used to inspect a file, or to check its geometry, before deciding whether to read the data.

Usage

```
read.nrrd.header(filepath)
```

Arguments

filepath	character string, path to the NRRD file (.nrrd or .nhdr).

Value

named list, the parsed header. The fields of the file are available under their normalized names (e.g. space_directions for the field 'space directions'), and fields that this function does not interpret are collected in the other_fields entry. The derived entries are vox2ras_matrix (the 4x4 matrix that maps 0-based voxel indices to RAS+ coordinates, or NULL when the file states no geometry), vox2ras_source (how the matrix was derived: 'space directions' or 'spacings'), data_files (the resolved paths of the data files for a detached header, or NULL), data_offset (the byte offset of the data in the header file), gzipped_file (whether the whole file is gzip-compressed), header_size (the size of the header in bytes) and dwi (the parsed diffusion metadata, or NULL, see read.fs.volume.nrrd).

Examples

```
nrrd_file <- system.file("extdata", "nrrd", "vol_u8_raw.nrrd",
  package = "freesurferformats", mustWork = TRUE
)
hdr <- read.nrrd.header(nrrd_file)
hdr$sizes
hdr$vox2ras_matrix
```

---

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://helpdesk.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(), 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(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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()`, `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_nisurface()`, `write.fs.surface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.mz3()`, `write.fs.surface.vtk()`

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: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfr

---

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: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfr

summary.fs.transform *Summarize an fs.transform instance.*

Description

Compute the properties of a transformation in a machine-readable form, for printing or for further processing.

Usage

```
## S3 method for class 'fs.transform'
summary(object, ...)
```

Arguments

object	an fs.transform instance.
...	ignored, only present for consistency with the generic.

Value

named list with the entries 'format', 'source', 'space_in', 'space_out', 'voxel_base', 'src', 'dst' (the corresponding fields of the transform), 'translation' (numerical vector of length 3, the last column of the matrix), 'determinant', 'is_affine' (logical, whether the last row of the matrix is '0 0 0 1') and 'is_invertible' (logical).

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
tf <- read.fs.transform(system.file("extdata", "register.dat",
  package = "freesurferformats", mustWork = TRUE
))
summary(tf)$space_out
```

`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/transfo
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/transfo

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

---

transform2lps *Convert a transformation to LPS world coordinates.*

Description

The reverse of transform2ras, for transformations that have to be expressed in the world coordinates that ITK and the tools built on it use. A transformation that is already in LPS coordinates is returned unchanged.

Usage

```
transform2lps(tf)
```

Arguments

tf	an fs.transform instance whose matrix operates on world coordinates, i.e. space_in and space_out are either 'lps' or 'ras'.

Value

an fs.transform instance whose matrix operates on LPS coordinates.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm

Examples

```
tf <- read.fs.transform(system.file("extdata", "talairach.xfm",
  package = "freesurferformats", mustWork = TRUE
))
# The matrix changes, because the sign of the first two axes changes.
max(abs(transform2lps(tf)$matrix - tf$matrix)) > 0
```

`transform2ras` *Convert a transformation to RAS world coordinates.*

Description

ITK and the tools built on it (3D Slicer, ANTs, SimpleITK and the workflows that use them) work in world coordinates that are left-posterior-superior (LPS), while the other file formats of this package use right-anterior-superior (RAS) coordinates. The two conventions differ in the sign of the first two axes only, so converting a transformation between them neither needs nor uses the geometry of a volume, unlike the conversion between voxel and world coordinates, see `transform2world`. A transformation that is already in RAS coordinates is returned unchanged.

Usage

```
transform2ras(tf)
```

Arguments

`tf`	an `fs.transform` instance whose matrix operates on world coordinates, i.e. `space_in` and `space_out` are either 'lps' or 'ras'.

Value

an `fs.transform` instance whose matrix operates on RAS coordinates.

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfr`

Examples

```
# An ITK transform operates on LPS coordinates, the FreeSurfer formats on RAS coordinates.
tf <- read.fs.transform(system.file("extdata", "talairach.xfm",
  package = "freesurferformats", mustWork = TRUE
))
summary(tf)$space_in
summary(transform2lps(tf))$space_in
summary(transform2ras(transform2lps(tf)))$space_in
```

`transform2voxel` *Convert a transformation to voxel coordinates.*

Description

The inverse operation of `transform2world`: given a transformation that operates on world coordinates, compute the matrix that maps voxel indices of one volume to voxel indices of another. The result can be saved as an FSL matrix, see `write.fs.transform.fslmat`.

Usage

```
transform2voxel(tf, src = NULL, dst = NULL)
```

Arguments

`tf`	an `fs.transform` instance with a matrix in voxel coordinates (`space_in` and `space_out` are 'voxel'). A transformation that already operates on world coordinates is returned unchanged.
`src`	`NULL` or the volume the transformation maps from (the `-in` image of FSL, the `src` volume of an LTA). Required for FSL matrices, since their files do not record the volumes.
`dst`	`NULL` or the volume the transformation maps to (the `-ref` image of FSL, the `dst` volume of an LTA). Required for FSL matrices.

Value

an `fs.transform` instance whose matrix operates on (zero-based) voxel coordinates.

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.itk()`, `write.fs.transform.lta()`, `write.fs.transform.xfr`

Examples

```
lta_file <- system.file("extdata", "talairach.lta", package = "freesurferformats", mustWork = TRUE)
tf_world <- transform2world(read.fs.transform(lta_file))
tf_voxel <- transform2voxel(tf_world)
max(abs(tf_voxel$matrix - read.fs.transform(lta_file)$matrix)) # back where we started
```

`transform2world` *Convert a transformation to the world (RAS) coordinate space.*

Description

Transformation files often store their matrix in voxel coordinates, which means that the matrix alone cannot be used to transform world coordinates (e.g. the coordinates of a brain surface vertex, or a peak coordinate from another study): the geometry of the volumes the matrix relates is required as well. This function converts such a transformation into one that operates on world coordinates.

The result depends on the format the transformation was read from, because the formats disagree about their world space, which is why this is not a pure matrix operation:

- For an FSL matrix ('fslmat'), the world space is the one FSL uses: unit voxel axes with a flipped first axis, see `fsl.scaled.voxel.matrix`. Both MRtrix3 and FreeSurfer implement exactly this, and the result is a transformation between FSL world coordinates. The `src` and `dst` entries of the result have the frame 'fsl'.
- For an LTA of type 0 (VOX2VOX), the world space is the RAS space of the two volume geometries: the result is `vox2ras_dst \%*\% matrix \%*\% solve(vox2ras_src)`. The descriptors have the frame 'scanner'.

Usage

```
transform2world(tf, src = NULL, dst = NULL)
```

Arguments

`tf`	an `fs.transform` instance with a matrix in voxel coordinates (`space_in` and `space_out` are 'voxel'). A transformation that already operates on world coordinates is returned unchanged.
`src`	`NULL` or the volume the transformation maps from (the `-in` image of FSL, the `src` volume of an LTA). Required for FSL matrices, since their files do not record the volumes.
`dst`	`NULL` or the volume the transformation maps to (the `-ref` image of FSL, the `dst` volume of an LTA). Required for FSL matrices.

Value

an `fs.transform` instance whose matrix operates on world coordinates.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm

Examples

```
# Read an LTA file, which records both volumes it relates, and convert it to world coordinates.
lta_file <- system.file("extdata", "talairach.lta", package = "freesurferformats", mustWork = TRUE)
tf <- read.fs.transform(lta_file)
tf_world <- transform2world(tf)
tf_world$space_in
tf_world$matrix
```

---

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.analyze *Write data to a file in ANALYZE 7.5 format.*

Description

Write an array or vector to the two files that make up an ANALYZE 7.5 image: a 348 byte header (<base>.hdr) and the raw voxel data in a separate file (<base>.img).

Usage

```
write.analyze(filepath, analyzedata, analyzeheader = NULL, ...)
```

Arguments

filepath	character string, the path of the output files. It must end with .hdr or .img (optionally followed by .gz), or contain no extension at all, in which case the two file names are derived from it by appending .hdr and .img. See analyze.pair.files.
analyzedata	array of numeric (integer or double) data, can have up to 7 dimensions. See analyzeheader.for.data for how the data type is chosen, and note that the data type and the number of bits per value in the header have to match the data, otherwise the file is not readable.

analyzeheader	optional ANALYZE 7.5 header that is suitable for the passed analyzedata, see analyzeheader.for.data and analyzeheader.template. If not given, one is generated from the data, with all other fields set to the defaults of the format, i.e. the voxel sizes in pix_dim will be zero and you most likely want to set them.
...	additional parameters passed to analyzeheader.for.data. Only used if analyzeheader is NULL.

Value

invisible named list with the entries header (the header that was written) and data (the data that was written). Use these to check what the function did with your input, they reflect the values that ended up in the file, e.g. values that had to be converted to the data type stated in the header.

Note

ANALYZE 7.5 cannot store a transformation matrix, and the format does not even define the direction of the voxel axes in world space. This writer therefore cannot write the geometry of an image: if you have a volume with a known vox2ras_matrix and you want to keep it, write a NIFTI v1 file with write.nifti1 instead (either a single file, or a .hdr/.img pair by setting the header magic to 'ni1', which is what write.fs.volume does for such file names). If you know that the data will be read by SPM or FreeSurfer, you can store the image origin in the spm_origin field of the header, see read.analyze.header.

Note that tools differ in how they recognize the format. FreeSurfer's mri_convert, for example, does not infer the ANALYZE format from the .hdr file extension and reports 'cannot determine file type' unless the input type is given explicitly: mri_convert -it analyze -ot mgz vol.hdr vol.mgz works, mri_convert -ot mgz vol.hdr vol.mgz does not. The header written by this function is the one FreeSurfer itself writes for -ot analyze, so this is a limitation of the reader, not of the file.

See Also

read.analyze.header, analyzeheader.for.data

Examples

```
outfiles <- tempfile()
data <- array(1:24, dim = c(4, 3, 2))
analyzeheader <- analyzeheader.for.data(data)
analyzeheader$pix_dim <- c(1., 1.5, 1.5, 2., 0., 0., 0., 0.)
write.analyze(outfiles, data, analyzeheader)
vol <- read.fs.volume.analyze(outfiles, with_header = TRUE)
vol$header$pix_dim
```

`write.atlas.to.lut.and.csv` *Write a brain atlas to a colortable (LUT) file and a per-vertex label file.*

Description

Write a brain atlas (surface annotation) to two text files: a FreeSurfer ASCII colortable lookup table (LUT) file and a per-vertex label file. This is the inverse of `atlas.from.lut.and.csv`, and is useful for exporting an annotation in the simple text format used e.g. by the Python package 'yabplot' (files like 'aparc_LUT.txt' and 'aparc_conte69.csv'). If the annotation was created by `atlas.from.lut.and.csv`, the original per-vertex label indices are stored in its metadata and are used for the export, which makes it lossless. Otherwise, the indices are recovered by matching the per-vertex color codes against the colortable, which is ambiguous if several regions share a color code (e.g., the left/right pairs of the Desikan-Killiany 'aparc' atlas).

Usage

