# Package: ieegio (via r-universe)

October 16, 2026

**Title** File IO for Intracranial Electroencephalography

**Version** 0.1.3

**Language** en-US

**Encoding** UTF-8

**Description** Integrated toolbox supporting common file formats used for intracranial Electroencephalography (iEEG) and deep-brain stimulation (DBS) study.

**URL** http://dipterix.org/ieegio/, https://github.com/dipterix/ieegio

**BugReports** https://github.com/dipterix/ieegio/issues

**License** MIT + file LICENSE

**Roxygen** list(markdown = TRUE)

**Imports** data.table (>= 1.16.0), digest, fastmap, filearray (>= 0.1.8), freesurferformats, fs, fst (>= 0.9.0), gifti (>= 0.8.0), grDevices, jsonlite, oro.nifti, R.matlab (>= 3.7.0), R6, readNSx (>= 0.0.5), rpyANTs (>= 0.0.3), stats, stringr, utils, yaml

**Suggests** h5lite, reticulate, ravetools, rgl, RNifti (>= 1.7.0), rpymat (>= 0.1.7), xml2, knitr, r3js, rmarkdown, tools, testthat (>= 3.0.0)

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**Config/roxygen2/version** 8.1.0

**Config/pak/sysreqs**
cmake make libicu-dev libpng-dev libuv1-dev libxml2-dev libssl-dev python3 libzmq3-dev

**Repository** https://dipterix.r-universe.dev

**Date/Publication** 2026-10-11 21:59:23 UTC

**RemoteUrl** https://github.com/dipterix/ieegio

**RemoteRef** HEAD

**RemoteSha** 71601e130d20385e70384faa984d9301163db181

# Contents

as_ieegio_colormap . . . 3
as_ieegio_colortable . . . 4
as_ieegio_lookup . . . 5
as_ieegio_roi . . . 6
as_ieegio_surface . . . 10
as_ieegio_transform . . . 13
as_ieegio_volume . . . 14
as_nifti_header . . . 16
burn_curve . . . 17
burn_volume . . . 19
calculate_color . . . 20
colorspace_from_rgb . . . 22
convert-fst . . . 23
detect_roi_overlap . . . 23
ieegio_sample_data . . . 26
imaging-streamlines . . . 27
imaging-surface . . . 29
imaging-volume . . . 31
io-trk . . . 35
io-tt . . . 38
io-vtk-polys . . . 38
io-vtk-streamlines . . . 40
io_h5_valid . . . 41
io_read_ants_transform . . . 42
io_read_flirt_transform . . . 44
io_read_fstarray_or_h5 . . . 46
io_read_h5 . . . 47
io_write_h5 . . . 48
LazyFST . . . 49
LazyH5 . . . 52
low-level-read-write . . . 54
merge.ieegio_surface . . . 58
merge.ieegio_volume . . . 60
new_space . . . 62
niml_find . . . 63
NWBHDF5IO . . . 65
plot.ieegio_colormap_continuous . . . 69
plot.ieegio_surface . . . 70
plot.ieegio_volume . . . 73
pynwb_module . . . 75
read_bci2000 . . . 76
read_brainvis . . . 77
read_colormap . . . 78
read_edf . . . 79
read_nsx . . . 80
read_nwb . . . 81

resample_volume . . . . . 84
resolve_roi_as . . . . . 86
SignalDataCache . . . . . 89
surface_to_surface . . . . . 90
transform_flirt2ras . . . . . 92
transform_orientation . . . . . 93
volume_to_surface . . . . . 96
write_colormap . . . . . 97
write_edf . . . . . 98

**Index** **102**

---

`as_ieegio_colormap` *Convert to* `ieegio_colormap`

---

## Description

Combine or convert objects into an `ieegio_colormap`.

## Usage

```
as_ieegio_colormap(
  x,
  lookup = NULL,
  colorspace = "RGB",
  data_range = NULL,
  type = c("auto", "discrete", "continuous"),
  ...
)

## S3 method for class 'threeBrain_colormap'
as_ieegio_colormap(
  x,
  lookup = NULL,
  colorspace = "RGB",
  data_range = NULL,
  type = c("auto", "discrete", "continuous"),
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | an `ieegio_colortable`, existing `ieegio_colormap`, or label-table `data.frame` |
| `lookup` | optional `ieegio_lookup` to attach; replaces existing lookup when `x` is already an `ieegio_colormap` |
| `colorspace` | one of `"RGB"` (default), `"sRGB"`, `"HSV"`, `"HCL"`, `"Lab"` |

| | |
|---|---|
| data_range | optional numeric(1 or 2) fixed data range for continuous color maps; NA/Inf positions are filled from data at call time; NULL defers to the lookup table |
| type | one of "auto" (default), "discrete", or "continuous". When lookup = NULL, this controls whether a discrete or continuous color map is returned. "auto" produces discrete unless data_range is also supplied. |
| ... | additional arguments |

## Value

An ieegio_colormap object.

## Examples

```
# From a color table alone (discrete colormap, no labels)
ct <- as_ieegio_colortable(data.frame(
  Key = 1:3,
  R = c(255L, 0L, 0L), G = c(0L, 200L, 0L),
  B = c(0L, 0L, 180L), A = c(255L, 255L, 255L)
))
cm <- as_ieegio_colormap(ct)
print(cm)

# Continuous colormap from the same color table
cm_cont <- as_ieegio_colormap(ct, type = "continuous", data_range = c(-10, 10))
print(cm_cont)

# From a label table data.frame (adds both colors and a discrete lookup)
lt <- data.frame(
  Key   = 1:2,
  Label = c("Region A", "Region B"),
  R = c(200L, 50L), G = c(50L, 200L),
  B = c(50L,  50L), A = c(255L, 255L)
)
cm2 <- as_ieegio_colormap(lt)
print(cm2)
```

---

# as_ieegio_colortable *Convert to ieegio color table*

---

## Description

Convert a data frame or existing label table to an ieegio_colortable object.

## Usage

```
as_ieegio_colortable(x, ...)
```

### Arguments

| | |
|---|---|
| x | object to convert |
| ... | additional arguments passed to methods |

### Value

An ieegio_colortable object.

### Examples

```
ct <- as_ieegio_colortable(data.frame(
  Key = 1:3,
  R   = c(255L, 0L, 0L),
  G   = c(0L, 255L, 0L),
  B   = c(0L, 0L, 255L),
  A   = c(255L, 255L, 255L)
))
print(ct)
```

---

## as_ieegio_lookup *Convert to ieegio lookup table*

---

### Description

Convert a data frame to an ieegio_lookup object.

### Usage

```
as_ieegio_lookup(x, type = "auto", ...)
```

### Arguments

| | |
|---|---|
| x | object to convert |
| type | "discrete" or "continuous" (or "auto" to detect) |
| ... | additional arguments |

### Value

An ieegio_lookup object.

## Examples

```
# Discrete lookup: integer keys mapped to region labels
lut_disc <- as_ieegio_lookup(data.frame(
  Key   = 1:3,
  Label = c("Cortex", "White matter", "CSF")
))
print(lut_disc)

# Continuous lookup: piecewise value-to-scaled mapping
lut_cont <- as_ieegio_lookup(data.frame(
  Value  = c(-5, 0, 5),
  Scaled = c(0, 0.5, 1)
))
print(lut_cont)
```

---

# as_ieegio_roi *Convert objects to 'ieegio' region of interest*

---

## Description

Marks an image volume, surface, point cloud, or streamlines object as a region of interest (ROI). The returned object is the underlying `'ieegio'` instance with an extra `"ieegio_roi"` class and an `"roi_info"` attribute; see 'Details'.

## Usage

```
as_ieegio_roi(x, ...)

## Default S3 method:
as_ieegio_roi(x, ...)

## S3 method for class 'ieegio_roi'
as_ieegio_roi(x, ...)

## S3 method for class 'character'
as_ieegio_roi(x, type = c("auto", "volume", "surface", "streamlines"), ...)

## S3 method for class 'ieegio_streamlines'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'list'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'matrix'
as_ieegio_roi(x, type = c("pointcloud", "streamlines"), ...)

## S3 method for class 'data.frame'
```

```
as_ieegio_roi(x, type = c("pointcloud", "streamlines"), ...)

## S3 method for class 'numeric'
as_ieegio_roi(x, ...)

## S3 method for class 'integer'
as_ieegio_roi(x, ...)

## S3 method for class 'array'
as_ieegio_roi(x, ...)

## S3 method for class 'ieegio_volume'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'niftiImage'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'nifti'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'ants.core.ants_image.ANTsImage'
as_ieegio_roi(x, threshold_lb = NA, threshold_ub = NA, ...)

## S3 method for class 'ieegio_surface'
as_ieegio_roi(
  x,
  threshold_expr = NULL,
  quoted = FALSE,
  env = parent.frame(),
  ...
)

## S3 method for class 'mesh3d'
as_ieegio_roi(
  x,
  threshold_expr = NULL,
  quoted = FALSE,
  env = parent.frame(),
  ...
)

## S3 method for class 'fs.surface'
as_ieegio_roi(
  x,
  threshold_expr = NULL,
  quoted = FALSE,
  env = parent.frame(),
  ...
```

```
)

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

## Arguments

| | |
|---|---|
| `x` | R object or file path to convert; when `x` is a character vector of file paths, only the first element is used and the remaining elements are ignored |
| `...` | passed to the underlying `as_ieegio_volume`, `as_ieegio_surface`, or `as_ieegio_streamlines` methods |
| `type` | type of the region of interest. For file paths, the choices are `"auto"` (default, inferred from the file extension), `"volume"`, `"surface"`, and `"streamlines"`; for matrices and data frames, the choices are `"pointcloud"` (default) and `"streamlines"`. There is no `"pointcloud"` choice for file paths: a point cloud on disk is simply a surface file that carries no face index, so the file is read as a surface and the label is corrected afterwards from the face count. |
| `threshold_lb, threshold_ub` | lower and upper bounds selecting the parts that belong to the region of interest; default is `NA` (no bound). Non-finite bounds are dropped. For image volumes the bounds select voxels by intensity; for streamlines they select tracts by arc length, in the unit of the world coordinate system |
| `threshold_expr` | expression selecting the vertices that belong to the region of interest, evaluated with `.x` bound to the surface, `.a` and `.m` bound to the annotation and measurement tables, and `.a1` and `.m1` bound to the first column of those tables; default is `NULL` (no selection) |
| `quoted` | whether `threshold_expr` has been quoted already; default is `FALSE` |
| `env` | environment in which `threshold_expr` is evaluated; default is the calling frame |

## Details

A region of interest is a decorator describing *how* to derive the region from the object it wraps; it does not alter the underlying data. The criteria are stored in the `"roi_info"` attribute, a list with the resolved `type` plus, where applicable, `threshold_lb`, `threshold_ub`, or `threshold_fun`. Applying them is left to the functions that consume a region of interest, and those functions document the behavior themselves; `resolve_roi_as` is the general one, applying the criteria and returning the region itself.

The two bounds are interpreted against whatever quantity is natural for the enclosed object: voxel intensity for an image volume, and tract arc length for streamlines. Surfaces and point clouds use `threshold_expr` instead, since their selection is driven by vertex annotations or measurements rather than by a single scalar.

Because the criteria live in an attribute, operations that rebuild the object drop them. This is intended: in most workflows the enclosed volume, surface, or streamlines object is what gets used, and the caller is responsible for carrying the decorator along when it matters.

## Value

`as_ieegio_roi` returns the converted `'ieegio'` object with `"ieegio_roi"` prepended to its class and an `"roi_info"` attribute. `print` returns the object invisibly.

## See Also

`resolve_roi_as` applies the criteria recorded here and converts the region into a chosen representation.

## Examples

```
# ---- Point cloud from a matrix ----------------------------------------
points <- matrix(rnorm(30), ncol = 3)
roi <- as_ieegio_roi(points)
attr(roi, "roi_info")$type

print(roi)

# ---- Streamlines: rows of `NA` separate the tracts --------------------
tracts <- rbind(
  matrix(rnorm(9), ncol = 3),
  NA,
  matrix(rnorm(15), ncol = 3)
)
roi <- as_ieegio_roi(tracts, type = "streamlines")
length(roi)

# ---- Image volume with threshold bounds -------------------------------
vox2ras <- matrix(
  c(-1, 0, 0, 5,
    0, 0, 1, -5,
    0, -1, 0, 5,
    0, 0, 0, 1),
  nrow = 4, byrow = TRUE
)
volume <- as_ieegio_roi(
  array(rnorm(1000), c(10, 10, 10)),
  vox2ras = vox2ras,
  threshold_lb = 0.5
)
attr(volume, "roi_info")$threshold_lb

# The criteria survive a round trip
attr(as_ieegio_roi(volume), "roi_info")$threshold_lb

# ---- Surface with a threshold expression ------------------------------
vertices <- matrix(rnorm(60), ncol = 3)
faces <- matrix(c(1, 2, 3, 2, 3, 4), ncol = 3, byrow = TRUE)
surface <- as_ieegio_surface(
  vertices,
  faces = faces,
  measurements = data.frame(Curv = rnorm(20))
```

```
)

roi <- as_ieegio_roi(surface, threshold_expr = .m1 > 0)
selection <- attr(roi, "roi_info")$threshold_fun(roi)
head(selection)
```

---

`as_ieegio_surface` *Convert other surface formats to* `ieegio` *surface*

---

## Description

Convert other surface formats to `ieegio` surface

## Usage

```
as_ieegio_surface(x, ...)

## Default S3 method:
as_ieegio_surface(
  x,
  vertices = x,
  faces = NULL,
  face_start = NA,
  transform = NULL,
  vertex_colors = NULL,
  annotation_labels = NULL,
  annotation_values = NULL,
  measurements = NULL,
  time_series_slice_duration = NULL,
  time_series_value = NULL,
  name = NULL,
  ...
)

## S3 method for class 'character'
as_ieegio_surface(x, ...)

## S3 method for class 'ieegio_surface'
as_ieegio_surface(x, ...)

## S3 method for class 'mesh3d'
as_ieegio_surface(x, ...)

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

```
## S3 method for class 'ieegio_niml'
as_ieegio_surface(x, type = NULL, name = "", ...)
```

## Arguments

| | |
|---|---|
| x | R object or file path |
| ... | passed to default method |
| vertices | n by 3 matrix, each row is a vertex node position |
| faces | (optional) face index, either zero or one-indexed (Matlab and R start counting from 1 while C and Python start indices from 0); one-index face order is recommended |
| face_start | (optional) either 0 or 1, indicating whether faces is zero or one-indexed; default is NA, which will check whether the minimum value of faces is 0. If so, then faces will be bumped by 1 internally |
| transform | (optional) a 4 by 4 matrix indicating the vertex position to scanner RAS transform. Default is missing (identity matrix), i.e. the vertex positions are already in the scanner RAS coordinate system. |
| vertex_colors | (optional) integer or color (hex) vector indicating the vertex colors |
| annotation_labels | (optional) a data frame containing at the following columns. Though optional, annotation_labels must be provided when annotation_values is provided<br>"Key" unique integers to appear in annotation_values, indicating the key of the annotation label<br>"Label" a character vector (strings) of human-readable labels of the corresponding key<br>"Color" hex string indicating the color of the key/label |
| annotation_values | (optional) an integer table where each column is a vector of annotation key (for example, 'FreeSurfer' segmentation key) and each row corresponds to a vertex node |
| measurements | (optional) a numeric table where each column represents a variable (for example, curvature) and each row corresponds to a vertex node. Unlike annotations, which is for discrete node values, measurements is for continuous values |
| time_series_slice_duration | (optional) a numeric vector indicating the duration of each slice; default is NA |
| time_series_value | (optional) a numeric matrix (n by m) where n is the number of vertices and m is the number of time points, hence each column is a time slice and each row is a vertex node. |
| name | (optional) name of the geometry |
| type | type of the data, either 'auto', NULL (default) for automatic detection, or 'annotations' 'measurements' for explicit data type |

## Value

An `ieeg_surface` object; see `read_surface` or 'Examples'.

## Examples

```
# ---- Simple usage
# vertices only
dodecahedron_vert <- matrix(
  ncol = 3, byrow = TRUE,
  c(-0.62, -0.62, -0.62, 0.62, -0.62, -0.62, -0.62, 0.62, -0.62,
    0.62, 0.62, -0.62, -0.62, -0.62, 0.62, 0.62, -0.62, 0.62,
    -0.62, 0.62, 0.62, 0.62, 0.62, 0.62, 0.00, -0.38, 1.00,
    0.00, 0.38, 1.00, 0.00, -0.38, -1.00, 0.00, 0.38, -1.00,
    -0.38, 1.00, 0.00, 0.38, 1.00, 0.00, -0.38, -1.00, 0.00,
    0.38, -1.00, 0.00, 1.00, 0.00, -0.38, 1.00, 0.00, 0.38,
    -1.00, 0.00, -0.38, -1.00, 0.00, 0.38)
)

point_cloud <- as_ieegio_surface(dodecahedron_vert)
plot(point_cloud, col = "red")

# with face index
dodecahedron_face <- matrix(
  ncol = 3L, byrow = TRUE,
  c(1, 11, 2, 1, 2, 16, 1, 16, 15, 1, 15, 5, 1, 5, 20, 1, 20, 19,
    1, 19, 3, 1, 3, 12, 1, 12, 11, 2, 11, 12, 2, 12, 4, 2, 4, 17,
    2, 17, 18, 2, 18, 6, 2, 6, 16, 3, 13, 14, 3, 14, 4, 3, 4, 12,
    3, 19, 20, 3, 20, 7, 3, 7, 13, 4, 14, 8, 4, 8, 18, 4, 18, 17,
    5, 9, 10, 5, 10, 7, 5, 7, 20, 5, 15, 16, 5, 16, 6, 5, 6, 9,
    6, 18, 8, 6, 8, 10, 6, 10, 9, 7, 10, 8, 7, 8, 14, 7, 14, 13)
)
mesh <- as_ieegio_surface(dodecahedron_vert,
                          faces = dodecahedron_face)
plot(mesh)

# with vertex colors
mesh <- as_ieegio_surface(dodecahedron_vert,
                          faces = dodecahedron_face,
                          vertex_colors = sample(20))
plot(mesh, name = "color")

# with annotations
mesh <- as_ieegio_surface(
  dodecahedron_vert,
  faces = dodecahedron_face,
  annotation_labels = data.frame(
    Key = 1:3,
    Label = c("A", "B", "C"),
    Color = c("red", "green", "blue")
  ),
```

```
  annotation_values = data.frame(
    MyVariable = c(rep(1, 7), rep(2, 7), rep(3, 6))
  )
)
plot(mesh, name = "annotations")

# with measurements
mesh <- as_ieegio_surface(
  dodecahedron_vert,
  faces = dodecahedron_face,
  measurements = data.frame(
    MyVariable = dodecahedron_vert[, 1]
  )
)
plot(mesh, name = "measurements",
     col = c("blue", "gray", "red"))
```

---

as_ieegio_transform *Convert to ieegio transform*

---

## Description

Generic function to convert various objects into `ieegio_transforms`.

## Usage

```
as_ieegio_transform(x, ...)

## S3 method for class '`NULL`'
as_ieegio_transform(x, space_from = "", space_to = "", ...)

## S3 method for class 'character'
as_ieegio_transform(x, format = c("ants", "flirt"), ...)

## S3 method for class 'matrix'
as_ieegio_transform(x, space_from = "", space_to = "", ...)

## S3 method for class 'array'
as_ieegio_transform(x, space_from = "", space_to = "", ...)

## S3 method for class 'list'
as_ieegio_transform(x, ...)

## S3 method for class 'ieegio_transforms'
as_ieegio_transform(x, ...)
```

### Arguments

| | |
|---|---|
| x | object to convert (character path, matrix, array, list, or existing transform) |
| ... | additional arguments passed to methods |
| space_from | source space for matrix/array methods. Default "" is a wildcard for arbitrary space name. |
| space_to | target space for matrix/array methods. Default "" is a wildcard for arbitrary space name. |
| format | character string specifying the file format for character paths. Supports "ants" (default) for ANTs format and "flirt" for FSL FLIRT format. Only used for character method. |

### Details

Methods available:

- character: Reads transform from file (uses io_read_ants_transform or io_read_flirt_transform depending on format)
- matrix: Creates transform from matrix
- array: Creates transform from 2D array
- list: Creates transform chain from list of transforms
- ieegio_transforms: Returns input unchanged

### Value

An ieegio_transforms object

---

## as_ieegio_volume *Convert objects to 'ieegio' image volumes*

---

### Description

Convert array, path, or 'NIfTI' images in other formats to 'ieegio' image volume instance

### Usage

```
as_ieegio_volume(x, ...)

## S3 method for class 'character'
as_ieegio_volume(x, ...)

## S3 method for class 'ieegio_volume'
as_ieegio_volume(x, ...)

## S3 method for class 'array'
as_ieegio_volume(x, vox2ras = NULL, as_color = is.character(x), ...)
```

```
## S3 method for class 'niftiImage'
as_ieegio_volume(x, ...)

## S3 method for class 'nifti'
as_ieegio_volume(x, ...)

## S3 method for class 'ants.core.ants_image.ANTsImage'
as_ieegio_volume(x, ...)
```

## Arguments

| | |
|---|---|
| x | R object such as array, image path, or objects such as 'RNifti' or 'oro.nifti' image instances |
| ... | passed to other methods |
| vox2ras | a 4x4 'affine' matrix representing the transform from 'voxel' index (column-row-slice) to 'RAS' (right-anterior-superior) coordinate. This transform is often called 'xform', 'sform', 'qform' in 'NIfTI' terms, or 'Norig' in 'FreeSurfer' |
| as_color | for converting arrays to volume, whether to treat x as array of colors; default is true when x is a raster matrix ( matrix of color strings) and false when x is not a character array. |

## Value

An ieegio volume object; see imaging-volume

## Examples

```
shape <- c(50, 50, 50)
vox2ras <- matrix(
  c(-1, 0, 0, 25,
    0, 0, 1, -25,
    0, -1, 0, 25,
    0, 0, 0, 1),
  nrow = 4, byrow = TRUE
)

# continuous
x <- array(rnorm(125000), shape)

volume <- as_ieegio_volume(x, vox2ras = vox2ras)
plot(volume, zoom = 3, pixel_width = 0.5)

# color rgb(a)
x <- array(
  sample(c("red","blue", "green", "cyan", "yellow"),
         12500, replace = TRUE),
  shape
)
```

```
rgb <- as_ieegio_volume(x, vox2ras = vox2ras)
plot(rgb, zoom = 3, pixel_width = 0.5)



# ---- When RNifti package is not available ---------------------------

# Emulate WebAssemply when RNifti is unavailable, using oro.nifti instead
old_opt <- options("ieegio.debug.emscripten" = TRUE)
on.exit({ options(old_opt) }, add = TRUE)

shape <- c(50, 50, 50)
vox2ras <- matrix(
  c(-1, 0, 0, 25,
    0, 0, 1, -25,
    0, -1, 0, 25,
    0, 0, 0, 1),
  nrow = 4, byrow = TRUE
)

# continuous
x <- array(rnorm(125000), shape)

# In WebAssemply, RNifti is not available, using oro.nifti instead
volume <- as_ieegio_volume(x, vox2ras = vox2ras)

stopifnot(volume$type[[1]] == "oro")

plot(volume, zoom = 3, pixel_width = 0.5)

# Cleanup: make sure the options are reset
options(old_opt)
```

---

`as_nifti_header` *Internal method to extract header information from a 'NIfTI' file*

---

## Description

Internal method to extract header information from a 'NIfTI' file

## Usage

```
as_nifti_header(x)
```

## Arguments

| | |
|---|---|
| x | file path or an R object |

## Value

A list containing the file header information

---

`burn_curve` *Burn a curve trajectory into a volume*

---

## Description

Burn density values along a `Catmull-Rom` spline trajectory into a numeric volume. The curve is sampled at sub-voxel resolution and voxels within `thickness` of the trajectory are assigned density values.

## Usage

```
burn_curve(
  image,
  curve,
  thickness = 1,
  density = 1,
  reshape = FALSE,
  antialias_type = c("reduced", "ignore", "fill", "threshold"),
  merge = c("mean", "max", "min"),
  n_samples = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| `image` | volume providing the coordinate space; see `imaging-volume` |
| `curve` | a `ravetools_curve` object (see `catmull_rom_3d`), or an $n \times 3$ numeric matrix of key points in 'RAS' world coordinates. If a matrix is supplied, `catmull_rom_3d` is called automatically (requires the **ravetools** package). |
| `thickness` | tube radius (half-width) around the curve in world ('RAS') coordinates (millimeters). Can be a positive scalar or a function of `t` in `[0, 1]` that returns a positive scalar, with the same call convention as `curve$get_point(t)`. |
| `density` | value to burn into voxels within the tube. Can be a numeric scalar or a function of `t` in `[0, 1]` that returns a numeric scalar. |
| `reshape` | whether to reshape the image before burning; identical semantics to `burn_volume`: `FALSE` (default, keep original resolution), `TRUE` (double each dimension), a single positive number (isotropic new size), or a length-3 integer vector. |
| `antialias_type` | how to handle voxels at the edge of the tube. One of `"reduced"` (default), `"ignore"`, `"fill"`, or `"threshold"`: |
| | `"ignore"` Only voxels whose center is strictly within `thickness` are burned at full density. |

"fill" Any voxel that overlaps the tube (center within thickness + half_voxel_diagonal) is fully burned.

"threshold" Voxels are burned at full density if more than 50\ within thickness; otherwise skipped.

"reduced" Density is scaled proportionally by the fraction of the voxel (8-corner approximation) that lies within thickness.

merge — how to combine density values when multiple curve samples illuminate the same voxel. One of "mean" (default, average of all contributing samples), "max" (maximum value), or "min" (minimum value).

n_samples — number of points to sample along the curve. Default NULL computes ceiling(arc_length / min_voxel_size * 3) (3x oversampling), with a minimum of 10.

... — passed to as_ieegio_volume, useful when image is an array.

## Value

A numeric ieegio_volume with the same spatial extent as image (or the reshaped extent). Background voxels contain 0. When multiple curve samples illuminate the same voxel, values are combined according to merge.

## Examples

```
if (interactive()) {

dm <- c(50, 50, 50)
image <- as_ieegio_volume(
  array(0, dm),
  vox2ras = rbind(cbind(diag(1, 3), -dm / 2),
                  c(0, 0, 0, 1))
)

# Three-point trajectory (Catmull-Rom from matrix)
key_pts <- rbind(c(-50, 0, 0), c(0, 20, 2), c(-10, 0, -1))

curve <- ravetools::catmull_rom_3d(key_pts)

plot(curve, use_rgl = FALSE)

# Constant thickness and density
burned <- burn_curve(image, key_pts, thickness = 3, density = 100)

plot(burned, zoom = 3, position = c(0, 0, 0), which = "axial")

# Varying thickness and density along the curve
burned2 <- burn_curve(
  image, curve,
  thickness      = function(t) 2 + t * 4,
  density        = function(t) 100 * (1 - t) + 100,
  antialias_type = "reduced"
)
plot(burned2, zoom = 3, position = c(0, 0, 0), which = "axial")
```

```
}
```

---

`burn_volume` *Burn image at given positions*

---

## Description

Burn image at given positions with given color and radius.

## Usage

```
burn_volume(
  image,
  ras_position,
  col = "red",
  radius = 1,
  reshape = FALSE,
  alpha = FALSE,
  blank_underlay = FALSE,
  ...,
  preview = NULL
)
```

## Arguments

| | |
|---|---|
| `image` | volume |
| `ras_position` | image-defined right-anterior-posterior positions, an `nx3` matrix, each row is an 'RAS' coordinate |
| `col` | vector of integer or characters, color of each contact |
| `radius` | vector of positive number indicating the burning radius |
| `reshape` | whether to reshape the image at a different resolution; default is false; can be `TRUE` (image resolution will be doubled), a single number (size of isotropic volume along one side), or a length of three defining the new shape. |
| `alpha` | whether to include alpha (transparent) channel. Default is false for compatibility concerns (legacy software might not support reading alpha channel). In this case, the background will be black. If `alpha=TRUE` is set, then the background will be fully transparent. |
| `blank_underlay` | whether to use blank image or the input `image` as underlay; default is `FALSE` (using `image` as underlay); alternative is `TRUE`, and use black or transparent background |
| `...` | passed to `as_ieegio_volume`, useful if `image` is an array |
| `preview` | indices (integer) of the position to visualize; default is `NULL` (no preview) |

## Value

Color image that is burnt; see imaging-volume.

## Examples

```
if(interactive()) {

dim <- c(6, 6, 6)
image <- as_ieegio_volume(
  array(rnorm(prod(dim)), dim),
  vox2ras = rbind(cbind(diag(1, 3), -dim / 2),
                  c(0, 0, 0, 1))
)

ras_positions <- rbind(c(1, -1, 1.5), c(-2.25, -1, -0.75))


burned <- burn_volume(
  image,
  ras_positions,
  col = c("red", "green"),
  radius = 0.5,
  reshape = c(24, 24, 24)
)

plot(
  burned,
  position = ras_positions[1, ],
  zoom = 15,
  pixel_width = 0.25
)

}
```

---

# calculate_color *Map data values or atlas keys to hex colors*

---

## Description

Apply an ieegio_colormap to a vector of values or atlas keys, returning a character vector of hex color strings.