```
write.atlas.to.lut.and.csv(fs.annot, lut_file, csv_file, unknown_index = 0L)
```

Arguments

`fs.annot`	an annotation, as returned by `read.fs.annot` or `atlas.from.lut.and.csv`. Must have a colortable (an entry named 'colortable_df').
`lut_file`	string, path to the output colortable file. Will be written in FreeSurfer ASCII LUT format, see `write.fs.colortable`.
`csv_file`	string, path to the output per-vertex label file. Will contain one integer per line: the struct index (label ID) of the region for that vertex, as defined in the first column of the LUT file. Unlabeled vertices (e.g., the medial wall) are encoded with the struct index given in parameter `unknown_index`.
`unknown_index`	integer, the struct index to use for unlabeled vertices (i.e., vertices whose label code has no entry in the colortable). Defaults to 0.

Value

named list with the following entries: 'lut_file' and 'csv_file' (the paths to the files that were written), 'num_vertices' (integer, the number of vertices in the per-vertex label file) and 'num_regions' (integer, the number of regions in the LUT file).

See Also

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

Examples

```
## Not run:
annot <- read.fs.annot("lh.aparc.annot")
write.atlas.to.lut.and.csv(annot, "myatlas_LUT.txt", "myatlas_vertices.csv")
# read it back into an annotation:
annot2 <- atlas.from.lut.and.csv("myatlas_LUT.txt", "myatlas_vertices.csv")

## End(Not run)
```

`write.cifti` *Write a CIFTI-2 file.*

Description

Write a data matrix and its axes to a CIFTI-2 file, i.e. a NIFTI-2 file with the CIFTI-2 XML in a header extension. The axes describe what the matrix dimensions contain; they can be given explicitly (see the `cifti.axis.*()` functions), taken from a template file (recommended for real data, see `cifti.axis.from.template`), or both.

The file type (and thus the NIFTI intent code that identifies it) follows from the axes: a scalars dimension and a brain model dimension make a `.dscalar` file, a series dimension and a brain model dimension a `.dtseries` file, and so on. If the file name has one of the standard CIFTI extensions, the axes have to match it: a file named `.pdconn.nii` whose dimensions are not (brain models, parcels) is an error, not a warning, because the file name is the only hint that other software has about the content.

Usage

```
write.cifti(filepath, data, axes = NULL, template = NULL, metadata = NULL)
```

Arguments

`filepath`	character string, the path of the file to write.
`data`	numeric or integer matrix or array, the data. The dimensions have to match the sizes of the axes, and the order is the same as for `read.cifti`: the first array dimension is CIFTI matrix dimension 0. A plain vector is interpreted as an array of that length (which is what you want for a file with a single map, e.g. a `.dscalar` with one map).
`axes`	list of axes, one per matrix dimension, or `NULL` (in which case `template` has to be given). See `cifti.header.from.axes` for the naming rules.
`template`	character string (the path of a CIFTI-2 file), an `fs.cifti` or an `fs.cifti.data` object or `NULL`. The axes of this file are used for the dimensions that `axes` does not describe, and also for the metadata if `metadata` is `NULL`.
`metadata`	named character vector or named list, or `NULL`, the matrix metadata to write. The default writes the metadata of the template file, if there is one, so that the provenance of the source data is kept.

Value

the axes that were written, invisibly. The file is written as a side effect.

Note

The data are written as 32 bit floating point values, which is what the reference implementations write for CIFTI-2 files (including for label files, whose keys are small integers). Values that need more than about 7 significant digits are rounded.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
axis_brain <- cifti.axis.brain.models(list(
  cifti.brain.model.surface("lh", 10L),
  cifti.brain.model.surface("rh", 12L)))
data_matrix <- matrix(1:22, nrow = 1L)
out_file <- file.path(tempdir(), "tiny_written.dscalar.nii")
write.cifti(out_file, data_matrix, axes = list(cifti.axis.scalars("my data"), axis_brain))

## Not run:
# Write data with the mapping of an existing file (recommended for real data):
template <- "Conte69.MyelinAndCorrThickness.32k_fs_LR.dscalar.nii"
data_matrix <- read.cifti(template)$data
write.cifti("copy.dscalar.nii", data_matrix, template = template)

## End(Not run)
```

---

write.dti.bval *Write b-values to a FSL-style bvals file.*

Description

Writes the diffusion weighting of each volume of a diffusion MRI dataset to a plain text file, which is the *bvals* part of the FSL bvecs/bvals format.

Usage

```
write.dti.bval(filepath, bval, layout = "components")
```

Arguments

filepath	character string, path to the output file. The conventional extension is .bval or .bvals, but that is not enforced. Files ending in .gz are gzip-compressed.
bval	numeric vector, the b-value of each volume, in s/mm^2.
layout	character string, one of 'components' or 'volumes'. components (the default) writes all values into a single line, which is what the FSL tools produce, while volumes writes one value per line, which is what the Human Connectome Project distributes.

Value

NULL, invisibly. Called for the side effect of writing the file.

See Also

Other dti functions: read.dti.bval(), read.dti.bvec(), read.dti.grad(), read.dti.gradients(), write.dti.bvec(), write.dti.grad()

---

write.dti.bvec *Write b-vectors to a FSL-style bvecs file.*

Description

Writes the diffusion gradient directions of a diffusion MRI dataset to a plain text file, which is the *bvecs* part of the FSL bvecs/bvals format. Note that these vectors are interpreted relative to the *image* axes, so the file is only meaningful together with the image it was derived from.

Usage

```
write.dti.bvec(filepath, bvec, layout = "components")
```

Arguments

filepath	character string, path to the output file. The conventional extension is .bvec or .bvecs, but that is not enforced. Files ending in .gz are gzip-compressed.
bvec	numeric matrix with 3 columns (one row per volume) or 3 rows (one column per volume), the gradient direction of each volume.
layout	character string, one of 'components' or 'volumes'. components (the default) writes the file with 3 lines, one per vector component, which is the FSL layout, while volumes writes one volume per line, which is what the Human Connectome Project distributes.

Value

NULL, invisibly. Called for the side effect of writing the file.

Note

A square 3x3 input is ambiguous. Since a matrix passed in memory follows the R convention of one row per volume, the rows are written as the volumes here; `read.dti.bvec` assumes the components for a square file, so reading such a file back yields the transposed matrix and warns about the ambiguity.

See Also

Other dti functions: `read.dti.bval()`, `read.dti.bvec()`, `read.dti.grad()`, `read.dti.gradients()`, `write.dti.bval()`, `write.dti.grad()`

`write.dti.grad` *Write a gradient table in MRtrix3 format.*

Description

Writes a diffusion gradient table to a plain text file with one line per DWI volume, each line holding the four space-separated values `[ x y z b ]`. This is the format that the `-grad` option of the MRtrix3 commands expects, and the format of the `dw_scheme` entry in the header of a `.mif` image.

Usage

```
write.dti.grad(filepath, bvec, bval = NULL, layout = "volumes")
```

Arguments

`filepath`	character string, path to the output file. Files ending in `.gz` are gzip-compressed.
`bvec`	numeric matrix with 3 columns (one row per volume) or 3 rows (one column per volume), the gradient directions. Alternatively a full gradient table, i.e. a numeric matrix with 4 columns named or ordered 'x', 'y', 'z' and 'b', as returned by `read.dti.grad`. If a full table is given, the `bval` parameter must be left at `NULL`.
`bval`	numeric vector, the b-value of each volume, in s/mm^2. Ignored if `bvec` is a full gradient table.
`layout`	character string, one of 'volumes' or 'components'. `volumes` (the default) writes one volume per line, which is the MRtrix layout, while `components` writes the transposed variant with 4 lines.

Value

NULL, invisibly. Called for the side effect of writing the file.

See Also

Other dti functions: `read.dti.bval()`, `read.dti.bvec()`, `read.dti.grad()`, `read.dti.gradients()`, `write.dti.bval()`, `write.dti.bvec()`

`write.dti.tck` *Write tracks to a file in MRtrix TCK format.*

Description

Writes streamlines in the TCK format described at `https://mrtrix.readthedocs.io/en/latest/getting_started/imag`
The format stores a triplet of NaN values between streamlines and a triplet of Inf values at the end.
The output is read by MRtrix and by `read.dti.tck`.

Usage

```
write.dti.tck(
  tracts,
  filepath,
  datatype = "Float32LE",
  gzip = NULL,
  header = list()
)
```

Arguments

`tracts`	the tracks to write, either an `fs.tracts` instance or a list of numeric matrices with 3 columns, see `write.dti.trk`.
`filepath`	character string, the path of the file to write.
`datatype`	character string, one of 'Float32LE' (the default, and what MRtrix writes), 'Float32BE', 'Float64LE' or 'Float64BE'.
`gzip`	logical or NULL, whether to gzip-compress the output. If `NULL` (the default), the file is compressed when the file name ends in '.gz'.
`header`	named list of additional header entries to store in the file, e.g., the header of the file the tracks were read from. The entries 'id', 'datatype', 'count', 'file' and 'derived' are always computed by the readers and cannot be set.

Value

the file path, invisibly.

Note

Tracts without any point cannot be represented in the TCK format: they are written as a bare delimiter, which every reader (including this package and 'nibabel') drops again, so the file reads back with fewer tracts than it was written from. Writing such a file raises a warning. `write.dti.trk` preserves empty tracts.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck", max_tracks = 1000);
write.dti.tck(tck$tracks, "first_1000.tck", header = tck$header);

# Round trip through a compressed file:
write.dti.tck(tck$tracks, "copy.tck.gz");

## End(Not run)
```

`write.dti.trk` *Write tracks to a file in TrackVis TRK format.*

Description

Writes streamlines in the TRK format described at `http://trackvis.org/docs/?subsect=fileformat`. The output is read by TrackVis, by DSI Studio and by `read.dti.trk`.

Usage

```
write.dti.trk(
  tracts,
  filepath,
  header = NULL,
  coords_space = NULL,
  endian = "little",
  gzip = NULL
)
```

Arguments

`tracts`	the tracks to write, either an `fs.tracts` instance as returned in the `tracks` entry of `read.dti.trk`, or a list of numeric matrices with 3 columns, or a single such matrix. Per-point scalars and per-track properties are written if the input is an `fs.tracts` instance that has them.
`filepath`	character string, the path of the file to write.
`header`	named list or NULL, the header of the file the tracks were read from, whose metadata (voxel sizes, dimensions, the voxel-to-RAS matrix, the voxel order, and the scalars and properties of the file) is reused. If `NULL`, a minimal header is written that describes coordinates in 1 mm isotropic voxels with an identity voxel-to-RAS matrix, which is the case that needs no coordinate correction at all.
`coords_space`	character string or NULL, the coordinate system the tracks are in, either 'native' (the coordinates are written as they are) or 'ras' (the coordinates are RAS+ mm and are transformed back to the space used in the file, which is what 'nibabel'

does when saving). If NULL (the default), the value stored in header$coords_space is used when present, and 'native' otherwise. This makes a read-write round trip with coords = 'ras' work without any further arguments.

endian — character string, 'little' (the default) or 'big'. TrackVis writes little endian files, big endian support is for reading files written on old big endian systems.

gzip — logical or NULL, whether to gzip-compress the output. If NULL (the default), the file is compressed when the file name ends in '.gz'. Note that the TrackVis tools and MRtrix do not read compressed track files, so this is useful for archiving and for passing the file back to read.dti.trk, but not for exchanging it with other software.

Value

the file path, invisibly.

Examples

```
## Not run:
trk <- read.dti.trk("brain.trk");
write.dti.trk(trk$tracks, "copy.trk", header = trk$header);

# Write RAS coordinates back to a file that uses voxelmm space:
trk_ras <- read.dti.trk("brain.trk", coords = "ras");
write.dti.trk(trk_ras$tracks, "copy.trk", header = trk_ras$header, coords_space = "ras");