## Usage

```
calculate_color(
  x,
  colormap,
  type = c("auto", "discrete", "continuous"),
  colorspace = NULL,
```

```
  keep_alpha = TRUE,
  na = "#00000000"
)
```

## Arguments

| | |
|---|---|
| x | numeric or integer vector of data values / atlas keys |
| colormap | an ieegio_colormap object |
| type | one of "auto" (default), "discrete", "continuous". Auto-detection: integer class or any value outside the 0-1 range maps to discrete; numeric values all within 0-1 map to continuous. |
| colorspace | override the color space stored in colormap; one of "RGB", "sRGB", "HSV", "HCL", "Lab". NULL (default) inherits from colormap$colorspace. |
| keep_alpha | logical; if TRUE (default), colors with alpha below 255 are returned as #RRGGBBAA. Set to FALSE to always return #RRGGBB, dropping the alpha channel (needed for formats that do not support alpha in hex). |
| na | color string used to replace NA inputs in the output; defaults to "#00000000" (transparent black). Set to NA to leave NA inputs as NA in the output. |

## Value

Character vector of #RRGGBB (or #RRGGBBAA when keep_alpha=TRUE and alpha is below 255) color strings, same length as x.

## Examples

```
# Build a red-to-green color table with 5 stops
ct <- as_ieegio_colortable(data.frame(
  Key = c(1L, 5L),
  R = c(255L, 0L), G = c(0L, 255L), B = c(0L, 0L), A = c(255L, 255L)
))
cm <- as_ieegio_colormap(ct)

# Discrete: Key 0 is always black; intermediate keys are interpolated
calculate_color(c(0L, 1L, 3L, 5L), cm)

# Continuous: attach a value-to-scaled lookup and map numeric data
lut <- as_ieegio_lookup(data.frame(Value = c(0, 1), Scaled = c(0, 1)))
cm_cont <- as_ieegio_colormap(ct, lookup = lut)
calculate_color(c(0, 0.25, 0.5, 0.75, 1), cm_cont)

# NA handling: default fills NA with transparent black
calculate_color(c(1L, NA_integer_, 5L), cm)
# Pass na = NA to keep NA in the output
calculate_color(c(1L, NA_integer_, 5L), cm, na = NA)
# Drop alpha channel for formats that do not support it
calculate_color(c(1L, 5L), cm, keep_alpha = FALSE)
```

`colorspace_from_rgb` *Color space conversion helpers*

## Description

Convert an n-by-3 RGB integer matrix (values 0-255) to a target color space, or convert back. All conversions use base-R `grDevices` - no additional packages required.

## Usage

```
colorspace_from_rgb(rgb_mat, to = c("RGB", "sRGB", "HSV", "HCL", "Lab"))

colorspace_to_rgb(mat, from = c("RGB", "sRGB", "HSV", "HCL", "Lab"))
```

## Arguments

| | |
|---|---|
| `rgb_mat` | integer matrix with columns R, G, B (values 0-255) |
| `to, from` | one of `"RGB"`, `"sRGB"`, `"HSV"`, `"HCL"`, `"Lab"` |
| `mat` | color space matrix returned by `colorspace_from_rgb` |

## Value

A numeric matrix with 3 columns in the target color space. For `colorspace_to_rgb`: an integer matrix (0-255).

## Examples

```
# RGB primaries as a 3x3 matrix (rows = colors, columns = R, G, B)
rgb_mat <- matrix(c(
  255,   0,   0,
    0, 255,   0,
    0,   0, 255
), ncol = 3, byrow = TRUE)

# Convert to HSV and back
hsv_mat  <- colorspace_from_rgb(rgb_mat, to = "HSV")
rgb_back <- colorspace_to_rgb(hsv_mat, from = "HSV")
print(rgb_back)

# Convert to HCL (perceptually uniform)
hcl_mat <- colorspace_from_rgb(rgb_mat, to = "HCL")
print(hcl_mat)
```

## convert-fst *Convert 'FST' files to other formats*

### Description

'HDF5', 'CSV' are common file formats that can be read into 'Matlab' or 'Python'

### Usage

```
convert_fst_to_hdf5(fst_path, hdf5_path, exclude_names = NULL)

convert_fst_to_csv(fst_path, csv_path, exclude_names = NULL)
```

### Arguments

| | |
|---|---|
| fst_path | path to 'FST' file |
| hdf5_path | path to 'HDF5' file; if file exists before the conversion, the file will be erased first. Please make sure the files are backed up. |
| exclude_names | table names to exclude |
| csv_path | path to 'CSV' file; if file exists before the conversion, the file will be erased first. Please make sure the files are backed up. |

### Value

`convert_fst_to_hdf5` will return a list of data saved to 'HDF5'; `convert_fst_to_csv` returns the normalized 'CSV' path.

## detect_roi_overlap *Detect whether two regions of interest overlap*

### Description

Reports, for every unit of `y`, whether it comes within `radius` of `x`, and hands back `y` with that answer written onto it. The test is asymmetric: `x` is indexed once and `y` is streamed against it, so the results line up with `y`, and the larger or repeatedly reused region belongs in `x`.

### Usage

```
detect_roi_overlap(
  x,
  y,
  mode_x = c("auto", "volume", "pointcloud", "streamlines", "surface"),
  mode_y = c("auto", "volume", "pointcloud", "streamlines", "surface"),
  radius = 0,
  early_stop = FALSE,
```

```
  include_interior = FALSE
)

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

## Arguments

| | |
|---|---|
| x, y | the two regions, or anything as_ieegio_roi accepts; each is put through resolve_roi_as before being tested |
| mode_x, mode_y | representation to resolve each region into first: "auto" (default, keep whatever the region already is), "volume", "pointcloud", "surface", or "streamlines" |
| radius | distance tolerance; an overlap is reported when the distance is at most radius. Default is 0, meaning literal contact. A volume raises this to a floor of its own; see 'Details' |
| early_stop | whether to stop measuring a unit at its first overlapping element instead of keeping the closest one; default is FALSE. Every unit is answered either way; see 'Details' |
| include_interior | whether geometry lying strictly inside a closed x counts as overlapping even when it never comes within radius of the surface itself; default is FALSE. Only a surface has an interior, so when x resolves to anything else this is reported and dropped rather than raised as an error, and the test falls back to contact alone. The setting that was actually used is recorded in collision_detection$summary$include_interior |
| ... | ignored, present for compatibility with print |

## Details

Answers are reported per *unit* of y, which is coarser than a vertex or a point: a point for a point cloud, a voxel center for a volume, a **face** for a surface, and a **whole tract** for streamlines. A surface of 642 vertices and 1280 faces therefore yields 1280 answers, and a bundle of 20 tracts yields 20, however many points each holds.

**Both regions must already share one coordinate space.** Nothing here transforms them into a common one: a volume contributes its vox2ras, streamlines their header$vox2ras, and a surface whichever transform geometry$transforms lists first, which for a GIFTI carrying several is not necessarily the scanner one. Comparing a region in "MNI152" against one in "ScannerAnat" returns confident nonsense rather than an error.

radius has a lower bound that the regions themselves set. **ravetools** has no volume mode, so a volume is tested as its voxel centers, and each center then stands in for a voxel it can no longer describe. The radius actually used is max(radius, inflation_x, inflation_y), where a volume's inflation is half its voxel diagonal and every other representation inflates by zero - a point cloud is genuinely points, and surfaces and streamlines are carried exactly. Two volumes therefore floor at the larger of their two, a volume against anything else floors at its own, and a pair without a volume leaves radius untouched.

early_stop chooses only which element inside a unit gets reported, not how many units are looked at: FALSE measures every element and keeps the closest, TRUE stops at the first that overlaps. Both a hit and a distance come back either way. Since a point and a face are each their own unit, the

setting changes nothing for them, and matters only for streamlines, where a tract's reported distance becomes that of its first overlapping segment rather than its nearest one.

Tracts that are not lines are dropped before the test: a tract needs at least two points to have any segment, and points with missing coordinates are removed first. A region can therefore report fewer tracts than it was built with.

## Value

An `"ieegio_roi_overlap_result"` list with

`overlapped` whether the two regions overlap at all, taking each as a whole.

`hit_ratio` the ratio, between 0 and 1, of the units of `y` that overlap `x`. Units that could not be tested are left out of both sides rather than counted as misses, and a region with nothing testable in it gives `0` rather than `NaN`. A surface reports two, `c(vertex = , face = )`, a face-level answer and its vertex-level reduction being different numbers.

`early_stop` the setting actually used.

`annotated` the resolved `y`, annotated as above.

`collision_detection` the raw `vcg_detect_collision` output, untouched.

`mode_x`, `mode_y` the representations the two regions resolved to, which is what was tested, rather than the arguments, which may both have said `"auto"`.

## Annotating `y`

`annotated` is the resolved `y`, in whatever `mode_y` asked for, carrying the result in that representation's own idiom:

**point cloud, surface** two vertex measurements, `Overlap` (`1` or `0`) and `distance` (the distance, or `NA` where clear). A surface is answered per face, so both are read at the vertices: a vertex overlaps if any face meeting there does, and takes the nearest of them.

**streamlines** the same two as per-tract *properties*, one value each, since a unit is a whole tract.

**volume** voxel values on the region's own grid and `vox2ras`. A volume cannot hold `NA` - it becomes `0`, which a distance field would read as exact contact - so two negative sentinels keep the three states apart, neither of which a measured distance can take: `-2` outside the region and never tested, `-1` inside it but not overlapping, and `>= 0` overlapping, the value being the distance. Thresholding at `>= -1` therefore recovers the region's own mask, and at `>= 0` the part of it that overlaps.

## See Also

`as_ieegio_roi` records how a region is derived, and `resolve_roi_as` applies that description; this function consumes the result of both.

## Examples

```
if (interactive()) {

# ---- Two spheres, offset so that they partly overlap -----------------
sphere <- ravetools::vcg_sphere()
```

```
x <- as_ieegio_roi(sphere)

sphere$vb[1:2, ] <- sphere$vb[1:2, ] + 1
y <- as_ieegio_roi(sphere)

result <- detect_roi_overlap(x, y)
result

# `y` is answered per face, and carries the answer at its vertices
length(result$collision_detection$hit_unit)
head(result$annotated$measurements$data_table)

# ---- The same question asked of a volume ----------------------------
mask <- array(0, c(20, 20, 20))
mask[8:13, 8:13, 8:13] <- 1
vox2ras <- rbind(cbind(diag(1, 3), c(-10, -10, -10)), c(0, 0, 0, 1))
region <- as_ieegio_roi(mask, vox2ras = vox2ras, threshold_lb = 0.5)

result <- detect_roi_overlap(x, region, mode_y = "volume")
result$hit_ratio

# `>= -1` is the region itself, `>= 0` the part of it that overlaps
values <- result$annotated[]
c(region = sum(values >= -1), overlapping = sum(values >= 0))

}
```

---

`ieegio_sample_data` *Download sample files*

---

## Description

Download sample files

## Usage

```
ieegio_sample_data(file, test = FALSE, cache_ok = TRUE)
```

## Arguments

| | |
|---|---|
| `file` | file to download; set to `NULL` to view all possible files |
| `test` | test whether the sample file exists instead of downloading them; default is `FALSE` |
| `cache_ok` | whether to use cache |

## Value

When `test` is false, returns downloaded file path (character); when `test` is true, returns whether the expected sample exists (logical).

## Examples

```
# list available files
ieegio_sample_data()

# check if file edfPlusD.edf exists
ieegio_sample_data("edfPlusD.edf", test = TRUE)

## Not run:

ieegio_sample_data("edfPlusD.edf")


## End(Not run)
```

---

`imaging-streamlines` *Read and write streamlines*

---

## Description

High-level functions to read and write streamlines, supporting `'TCK'`, `'TRK'`, `'TT'` (read-only), `'VTK'` poly-data (including legacy `'.vtk'`, 'XML'-based `'.vtp'`, 'HDF5'-based `'.vtpb'`)

## Usage

```
read_streamlines(file, ...)

write_streamlines(
  x,
  con,
  format = c("auto", "tck", "trk", "vtk", "vtp", "vtpb"),
  ...
)

as_ieegio_streamlines(x, ...)

## Default S3 method:
as_ieegio_streamlines(x, vox2ras = NULL, ..., class = NULL)

## S3 method for class 'character'
as_ieegio_streamlines(x, sanitize = TRUE, ...)
```

## Arguments

| | |
|---|---|
| `file`, `con` | path to the streamline data |
| `...` | passed to low-level functions accordingly |
| `x` | R object that can be converted into an `ieegio` streamlines instance |

| | |
|---|---|
| format | format to write to file, the file extensions must match with the format |
| vox2ras | volume index to 'RAS' coordinate transform matrix; default is identity matrix and used by 'TRK' format |
| class | additional class to be added to the instance |
| sanitize | whether to discard the tracts in a file that are not lines; default is TRUE. A tract needs at least two points to have any segment, so points with missing coordinates are dropped, and a tract left with fewer than two points is dropped entirely. Reading a file may therefore return fewer tracts than it stores. Set to FALSE to read the file exactly as written |

## Value

read_streamlines and as_ieegio_streamlines returns a streamlines instance.

## Examples

```
# toy example
curve <- function(t) {
  x <- sin(4 * t + sample(300, 1) / 100) + t + sample(seq_along(t)) / length(t) / 10
  y <- cos(sin(t) + 5 * t) + sample(seq_along(t)) / length(t) / 10
  z <- t * 3
  cbind(x, y, z)
}

# 10 lines, each line is represented by nx3 matrix
tracts <- lapply(seq(100, 109), function(n) {
  curve(seq_len(n) / 100)
})

# convert to streamline
x <- as_ieegio_streamlines(tracts)

# Display
print(x)
plot(x, col = 1:10)

if(system.file(package = "r3js") != '') {
  plot(x, method = "r3js")
}

# Subset the first line (transformed)
coords <- x[[1]]$coords
head(coords)

# Save different formats
tdir <- tempfile()
dir.create(tdir, showWarnings = FALSE, recursive = TRUE)

write_streamlines(x, file.path(tdir, "sample.tck"))
write_streamlines(x, file.path(tdir, "sample.trk"))
```

```
write_streamlines(x, file.path(tdir, "sample.trk.gz"))

## Not run:

  # Require Python
  write_streamlines(x, file.path(tdir, "sample.vtk"))
  write_streamlines(x, file.path(tdir, "sample.vtp"))
  write_streamlines(x, file.path(tdir, "sample.vtpb"))