## End(Not run)
```

---

write.dti.tsf *Write per-point track values to a file in MRtrix TSF format.*

Description

The TSF format stores one scalar value per point of a streamline, e.g., the fractional anisotropy, the distance along the track or a value sampled from an image at the point coordinates. It is the companion format of the TCK format: a TSF file contains no coordinates and no track boundaries, it is just a stream of values that has to be read together with the tractogram it describes. The number of values per track is therefore required to write the file, and a TSF file without the matching TCK file is meaningless to every reader (MRtrix checks this, see the note below).

Usage

```
write.dti.tsf(
  tracts,
  filepath,
  lengths = NULL,
  datatype = "Float32LE",
  gzip = NULL,
```

```
  header = list()
)
```

Arguments

tracts	the values to write. This can be an fs.tracts instance whose scalars entry holds a single column of values (as returned by read.dti.trk for a file with one scalar, see fs.tracts to construct one), the scalars entry of the result of read.dti.tsf (so that a scalar file can be read and written back), a list of numeric vectors (one per track), or a single numeric vector of all values concatenated (in which case lengths is required).
filepath	character string, the path of the file to write.
lengths	integer vector or NULL, the number of values per track. This is ignored unless tracts is a plain vector, and has to be given in that case.
datatype	character string, one of 'Float32LE' (the default, and what MRtrix writes), 'Float32BE', 'Float64LE' or 'Float64BE'.
gzip	logical or NULL, whether to gzip-compress the output. If NULL (the default), the file is compressed when the file name ends in '.gz'.
header	named list of additional header entries to store in the file, e.g., the header of the file the tracks were read from. The entries 'id', 'datatype', 'count', 'file' and 'derived' are always computed by the readers and cannot be set.

Value

the file path, invisibly.

Note

The TSF format stores a NaN value after every track, and unlike the TCK format it has no Inf terminator: the reader relies on the delimiters to split the value stream into tracks. A file whose values are not delimited exactly like the tracks of the tractogram can therefore not be detected as broken by this package, but MRtrix reports the mismatch of the track counts when the file is used (e.g., in tcksample or tsfvalidate).

See Also

read.dti.tsf, write.dti.tck

Examples

```
# A TSF file stores one value per point. Since the format contains no track
# boundaries, the track lengths have to be provided:
tsff <- tempfile(fileext = ".tsf");
values_by_track <- list(c(0.1, 0.2, 0.3), c(0.4, 0.5));
write.dti.tsf(values_by_track, tsff);
read.dti.tsf(tsff)$scalars$scalar_list;

# The same file can be written from one vector of values and the lengths:
write.dti.tsf(c(0.1, 0.2, 0.3, 0.4, 0.5), tsff, lengths = c(3L, 2L));
```

```
## Not run:
# Read the values of a track scalar file, modify them and write them back:
tsf <- read.dti.tsf("brain.tsf");
tsf$scalars$merged <- tsf$scalars$merged * 2;
write.dti.tsf(tsf$scalars, "brain_doubled.tsf");

# Sample an image along the tracks of a tractogram and store the result. The
# values of a TRK file that has one scalar are accepted as they are:
trk <- read.dti.trk("brain.trk");
write.dti.tsf(trk$tracks, "brain.trk.tsf");

## End(Not run)
```

`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: `atlas.from.lut.and.csv()`, `colortable.from.annot()`, `read.fs.annot()`, `read.fs.colortable()`, `write.atlas.to.lut.and.csv()`, `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: atlas.from.lut.and.csv(), colortable.from.annot(), read.fs.annot(), read.fs.colortable(), write.atlas.to.lut.and.csv(), 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: atlas.from.lut.and.csv(), colortable.from.annot(), read.fs.annot(), read.fs.colortable(), write.atlas.to.lut.and.csv(), write.fs.annot(), write.fs.annot.gii()

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

---

write.fs.connectome.cifti

Write a CIFTI-2 connectome file.

Description

Write a connectome matrix to a CIFTI-2 file: a dense connectome (.dconn), a parcellated connectome (.pconn), or one of the mixed types (.pdconn, .dpconn) whose two dimensions hold brainordinates and parcels. The file type follows from the axes (and has to match the file name, see write.cifti), and the mapping of the two dimensions has to be given: it cannot be derived from the matrix, and guessing it (e.g. from the size of the matrix) would silently produce a file whose rows and columns describe the wrong brain regions.

Use template for the normal case, i.e. to write a connectome of the brainordinates or parcels of an existing file: this is how a .dconn of a subject gets the mapping of that subject, and how a .pconn gets the parcels of the .ptseries it was computed from. A template with a single brainordinate dimension (a .dscalar, .dtseries or .dlabel, i.e. the files that actually exist for a subject) is accepted as well: its mapping is then used for both dimensions of the connectome, which is what those files describe. Use axes if the mapping has to be built, e.g. a parcels axis from annotations (see cifti.axis.parcels.from.annot).

The complete matrix is written: a connectome file stores both halves of the matrix and its diagonal, so the symmetry of the matrix is not exploited anywhere, and the matrix of a .dconn can be huge (33 GB for the 91,282 grayordinates of an HCP subject, which is more than any implementation can hold in memory). Writing such a file is not supported yet; a connectome of a reduced set of brainordinates can be written.

Usage

```
write.fs.connectome.cifti(
  filepath,
  data,
  template = NULL,
  axes = NULL,
  metadata = NULL
)
```

Arguments

filepath	character string, the path of the file to write. The name should be one of .dconn.nii, .pconn.nii, .pdconn.nii or .dpconn.nii (the file type is derived from the axes, and the name has to agree with it).
data	numeric matrix, the connectome: the first matrix dimension (the rows) is CIFTI matrix dimension 0, which the axes describe. An fs.connectome instance (see read.fs.connectome.cifti) is accepted as well, in which case its data and its header are used.
template	character string (the path of a CIFTI-2 file), an fs.cifti or an fs.cifti.data object, or NULL. The axes of this file are used; a file with a single brainordinate dimension has that mapping used for both dimensions of the connectome.
axes	list of two axes (see cifti.header.from.axes) or a single axis that is used for both dimensions, or NULL. Use this instead of template to write a file with a mapping that is not in a file, e.g. a parcels axis built with cifti.axis.parcels.from.annot.
metadata	named character vector or named list, or NULL, the matrix metadata, see write.cifti. The default keeps the metadata of the template file.

Value

the axes that were written, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
pconn_file <- system.file("extdata", "cifti", "tiny.pconn.nii", package = "freesurferformats")
conn <- read.fs.connectome.cifti(pconn_file)
out_file <- file.path(tempdir(), "written.pconn.nii")
# The values are squared, the mapping is the one of the original file:
```

```
write.fs.connectome.cifti(out_file, conn$data^2, template = pconn_file)
read.fs.connectome.cifti(out_file)$data[1:2, 1:2]


## Not run:
# The grayordinates of a subject are the same in all its dense files, so a dtseries
# can define the mapping of the dconn:
write.fs.connectome.cifti("sub-01_dconn.nii", connectivity_matrix,
                          template = "sub-01_task-rest_dtseries.nii")

## End(Not run)
```

`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()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph()`, `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()`, `write.fs.weight.asc()`

`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()`, `read.fs.label.gii()`, `read.fs.label.native()`

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.mgh(), write.fs.morph(), 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(), write.fs.weight.asc()

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()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.morph()`, `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()`, `write.fs.weight.asc()`

---

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").

Note

A file name that is one of the standard CIFTI-2 names (e.g. `.dscalar.nii` or `.dlabel.nii`) is an error: such a file has to contain the CIFTI XML metadata, and a NIFTI file with that name is read as a CIFTI file (and refused) by this package and by other software. Use `write.fs.morph.cifti` to write morphometry data to a CIFTI-2 file, or `write.cifti` for the other CIFTI-2 types.

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.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.weight()`, `write.fs.weight.asc()`

---

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()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.ni2()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.weight()`, `write.fs.weight.asc()`

`write.fs.morph.cifti` *Write morphometry data to a CIFTI-2* `.dscalar` *file.*

Description

Write a per-vertex data vector (or several of them, as the maps of the file) to a CIFTI-2 dense scalar file. This is the inverse of `read.fs.morph.cifti`: the data are given for the *complete* surface, and the vertices that the file contains are selected with the mapping of a template file (which is what a grayordinates file needs, since it leaves out the medial wall).

Usage

```
write.fs.morph.cifti(
  filepath,
  data,
  template = NULL,
  structure = NULL,
  map_names = NULL,
  metadata = NULL
)
```

Arguments

`filepath`	character string, the path of the file to write. The name should end with `.dscalar.nii`.
`data`	numeric vector or matrix or named list, the per-vertex data: • a vector: one value per vertex of the complete surface of one structure, • a matrix: one row per vertex of the complete surface, one column per map, • a named list (e.g. `list(lh = ..., rh = ...)`): a vector or matrix as above per structure. The vertex order is the order of the surface mesh, which is also the order in which `read.fs.morph.cifti` returns the data.
`template`	character string (the path of a CIFTI-2 file), an `fs.cifti` or an `fs.cifti.data` object, or `NULL`. The grayordinate mapping of this file is used, so pass the file the data comes from (e.g. an HCP file of the same subject): the mapping of such a file cannot be invented. Without a template, the file covers *all* vertices of the surfaces, and `structure` has to be given.
`structure`	character string or `NULL`, the brain structure the data belongs to (e.g. `'lh'`), needed if `data` is not a named list and there is no template to take the structures from.
`map_names`	character vector or `NULL`, the names of the maps. The number of names has to match the number of maps in the data. Without this, the map names of the template file are kept if it has as many maps as the data.
`metadata`	named character vector or `NULL`, the matrix metadata, see `write.cifti`. The default keeps the metadata of the template file.