## End(Not run)


# Read formats
y <- read_streamlines(file.path(tdir, "sample.trk"))

# Compare x and y
diffs <- mapply(
  x = as.vector(x),
  y = as.vector(y),
  function(x, y) {
    range(x$coords - y$coords)
  }
)
# Should be floating errors
max(abs(diffs))

unlink(tdir, recursive = TRUE)
```

---

`imaging-surface` *Read and write surface files*

---

## Description

Supports surface geometry, annotation, measurement, and time-series data. Please use the high-level function `read_surface`, which calls other low-level functions internally.

## Usage

```
read_surface(file, format = "auto", type = NULL, ...)

write_surface(
  x,
  con,
  format = c("gifti", "freesurfer", "vtk"),
  type = c("geometry", "annotations", "measurements", "color", "time_series"),
  ...,
  name = 1
```

```
)

io_read_fs(
  file,
  type = c("geometry", "annotations", "measurements"),
  format = "auto",
  name = basename(file),
  ...
)

io_read_gii(file)

io_write_gii(x, con, ...)
```

## Arguments

| | |
|---|---|
| file, con | path the file |
| format | format of the file, for write_surface, this is 'gifti', 'freesurfer', or 'vtk'; for read_surface, use 'niml' to force the NIML reader, or see 'Arguments' section in read.fs.surface (when file type is 'geometry') and read.fs.curv (when file type is 'measurements') |
| type | type of the data; ignored if the file format is 'GIfTI'. For 'FreeSurfer' files, supported types are<br>'geometry' contains positions of mesh vertex nodes and face indices;<br>'annotations' annotation file (usually with file extension 'annot') containing a color look-up table and an array of color keys. These files are used to display discrete values on the surface such as brain atlas;<br>'measurements' measurement file such as 'sulc' and 'curv' files, containing numerical values (often with continuous domain) for each vertex node<br>For AFNI/SUMA NIML files (file name ending with '.niml.dset'), the type is resolved automatically from the dataset when left unspecified: datasets whose dset_type denotes labels or regions of interest, or that carry an AFNI_labeltable, are read as 'annotations'; the rest are read as 'measurements'. See io_read_niml for the low-level reader. |
| ... | for read_surface, the arguments will be passed to io_read_fs if the file is a 'FreeSurfer' file, or to io_read_vtk_polys if the file is a 'VTK' mesh. |
| x | surface (geometry, annotation, measurement) data |
| name | name of the data; for io_read_fs, this argument must be a character, and default is the file name; for write_surface, this argument can be an integer or a character, representing the index or name of the corresponding measurement or annotation column. |

## Value

A surface object container for read_surface, and the file path for write_surface

## Examples

```
library(ieegio)

# geometry
geom_file <- "gifti/GzipBase64/sujet01_Lwhite.surf.gii"

# measurements
shape_file <- "gifti/GzipBase64/sujet01_Lwhite.shape.gii"

# time series
ts_file <- "gifti/GzipBase64/fmri_sujet01_Lwhite_projection.time.gii"

if(ieegio_sample_data(geom_file, test = TRUE)) {

  geometry <- read_surface(ieegio_sample_data(geom_file))
  print(geometry)

  measurement <- read_surface(ieegio_sample_data(shape_file))
  print(measurement)

  time_series <- read_surface(ieegio_sample_data(ts_file))
  print(time_series)

  # merge measurement & time_series into geometry
  merged <- merge(geometry, measurement, time_series)
  print(merged)

  # make sure you install `rgl` package
  plot(merged, name = c("measurements", "Shape001"))

  plot(merged, name = "time_series",
       slice_index = c(1, 11, 21, 31))

}
```

---

`imaging-volume` *Read and write volume data*

---

## Description

Read and write volume data ('MRI', 'CT', etc.) in 'NIfTI' or 'MGH' formats. Please use `read_volume` and `write_volume` for high-level function. These functions will call other low-level functions internally.

## Usage

```
read_volume(file, header_only = FALSE, format = c("auto", "nifti", "mgh"), ...)

write_volume(x, con, format = c("auto", "nifti", "mgh"), ...)

io_read_mgz(file, header_only = FALSE)

io_write_mgz(x, con, ...)

## S3 method for class 'ieegio_volume'
io_write_mgz(x, con, ...)

## S3 method for class 'ieegio_mgh'
io_write_mgz(x, con, ...)

## S3 method for class 'nifti'
io_write_mgz(x, con, ...)

## S3 method for class 'niftiImage'
io_write_mgz(x, con, ...)

## S3 method for class 'ants.core.ants_image.ANTsImage'
io_write_mgz(x, con, ...)

## S3 method for class 'array'
io_write_mgz(x, con, vox2ras = NULL, ...)

io_read_nii(
  file,
  method = c("rnifti", "oro", "ants"),
  header_only = FALSE,
  ...
)

io_write_nii(x, con, ...)

## S3 method for class 'ieegio_nifti'
io_write_nii(x, con, ...)

## S3 method for class 'ants.core.ants_image.ANTsImage'
io_write_nii(x, con, ...)

## S3 method for class 'niftiImage'
io_write_nii(x, con, ...)

## S3 method for class 'nifti'
io_write_nii(x, con, gzipped = NA, ...)
```

```
## S3 method for class 'ieegio_mgh'
io_write_nii(x, con, ...)

## S3 method for class 'array'
io_write_nii(
  x,
  con,
  vox2ras = NULL,
  datatype_code = NULL,
  xyzt_units = c("NIFTI_UNITS_MM", "NIFTI_UNITS_SEC"),
  intent_code = "NIFTI_INTENT_NONE",
  ...,
  gzipped = NA
)
```

## Arguments

| | |
|---|---|
| `file` | file path to read volume data |
| `header_only` | whether to read header data only; default is `FALSE` |
| `format` | format of the file to be written; choices are `'auto'`, `'nifti'` or `'mgh'`; default is to `'auto'` detect the format based on file names, which will save as a 'MGH' file when file extension is `'mgz'` or `'mgh'`, otherwise 'NIfTI' format. We recommend explicitly setting this argument |
| `...` | passed to other methods |
| `x` | volume data (such as 'NIfTI' image, array, or 'MGH') to be saved |
| `con` | file path to store image |
| `vox2ras` | a `4x4` transform matrix from voxel indexing (column, row, slice) to scanner (often 'T1-weighted' image) 'RAS' (right-anterior-superior) coordinate |
| `method` | method to read the file; choices are `'oro'` (using `readNIfTI`), `'rnifti'` (using `readNifti`), and `'ants'` (using `as_ANTsImage`). |
| `gzipped` | for writing `'nii'` data: whether the file needs to be compressed; default is inferred from the file name. When the file ends with `'nii'`, then no compression is used; otherwise the file will be compressed. If the file name does not end with `'nii'` nor `'nii.gz'`, then the file extension will be added automatically. |
| `datatype_code`, `xyzt_units`, `intent_code` | additional flags for 'NIfTI' headers, for advanced users |

## Format

format of the file; default is auto-detection, other choices are `'nifti'` and `'mgh'`;

## Value

Imaging readers return `ieegio_volume` objects. The writers return the file path to where the file is saved to.

## Examples

```
library(ieegio)

nifti_file <- "brain.demosubject.nii.gz"

# Use `ieegio_sample_data(nifti_file)`
#   to download sample data


if( ieegio_sample_data(nifti_file, test = TRUE) ) {

# ---- NIfTI examples -------------------------------------------

file <- ieegio_sample_data(nifti_file)

# basic read
vol <- read_volume(file)

# voxel to scanner RAS
vol$transforms$vox2ras

# to freesurfer surface
vol$transforms$vox2ras_tkr

# to FSL
vol$transforms$vox2fsl

plot(vol, position = c(10, 0, 30))

# ---- using other methods --------------------------------------
# default
vol <- read_volume(file, method = "rnifti", format = "nifti")
vol$header

# lazy-load nifti
vol2 <- read_volume(file, method = "oro", format = "nifti")
vol2$header

## Not run:
# requires additional python environment

# Using ANTsPyx
vol3 <- read_volume(file, method = "ants", format = "nifti")
vol3$header


## End(Not run)

# ---- write ----------------------------------------------------

# write as NIfTI
```

```
f <- tempfile(fileext = ".nii.gz")

write_volume(vol, f, format = "nifti")

# alternative method
write_volume(vol$header, f, format = "nifti")

# write to mgz/mgh
f2 <- tempfile(fileext = ".mgz")

write_volume(vol, f, format = "mgh")

# clean up
unlink(f)
unlink(f2)

# ---- Special case in WebAsssembly --------------------------------
# oro.nifti backend is always used

# Emulate WebAssemply when RNifti is unavailable, using oro.nifti instead
old_opt <- options("ieegio.debug.emscripten" = TRUE)
on.exit({ options(old_opt) }, add = TRUE)


# In WebAssemply, RNifti is not available, using oro.nifti instead
vol <- read_volume(file)

stopifnot(vol$type[[1]] == "oro")

# Cleanup: make sure the options are reset
options(old_opt)

}
```

---

`io-trk` *Read or write 'TCK' streamlines*

---

## Description

Low-level functions; for high-level functions, please use `read_streamlines` or `as_ieegio_streamlines` instead.

Low-level functions, supports compressed files; for high-level functions, please use `read_streamlines` or `as_ieegio_streamlines` instead.

## Usage

```
io_read_tck(file)

io_write_tck(
```

```
  x,
  con,
  datatype = c("Float32LE", "Float32BE", "Float64LE", "Float64BE")
)

io_read_trk(file, half_voxel_offset = TRUE)

io_write_trk(x, con, half_voxel_offset = NA)
```

## Arguments

| | |
|---|---|
| file, con | file path to the streamline file |
| x | imaging-streamlines instance |
| datatype | data storage type to write, default is 'Float32LE', 4-byte little 'endian' float; other choices are 'Float32BE', 'Float64LE', and 'Float64BE' |
| half_voxel_offset | whether to add 0.5 millimeter shift on each side, default is TRUE. See 'Details' for explanation. |

## Details

'TRK' gains popularity due to its ability to store streamline attributes. However, this file format suffer from ambiguous definition in the initial 'TrackVis' implementation. Typically in a medical image file, there might exists a 4-by-4 matrix that maps the volume indices to the corresponding anatomical right-anterior-superior 'RAS' locations. However, the original definition of 'TRK' does not have this. Since version 2, 'TRK' introduced such matrix, but it was interpreted differently. Instead of the volume index space, the source space is conformed 1 millimeter space, with the origin at the first 'voxel' corner instead of the center. Therefore there is a 0.5 mm shift at each direction, and half_voxel_offset is designed to offset this shift.

What has made this issue complicated was that some software, such as 'DSI-studio', seemed to ignore that offset when converting from their own format to the 'TRK' format. If the file is generated in such way, please set half_voxel_offset=FALSE to turn off the offset correction. We always recommend that user store data in 'TCK' format.

## Value

io_read_tck returns a ieegio streamline object, io_write_tck returns the connection or file path.

io_read_trk returns an imaging-streamlines instance.

## Examples

```
# run `ieegio_sample_data("streamlines/CNVII_R.tck")` to
# download sample data

if( ieegio_sample_data("streamlines/CNVII_R.tck", test = TRUE) ) {

  path <- ieegio_sample_data("streamlines/CNVII_R.tck")
```

```
  # Read
  streamlines <- io_read_tck(path)

  plot(streamlines)

  # write
  tfile <- tempfile(fileext = ".tck")
  io_write_tck(streamlines, tfile, datatype = streamlines$header$datatype)

  # verify two files are identical
  digest::digest(file = tfile) == digest::digest(file = path)


  unlink(tfile)

}




# This example uses sample data, run
# `ieegio_sample_data("streamlines/CNVII_R.trk")` to download

if( ieegio_sample_data("streamlines/CNVII_R.trk", test = TRUE) ) {

  path <- ieegio_sample_data("streamlines/CNVII_R.trk")
  tfile <- tempfile(fileext = ".trk")

  # read
  x <- io_read_trk(path)

  # write
  io_write_trk(x, tfile)

  # compare two files
  file.size(path) == file.size(tfile)

  src_raw <- readBin(path, "raw", n = file.size(path))
  dst_raw <- readBin(tfile, "raw", n = file.size(tfile))

  equal_raw <- src_raw == dst_raw

  # Some reserved information are removed
  all(equal_raw[-c(945:947)])

  unlink(tfile)

}
```

## io-tt *Read 'TT' streamline file*

### Description

Writer is not implemented yet. Please save as a 'TCK' file.

### Usage

```
io_read_tt(file)
```

### Arguments

| | |
|---|---|
| file | path to the streamline file |

### Value

An imaging-streamlines instance.

### Examples

```
# This example uses sample data, run
# `ieegio_sample_data("streamlines/CNVII_R.trk")` to download

if( ieegio_sample_data("streamlines/CNVII_R.tt.gz", test = TRUE) ) {

  path <- ieegio_sample_data("streamlines/CNVII_R.tt.gz")

  # read
  x <- io_read_tt(path)

  plot(x)

}
```

## io-vtk-polys *Read or write surface mesh data in 'VTK' format*

### Description

Reads and writes the polygon ('Polys') part of a 'vtkPolyData' object. Like io_read_vtk_streamlines, this reader uses 'Python' 'vtk' package, and supports '.vtk', '.vtp', '.pvtp', '.vtu', '.vtpb' formats.

## Usage

```
io_read_vtk_polys(file, name = basename(file), transform = NULL)

io_write_vtk_polys(x, con, binary = TRUE, transform = 1)
```

## Arguments

| | |
|---|---|
| file, con | file path to the 'VTK' file, the format will be inferred from the file extension (with default '.vtk') |
| name | name of the geometry; default is the file name |
| transform | for io_read_vtk_polys, an optional 4 by 4 matrix indicating the vertex position to scanner 'RAS' transform; default is NULL (identity matrix), i.e. the vertex positions are used as-is. For io_write_vtk_polys, since 'VTK' files cannot store any coordinate system information, this is the transform applied to the vertex positions before writing; the value is either a 4 by 4 matrix, or the index or name of the transform stored in the surface object (default is 1, the first transform); use NULL to write the vertex positions as-is |
| x | an imaging-surface object, or anything that can be converted via as_ieegio_surface |
| binary | for legacy '.vtk' file only, whether to store the data as binary file or 'ASCII' plain text; default is true (binary). |

## Details

'VTK' allows polygons with arbitrary number of nodes as well as triangle strips, while 'ieegio' surface objects only support triangular faces. Therefore the mesh is triangulated automatically whenever needed.

'VTK' files cannot store the vertex position to scanner 'RAS' transform, therefore io_write_vtk_polys applies the transform to the vertex positions before writing, and io_read_vtk_polys reads the vertex positions as-is (with an identity transform).

Vertex-wise attributes stored as point data are imported as well: arrays with one component become measurements, and unsigned character arrays with three or four components (i.e. 'RGB' or 'RGBA' colors) become the vertex color. Other arrays, such as vertex normals, and cell (face-wise) data are ignored.

## Value

io_read_vtk_polys returns an imaging-surface object, while io_write_vtk_polys writes the data to file and returns the file path.

## Examples