Value

the axes that were written, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
template <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
out_file <- file.path(tempdir(), "written.dscalar.nii")
data <- read.fs.morph.cifti(template, "lh")
data[2] <- 555 # change one value
write.fs.morph.cifti(out_file, data, template = template, structure = "lh")
read.fs.morph.cifti(out_file, "lh")[1:3]
```

---

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), write.fs.morph.asc(), write.fs.morph.ni1(), write.fs.morph.ni2(), write.fs.morph.smp(), write.fs.morph.txt(), write.fs.weight(), write.fs.weight.asc()

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), write.fs.morph.asc(), write.fs.morph.gii(), write.fs.morph.ni2(), write.fs.morph.smp(), write.fs.morph.txt(), write.fs.weight(), write.fs.weight.asc()

Other nifti1 writers: nii1header.for.mgh(), 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()`, `read.fs.morph.gii()`, `read.fs.volume()`, `read.fs.weight()`, `write.fs.curv()`, `write.fs.label.gii()`, `write.fs.mgh()`, `write.fs.morph()`, `write.fs.morph.asc()`, `write.fs.morph.gii()`, `write.fs.morph.ni1()`, `write.fs.morph.smp()`, `write.fs.morph.txt()`, `write.fs.weight()`, `write.fs.weight.asc()`

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), write.fs.morph.asc(), write.fs.morph.gii(), write.fs.morph.ni1(), write.fs.morph.ni2(), write.fs.morph.txt(), write.fs.weight(), write.fs.weight.asc()

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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), write.fs.morph.asc(), write.fs.morph.gii(), write.fs.morph.ni1(), write.fs.morph.ni2(), write.fs.morph.smp(), write.fs.weight(), write.fs.weight.asc()

---

write.fs.parcellated.cifti

Write a CIFTI-2 parcellated map or series file.

Description

Write data that has one value (or one time series) per parcel to a parcellated CIFTI-2 file: a .pscalar (scalar maps, e.g. a mean value per region) or a .ptseries (a time series per region). The parcellated dimension of these file types is matrix dimension 1, and the order of the parcels is the order of the parcels axis, which comes from a template file or from a parcels axis that you build (see cifti.axis.parcels.from.annot for a parcellation of this package, e.g. the annotations of a FreeSurfer atlas).

The file type is derived from the file name: a name ending in .pscalar.nii gets a scalars dimension (the map names come from map_names), one ending in .ptseries.nii a series dimension (described by start, step and unit). Use write.cifti directly for the other parcellated types (.pconn, the mixed connectome types) or for a file with a non-standard name.

Usage

```
write.fs.parcellated.cifti(
  filepath,
  data,
  template = NULL,
  axes = NULL,
  map_names = NULL,
  start = 0,
  step = 1,
  unit = "SECOND",
  metadata = NULL
)
```

Arguments

filepath	character string, the path of the file to write, ending in .pscalar.nii or .ptseries.nii.
data	numeric vector or matrix, the data: a vector with one value per parcel (a .pscalar with a single map), or a matrix with one row per map (for a .pscalar) or series point (for a .ptseries) and one column per parcel. The first matrix dimension is CIFTI matrix dimension 0, like for every other reader and writer of this package.

template	character string (the path of a CIFTI-2 file), an fs.cifti or an fs.cifti.data object, or NULL. The parcels of this file are used, so pass the parcellated file the data belongs to (e.g. a .ptseries or .pconn of the same subject).
axes	list of two axes or a single parcels axis (see cifti.axis.parcels), or NULL. Use this instead of template to write data for a parcellation that is not in a CIFTI-2 file, e.g. one built from annotations with cifti.axis.parcels.from.annot.
map_names	character vector or NULL, the names of the maps of a .pscalar. The number of names has to match the number of rows of the data. Without this, the map names of the template are kept if it is a .pscalar with as many maps as the data.
start	numeric, the value of the first series point of a .ptseries, see cifti.axis.series.
step	numeric, the difference between consecutive series points.
unit	character string, the unit of the series, one of 'SECOND', 'HERTZ', 'METER' or 'RADIAN'.
metadata	named character vector or named list, or NULL, the matrix metadata, see write.cifti. The default keeps the metadata of the template file.

Value

the axes that were written, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellation.cifti(), write.fs.series.cifti()

Examples

```
template <- system.file("extdata", "cifti", "tiny.ptseries.nii", package = "freesurferformats")
data <- matrix(seq_len(3 * 4), nrow = 4L) # 4 series points, 3 parcels
out_file <- file.path(tempdir(), "written.ptseries.nii")
write.fs.parcellated.cifti(out_file, data, template = template, step = 0.5)
dim(read.cifti(out_file)$data)
```

`write.fs.parcellation.cifti` *Write a parcellation to a CIFTI-2* `.dlabel` *file.*

Description

Write per-vertex label keys (one per vertex of the complete surface, per structure) to a CIFTI-2 dense label file. This is the inverse of `read.fs.parcellation.cifti`.

Usage

```
write.fs.parcellation.cifti(
  filepath,
  data,
  template = NULL,
  structure = NULL,
  label_table = NULL,
  map_name = NULL,
  metadata = NULL
)
```

Arguments

`filepath`	character string, the path of the file to write. The name should end with `.dscalar.nii`.
`data`	integer vector, matrix or named list, the label keys per vertex, see `data` in `write.fs.morph.cifti`. The values have to be the label keys of the label table (they are written as integers, and a key of 0 usually means unknown).
`template`	character string (the path of a CIFTI-2 file), an `fs.cifti` or an `fs.cifti.data` object, or `NULL`. The grayordinate mapping of this file is used, so pass the file the data comes from (e.g. an HCP file of the same subject): the mapping of such a file cannot be invented. Without a template, the file covers *all* vertices of the surfaces, and `structure` has to be given.
`structure`	character string or `NULL`, the brain structure the data belongs to (e.g. `'lh'`), needed if `data` is not a named list and there is no template to take the structures from.
`label_table`	data.frame or `NULL`, the label table, see `cifti.axis.labels`: the columns 'key', 'red', 'green', 'blue', 'alpha' (in the range 0 to 1) and 'label'. The colortable of an `fs.annot` instance (see `read.fs.annot`) is accepted as well, its colors are then divided by 255. Without a label table the file still stores the keys, but nothing in it explains what they mean (Connectome Workbench writes one, and its label files rely on it).
`map_name`	character string or `NULL`, the name of the label map.
`metadata`	named character vector or `NULL`, the matrix metadata, see `write.cifti`. The default keeps the metadata of the template file.

Value

the axes that were written, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.series.cifti()

Examples

```
template <- system.file("extdata", "cifti", "tiny.dlabel.nii", package = "freesurferformats")
keys <- read.fs.parcellation.cifti(template, "lh")
label_table <- read.fs.parcellation.cifti(template, "lh", with_label_table = TRUE)$label_table
out_file <- file.path(tempdir(), "written.dlabel.nii")
write.fs.parcellation.cifti(out_file, keys, template = template, structure = "lh",
                            label_table = label_table)
table(read.fs.parcellation.cifti(out_file, "lh"))
```

---

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

`write.fs.series.cifti` *Write time series data to a CIFTI-2* `.dtseries` *file.*

Description

Write a per-vertex time series (or several structures at once) to a CIFTI-2 dense series file. This is the inverse of `read.fs.series.cifti`.

Usage

```
write.fs.series.cifti(
  filepath,
  data,
  template = NULL,
  structure = NULL,
  start = 0,
  step = 1,
  unit = "SECOND",
  metadata = NULL
)
```

Arguments

`filepath`	character string, the path of the file to write. The name should end with `.dscalar.nii`.
`data`	numeric matrix or named list, the data: a matrix with one row per vertex of the complete surface and one column per series point (time point), or a named list of such matrices per structure.
`template`	character string (the path of a CIFTI-2 file), an `fs.cifti` or an `fs.cifti.data` object, or `NULL`. The grayordinate mapping of this file is used, so pass the file the data comes from (e.g. an HCP file of the same subject): the mapping of such a file cannot be invented. Without a template, the file covers *all* vertices of the surfaces, and `structure` has to be given.
`structure`	character string or `NULL`, the brain structure the data belongs to (e.g. `'lh'`), needed if `data` is not a named list and there is no template to take the structures from.
`start`	numeric, the value of the first series point, see `cifti.axis.series`.
`step`	numeric, the difference between consecutive series points.
`unit`	character string, the unit of the series, one of 'SECOND', 'HERTZ', 'METER' or 'RADIAN'.
`metadata`	named character vector or `NULL`, the matrix metadata, see `write.cifti`. The default keeps the metadata of the template file.

Value

the axes that were written, invisibly.

See Also

Other cifti functions: cifti.axis.brain.models(), cifti.axis.from.template(), cifti.axis.labels(), cifti.axis.parcels(), cifti.axis.parcels.from.annot(), cifti.axis.scalars(), cifti.axis.series(), cifti.brain.model.surface(), cifti.brain.model.volume(), cifti.dim.labels(), cifti.file.type.for.axes(), cifti.grayordinates(), cifti.header.from.axes(), cifti.label.table(), cifti.parcel(), cifti.parcels(), cifti.series.info(), cifti.structure.data(), cifti.structures(), cifti.volume(), print.fs.cifti(), print.fs.cifti.data(), print.fs.connectome(), read.cifti(), read.cifti.header(), read.cifti.rows(), read.fs.connectome.cifti(), write.cifti(), write.fs.connectome.cifti(), write.fs.morph.cifti(), write.fs.parcellated.cifti(), write.fs.parcellation.cifti()

Examples

```
template <- system.file("extdata", "cifti", "tiny.dscalar.nii", package = "freesurferformats")
series <- matrix(seq_len(10 * 3), nrow = 10L) # 10 vertices (lh), 3 time points
out_file <- file.path(tempdir(), "written.dtseries.nii")
write.fs.series.cifti(out_file, series, template = template, structure = "lh", step = 0.72)
dim(read.fs.series.cifti(out_file, "lh"))
```

---

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, 'stl' for the stereolithography (STL) format used for 3D printing, 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. A path ending in '.stla' is interpreted as the ASCII variant of the STL format and one ending in '.stlb' or '.stl' as the binary variant. 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(), 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_nisurface(), read_nisurfacefile(), 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(), write.fs.surface.off.ply2(), write.fs.surface.ply(), write.fs.surface.ply2(), write.fs.surface.stl(), write.fs.surface.vtk()

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(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.gii(), write.fs.surface.mz3(), write.fs.surface.vtk()

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(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.mz3(), write.fs.surface.vtk()

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()`, `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_nisurface()`, `read_nisurfacefile()`, `write.fs.surface()`, `write.fs.surface.asc()`, `write.fs.surface.byu()`, `write.fs.surface.gii()`, `write.fs.surface.vtk()`

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()`, `write.fs.surface.off()`, `write.fs.surface.off.ply2()`, `write.fs.surface.ply()`, `write.fs.surface.ply2()`, `write.fs.surface.stl()`, `write.fs.surface.vtk()`

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(), write.fs.surface.obj(), write.fs.surface.off.ply2(), write.fs.surface.ply(), write.fs.surface.ply2(), write.fs.surface.stl(), write.fs.surface.vtk()

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(), write.fs.surface.obj(), write.fs.surface.off(), write.fs.surface.off.ply2(), write.fs.surface.ply2(), write.fs.surface.stl(), write.fs.surface.vtk()

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()`, `write.fs.surface.obj()`, `write.fs.surface.off()`, `write.fs.surface.off.ply2()`, `write.fs.surface.ply()`, `write.fs.surface.stl()`, `write.fs.surface.vtk()`

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.stl *Write mesh to file in STL format (ASCII or binary).*

Description

The STL format (stereolithography, the format used for 3D printing) stores a triangular mesh as a list of triangles, each with its vertex coordinates repeated and with a normal vector, instead of storing a vertex list and indices into it. Both the ASCII and the binary version of the format are written by this function, the binary one being the default since it is much smaller and it is what most software uses. The resulting files can be read back with `read.fs.surface.stl` and are accepted by mesh viewers and slicers.

Usage

```
write.fs.surface.stl(
  filepath,
  vertex_coords,
  faces,
  ascii = FALSE,
  solid_name = "mesh"
)
```

Arguments

filepath	character string, the path of the file to write.
vertex_coords	n x 3 matrix of doubles, the vertex coordinates.
faces	n x 3 matrix of integers, the vertex indices of the triangles. The STL format has no support for polygons with more than 3 vertices, so a quad mesh has to be converted first with faces.quad.to.tris.
ascii	logical, whether to write the ASCII version of the format. The default is the binary version, which is smaller by a factor of about 5 and which is what most mesh processing software writes. Use the ASCII version if the file has to be readable by humans or by software that supports only the ASCII variant.
solid_name	character string, the name of the mesh. Only used in the ASCII version, where the format requires the name in the first and the last line of the file.

Value

character string, the format that was written: 'tris'.

Note

The normals of the triangles are computed from the vertex coordinates (the STL format stores them, but no reader has to trust them). A degenerate triangle, i.e., one whose vertices are collinear or identical, has no normal, so a zero vector is written for it.

An indexed mesh is stored as a polygon soup in an STL file: every triangle repeats the coordinates of its vertices. Reading such a file back with read.fs.surface.stl merges the repeated vertices again (using the digits precision of that function), so a round trip through an STL file preserves the geometry of the mesh, but not the order or the count of the vertices in the vertex list.

See Also

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

Examples