```
# This example shows how to convert a `GIfTI` surface to `VTK`

# run `ieegio_sample_data("gifti/GzipBase64/sujet01_Lwhite.surf.gii")`
# to download sample data

geom_file <- "gifti/GzipBase64/sujet01_Lwhite.surf.gii"
```

```
if( requireNamespace("rpymat", quietly = TRUE) &&
    dir.exists(rpymat::env_path()) &&
    ieegio_sample_data(geom_file, test = TRUE) ) {

  surface <- read_surface(ieegio_sample_data(geom_file))

  # write to vtk
  tfile <- tempfile(fileext = ".vtk")
  io_write_vtk_polys(surface, con = tfile)

  # read
  vtk_surface <- io_read_vtk_polys(tfile)

  print(vtk_surface)

  # 0 0
  range(surface$geometry$faces - vtk_surface$geometry$faces)

  # the vertex positions are written in the transformed space
  # 0 0
  range(
    surface$geometry$transforms[[1]] %*% surface$geometry$vertices -
      vtk_surface$geometry$vertices
  )

  unlink(tfile)

}
```

---

`io-vtk-streamlines` *Read or write streamline data in 'VTK' format*

---

## Description

This reader uses 'Python' `'vtk'` package, supports `'.vtk'`, `'.vtp'`, `'.pvtp'`, `'.vtpb'` formats.

## Usage

```
io_read_vtk_streamlines(file)

io_write_vtk_streamlines(x, con, binary = TRUE)
```

## Arguments

| | |
|---|---|
| `file`, `con` | file path to the `'VTK'` file, the format will be inferred from the file extension (with default `'.vtk'`) |
| `x` | An `imaging-streamlines` object |
| `binary` | for legacy `'.vtk'` file only, whether to store the data as binary file or 'ASCII' plain text; default is true (binary). |

## Value

io_read_vtk_streamlines returns an imaging-streamlines object, while io_write_vtk_streamlines writes the data to file

## Examples

```
# This example shows how to convert tck to vtk

# run `ieegio_sample_data("streamlines/CNVII_R.tck")` to
# download sample data

if( ieegio_sample_data("streamlines/CNVII_R.tck", test = TRUE) ) {

  path <- ieegio_sample_data("streamlines/CNVII_R.tck")

  streamlines <- as_ieegio_streamlines(path)

  # write to vtk
  tfile <- tempfile(fileext = ".vtk")
  io_write_vtk_streamlines(streamlines, con = tfile)

  # read
  vtk_streamlines <- io_read_vtk_streamlines(tfile)

  # compare
  plot(streamlines)
  plot(vtk_streamlines)

  # 0 0
  range(streamlines[[1]]$coords - vtk_streamlines[[1]]$coords)

}
```

---

# io_h5_valid *Check whether a 'HDF5' file can be opened for read/write*

---

## Description

Check whether a 'HDF5' file can be opened for read/write

## Usage

```
io_h5_valid(file, mode = c("r", "w"), close_all = FALSE)

io_h5_names(file)
```

### Arguments

| | |
|---|---|
| file | path to file |
| mode | 'r' for read access and 'w' for write access |
| close_all | whether to close all connections or just close current connection; default is false. Set this to TRUE if you want to close all other connections to the file. This only applies to the 'hdf5r' backend; 'h5lite' never holds the file open |

### Value

io_h5_valid returns a logical value indicating whether the file can be opened. io_h5_names returns a character vector of dataset names.

### Examples

```
x <- array(1:27, c(3,3,3))
f <- tempfile()

# No data written to the file, hence invalid
io_h5_valid(f, 'r')

io_write_h5(x, f, 'dset')
io_h5_valid(f, 'w')

# clean up
unlink(f)
```

---

## io_read_ants_transform

*Read* ANTs *transform file*

---

### Description

Reads spatial transformation files in ANTs (Advanced Normalization Tools) format, including affine matrices (.mat) and deformation fields (.h5, .nii.gz).

### Usage

```
io_read_ants_transform(
  file,
  space_from,
  space_to,
  interpretation = c("passive", "active")
)
```

## Arguments

| | |
|---|---|
| file | character string specifying the path to the transform file. Supported formats include:<br>• .mat: ITK/ANTs affine transform (4x4 matrix)<br>• .h5: HDF5 composite transform (may contain affine and/or deformation components)<br>• .nii, .nii.gz: Deformation field images |
| space_from | character string or ieegio_space object identifying the source space. If missing, will be inferred from the filename using BIDS-style from-<space> entity (e.g., "sub-01_from-T1w_to-MNI_xfm.h5" yields "T1w"). |
| space_to | character string or ieegio_space object identifying the target space. If missing, will be inferred from the filename using BIDS-style to-<space> entity. |
| interpretation | character string specifying how to interpret the transform:<br>• "passive" (default): Axis/coordinate frame transform. Represents how coordinate systems relate to each other. This is the typical interpretation for brain imaging registration transforms.<br>• "active": Point transform. Directly transforms point coordinates from source to target space. |

## Details

ANTs transforms operate in LPS (Left-Posterior-Superior) coordinate convention. The returned transform object automatically sets orientation to "LPS" for both source and target spaces.

For composite transforms (e.g., .h5 files containing both affine and deformation components), the function returns a single transform object. Use as_ieegio_transform with a list to combine multiple transforms.

This function requires the rpyANTs package and a configured Python environment.

## Value

An ieegio_transforms object with:

**data** List containing the transform data (matrix for affine, ANTsTransform object for deformation)

**type** "affine" or "deformation"

**interpretation** "active" or "passive"

**space_from** Source space (with "LPS" orientation for ANTs)

**space_to** Target space (with "LPS" orientation for ANTs)

**dimension** Spatial dimension (typically 3)

## BIDS Support

The function can automatically infer space names from BIDS-compliant file names:

- from-<source>: Source space identifier
- to-<target>: Target space identifier

Example: "sub-01_from-T1w_to-MNI152NLin2009cAsym_mode-image_xfm.h5"

## See Also

as_ieegio_transform for converting objects to transforms and chaining transform_orientation for orientation conversion transforms

## Examples

```
## Not run:
# Read an affine transform
xfm <- io_read_ants_transform("sub-01_from-T1w_to-MNI_xfm.mat")

# Explicitly specify spaces
xfm <- io_read_ants_transform(
  "transform.h5",
  space_from = "native",
  space_to = "MNI152"
)

# Read as active (point) transform
xfm <- io_read_ants_transform(
  "transform.mat",
  interpretation = "active"
)

# Chain multiple transforms
xfm1 <- io_read_ants_transform("from-T1w_to-T2w_xfm.mat")
xfm2 <- io_read_ants_transform("from-T2w_to-MNI_xfm.h5")
combined <- as_ieegio_transform(list(xfm1, xfm2))

## End(Not run)
```

---

# io_read_flirt_transform

*Read FSL FLIRT transformation matrix*

---

## Description

Reads a 4x4 affine transformation matrix from an FSL FLIRT output file. FLIRT matrices operate in FSL scaled-voxel coordinate system and require source and reference images to convert to world (RAS) coordinates.

## Usage

```
io_read_flirt_transform(file, space_from, space_to)
```

## Arguments

| | |
|---|---|
| file | character string specifying the path to the FLIRT matrix file. This is a plain text file containing a 4x4 affine matrix. |
| space_from | character string or ieegio_space object identifying the source (moving) space. If missing, will be inferred from the filename using BIDS-style from-<space> entity. |
| space_to | character string or ieegio_space object identifying the target (reference) space. If missing, will be inferred from the filename using BIDS-style to-<space> entity. |

## Details

FLIRT matrices operate in FSL scaled-voxel coordinate system, which is:

- Voxel indices multiplied by voxel sizes (pixdim)
- X-axis inverted if the image has positive sform determinant (neurological convention)

The returned transform has "FSL" orientation for both source and target spaces. To convert to world (RAS) coordinates, use transform_flirt2ras with the source and/or reference images.

FLIRT matrices are **active** transforms: they map point coordinates from the source (moving) image space to the reference (fixed) image space.

## Value

An ieegio_transforms object with:

**data** List containing the 4x4 FLIRT matrix

**type** "affine"

**interpretation** "active" (FLIRT matrices are point transforms)

**space_from** Source space (with "FSL" orientation)

**space_to** Target space (with "FSL" orientation)

**dimension** 3

## BIDS Support

The function can automatically infer space names from BIDS-compliant file names:

- from-<source>: Source space identifier
- to-<target>: Target space identifier

## See Also

transform_flirt2ras for converting to world coordinates io_read_ants_transform for reading ANTs format transforms as_ieegio_transform for converting objects to transforms

## Examples

```
## Not run:
# Read a FLIRT matrix
xfm <- io_read_flirt_transform("source_to_reference.mat")

# Convert to RAS coordinates (requires source and reference images)
xfm_ras <- transform_flirt2ras(xfm, source = "source.nii.gz",
                               reference = "reference.nii.gz")

# Explicitly specify spaces
xfm <- io_read_flirt_transform(
  "transform.mat",
  space_from = "T1w",
  space_to = "MNI152"
)

## End(Not run)
```

---

# io_read_fstarray_or_h5

*Function try to load 'FST' arrays, if not found, read 'HDF5' arrays*

---

## Description

Experimental function; use with caution.

## Usage

```
io_read_fstarray_or_h5(
  fst_path,
  h5_path,
  h5_name,
  fst_need_transpose = FALSE,
  fst_need_drop = FALSE,
  ram = FALSE
)
```

## Arguments

| | |
|---|---|
| fst_path | 'FST' file cache path |
| h5_path | alternative 'HDF5' file path |
| h5_name | 'HDF5' data name |
| fst_need_transpose | does 'FST' data need transpose? |
| fst_need_drop | drop dimensions |
| ram | whether to load to memory directly or perform lazy loading |

## Details

RAVE stores data with redundancy. One electrode data is usually saved with two copies in different formats: 'HDF5' and 'FST', where 'HDF5' is cross-platform and supported by multiple languages such as `Matlab`, `Python`, etc, while 'FST' format is supported by R only, with super high read/write speed. `load_fst_or_h5` checks whether the presence of 'FST' file, if failed, then it reads data from persistent 'HDF5' file.

## Value

If 'FST' cache file exists, returns `LazyFST` object, otherwise returns `LazyH5` instance

---

`io_read_h5` *Lazy Load 'HDF5' File*

---

## Description

Wrapper for class `LazyH5`, which load data with "lazy" mode - only read part of dataset when needed. The underlying 'HDF5' backend is resolved at run-time; see `LazyH5`.

## Usage

```
io_read_h5(file, name, read_only = TRUE, ram = FALSE, quiet = FALSE)
```

## Arguments

| | |
|---|---|
| `file` | 'HDF5' file |
| `name` | `group/data_name` path to dataset (`H5D` data) |
| `read_only` | only used if `ram=FALSE`, whether the returned `LazyH5` instance should be read only |
| `ram` | load data to memory immediately, default is false |
| `quiet` | whether to suppress messages |

## Value

If `ram` is true, then return data as arrays, otherwise return a `LazyH5` instance.

## See Also

`io_write_h5`

## Examples

```
file <- tempfile()
x <- array(1:120, dim = c(4,5,6))

# save x to file with name /group/dataset/1
io_write_h5(x, file, '/group/dataset/1', quiet = TRUE)

# read data
y <- io_read_h5(file, '/group/dataset/1', ram = TRUE)
class(y)   # array

z <- io_read_h5(file, '/group/dataset/1', ram = FALSE)
class(z)   # LazyH5

dim(z)

# clean up
unlink(file)
```

---

`io_write_h5` *Save objects to 'HDF5' file without trivial checks*

---

## Description

Save objects to 'HDF5' file without trivial checks

## Usage

```
io_write_h5(
  x,
  file,
  name,
  chunk = "auto",
  level = 4,
  replace = TRUE,
  new_file = FALSE,
  ctype = NULL,
  quiet = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | an array, a matrix, or a vector |
| `file` | path to 'HDF5' file |
| `name` | path/name of the data; for example, `"group/data_name"` |

| | |
|---|---|
| chunk | chunk size; only honored by the 'hdf5r' backend, as 'h5lite' has no per-dimension chunking |
| level | compress level from 0 - no compression to 10 - max compression |
| replace | should data be replaced if exists |
| new_file | should removing the file if old one exists |
| ctype | data type such as "character", "integer", or "numeric". If set to NULL then automatically detect types. Note for complex data please store separately the real and imaginary parts. |
| quiet | whether to suppress messages, default is false |
| ... | passed to other LazyH5$save |

## Value

Absolute path of the file saved

## See Also

io_read_h5

## Examples

```
file <- tempfile()
x <- array(1:120, dim = 2:5)

# save x to file with name /group/dataset/1
io_write_h5(x, file, '/group/dataset/1', chunk = dim(x))

# load data
y <- io_read_h5(file, '/group/dataset/1')

# read data to memory
y[]

# clean up
unlink(file)
```

---

# LazyFST *R6 Class to Load 'FST' Files*

---

## Description

provides low-level hybrid array loading for 'FST' file; used internally

## Methods

### Public methods:

- LazyFST$open()
- LazyFST$close()
- LazyFST$save()
- LazyFST$new()
- LazyFST$get_dims()
- LazyFST$subset()

`LazyFST$open()`: to be compatible with LazyH5

*Usage:*

```
LazyFST$open(...)
```

*Arguments:*

`...` ignored

*Returns:* none

`LazyFST$close()`: close the connection

*Usage:*

```
LazyFST$close(..., .remove_file = FALSE)
```

*Arguments:*

`...` ignored

`.remove_file` whether to remove the file when garbage collected

*Returns:* none

`LazyFST$save()`: to be compatible with LazyH5

*Usage:*

```
LazyFST$save(...)
```

*Arguments:*

`...` ignored

*Returns:* none

`LazyFST$new()`: constructor

*Usage:*

```
LazyFST$new(file_path, transpose = FALSE, dims = NULL, ...)
```

*Arguments:*

`file_path` where the data is stored

`transpose` whether to load data transposed

`dims` data dimension, only support 1 or 2 dimensions

`...` ignored

`LazyFST$get_dims()`: get data dimension

*Usage:*

```
LazyFST$get_dims(...)
```

*Arguments:*

`...` ignored

*Returns:* vector, dimensions

`LazyFST$subset()`: subset data

*Usage:*

```
LazyFST$subset(i = NULL, j = NULL, ..., drop = TRUE)
```

*Arguments:*

`i, j, ...` index along each dimension

`drop` whether to apply `drop` the subset

*Returns:* subset of data

## Author(s)

Zhengjia Wang

## Examples

```
library(ieegio)

# Data to save, 8 MB
x <- matrix(rnorm(1000000), ncol = 100)

# Save to local disk
f <- tempfile()
io_write_fst(as.data.frame(x), con = f)

# Load via LazyFST
dat <- LazyFST$new(file_path = f, dims = c(10000, 100))

# dat < 1 MB

# Check whether the data is identical
range(dat[] - x)

system.time(dat[,1])

system.time(dat[1:100,])
```

LazyH5 *Lazy 'HDF5' file loader*

## Description

Provides hybrid data structure for 'HDF5' file. The class is not intended for direct-use. Please see `io_read_h5` and `io_write_h5`.

## Public fields

`quiet` whether to suppress messages

## Methods

### Public methods:

- `LazyH5$do_finalize()`
- `LazyH5$print()`
- `LazyH5$new()`
- `LazyH5$save()`
- `LazyH5$open()`
- `LazyH5$close()`
- `LazyH5$subset()`
- `LazyH5$get_dims()`
- `LazyH5$get_type()`

`LazyH5$do_finalize()`: garbage collection method

*Usage:*

```
LazyH5$do_finalize()
```

*Returns:* none

`LazyH5$print()`: overrides print method

*Usage:*

```
LazyH5$print()
```

*Returns:* self instance

`LazyH5$new()`: constructor

*Usage:*

```
LazyH5$new(file_path, data_name, read_only = FALSE, quiet = FALSE)
```

*Arguments:*

`file_path` where data is stored in 'HDF5' format

`data_name` the data stored in the file

`read_only` whether to open the file in read-only mode. It's highly recommended to set this to be true, otherwise the file connection is exclusive.

`quiet` whether to suppress messages, default is false

*Returns:* self instance

`LazyH5$save()`: save data to a 'HDF5' file

*Usage:*

```
LazyH5$save(
  x,
  chunk = "auto",
  level = 7,
  replace = TRUE,
  new_file = FALSE,
  force = TRUE,
  ctype = NULL,
  size = NULL,
  ...
)
```

*Arguments:*

`x` vector, matrix, or array

`chunk` chunk size, length should matches with data dimension

`level` compress level, from 1 to 9

`replace` if the data exists in the file, replace the file or not

`new_file` remove the whole file if exists before writing?

`force` if you open the file in read-only mode, then saving objects to the file will raise error. Use `force=TRUE` to force write data

`ctype` data type, see `mode`, usually the data type of `x`. Try `mode(x)` or `storage.mode(x)` as hints.

`size` deprecated, for compatibility issues

`...` passed to self `open()` method

`LazyH5$open()`: open connection

*Usage:*

```
LazyH5$open(new_dataset = FALSE, robj, ...)
```

*Arguments:*

`new_dataset` only used when the internal pointer is closed, or to write the data

`robj` data array to save

`...` backend-specific arguments; for example, passed to `createDataSet` in **hdf5r**

`LazyH5$close()`: close connection

*Usage:*

```
LazyH5$close(all = TRUE)
```

*Arguments:*

`all` whether to close all connections associated to the data file. If true, then all connections, including access from other programs, will be closed

`LazyH5$subset()`: subset data

*Usage:*

```
LazyH5$subset(..., drop = FALSE, stream = FALSE, envir = parent.frame())
```

*Arguments:*

`drop` whether to apply `drop` the subset

`stream` whether to read partial data at a time

`envir` if `i,j,...` are expressions, where should the expression be evaluated

`i, j, ...` index along each dimension

*Returns:* subset of data

`LazyH5$get_dims()`: get data dimension

*Usage:*

```
LazyH5$get_dims(stay_open = TRUE)
```

*Arguments:*

`stay_open` whether to leave the connection opened

*Returns:* dimension of the array

`LazyH5$get_type()`: get data type

*Usage:*

```
LazyH5$get_type(stay_open = TRUE)
```

*Arguments:*

`stay_open` whether to leave the connection opened

*Returns:* data type, currently only character, integer, raw, double, and complex are available, all other types will yield "unknown"

---

`low-level-read-write` *Low-level file read and write*

---

## Description

Interfaces to read from or write to files with common formats.

## Usage

```
io_read_fst(
  con,
  method = c("proxy", "data_table", "data_frame", "header_only"),
  ...,
  old_format = FALSE
)

io_write_fst(x, con, compress = 50, ...)
```

```
io_read_ini(con, ...)

io_read_json(con, ...)

io_write_json(
  x,
  con = stdout(),
  ...,
  digits = ceiling(-log10(.Machine$double.eps)),
  pretty = TRUE,
  serialize = TRUE
)

io_read_mat(
  con,
  method = c("auto", "R.matlab", "pymatreader", "mat73"),
  verbose = TRUE,
  on_convert_error = c("warning", "error", "ignore"),
  ...
)

io_write_mat(x, con, method = c("R.matlab", "scipy"), ...)

io_read_yaml(con, ...)

io_write_yaml(x, con, ..., sorted = FALSE)
```

## Arguments

| | |
|---|---|
| con | connection or file |
| method | method to read table. For 'fst', the choices are |
| | 'proxy' do not read data to memory, query the table when needed; |
| | 'data_table' read as data.table; |
| | 'data_frame' read as data.frame; |
| | 'header_only' read 'fst' table header. |
| | For 'mat', the choices are |
| | 'auto' automatically try the native option, and then 'pymatreader' if fails; |
| | 'R.matlab' use the native method (provided by readMat); only support 'MAT 5.0' format; |
| | 'pymatreader' use 'Python' library 'pymatreader'; |
| | 'mat73' use 'Python' library 'mat73'. |
| ... | passed to internal function calls |
| old_format | see fst |
| x | data to write to disk |
| compress | compress level from 0 to 100; default is 50 |

| | |
|---|---|
| digits, pretty | for writing numeric values to 'json' format |
| serialize | set to TRUE to serialize the data to 'json' format (with the data types, default); or FALSE to save the values without types |
| verbose | whether to print out the process |
| on_convert_error | for reading 'mat' files with 'Python' modules, the results will be converted to R objects in the end. Not all objects can be converted. This input defines the behavior when the conversion fails; choices are "error", "warning", or "ignore" |
| sorted | whether to sort the list; default is FALSE |

## Value

The reader functions returns the data extracted from files, mostly as R objects, with few exceptions on some 'Matlab' files. When reading a 'Matlab' file requires using 'Python' modules, io_read_mat will try its best effort to convert 'Python' objects to R. However, such conversion might fail. In this case, the result might partially contain 'Python' objects with warnings.

## Examples

```
# ---- fst -----------------------------------------------------------------


f <- tempfile(fileext = ".fst")
x <- data.frame(
  a = 1:10,
  b = rnorm(10),
  c = letters[1:10]
)

io_write_fst(x, con = f)

# default reads in proxy
io_read_fst(f)

# load as data.table
io_read_fst(f, "data_table")

# load as data.frame
io_read_fst(f, "data_frame")

# get header
io_read_fst(f, "header_only")

# clean up
unlink(f)



# ---- json ----------------------------------------------------------------
```

```
f <- tempfile(fileext = ".json")

x <- list(a = 1L, b = 2.3, c = "a", d = 1+1i)

# default is serialize
io_write_json(x, f)

io_read_json(f)

cat(readLines(f), sep = "\n")

# just values
io_write_json(x, f, serialize = FALSE, pretty = FALSE)

io_read_json(f)

cat(readLines(f), sep = "\n")

# clean up
unlink(f)



# ---- Matlab .mat ------------------------------------------------------

## Not run:

f <- tempfile(fileext = ".mat")

x <- list(a = 1L, b = 2.3, c = "a", d = 1+1i)

# save as MAT 5.0
io_write_mat(x, f)

io_read_mat(f)

# require setting up Python environment

io_read_mat(f, method = "pymatreader")

# MAT 7.3 example
sample_data <- ieegio_sample_data("mat_v73.mat")
io_read_mat(sample_data)

# clean up
unlink(f)


## End(Not run)



# ---- yaml -------------------------------------------------------------
```

```
f <- tempfile(fileext = ".yaml")

x <- list(a = 1L, b = 2.3, c = "a")
io_write_yaml(x, f)

io_read_yaml(f)

# clean up
unlink(f)
```

---

`merge.ieegio_surface` *Merge two* `'ieegio'` *surfaces*

---

## Description

Either merge surface objects by attributes or merge geometries

## Usage

```
## S3 method for class 'ieegio_surface'
merge(
  x,
  y,
  ...,
  merge_type = c("attribute", "geometry"),
  merge_space = c("model", "world"),
  transform_index = 1,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| `x, y, ...` | `'ieegio'` surface objects, see `as_ieegio_surface` or `read_surface`. Object `x` must contain geometry information. |
| `merge_type` | type of merge: |
| | `"attribute"` merge `y,...` into x by attributes such as color, measurements, annotations, or time-series data, assuming `x,y,...` all refer to the same geometry, hence the underlying number of vertices should be the same. |
| | `"geometry"` merge `y,...` into x by geometry; this requires the surfaces to merge have geometries and cannot be only surface attributes. Two mesh objects will be merged into one, and face index will be re-calculated. The merge happens in transformed space, Notice the attributes will be ignored and eventually discarded during merge. |

`merge_space` space to merge the geometries; only used when `merge_type` is `"geometry"`. Default is to directly merge the surfaces in `"model"` space, i.e. assuming the surfaces share the same transform; alternatively, if the model to world transforms are different, users can choose to merge in `"world"` space, then all the surfaces will be transformed into world space and mapped back to the model space in `x`

`transform_index` which local-to-world transform to use when merging geometries in the world space; default is the first transform for each surface object. The transform list can be obtained from `surface$geometry$transforms` and `transform_index` indicates the index of the transform matrices. The length of `transform_index` can be either 1 (same for all surfaces) or the length of all the surfaces, (i.e. length of `list(x,y,...)`), when the index needs to be set for each surface respectively. If any index is set to `NA`, then it means no transform is to be applied and that surface will be merged assuming its model space is the world space.

`verbose` whether to verbose the messages

## Value

A merged surface object

## Examples

```
# Construct example geometry
dodecahedron_vert <- matrix(
  ncol = 3, byrow = TRUE,
  c(-0.62, -0.62, -0.62, 0.62, -0.62, -0.62, -0.62, 0.62, -0.62,
    0.62, 0.62, -0.62, -0.62, -0.62, 0.62, 0.62, -0.62, 0.62,
    -0.62, 0.62, 0.62, 0.62, 0.62, 0.62, 0.00, -0.38, 1.00,
    0.00, 0.38, 1.00, 0.00, -0.38, -1.00, 0.00, 0.38, -1.00,
    -0.38, 1.00, 0.00, 0.38, 1.00, 0.00, -0.38, -1.00, 0.00,
    0.38, -1.00, 0.00, 1.00, 0.00, -0.38, 1.00, 0.00, 0.38,
    -1.00, 0.00, -0.38, -1.00, 0.00, 0.38)
)

dodecahedron_face <- matrix(
  ncol = 3L, byrow = TRUE,
  c(1, 11, 2, 1, 2, 16, 1, 16, 15, 1, 15, 5, 1, 5, 20, 1, 20, 19,
    1, 19, 3, 1, 3, 12, 1, 12, 11, 2, 11, 12, 2, 12, 4, 2, 4, 17,
    2, 17, 18, 2, 18, 6, 2, 6, 16, 3, 13, 14, 3, 14, 4, 3, 4, 12,
    3, 19, 20, 3, 20, 7, 3, 7, 13, 4, 14, 8, 4, 8, 18, 4, 18, 17,
    5, 9, 10, 5, 10, 7, 5, 7, 20, 5, 15, 16, 5, 16, 6, 5, 6, 9,
    6, 18, 8, 6, 8, 10, 6, 10, 9, 7, 10, 8, 7, 8, 14, 7, 14, 13)
)

x0 <- as_ieegio_surface(dodecahedron_vert, faces = dodecahedron_face)

plot(x0)
```

```
# ---- merge by attributes ----------------------------------

# point-cloud but with vertex measurements
y1 <- as_ieegio_surface(
  dodecahedron_vert,
  measurements = data.frame(MyVariable = dodecahedron_vert[, 1]),
  transform = diag(c(2,1,0.5,1))
)

plot(y1)

# the geometry of `y1` will be discarded and only attributes
# (in this case, measurements:MyVariable) will be merged to `x`

z1 <- merge(x0, y1, merge_type = "attribute")

plot(z1)

# ---- merge by geometry ------------------------------------

y2 <- as_ieegio_surface(
  dodecahedron_vert + 4, faces = dodecahedron_face,
  transform = diag(c(2, 1, 0.5, 1))
)

plot(y2)

# merge directly in model space: transform matrix of `y2` will be ignored
z2 <- merge(x0, y2, merge_type = "geometry", merge_space = "model")

plot(z2)

# merge x, y2 in the world space where transforms will be respected
z3 <- merge(x0, y2, merge_type = "geometry", merge_space = "world")

plot(z3)
```

---

`merge.ieegio_volume` *Merge* `'ieegio'` *volumes*

---

## Description

Merge volume data into base image. The images must be static 3-dimensional volume data. Currently time-series or 4-dimensional data is not supported.

## Usage

```
## S3 method for class 'ieegio_volume'
merge(x, y, ..., thresholds = 0, reshape = dim(x), na_fill = NA)
```

## Arguments

| | |
|---|---|
| x | base image to be merged |
| y, ... | images to be merged into x |
| thresholds | numerical threshold for y, ..., can be length of one or more, if images to overlay is more than one. The image values lower than the threshold will be trimmed out |
| reshape | output shape, default is the dimension of x; if changed, then the underlay will be sampled into the new shape |
| na_fill | how to handle missing values; default is NA; for compatibility, you might want to set to 0 |

## Value

Merged volume with dimension reshape.

## Examples

```
base_array <- array(0, c(15, 15, 15))
base_array[4:6, 4:6, 4:6] <- runif(27) * 255

# generate a 15x15x15 mask with 1mm spacing
vox2ras1 <- diag(1, 4)
vox2ras1[1:3, 4] <- -5
x <- as_ieegio_volume(base_array, vox2ras = vox2ras1)