```
## Not run:
# Write a mesh as binary and as ASCII STL:
mesh <- read.fs.surface(system.file("extdata", "cube.stl", package = "freesurferformats"));
write.fs.surface.stl(tempfile(fileext = ".stl"), mesh$vertices, mesh$faces);
write.fs.surface.stl(tempfile(fileext = ".stl"), mesh$vertices, mesh$faces, ascii = TRUE);

# The file format is also chosen by the file name when using the generic
# writer:
write.fs.surface(tempfile(fileext = ".stl"), mesh$vertices, mesh$faces);

## End(Not run)
```

`write.fs.surface.vtk` *Write mesh to file in VTK legacy format*

Description

The VTK legacy format is the plain text/binary format that is supported by all versions of the VTK library; the XML based VTK format (.vtp) is not the same thing. Two variants of the file layout exist and are both still written by software in use today: the old layout introduced in VTK 4.2, and the layout that VTK produces since version 5.1 (released 2015). The parameter 'version' selects which one to write.

Usage

```
write.fs.surface.vtk(
  filepath,
  vertex_coords,
  faces,
  version = 4.2,
  binary = FALSE
)
```

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 VTK style, i.e., all indices will have 1 substracted, so that the index of the first vertex will be zero.
`version`	double, the VTK version whose file layout to write. Either 4.2 (the default) or 5.1. Version 4.2 writes the cell array layout that every VTK version can read, version 5.1 writes the `OFFSETS/CONNECTIVITY` layout that VTK itself has been producing since 2015. Only change this if you know that the software you hand the file to requires the newer layout.
`binary`	logical, whether to write the data in binary form instead of the ASCII text form. Binary files are much smaller and much faster to read and write, but they are not human readable. Defaults to FALSE.

Value

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

Note

Binary data in the VTK legacy format is always big endian, the format has no way of expressing a different byte order. The vertex coordinates are written as single precision (4 byte) floats in both encodings, which is what VTK itself does.

See Also

Other mesh functions: faces.quad.to.tris(), read.fs.surface(), 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_nisurface(), read_nisurfacefile(), write.fs.surface(), write.fs.surface.asc(), write.fs.surface.byu(), write.fs.surface.gii(), write.fs.surface.mz3()

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

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)
# Or as a binary file using the layout of VTK 5.1:
write.fs.surface.vtk(tempfile(fileext = ".vtk"), mesh$vertices, mesh$faces,
  version = 5.1, binary = TRUE
)

## End(Not run)
```

---

write.fs.transform *Write a transformation matrix to a file.*

Description

Save an fs.transform instance in one of the supported transformation file formats.

A transformation stores its matrix together with the coordinate spaces it maps between, and the formats disagree about which spaces they can express. A transformation is only written if the format can represent it exactly, because a silent conversion would change the meaning of the matrix: FSL matrix files, for example, store voxel-to-voxel matrices, so a transformation in world coordinates must be converted first with transform2voxel. A format that cannot express the transformation at all is an error, not a warning.

Usage

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

Arguments

tf	an fs.transform instance, the transformation to write.
filepath	character string, the full path of the file to write.
format	character string, the file format, one of 'auto' (guess from the file extension), 'fslmat' (an FSL/FLIRT matrix file, i.e. a plain text 4x4 matrix as written by FSL's flirt -omat), 'lta' (write.fs.transform.lta), 'dat' (write.fs.transform.dat), 'xfm' (write.fs.transform.xfm) or 'itk' (write.fs.transform.itk).

Value

the fs.transform instance tf, invisibly.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform.dat(), write.fs.transform.fslm write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
tf <- read.fs.transform(system.file("extdata", "talairach.lta",
  package = "freesurferformats", mustWork = TRUE
))
out_file <- tempfile(fileext = ".mat")
# An LTA of type 0 is a voxel-to-voxel transformation, so it can be written as an FSL matrix.
write.fs.transform(tf, out_file, format = "fslmat")
read.fs.transform(out_file)$matrix
unlink(out_file)

# The same transformation can be written in the FreeSurfer formats of the spaces it maps between.
out_file <- tempfile(fileext = ".lta")
write.fs.transform(tf, out_file)
unlink(out_file)
```

write.fs.transform.dat *Write a tkregister dat file.*

Description

Write a transformation in the FreeSurfer tkregister format (`register.dat`), the format that `mri_vol2vol --reg`, `tkregister2` and `bbregister` use. Such a matrix maps the voxel coordinates of the movable volume (the source) to RAS coordinates in the tkregister frame of the target volume, see `mghheader.vox2ras.tkreg`, so a transformation can only be written if this is what it maps.

Usage

```
write.fs.transform.dat(
  tf,
  filepath,
  subject = NULL,
  in_plane_resolution = NULL
)
```

Arguments

`tf`	an `fs.transform` instance whose matrix maps voxel coordinates to RAS coordinates.
`filepath`	character string, the full path of the file to write.
`subject`	`NULL` or character string, the subject identifier to store in the first line of the file. This is metadata for the tools that read the file and does not influence the transformation. If `NULL` and `tf` has a `subject` field (as read by `read.fs.transform.dat`), that value is used.
`in_plane_resolution`	`NULL` or numerical vector of length 2, the in-plane and between-plane resolution of the movable volume in millimeters, stored in the second and third line of the file. These are metadata that describe the acquisition and do not influence the transformation. If `NULL` and `tf` has the corresponding fields, they are used, otherwise they are derived from the geometry of the source volume, and set to 1 if that is not available either.

Value

the `fs.transform` instance `tf`, invisibly.

See Also

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

read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
dat_file <- system.file("extdata", "register.dat", package = "freesurferformats", mustWork = TRUE)
tf <- read.fs.transform(dat_file)
out_file <- tempfile(fileext = ".dat")
write.fs.transform.dat(tf, out_file)
max(abs(read.fs.transform(out_file)$matrix - tf$matrix)) # 0
unlink(out_file)
```

---

write.fs.transform.fslmat

Write a transformation matrix in FSL format.

Description

Write a 4x4 matrix as an FSL matrix file, i.e. as the plain text file that FSL's `flirt` writes with the `-omat` option and that FSL, MRtrix3 and FreeSurfer read as the registration between two images. The matrix must map voxel coordinates to voxel coordinates (`space_in` and `space_out` are 'voxel'), because that is what an FSL matrix stores: it relates the voxel grid of the image given to `flirt -in` to the voxel grid of the image given to `flirt -ref`, and it does not record which images those were. Use `transform2voxel` to convert a transformation in world coordinates into one that can be written.

Usage

```
write.fs.transform.fslmat(tf, filepath)
```

Arguments

`tf`	an `fs.transform` instance whose matrix maps voxel coordinates to voxel coordinates.
`filepath`	character string, the full path of the file to write.

Value

the `fs.transform` instance `tf`, invisibly.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.itk(), write.fs.transform.lta(), write.fs.transform.xfm()

Examples

```
tf_file <- system.file("extdata", "talairach.lta", package = "freesurferformats", mustWork = TRUE)
out_file <- tempfile(fileext = ".mat")
write.fs.transform.fslmat(read.fs.transform(tf_file), out_file)
readLines(out_file)
unlink(out_file)
```

---

write.fs.transform.itk

Write an ITK text transform file.

Description

Write a transformation in the text file format of ITK, which the tools built on ITK read: 3D Slicer (which calls it the 'ITK Transform' format), ANTs (via `ConvertTransformFile`), SimpleITK, and the workflows that write their transformations with them. The file name extension is usually `.tfm` or `.txt`.

An ITK transform operates on the world coordinates of the images, which in ITK are left-posterior-superior, so only a transformation that maps LPS coordinates can be written. Use transform2lps to convert a transformation in RAS coordinates.

Usage

```
write.fs.transform.itk(tf, filepath)
```

Arguments

`tf`	an `fs.transform` instance whose matrix maps LPS coordinates to LPS coordinates.
`filepath`	character string, the full path of the file to write.

Value

the `fs.transform` instance `tf`, invisibly.

Note

The file that is written uses the class 'AffineTransform_double_3_3', i.e. the parameters are stored in double precision, and it states a center of rotation of zero, with the center folded into the translation. This is exactly the form that FreeSurfer's `lta_convert --outitk` writes, and the form that `lta_convert --initk` can read: it rejects the 'float' variant of the classes and ignores a non-zero center of rotation, see the note in `read.fs.transform.itk`.

See Also

Other header coordinate space: `invert.fs.transform()`, `is.fs.transform()`, `mghheader.is.ras.valid()`, `mghheader.ras2vox()`, `mghheader.ras2vox.tkreg()`, `mghheader.scanner2tkreg()`, `mghheader.tkreg2scanner()`, `mghheader.vox2ras()`, `mghheader.vox2ras.tkreg()`, `print.fs.transform()`, `read.fs.transform()`, `read.fs.transform.dat()`, `read.fs.transform.fslmat()`, `read.fs.transform.itk()`, `read.fs.transform.lta()`, `read.fs.transform.xfm()`, `sm0to1()`, `sm1to0()`, `summary.fs.transform()`, `transform2lps()`, `transform2ras()`, `transform2voxel()`, `transform2world()`, `write.fs.transform()`, `write.fs.transform.dat()`, `write.fs.transform.fslmat()`, `write.fs.transform.lta()`, `write.fs.transform.xfm()`

Examples

```
xfm_file <- system.file("extdata", "talairach.xfm", package = "freesurferformats", mustWork = TRUE)
out_file <- tempfile(fileext = ".tfm")
write.fs.transform.itk(transform2lps(read.fs.transform(xfm_file)), out_file)
readLines(out_file)
unlink(out_file)
```

---

write.fs.transform.lta

Write a FreeSurfer linear transform array (LTA) file.

Description

Write a transformation in the LTA format, which is the format FreeSurfer uses to exchange linear transformations and that is read by `mri_vol2vol`, `tkregister2`, `mri_register` and `lta_convert`.

The file states in its header whether the matrix operates on voxel indices (type 0, LINEAR_VOX_TO_VOX) or on RAS coordinates (type 1, LINEAR_RAS_TO_RAS), so the spaces of the transformation determine the type that is written. The geometry of the volumes is written from the `src` and `dst` descriptors, which makes the file self-contained: FreeSurfer can convert it to other spaces without being given the volumes again.

Usage

```
write.fs.transform.lta(tf, filepath)
```

Arguments

tf	an fs.transform instance whose matrix maps either voxel coordinates to voxel coordinates, or RAS coordinates to RAS coordinates.
filepath	character string, the full path of the file to write.

Value

the fs.transform instance tf, invisibly.

Note

The mean and sigma header entries of an LTA file describe the registration that produced the matrix and are not used to interpret it. They are taken from the header field of tf if it has them, and are otherwise set to the center of the source volume and to 10000.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.xfm()

Examples

```
lta_file <- system.file("extdata", "talairach.lta", package = "freesurferformats", mustWork = TRUE)
tf <- read.fs.transform(lta_file)
out_file <- tempfile(fileext = ".lta")
write.fs.transform.lta(tf, out_file)
max(abs(read.fs.transform(out_file)$matrix - tf$matrix)) # 0, the matrix survives the round trip
unlink(out_file)
```

---

write.fs.transform.xfm

Write an MNI transform (xfm) file.