# 15x15x15 mask with 0.5mmx1mmx1mm spacing but oblique to `x`
vox2ras2 <- matrix(
  nrow = 4, byrow = TRUE,
  c(
    2, 0.2, -0.1, -3,
    -0.2, 1, 0.4, -4,
    0.3, -0.1, 1, -1,
    0, 0, 0, 1
  )
)
# vox2ras2[1:3, 4] <- c(-3,-4, -1)
base_array[4:6, 4:6, 4:6] <- runif(27) * 255
y <- as_ieegio_volume(base_array, vox2ras = vox2ras2)


# merge y into x and up-sample mask to 64^3 volume
# set to higher number to get better interpolation quality
# Only voxels of y>0 will be merged to x
z <- merge(x, y, reshape = c(64, 64, 64), thresholds = 0)
```

```
# Visualize

oldpar <- par(mfrow = c(1, 3), mar = c(0, 0, 2, 0))

zoom <- 10
crosshair_ras <- c(0, 0, 0)
pixel_width <- 2

plot(x,
     zoom = zoom,
     position = crosshair_ras,
     pixel_width = pixel_width,
     main = "Original - underlay")
plot(y,
     zoom = zoom,
     position = crosshair_ras,
     pixel_width = pixel_width,
     main = "Original - overlay")
plot(
  z,
  zoom = zoom,
  position = crosshair_ras,
  pixel_width = pixel_width,
  main = "Merged & up-sampled")

# reset graphical state
par(oldpar)
```

---

new_space *Define a coordinate space*

---

## Description

Creates an object representing a coordinate space/reference frame used in medical imaging. The orientation defines the anatomical meaning of the coordinate axes.

## Usage

```
new_space(name = "", orientation = ORIENTATION_CODES, dimension = 3, ...)
```

## Arguments

| | |
|---|---|
| name | character string identifying the coordinate space (e.g., "T1w", "MNI152NLin2009cAsym", "scanner"); default is "", a wildcard that indicates arbitrary space |
| orientation | character string specifying the axis orientation convention. Common orientations in brain imaging: |

- "RAS": Right-Anterior-Superior (FreeSurfer, NIfTI default)
- "LAS": Left-Anterior-Superior
- "LPS": Left-Posterior-Superior (DICOM, ANTs, ITK)
- "RPS": Right-Posterior-Superior
- "LPI": Left-Posterior-Inferior
- "RPI": Right-Posterior-Inferior
- "LAI": Left-Anterior-Inferior
- "RAI": Right-Anterior-Inferior

| | |
|---|---|
| dimension | integer dimension of the space (typically 3 for 3D imaging) |
| ... | additional attributes to attach to the space object |

## Details

Orientation codes use three letters to define the positive direction of the x, y, and z axes respectively:

- First letter (x-axis): **L**eft or **R**ight
- Second letter (y-axis): **A**nterior or **P**osterior
- Third letter (z-axis): **S**uperior or **I**nferior

For example, "RAS" means: +x points Right, +y points Anterior (toward face), +z points Superior (toward top of head).

## Value

An S3 object of class "ieegio_space" with attributes orientation and dimension

## Examples

```
# FreeSurfer/NIfTI convention
scanner_space <- new_space("scanner", orientation = "RAS")
print(scanner_space)

# DICOM/ANTs convention
mni_space <- new_space("MNI152NLin2009cAsym", orientation = "LPS", dimension = 3)
format(mni_space)
```

---

# niml_find *Read an* AFNI/SUMA NIML *dataset*

---

## Description

Reads a NIML ('.niml.dset') file into a nested tree of elements. All NIML storage forms are supported: plain text, binary.lsbfirst, binary.msbfirst, base64.lsbfirst, and base64.msbfirst; the file may additionally be gzip compressed. All NIML column types are supported, including 'String', 'Line', 'complex', 'rgb', and 'rgba'.

Most users should call read_surface or read_colormap instead, which build surface annotation and color map objects on top of this function.

## Usage

```
niml_find(x, name, recursive = TRUE, groups = FALSE)

io_read_niml(file)

niml_as_surface(file, type = NULL, name = path_ext_remove(basename(file)))
```

## Arguments

| | |
|---|---|
| x | an 'ieegio_niml' object, or an element node within one |
| name | name of the data; used when the name cannot be inferred from the data file to set surface data names |
| recursive | whether to descend into nested ni_group elements; default is true. Use FALSE to restrict the search to the immediate children, for example to select the data element belonging to a dataset itself rather than the one inside its label table |
| groups | whether to return ni_group elements instead of data elements; default is false |
| file | path to a NIML file |
| type | type of the data table, either 'annotations' for discrete data with look-up color table, or 'measurements' for continuous values; set to NULL or 'auto' for automated detection; default is NULL |

## Value

io_read_niml returns an 'ieegio_niml' object: a list of element nodes. Each node has a name, an attributes character vector, and either children (for ni_group elements) or a value data.frame with one column per ni_type entry and ni_dimen rows. niml_find returns a list of the matching element nodes.

## Examples

```
# Build a small NIML dataset with a nested label table
path <- tempfile(fileext = ".niml.dset")
writeLines(c(
  '<AFNI_dataset dset_type="Node_Label" ni_form="ni_group" >',
  '<SPARSE_DATA ni_type="int" ni_dimen="4" >',
  ' 0 1 2 1',
  '</SPARSE_DATA>',
  '<AFNI_labeltable ni_form="ni_group" >',
  '<SPARSE_DATA ni_type="4*float,int,String" ni_dimen="3" >',
  ' 0 0 0 1 0 "Unknown"',
  ' 1 0 0 1 1 "Left Insula"',
  ' 0 0 1 1 2 "Right Insula"',
  '</SPARSE_DATA>',
  '</AFNI_labeltable>',
  '</AFNI_dataset>'
), path)

x <- io_read_niml(path)
print(x)
```

```
# the data belonging to the dataset itself, not to the label table
dset <- x[[1]]
niml_find(dset, "SPARSE_DATA", recursive = FALSE)[[1]]$value

unlink(path)

# ---- Read as a surface annotation/measurement --------------

# This example requires extra sample data. Please run
# `ieegio_sample_data("niml/rh.std.141.Glasser_HCP.lbl.niml.dset")`
# to download sample NIML data


has_niml_file <- ieegio_sample_data(
  file = "niml/rh.std.141.Glasser_HCP.lbl.niml.dset",
  test = TRUE
)

if (has_niml_file) {

  niml_file <- ieegio_sample_data(
    file = "niml/rh.std.141.Glasser_HCP.lbl.niml.dset"
  )

  surf_file <- ieegio_sample_data("gifti/std.141.rh.inf_200.gii")

  # Read in surface mesh
  mesh <- read_surface(surf_file)

  # read_surface internally uses `niml_as_surface` for niml.dset
  annot <- niml_as_surface(niml_file)

  merged <- merge(mesh, annot)

  plot(merged)

}
```

---

NWBHDF5IO *Creates a* NWBHDF5IO *file container*

---

## Description

Class definition for 'PyNWB' container; use `read_nwb` for construction function.

## Active bindings

opened Whether the container is opened.

## Methods

### Public methods:


- NWBHDF5IO$new()
- NWBHDF5IO$get_handler()
- NWBHDF5IO$open()
- NWBHDF5IO$close()
- NWBHDF5IO$close_linked_files()
- NWBHDF5IO$read()
- NWBHDF5IO$with()
- NWBHDF5IO$clone()


`NWBHDF5IO$new()`: Initialize the class

*Usage:*

```
NWBHDF5IO$new(path = NULL, mode = c("r", "w", "r+", "a", "w-", "x"), ...)
```

*Arguments:*

`path` Path to a `'.nwb'` file

`mode` Mode for opening the file

`...` Other parameters passed to `nwb$NWBHDF5IO`

`NWBHDF5IO$get_handler()`: Get internal file handler. Please make sure you close the handler correctly.

*Usage:*

```
NWBHDF5IO$get_handler()
```

*Returns:* File handler, i.e. 'PyNWB' `NWBHDF5IO` instance.

`NWBHDF5IO$open()`: Open the connections, must be used together with `$close` method. For high-level method, see `$with`

*Usage:*

```
NWBHDF5IO$open()
```

*Returns:* container itself

*Examples:*

```
# low-level method to open NWB file, for safer methods, see
# `container$with()` below

container$open()

data <- container$read()

# process data...

# Make sure the container is closed!
container$close()
```

`NWBHDF5IO$close()`: Close the connections (low-level method, see 'with' method below)

*Usage:*

```
NWBHDF5IO$close(close_links = TRUE)
```

*Arguments:*

`close_links` Whether to close all files linked to from this file; default is true

*Returns:* Nothing

`NWBHDF5IO$close_linked_files()`: Close all opened, linked-to files. `'MacOS'` and `'Linux'` automatically release the linked-to file after the linking file is closed, but `'Windows'` does not, which prevents the linked-to file from being deleted or truncated. Use this method to close all opened, linked-to files.

*Usage:*

```
NWBHDF5IO$close_linked_files()
```

*Returns:* Nothing

`NWBHDF5IO$read()`: Read the `'NWB'` file from the 'IO' source. Please use along with `'$with'` method

*Usage:*

```
NWBHDF5IO$read()
```

*Returns:* `'NWBFile'` container

`NWBHDF5IO$with()`: Safe wrapper for reading and handling `'NWB'` file. See class examples.

*Usage:*

```
NWBHDF5IO$with(expr, quoted = FALSE, envir = parent.frame())
```

*Arguments:*

`expr` R expression to evaluate

`quoted` Whether `expr` is quoted; default is false

`envir` environment for `expr` to evaluate; default is the parent frame (see `parent.frame`)

*Returns:* Whatever results generated by `expr`

*Examples:*

```
container$with({
  data <- container$read()
  # process data
})
```

`NWBHDF5IO$clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
NWBHDF5IO$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## Not run:

# Running this example requires a .nwb file

library(rnwb)
container <- NWBHDF5IO$new(path = file)
container$with({

  data <- container$read()
  electrode_table <- data$electrodes[convert = TRUE]

})

print(electrode_table)


## End(Not run)

## ------------------------------------------------
## Method `NWBHDF5IO$open()`
## ------------------------------------------------

## Not run:

# low-level method to open NWB file, for safer methods, see
# `container$with()` below

container$open()

data <- container$read()

# process data...

# Make sure the container is closed!
container$close()


## End(Not run)

## ------------------------------------------------
## Method `NWBHDF5IO$with()`
## ------------------------------------------------

## Not run:

container$with({
  data <- container$read()
  # process data
})
```

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

---

`plot.ieegio_colormap_continuous` *Plot a color map legend*

---

## Description

Plot a visual legend for an `ieegio_colormap` object.

The continuous method draws a vertical gradient bar with tick marks on the requested axis side (default left); the bar is rendered flush to the opposite edge. Ticks are placed at the minimum, maximum, and zero (when zero falls strictly inside the range).

The discrete method draws vertically stacked labeled color squares. Labels appear to the right of each square. When there are more labels than `max_labels`, the excess are shown as `"... (N omitted)"`.

## Usage

```
## S3 method for class 'ieegio_colormap_continuous'
plot(x, data_range = NULL, side = 2L, las = 1L, n_colors = 256L, ...)

## S3 method for class 'ieegio_colormap_discrete'
plot(x, max_labels = NA, cex = 1, border = NA, ...)
```

## Arguments

| | |
|---|---|
| `x` | an `ieegio_colormap_continuous` or `ieegio_colormap_discrete` object |
| `data_range` | numeric(2); overrides the color bar range. Falls back to `x$data_range`, then to `c(0, 1)` if still missing. |
| `side` | integer; axis side for tick marks (see `axis`). `2` (left, default) places ticks on the left; `4` on the right. |
| `las` | integer; label orientation passed to `axis`. Default `1` (always horizontal). |
| `n_colors` | integer; number of color samples used to render the gradient bar. Default `256`. |
| `...` | currently unused; reserved for future arguments. |
| `max_labels` | integer or `NA`; maximum number of labels to display in the discrete legend. `NA` (default) auto-sizes from the current plot-region height. |
| `cex` | numeric; character expansion for label text. Default `1`. |
| `border` | color for the box border in the discrete legend; `NA` (default) draws no border. |

## Value

Invisibly returns `x`.

## Examples

```
# Continuous legend ---------------------------------------------------
ct <- as_ieegio_colortable(data.frame(
  Key = c(0L, 50L, 100L),
  R = c(0L, 255L, 255L), G = c(0L, 255L, 0L),
  B = c(255L, 0L, 0L),   A = c(255L, 255L, 255L)
))
lut <- as_ieegio_lookup(data.frame(
  Value = c(-3, 0, 3), Scaled = c(0, 0.5, 1)
))
cm_cont <- as_ieegio_colormap(ct, lookup = lut, data_range = c(-3, 3))
grDevices::pdf(NULL)
plot(cm_cont)
grDevices::dev.off()

# Discrete legend -----------------------------------------------------
ct2 <- as_ieegio_colortable(data.frame(
  Key = 1:4,
  R = c(255L, 0L, 0L, 128L), G = c(0L, 200L, 0L, 0L),
  B = c(0L, 0L, 180L, 180L), A = c(255L, 255L, 255L, 255L)
))
lut2 <- as_ieegio_lookup(data.frame(
  Key   = 1:4,
  Label = c("Cortex", "White matter", "CSF", "Hippocampus")
))
cm_disc <- as_ieegio_colormap(ct2, lookup = lut2)
grDevices::pdf(NULL)
plot(cm_disc)
grDevices::dev.off()
```

---

# plot.ieegio_surface *Plot '3D' surface objects*

---

## Description

Plot '3D' surface objects

## Usage

```
## S3 method for class 'ieegio_surface'
plot(
  x,
  method = c("auto", "base", "r3js", "rgl_basic", "rgl_full"),
  transform = 1L,
  name = "auto",
  vlim = NULL,
  col = c("black", "white"),
  slice_index = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| x | 'ieegio_surface' object, see read_surface |
| method | plot method; 'base' for using base-R to plot (requiring package **ravetools**); 'r3js' for rendering a surface viewer using package **r3js**; For **rgl** users, 'basic' for just rendering the surfaces, 'full' for rendering with axes and title |
| transform | which transform to use, can be a 4-by-4 matrix; if the surface contains transform matrix, then this argument can be an integer index of the transform embedded, or the target (transformed) space name; print names(x$transforms) for choices |
| name | attribute and name used for colors, options can be 'color' if the surface has color matrix; c('annotations', varname) for rendering colors from annotations with variable varname; c('measurements', varname) for rendering colors from measurements with variable varname; 'time_series' for plotting time series slices; or "flat" for flat color; default is 'auto', which will plot the first available data. More details see 'Examples'. |
| vlim | when plotting with continuous data (name is measurements or time-series), the value limit used to generate color palette; default is NULL: the range of the values. This argument can be length of 1 ( creating symmetric value range) or 2. If set, then values exceeding the range will be trimmed to the limit |
| col | color or colors to form the color palette when value data is continuous; when name="flat", the last color will be used |
| slice_index | when plotting the name="time_series" data, the slice indices to plot; default is to select a maximum of 4 slices |
| ... | ignored |

## Examples