Description

Write a transformation in the FreeSurfer xfm format, the format of talairach.xfm files, which store the transformation from the RAS space of a subject to the RAS space of an MNI or Talairach template. Only the first three rows of the matrix are stored, so the transformation must be affine, and it must map RAS coordinates to RAS coordinates.

Usage

```
write.fs.transform.xfm(tf, filepath, type = "Linear")
```

Arguments

tf	an fs.transform instance whose matrix maps RAS coordinates to RAS coordinates.
filepath	character string, the full path of the file to write.
type	character string, the value of the Transform_Type entry, 'Linear' by default.

Value

the fs.transform instance tf, invisibly.

See Also

Other header coordinate space: invert.fs.transform(), is.fs.transform(), mghheader.is.ras.valid(), mghheader.ras2vox(), mghheader.ras2vox.tkreg(), mghheader.scanner2tkreg(), mghheader.tkreg2scanner(), mghheader.vox2ras(), mghheader.vox2ras.tkreg(), print.fs.transform(), read.fs.transform(), read.fs.transform.dat(), read.fs.transform.fslmat(), read.fs.transform.itk(), read.fs.transform.lta(), read.fs.transform.xfm(), sm0to1(), sm1to0(), summary.fs.transform(), transform2lps(), transform2ras(), transform2voxel(), transform2world(), write.fs.transform(), write.fs.transform.dat(), write.fs.transform.fslmat(), write.fs.transform.itk(), write.fs.transform.lta()

Examples

```
xfm_file <- system.file("extdata", "talairach.xfm", package = "freesurferformats", mustWork = TRUE)
tf <- read.fs.transform(xfm_file)
out_file <- tempfile(fileext = ".xfm")
write.fs.transform.xfm(tf, out_file)
max(abs(read.fs.transform(out_file)$matrix - tf$matrix)) # 0
unlink(out_file)
```

---

write.fs.volume — *Write an fs.volume instance to a file in MGH, MGZ or NIFTI v1 format.*

Description

Write brain volume data to a file. The format is determined from the file extension of filepath.

Usage

```
write.fs.volume(filepath, fs_vol)
```

Arguments

filepath	string. Full path to the output file. The file extension determines the format: '.mgh' or '.mgz' for FreeSurfer MGH/MGZ format, '.nii' or '.nii.gz' for NIFTI v1 format, and '.hdr', '.img', '.hdr.gz' or '.img.gz' for a NIFTI v1 pair, i.e. a header file with a separate data file. A pair is written instead of a plain ANALYZE 7.5 file because ANALYZE cannot store the geometry of the volume, see write.analyze if you need real ANALYZE output.
fs_vol	an fs.volume instance, as returned by read.fs.volume with parameter with_header=TRUE.

Note

When writing NIFTI files, this function uses nii1header.for.mgh to compute a NIFTI v1 header from the MGH header information.

A file name that is one of the standard CIFTI-2 names (e.g. .dscalar.nii) is an error: such a file has to contain the CIFTI XML metadata, so a NIFTI file with that name is refused by this package and misread by other software. Use write.cifti for CIFTI-2 files.

Examples

```
## Not run:
mgh_file <- system.file("extdata", "brain.mgz",
  package = "freesurferformats",
  mustWork = TRUE
)
fs_vol <- read.fs.volume(mgh_file, with_header = TRUE)
write.fs.volume(tempfile(fileext = ".mgz"), fs_vol)
write.fs.volume(tempfile(fileext = ".nii.gz"), fs_vol)
write.fs.volume(tempfile(fileext = ".hdr"), fs_vol)

## 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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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.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(), read.fs.morph.gii(), read.fs.volume(), read.fs.weight(), write.fs.curv(), write.fs.label.gii(), write.fs.mgh(), write.fs.morph(), 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()

`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' for a single file NIFTI image, and '.hdr', '.img' (optionally followed by '.gz') for a NIFTI v1 pair, i.e. a header file with the voxel data in a separate '.img' file. Which of the two is written follows from the `magic` field of the header: 'n+1' means a single file, 'ni1' means a pair. See `ni1header.template`.
`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`.

Value

invisible named list with the entries `header` (the header that was written) and `data` (the data that was written, or `NULL` if no data was passed).

See Also

Other nifti1 writers: `nii1header.for.mgh()`, `write.fs.morph.ni1()`

Examples

```
# A single file NIFTI image (the default):
data <- array(1:24, dim = c(4, 3, 2))
nii_file <- tempfile(fileext = ".nii")
write.nifti1(nii_file, data)

# A NIFTI v1 pair, i.e. a header plus a separate data file. Ask for a pair
# header by passing 'pair = TRUE' (this is passed on to the header generator):
write.nifti1(tempfile(fileext = ".hdr"), data, pair = TRUE)
```

```
# The NIFTI v1 pair files can be read back with the volume reader:
hdr_file <- tempfile(fileext = ".hdr")
write.nifti1(hdr_file, data, pair = TRUE)
vol <- read.fs.volume(hdr_file, with_header = TRUE)
print(dim(vol$data))
```

`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, extensions = 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`.
`extensions`	optional list of NIFTI v2 header extensions to write between the header and the data, each created with `nifti2.extension`. If left at `NULL` and the 'niiheader' has a field named 'extensions', that field is used, so that a header read with `read.nifti2.header` can be written back to a new file without losing its extensions. The 'vox_offset' field of the header is adapted if it is too small to fit the extensions, and the data is written directly after them.

See Also

`read.nifti2.header`, `nifti2.get.extension`

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

[.fs.tracts *Subset an fs.tracts instance.*

Description

Subset an fs.tracts instance.

Usage

```
## S3 method for class 'fs.tracts'
x[i]
```

Arguments

x	an `fs.tracts` instance.
i	index vector (integer, numeric or logical), as usual in R.

Value

a new `fs.tracts` instance containing the selected tracts.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck");
first_ten <- tck$tracks[1:10];

## End(Not run)
```

`[[.fs.tracts` *Access a single tract of an fs.tracts instance.*

Description

For TCK data (see `read.dti.tck`), a tract is an n x 3 numeric matrix of coordinates. For TRK data (see `read.dti.trk`), a tract is a named list with the entries `coords` (n x 3 matrix), `num_points` (integer), `scalars` (n x n_scalars matrix or NULL) and `properties` (numeric vector or NULL).

This returns a copy of the requested tract, the data is stored in a single matrix internally.

Usage

```
## S3 method for class 'fs.tracts'
x[[i]]
```

Arguments

`x`	an `fs.tracts` instance.
`i`	positive integer, the index of the tract to retrieve.

Value

the tract, see the description.

Examples

```
## Not run:
tck <- read.dti.tck("brain.tck");
first_tract_coords <- tck$tracks[[1]];

## End(Not run)
```

Index

∗ **Euclidean distance util functions**
closest.vert.to.point, 35
vertex.euclid.dist, 172
vertexdists.to.point, 173
∗ **NIFTI helper functions**
nifti.datadim.from.dimfield, 74
nifti.datadim.to.dimfield, 75
∗ **atlas functions**
atlas.from.lut.and.csv, 11
colortable.from.annot, 36
read.fs.annot, 105
read.fs.colortable, 107
write.atlas.to.lut.and.csv, 175
write.fs.annot, 184
write.fs.annot.gii, 185
write.fs.colortable, 186
∗ **cifti functions**
cifti.axis.brain.models, 13
cifti.axis.from.template, 15
cifti.axis.labels, 16
cifti.axis.parcels, 17
cifti.axis.parcels.from.annot, 18
cifti.axis.scalars, 19
cifti.axis.series, 20
cifti.brain.model.surface, 21
cifti.brain.model.volume, 22
cifti.dim.labels, 23
cifti.file.type.for.axes, 24
cifti.grayordinates, 25
cifti.header.from.axes, 26
cifti.label.table, 27
cifti.parcel, 28
cifti.parcels, 29
cifti.series.info, 30
cifti.structure.data, 31
cifti.structures, 33
cifti.volume, 34
print.fs.cifti, 81
print.fs.cifti.data, 81
print.fs.connectome, 82
read.cifti, 90
read.cifti.header, 91
read.cifti.rows, 93
read.fs.connectome.cifti, 108
write.cifti, 176
write.fs.connectome.cifti, 187
write.fs.morph.cifti, 195
write.fs.parcellated.cifti, 200
write.fs.parcellation.cifti, 202
write.fs.series.cifti, 204
∗ **colorLUT functions**
colortable.from.annot, 36
read.fs.colortable, 107
write.fs.colortable, 186
∗ **dti functions**
read.dti.bval, 95
read.dti.bvec, 95
read.dti.grad, 96
read.dti.gradients, 97
write.dti.bval, 177
write.dti.bvec, 178
write.dti.grad, 179
∗ **gifti readers**
read.fs.annot.gii, 106
read.fs.label.gii, 112
read.fs.morph.gii, 119
read.fs.surface.gii, 130
∗ **gifti writers**
write.fs.annot.gii, 185
write.fs.label.gii, 191
write.fs.morph.gii, 196
write.fs.surface.gii, 209
∗ **header coordinate space**
invert.fs.transform, 53
is.fs.transform, 58
mghheader.is.ras.valid, 62
mghheader.ras2vox, 64
mghheader.ras2vox.tkreg, 65

mghheader.scanner2tkreg, 66
mghheader.tkreg2scanner, 67
mghheader.vox2ras, 68
mghheader.vox2ras.tkreg, 69
print.fs.transform, 85
read.fs.transform, 138
read.fs.transform.dat, 139
read.fs.transform.fslmat, 140
read.fs.transform.itk, 141
read.fs.transform.lta, 142
read.fs.transform.xfm, 143
sm0to1, 163
sm1to0, 164
summary.fs.transform, 165
transform2lps, 168
transform2ras, 169
transform2voxel, 170
transform2world, 171
write.fs.transform, 219
write.fs.transform.dat, 221
write.fs.transform.fslmat, 222
write.fs.transform.itk, 223
write.fs.transform.lta, 224
write.fs.transform.xfm, 225

∗ **label functions**
read.fs.label, 111
read.fs.label.gii, 112
read.fs.label.native, 113
write.fs.label, 190

∗ **mesh export functions**
write.fs.surface, 205
write.fs.surface.obj, 212
write.fs.surface.off, 213
write.fs.surface.ply, 214
write.fs.surface.ply2, 215
write.fs.surface.stl, 216
write.fs.surface.vtk, 218

∗ **mesh functions**
faces.quad.to.tris, 41
read.fs.surface, 126
read.fs.surface.asc, 127
read.fs.surface.bvsrf, 128
read.fs.surface.geo, 129
read.fs.surface.gii, 130
read.fs.surface.ico, 131
read.fs.surface.obj, 132
read.fs.surface.off, 133
read.fs.surface.ply, 134
read.fs.surface.vtk, 136
read.mesh.brainvoyager, 153
read_nisurface, 159
read_nisurfacefile, 160
write.fs.surface, 205
write.fs.surface.asc, 206
write.fs.surface.byu, 208
write.fs.surface.gii, 209
write.fs.surface.mz3, 211
write.fs.surface.vtk, 218

∗ **morphometry functions**
fs.get.morph.file.ext.for.format, 44
fs.get.morph.file.format.from.filename, 44
read.fs.curv, 110
read.fs.mgh, 114
read.fs.morph, 116
read.fs.morph.gii, 119
read.fs.volume, 144
read.fs.weight, 152
write.fs.curv, 189
write.fs.label.gii, 191
write.fs.mgh, 192
write.fs.morph, 193
write.fs.morph.asc, 194
write.fs.morph.gii, 196
write.fs.morph.ni1, 197
write.fs.morph.ni2, 198
write.fs.morph.smp, 199
write.fs.morph.txt, 199
write.fs.weight, 227
write.fs.weight.asc, 228

∗ **nifti1 writers**
nii1header.for.mgh, 79
write.fs.morph.ni1, 197
write.nifti1, 229

∗ **nifti2 extensions**
nifti2.extension, 77
nifti2.extension.text, 78
nifti2.get.extension, 78

∗ **nifti2 writers**
write.fs.morph.ni2, 198
write.nifti2, 230

∗ **patch functions**
fs.patch, 45
read.fs.patch, 124
read.fs.patch.asc, 124

write.fs.patch, 203
∗ **volume export functions**
write.fs.volume, 226
∗ **volume math**
flip3D, 43
rotate3D, 163
[.fs.tracts, 232
[[.fs.tracts, 233

analyze.pair.files, *55*, *88*, *146*, *173*
analyzeheader.for.data, 7, *8*, *173*, *174*
analyzeheader.template, *7*, *8*, 8, *89*, *174*
annot.max.region.idx, 9
as.fs.tracts, 9
as.list.fs.tracts, 10
atlas.from.lut.and.csv, 11, *175*
atlas.from.lut.and.csv(), *36*, *106*, *108*, *175*, *185–187*

bvsmp, 12

cdata, 13
cifti.axis.brain.models, 13, *15*, *17*, *22*, *23*, *34*
cifti.axis.brain.models(), *15–17*, *19–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.from.template, 15, *26*, *34*, *176*
cifti.axis.from.template(), *14*, *16*, *17*, *19–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.labels, 16, *202*
cifti.axis.labels(), *14*, *15*, *17*, *19–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.parcels, 17, *19*, *29*, *34*, *201*
cifti.axis.parcels(), *14–16*, *19–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.parcels.from.annot, 18, *187*, *188*, *200*, *201*
cifti.axis.parcels.from.annot(), *14–17*, *20–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.scalars, *16*, 19
cifti.axis.scalars(), *14–17*, *19*, *21–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.axis.series, 20, *201*, *204*
cifti.axis.series(), *14–17*, *19*, *20*, *22–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.brain.model.surface, *14*, 21
cifti.brain.model.surface(), *14–17*, *19–21*, *23–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.brain.model.volume, *14*, 22
cifti.brain.model.volume(), *14–17*, *19–22*, *24–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.dim.labels, 23, *91*, *109*
cifti.dim.labels(), *14–17*, *19–23*, *25–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.file.type.for.axes, 24
cifti.file.type.for.axes(), *14–17*, *19–24*, *26–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.file.types, *24*
cifti.grayordinates, 25, *109*
cifti.grayordinates(), *14–17*, *19–25*, *27–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.header.from.axes, *14*, *16*, *17*, *20*, *21*, *24*, 26, *176*, *188*
cifti.header.from.axes(), *14–17*, *19–26*, *28–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.label.table, *16*, 27, *91*, *92*
cifti.label.table(), *14–17*, *19–27*, *29–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.map.for.dim, *90*
cifti.parcel, *17*, 28
cifti.parcel(), *14–17*, *19–28*, *30–33*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.parcels, 29, *91*, *109*
cifti.parcels(), *14–17*, *19–29*, *31*, *32*, *34*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.parse.xml, *26*
cifti.region.name.without.hemisphere, *18*
cifti.series.info, 30, *90*, *91*
cifti.series.info(), *14–17*, *19–30*, *32*, *34*, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*,

196, *201*, *203*, *205*
cifti.structure.canonical, *22, 23*, *32*
cifti.structure.data, 31, *90*
cifti.structure.data(), *14–17*, *19–31*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.structures, *25*, 33, *91*
cifti.structures(), *14–17*, *19–32*, *35*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
cifti.volume, *14*, *17*, *22*, 34
cifti.volume(), *14–17*, *19–32*, *34*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
closest.vert.to.point, 35
closest.vert.to.point(), *172, 173*
colortable.from.annot, 36
colortable.from.annot(), *12*, *106*, *108*, *175*, *185–187*
convert_endian, *51*
convert_intent, *51*

delete_all_opt_data, 37
doapply.transform.mtx, 37
download_opt_data, 38
drop, *88*
dti.track.bbox, 38
dti.track.count, 39, *40*
dti.track.iterator, 40

faces.quad.to.tris, 41, *217*
faces.quad.to.tris(), *127, 128*, *130, 131*, *133, 134*, *137*, *154*, *159, 160*, *206, 207*, *209–211*, *219*
faces.tris.to.quad, 42
flip2D, 43
flip3D, 43
flip3D(), *163*
fs.get.morph.file.ext.for.format, 44
fs.get.morph.file.ext.for.format(), *45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
fs.get.morph.file.format.from.filename, 44
fs.get.morph.file.format.from.filename(), *44*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
fs.patch, 45
fs.patch(), *124, 125*, *204*
fs.surface.to.tmesh3d, 46
fs.tracts, 46, *137*, *183*
fs.tracts.coords, 47
fs.tracts.lengths, 48, *137*
fs.transform, *138*
fsl.scaled.voxel.matrix, *171*

get_opt_data_filepath, 49
gifti_writer, 50, *51*
gifti_xml, *49, 50*, 50
gifti_xml_add_global_metadata, 52
gifti_xml_write, *50, 51*, 53
giftixml_add_labeltable_from_annot, 49

invert.fs.transform, 53
invert.fs.transform(), *58*, *63–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
is.analyze.file, 54, *89*
is.bvsmp, 55
is.fs.annot, 56
is.fs.label, 56
is.fs.surface, 57
is.fs.tracts, 57, *99*, *102*
is.fs.transform, 58
is.fs.transform(), *54*, *63–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
is.fs.volume, 59
is.mghheader, 59

length.fs.tracts, 60
list_opt_data, 60

mesh.face.normals, *136*
mghheader.centervoxelRAS.from.firstvoxelRAS, 61, *86*, *166, 167*
mghheader.crs.orientation, 61
mghheader.is.conformed, 62
mghheader.is.ras.valid, 62
mghheader.is.ras.valid(), *54*, *58*, *64–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.primary.slice.direction, 63
mghheader.ras2vox, 64, *68*
mghheader.ras2vox(), *54*, *58*, *63*, *65–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.ras2vox.tkreg, 65, *69*

mghheader.ras2vox.tkreg(), *54*, *58*, *63, 64*, *66, 67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.scanner2tkreg, 66, *67*
mghheader.scanner2tkreg(), *54*, *58*, *63–65*, *67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.tkreg2scanner, *66*, 67
mghheader.tkreg2scanner(), *54*, *58*, *63–66*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.update.from.vox2ras, 68
mghheader.vox2ras, *64*, 68, *148*, *163*
mghheader.vox2ras(), *54*, *58*, *63–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.vox2ras.tkreg, *65*, 69, *115*, *139*, *145*, *147*, *149*, *221*
mghheader.vox2ras.tkreg(), *54*, *58*, *63–67*, *69*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
mghheader.vox2vox, 70
mni152reg, 70

ni1header.for.data, *8*, 71, *72*, *229*
ni1header.template, *71*, 72, *79*, *229*
ni2header.for.data, *73*, 73, *74*, *230*
ni2header.template, *73*, 73, *78*
nifti.datadim.from.dimfield, 74
nifti.datadim.from.dimfield(), *75*
nifti.datadim.to.dimfield, 75
nifti.datadim.to.dimfield(), *74*
nifti.file.uses.fshack, 75
nifti.file.version, *76*, 76, *198*
nifti.header.check, 76
nifti2.extension, 77, *78, 79*, *156*, *230*
nifti2.extension(), *78, 79*
nifti2.extension.content(), *77–79*
nifti2.extension.size(), *77–79*
nifti2.extension.text, *77*, 78, *79*
nifti2.extension.text(), *77*, *79*
nifti2.get.extension, *77*, 78, *156*, *230*
nifti2.get.extension(), *77, 78*
nii1header.for.mgh, 79, *227*
nii1header.for.mgh(), *197*, *229*

print.dti.track.iterator, 80
print.fs.annot, 80
print.fs.cifti, 81
print.fs.cifti(), *14–17*, *19–32*, *34, 35*, *82, 83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
print.fs.cifti.data, 81
print.fs.cifti.data(), *14–17*, *19–32*, *34, 35*, *81*, *83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
print.fs.connectome, 82
print.fs.connectome(), *14–17*, *19–32*, *34, 35*, *81, 82*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
print.fs.label, 83
print.fs.patch, 83
print.fs.surface, 84
print.fs.tracts, 84
print.fs.transform, 85
print.fs.transform(), *54*, *58*, *63–67*, *69*, *139–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
print.fs.volume, 86

ras.to.surfaceras, 86
ras.to.talairachras, 87
read.analyze.data, 87, *89*, *148*
read.analyze.header, *8*, *88*, 88, *146–148*, *174*
read.cifti, *15*, *23*, *31, 32*, *82*, 90, *93, 94*, *109*, *116*, *119*, *123*, *125*, *145*, *176*
read.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
read.cifti.header, *15, 16*, *19*, *23*, *25–27*, *29, 30*, *33*, *81*, *90, 91*, 91, *109*, *119*, *123*, *125*
read.cifti.header(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *94*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
read.cifti.rows, *90*, 93
read.cifti.rows(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93*, *109*, *177*, *188*, *196*, *201*, *203*, *205*
read.dti.bval, 95, *97*
read.dti.bval(), *96–98*, *178, 179*
read.dti.bvec, 95, *97*, *179*
read.dti.bvec(), *95*, *97, 98*, *178, 179*
read.dti.grad, 96, *179*
read.dti.grad(), *95, 96*, *98*, *178, 179*
read.dti.gradients, *96*, 97, *152*
read.dti.gradients(), *95–97*, *178, 179*

read.dti.tck, *39*, *40*, *46*, 98, *180*, *233*
read.dti.tck.header, 100, *104*
read.dti.trk, *39*, *46*, 100, *103*, *181–183*, *233*
read.dti.trk.header, 102
read.dti.tsf, *46*, 103, *183*
read.dti.tsf.header, 104
read.fs.annot, *12*, *18*, *36*, 105, *175*, *184*, *202*
read.fs.annot(), *12*, *36*, *108*, *175*, *185–187*
read.fs.annot.gii, 106, *113*
read.fs.annot.gii(), *113*, *120*, *131*
read.fs.colortable, *11*, 107, *187*
read.fs.colortable(), *12*, *36*, *106*, *175*, *185–187*
read.fs.connectome.cifti, *82*, 108, *188*
read.fs.connectome.cifti(), *14–17*, *19–32*, *34*, *35*, *81–83*, *91*, *93*, *94*, *177*, *188*, *196*, *201*, *203*, *205*
read.fs.curv, 110
read.fs.curv(), *44*, *45*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
read.fs.gca, 111
read.fs.label, 111, *190*
read.fs.label(), *113*, *114*, *191*
read.fs.label.gii, 112
read.fs.label.gii(), *107*, *112*, *114*, *120*, *131*, *191*
read.fs.label.native, *112*, 113
read.fs.label.native(), *112*, *113*, *191*
read.fs.mgh, *61–63*, *68*, 114, *150*
read.fs.mgh(), *44*, *45*, *110*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
read.fs.morph, *91*, 116, *150*
read.fs.morph(), *44*, *45*, *110*, *115*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
read.fs.morph.asc, 117
read.fs.morph.bvsmp, 118
read.fs.morph.cifti, *116*, 118, *195*
read.fs.morph.gii, 119
read.fs.morph.gii(), *44*, *45*, *107*, *110*, *113*, *115*, *117*, *131*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
read.fs.morph.ni1, 120
read.fs.morph.ni2, 121
read.fs.morph.nii, 121
read.fs.morph.txt, 122
read.fs.parcellation.cifti, 122, *202*
read.fs.patch, *45*, 124, *125*, *203*
read.fs.patch(), *45*, *125*, *204*
read.fs.patch.asc, *45*, *124*, 124
read.fs.patch.asc(), *45*, *124*, *204*
read.fs.series.cifti, 125, *204*
read.fs.surface, 126, *159–162*
read.fs.surface(), *42*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.asc, 127, *161*
read.fs.surface.asc(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.bvsrf, 128
read.fs.surface.bvsrf(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.byu, 129
read.fs.surface.geo, 129
read.fs.surface.geo(), *42*, *127*, *128*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.gii, 130
read.fs.surface.gii(), *42*, *107*, *113*, *120*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.ico, 131
read.fs.surface.ico(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.mz3, 132
read.fs.surface.obj, 132
read.fs.surface.obj(), *42*, *127*, *128*, *130*, *131*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.off, 133
read.fs.surface.off(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.ply, 134
read.fs.surface.ply(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*
read.fs.surface.stl, 135, *216*, *217*
read.fs.surface.stl.bin, 135
read.fs.surface.vtk, 136
read.fs.surface.vtk(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *154*, *159*, *160*, *206*, *207*, *209–211*, *219*

read.fs.tracts.vtk, *137*, 137
read.fs.transform, 138
read.fs.transform(), *54*, *58*, *63–67*, *69*, *85*, *140–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
read.fs.transform.dat, 139, *221*
read.fs.transform.dat(), *54*, *58*, *63–67*, *69*, *85*, *139*, *141–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
read.fs.transform.fslmat, 140
read.fs.transform.fslmat(), *54*, *58*, *63–67*, *69*, *85*, *139, 140*, *142–144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
read.fs.transform.itk, 141, *224*
read.fs.transform.itk(), *54*, *58*, *63–67*, *69*, *85*, *139–141*, *143, 144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
read.fs.transform.lta, 142
read.fs.transform.lta(), *54*, *58*, *63–67*, *69*, *85*, *139–142*, *144*, *164, 165*, *168–170*, *172*, *220, 221*, *223–226*
read.fs.transform.xfm, 143
read.fs.transform.xfm(), *54*, *58*, *63–67*, *69*, *85*, *139–143*, *164, 165*, *168–170*, *172*, *220*, *222–226*
read.fs.volume, *79*, *91*, 144, *148*, *227*
read.fs.volume(), *44, 45*, *110*, *115*, *117*, *120*, *153*, *189*, *192–194*, *197–200*, *228*
read.fs.volume.analyze, *55*, *88, 89*, *145*, 146
read.fs.volume.nii, 148
read.fs.volume.nrrd, 151, *157*
read.fs.weight, 152
read.fs.weight(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *189*, *192–194*, *197–200*, *228*
read.mesh.brainvoyager, 153
read.mesh.brainvoyager(), *42*, *127, 128*, *130, 131*, *133, 134*, *137*, *159, 160*, *206, 207*, *209–211*, *219*
read.nifti1.data, *121, 122*, 154
read.nifti1.header, *55*, *74*, *76*, *147*, *154*, 155
read.nifti2.data, *121, 122*, 155
read.nifti2.header, *74*, *77, 78*, *92*, *155*, 156, *230*
read.nrrd.header, *151, 152*, 156
read.smp.brainvoyager, *118*, 157, *231*
read_nisurface, *120*, 159
read_nisurface(), *42*, *127, 128*, *130, 131*, *133, 134*, *137*, *154*, *160*, *206, 207*, *209–211*, *219*
read_nisurfacefile, *159*, 160
read_nisurfacefile(), *42*, *127, 128*, *130, 131*, *133, 134*, *137*, *154*, *159*, *206, 207*, *209–211*, *219*
read_nisurfacefile.fsascii, 161
read_nisurfacefile.fsnative, 161
read_nisurfacefile.gifti, 162
readable.files, 158
rotate2D, 162
rotate3D, 163
rotate3D(), *44*

sm0to1, *69*, 163, *164*
sm0to1(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220*, *222–226*
sm1to0, *64, 65*, *164*, 164
sm1to0(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *168–170*, *172*, *220*, *222–226*
summary.fs.transform, 165
summary.fs.transform(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164*, *168–170*, *172*, *220*, *222–226*
surfaceras.to.ras, 166
surfaceras.to.talairach, 167

talairachras.to.ras, 167
trackvis.affine.to.rasmm, *102*
transform2lps, 168, *223*
transform2lps(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *169, 170*, *172*, *220*, *222–226*
transform2ras, *141*, *168*, 169
transform2ras(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *168*, *170*, *172*, *220*, *222–226*
transform2voxel, 170, *219*, *222*
transform2voxel(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *168, 169*, *172*, *220*, *222–226*
transform2world, *140*, *169, 170*, 171

transform2world(), *54*, *58*, *63–67*, *69, 70*, *85*, *139–144*, *164, 165*, *168–170*, *220*, *222–226*

vertex.euclid.dist, 172
vertex.euclid.dist(), *35*, *173*
vertexdists.to.point, 173
vertexdists.to.point(), *35*, *172*

write.analyze, *8*, 173, *227*
write.atlas.to.lut.and.csv, 175
write.atlas.to.lut.and.csv(), *12*, *36*, *106*, *108*, *185–187*
write.cifti, *14*, *16, 17*, *19–21*, 176, *187, 188*, *194, 195*, *200–202*, *204*, *227*
write.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *188*, *196*, *201*, *203*, *205*
write.dti.bval, 177
write.dti.bval(), *95–98*, *179*
write.dti.bvec, 178
write.dti.bvec(), *95–98*, *178, 179*
write.dti.grad, 179
write.dti.grad(), *95–98*, *178, 179*
write.dti.tck, *46*, 180, *183*
write.dti.trk, *46*, *180*, 181
write.dti.tsf, 182
write.fs.annot, *12*, 184
write.fs.annot(), *12*, *36*, *106*, *108*, *175*, *186, 187*
write.fs.annot.gii, 185
write.fs.annot.gii(), *12*, *36*, *106*, *108*, *175*, *185*, *187*, *192*, *197*, *210*
write.fs.colortable, *175*, 186
write.fs.colortable(), *12*, *36*, *106*, *108*, *175*, *185, 186*
write.fs.connectome.cifti, 187
write.fs.connectome.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *177*, *196*, *201*, *203*, *205*
write.fs.curv, 189
write.fs.curv(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *192–194*, *197–200*, *228*
write.fs.label, 190
write.fs.label(), *112–114*
write.fs.label.gii, 191
write.fs.label.gii(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *186*, *189*, *193, 194*, *197–200*, *210*, *228*
write.fs.mgh, 192
write.fs.mgh(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192*, *194*, *197–200*, *228*
write.fs.morph, 193
write.fs.morph(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
write.fs.morph.asc, 194
write.fs.morph.asc(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
write.fs.morph.cifti, *194*, 195, *202*
write.fs.morph.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *201*, *203*, *205*
write.fs.morph.gii, 196
write.fs.morph.gii(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *186*, *189*, *192–194*, *197–200*, *210*, *228*
write.fs.morph.ni1, 197
write.fs.morph.ni1(), *44, 45*, *79*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228, 229*
write.fs.morph.ni2, 198
write.fs.morph.ni2(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197*, *199, 200*, *228*, *230*
write.fs.morph.smp, 199, *231*
write.fs.morph.smp(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197, 198*, *200*, *228*
write.fs.morph.txt, 199
write.fs.morph.txt(), *44, 45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–199*, *228*
write.fs.parcellated.cifti, *19*, 200
write.fs.parcellated.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *203*, *205*
write.fs.parcellation.cifti, 202
write.fs.parcellation.cifti(), *14–17*, *19–32*, *34, 35*, *81–83*, *91*, *93, 94*, *109*, *177*, *188*, *196*, *201*, *205*
write.fs.patch, 203
write.fs.patch(), *45*, *124, 125*
write.fs.series.cifti, 204
write.fs.series.cifti(), *14–17*, *19–32*,

34, *35*, *81–83*, *91*, *93*, *94*, *109*, *177*, *188*, *196*, *201*, *203*
write.fs.surface, 205
write.fs.surface(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *207*, *209–213*, *215–217*, *219*
write.fs.surface.asc, 206
write.fs.surface.asc(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *209–211*, *219*
write.fs.surface.bvsrf, 207
write.fs.surface.byu, 208
write.fs.surface.byu(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *210*, *211*, *219*
write.fs.surface.gii, 209
write.fs.surface.gii(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *186*, *192*, *197*, *206*, *207*, *209*, *211*, *219*
write.fs.surface.mz3, 211
write.fs.surface.mz3(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209*, *210*, *219*
write.fs.surface.obj, 212
write.fs.surface.obj(), *206*, *213*, *215–217*, *219*
write.fs.surface.off, 213
write.fs.surface.off(), *206*, *212*, *215–217*, *219*
write.fs.surface.off.ply2(), *206*, *212*, *213*, *215–217*, *219*
write.fs.surface.ply, *134*, 214
write.fs.surface.ply(), *206*, *212*, *213*, *216*, *217*, *219*
write.fs.surface.ply2, 215
write.fs.surface.ply2(), *206*, *212*, *213*, *215*, *217*, *219*
write.fs.surface.stl, 216
write.fs.surface.stl(), *206*, *212*, *213*, *215*, *216*, *219*
write.fs.surface.vtk, 218
write.fs.surface.vtk(), *42*, *127*, *128*, *130*, *131*, *133*, *134*, *137*, *154*, *159*, *160*, *206*, *207*, *209–213*, *215–217*
write.fs.transform, 219
write.fs.transform(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *222–226*
write.fs.transform.dat, *220*, 221
write.fs.transform.dat(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *220*, *223–226*
write.fs.transform.fslmat, *170*, 222
write.fs.transform.fslmat(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *220*, *222*, *224–226*
write.fs.transform.itk, *142*, *220*, 223
write.fs.transform.itk(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *220*, *222*, *223*, *225*, *226*
write.fs.transform.lta, *220*, 224
write.fs.transform.lta(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *220*, *222–224*, *226*
write.fs.transform.xfm, *220*, 225
write.fs.transform.xfm(), *54*, *58*, *63–67*, *69*, *70*, *85*, *139–144*, *164*, *165*, *168–170*, *172*, *220*, *222–225*
write.fs.volume, *174*, 226
write.fs.weight, 227
write.fs.weight(), *44*, *45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
write.fs.weight.asc, 228
write.fs.weight.asc(), *44*, *45*, *110*, *115*, *117*, *120*, *146*, *153*, *189*, *192–194*, *197–200*, *228*
write.nifti1, *72*, *79*, *174*, *197*, 229
write.nifti1(), *79*, *197*
write.nifti2, *73*, *77*, *156*, *198*, 230
write.nifti2(), *198*
write.smp.brainvoyager, *12*, *199*, 231
write_xml, *53*

xml_cdata, *13*
xml_node_gifti_coordtransform, 231