```
library(ieegio)

# geometry
geom_file <- "gifti/GzipBase64/sujet01_Lwhite.surf.gii"

# measurements
shape_file <- "gifti/GzipBase64/sujet01_Lwhite.shape.gii"

# time series
ts_file <- "gifti/GzipBase64/fmri_sujet01_Lwhite_projection.time.gii"

if (ieegio_sample_data(geom_file, test = TRUE)) {

  geometry <- read_surface(ieegio_sample_data(geom_file))
  geometry$geometry$transforms[[1]] <- diag(c(1, -1, -1, 1))
  measurement <- read_surface(ieegio_sample_data(shape_file))
  time_series <- read_surface(ieegio_sample_data(ts_file))
  ts_demean <- apply(
    time_series$time_series$value,
```

```
  MARGIN = 1L,
  FUN = function(x) {
    x - mean(x)
  }
)
time_series$time_series$value <- t(ts_demean)

# merge measurement & time_series into geometry (optional)
merged <- merge(geometry, measurement, time_series)
print(merged)

# ---- plot method/style -----------------------------------
plot(merged)

# ---- plot data -------------------------------------------

## Measurements or annotations

# the first column of `measurements`
plot(merged, name = "measurements")

# equivalent to
plot(merged, name = list("measurements", 1L))

# equivalent to
measurement_names <- names(merged$measurements$data_table)
plot(merged, name = list("measurements", measurement_names[[1]]))

## Time-series

# automatically select 4 slices, trim the color palette
# from -25 to 25
plot(merged, name = "time_series", vlim = c(-25, 25),
     slice_index = 1L)

plot(
  merged,
  name = "time_series",
  vlim = c(-25, 25),
  slice_index = seq(1, 128, by = 11),
  col = c("#053061", "#2166ac", "#4393c3",
          "#92c5de", "#d1e5f0", "#ffffff",
          "#fddbc7", "#f4a582", "#d6604d",
          "#b2182b", "#67001f"),
  method = "base",
  eye = c(1000, 0, 0),
  up = c(0, 0, 1),
  side = "front",
  mesh_clipping = 0.3,
  ambient_intensity = 0.7
)
```

```
}
```

---

`plot.ieegio_volume` *Plot '3D' volume in anatomical slices*

---

## Description

Plot '3D' volume in anatomical slices

## Usage

```
## S3 method for class 'ieegio_volume'
plot(
  x,
  position = c(0, 0, 0),
  center_position = FALSE,
  which = c("coronal", "axial", "sagittal"),
  slice_index = 1L,
  transform = "vox2ras",
  zoom = 1,
  pixel_width = max(zoom/2, 1),
  col = c("black", "white"),
  alpha = NA,
  crosshair_gap = 4,
  crosshair_lty = 2,
  crosshair_col = "#00FF00A0",
  label_col = crosshair_col,
  continuous = TRUE,
  vlim = NULL,
  add = FALSE,
  main = "",
  axes = FALSE,
  background = col[[1]],
  foreground = col[[length(col)]],
  ...,
  .xdata = x$data
)
```

## Arguments

| | |
|---|---|
| `x` | `'ieegio_volume'` object; see `read_volume` |
| `position` | position in `'RAS'` (right-anterior-superior) coordinate system on which crosshair should focus |

center_position
: whether to center canvas at position, default is FALSE

which
: which slice to plot; choices are "coronal", "axial", and "sagittal"

slice_index
: length of 1: if x has fourth dimension (e.g. 'fMRI'), then which slice index to draw

transform
: which transform to apply, can be a 4-by-4 matrix, an integer or name indicating the matrix in x$transforms; this needs to be the transform matrix from voxel index to 'RAS' (right-anterior-superior coordinate system), often called 'xform', 'sform', 'qform' in 'NIfTI' terms, or 'Norig' in 'FreeSurfer'

zoom
: zoom-in level

pixel_width
: pixel size, ranging from 0.05 to 50; default is the half of zoom or 1, whichever is greater; the unit of pixel_width divided by zoom is milliliter

col
: color palette for continuous x values

alpha
: opacity value if the image is to be displayed with transparency

crosshair_gap
: the cross-hair gap in milliliter

crosshair_lty
: the cross-hair line type

crosshair_col
: the cross-hair color; set to NA to hide

label_col
: the color of anatomical axis labels (i.e. "R" for right, "A" for anterior, and "S" for superior); default is the same as crosshair_col

continuous
: reserved

vlim
: the range limit of the data; default is computed from range of x$data; data values exceeding the range will be trimmed

add
: whether to add the plot to existing underlay; default is FALSE

main, ...
: passed to image

axes
: whether to draw axes; default is FALSE

background, foreground
: background and foreground colors; default is the first and last elements of col

.xdata
: default is x$data, used to speed up the calculation when multiple different angles are to be plotted

## Examples

```
library(ieegio)

nifti_file <- "nifti/rnifti_example.nii.gz"
nifti_rgbfile <- "nifti/rnifti_example_rgb.nii.gz"

# Use
#   `ieegio_sample_data(nifti_file)`
# and
#   `ieegio_sample_data(nifti_rgbfile)`
# to download sample data


if(
```

```
  ieegio_sample_data(nifti_file, test = TRUE) &&
  ieegio_sample_data(nifti_rgbfile, test = TRUE)
) {

# ---- NIfTI examples -------------------------------------------

underlay_path <- ieegio_sample_data(nifti_file)
overlay_path <- ieegio_sample_data(nifti_rgbfile)

# basic read
underlay <- read_volume(underlay_path)
overlay <- read_volume(overlay_path)

par(mfrow = c(1, 3), mar = c(0, 0, 3.1, 0))

ras_position <- c(50, -10, 15)

ras_str <- paste(sprintf("%.0f", ras_position), collapse = ",")

for(which in c("coronal", "axial", "sagittal")) {
  plot(x = underlay, position = ras_position, crosshair_gap = 10,
       crosshair_lty = 2, zoom = 3, which = which,
       main = sprintf("%s T1RAS=[%s]", which, ras_str))
  plot(x = overlay, position = ras_position,
       crosshair_gap = 10, label_col = NA,
       add = TRUE, alpha = 0.9, zoom = 5, which = which)
}


}
```

---

pynwb_module *Install* 'NWB' *via* 'pynwb'

---

## Description

Install 'NWB' via 'pynwb'

## Usage

```
install_pynwb(python_ver = "auto", verbose = TRUE)

pynwb_module(force = FALSE, error_if_missing = TRUE)
```

## Arguments

| | |
|---|---|
| python_ver | 'Python' version, see configure_conda; default is "auto", which is suggested |
| verbose | whether to print the installation messages |

| | |
|---|---|
| force | whether to force-reload the module |
| error_if_missing | whether to raise errors when the module fails to load; default is true |

## Value

A 'Python' module pynwb.

---

`read_bci2000` *Read 'BCI2000' data file*

---

## Description

Read 'BCI2000' data file

## Usage

```
read_bci2000(
  file,
  extract_path = getOption("ieegio.extract_path", NULL),
  header_only = FALSE,
  cache_ok = TRUE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| file | file path to the data file |
| extract_path | location to where the extracted information is to be stored |
| header_only | whether to only load header data |
| cache_ok | whether existing cache should be reused; default is TRUE. This input can speed up reading large data files; set to FALSE to delete cache before importing. |
| verbose | whether to print processing messages; default is TRUE |

## Value

A cached object that is readily to be loaded to memory; see SignalDataCache for class definition.

## Examples

```
if( ieegio_sample_data("bci2k.dat", test = TRUE) ) {
  file <- ieegio_sample_data("bci2k.dat")

  x <- read_bci2000(file)
  print(x)
```

```
  channel <- x$get_channel(1)

  plot(
    channel$time,
    channel$value,
    type = "l",
    main = channel$info$Label,
    xlab = "Time",
    ylab = channel$info$Unit
  )
}
```

---

`read_brainvis` *Read 'BrainVision' data*

---

## Description

Read 'BrainVision' data

## Usage

```
read_brainvis(
  file,
  extract_path = getOption("ieegio.extract_path", NULL),
  header_only = FALSE,
  cache_ok = TRUE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| `file` | file path to the data file |
| `extract_path` | location to where the extracted information is to be stored |
| `header_only` | whether to only load header data |
| `cache_ok` | whether existing cache should be reused; default is `TRUE`. This input can speed up reading large data files; set to `FALSE` to delete cache before importing. |
| `verbose` | whether to print processing messages; default is `TRUE` |

## Value

A cached object that is readily to be loaded to memory; see `SignalDataCache` for class definition.

## Examples

```
if( ieegio_sample_data("brainvis.dat", test = TRUE) ) {
  # ensure the header and marker files are downloaded as well
  ieegio_sample_data("brainvis.vhdr")
  ieegio_sample_data("brainvis.dat")
  file <- ieegio_sample_data("brainvis.vmrk")

  x <- read_brainvis(file)
  print(x)

  x$get_header()

  x$get_channel_table()

  x$get_annotations()

  channel <- x$get_channel(10)

  plot(
    channel$time,
    channel$value,
    type = "l",
    main = channel$info$Label,
    xlab = "Time",
    ylab = channel$info$Unit
  )
}
```

---

`read_colormap` *Read a color map file*

---

## Description

Read a color map or color lookup table from a file. The format is auto-detected from the file extension and content.

## Usage

```
read_colormap(file, type = "auto", ...)
```

## Arguments

| | |
|---|---|
| `file` | path to the color map file |
| `type` | one of `"auto"` (default), `"fs_lut"`, `"fs_annot"`, `"afni"`, `"fsl"`, `"threebrain"` |
| `...` | additional arguments passed to the format-specific reader |

## Value

An ieegio_colormap or ieegio_colortable object.

## Examples

```
# Write a colormap to a temp file and read it back
cm <- as_ieegio_colormap(as_ieegio_colortable(data.frame(
  Key = 1:3,
  R = c(255L, 0L, 0L), G = c(0L, 200L, 0L),
  B = c(0L, 0L, 180L), A = c(255L, 255L, 255L)
)))
tmp <- tempfile(fileext = ".json")
write_colormap(cm, tmp)
cm2 <- read_colormap(tmp)
print(cm2)
unlink(tmp)
```

---

# read_edf *Read 'EDF' or 'BDF' data file*

---

## Description

Read 'EDF' or 'BDF' data file

## Usage

```
read_edf(
  con,
  extract_path = getOption("ieegio.extract_path", NULL),
  header_only = FALSE,
  cache_ok = TRUE,
  begin = 0,
  end = Inf,
  convert = TRUE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| con | file or connection to the data file |
| extract_path | location to where the extracted information is to be stored |
| header_only | whether to only load header data |
| cache_ok | whether existing cache should be reused; default is TRUE. This input can speed up reading large data files; set to FALSE to delete cache before importing. |
| begin, end | begin and end of the data to read |
| convert | whether to convert digital numbers to analog signals; default is TRUE |
| verbose | whether to print processing messages; default is TRUE |

## Value

A cached object that is readily to be loaded to memory; see SignalDataCache for class definition.

## Examples

```
# ---- EDF/BDF(+) ------------------------------------------------------

# Run `ieegio_sample_data("edfPlusD.edf")` to download sample data

# Tun example if the sample data exists
if(ieegio_sample_data("edfPlusD.edf", test = TRUE)) {

  edf_path <- ieegio_sample_data("edfPlusD.edf")

  data <- read_edf(edf_path)

  data$get_header()

  data$get_annotations()

  data$get_channel_table()

  channel <- data$get_channel(1)

  plot(
    channel$time,
    channel$value,
    type = "l",
    main = channel$info$Label,
    xlab = "Time",
    ylab = channel$info$Unit
  )

}
```

---

# read_nsx *Read ('BlackRock') 'NEV' 'NSx' data*

---

## Description

Read ('BlackRock') 'NEV' 'NSx' data

## Usage

```
read_nsx(
  file,
  extract_path = getOption("ieegio.extract_path", NULL),
```

```
  header_only = FALSE,
  cache_ok = TRUE,
  include_waveform = FALSE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| file | file path to the data file |
| extract_path | location to where the extracted information is to be stored |
| header_only | whether to only load header data |
| cache_ok | whether existing cache should be reused; default is TRUE. This input can speed up reading large data files; set to FALSE to delete cache before importing. |
| include_waveform | whether to include 'waveform' data (usually for online spike sorting); default is FALSE |
| verbose | whether to print processing messages; default is TRUE |

## Value

A cached object that is readily to be loaded to memory; see SignalDataCache for class definition.

---

# read_nwb *Read 'NWB' format*

---

## Description

Life cycle: experimental. Read "Neurodata Without Borders" ('NWB' format) file. Unlike other readers read_nwb returns low-level 'Python' class handler via pynwb module.

## Usage

```
read_nwb(file, mode = c("r", "w", "r+", "a", "w-", "x"), ...)
```

## Arguments

| | |
|---|---|
| file | path to 'NWB' file |
| mode | file open mode; default is 'r' (read-only) |
| ... | passed to NWBHDF5IO initialize function |

## Value

A NWBHDF5IO instance

## Examples

```
if(ieegio_sample_data("nwb_sample.nwb", test = TRUE)) {
  file <- ieegio_sample_data("nwb_sample.nwb")

  # Create NWBIO container
  container <- read_nwb(file)

  # Open connection
  container$open()

  # read meta data
  data <- container$read()
  data

  # get `test_timeseries` data
  ts_data <- data$get_acquisition("test_timeseries")
  ts_data

  # read timeseries data into memory
  ts_arr <- ts_data$data[]
  ts_arr

  # Convert Python array to R
  # using `rpymat::py_to_r(ts_arr)` or
  as.numeric(ts_arr)

  # Make sure you close the connection
  container$close()

}


# Requires setting up Python environment
# run `ieegio::install_pynwb()` to set up environment first

## Not run:


# Replicating tutorial
# https://pynwb.readthedocs.io/en/stable/tutorials/general/plot_file.html

library(rpymat)

# Load Python module
pynwb <- import("pynwb")
uuid <- import("uuid")
datetime <- import("datetime")
np <- import("numpy")
tz <- import("dateutil.tz")
```

```
# 2018L is 2018 as integer
session_start_time <- datetime$datetime(
  2018L, 4L, 25L, 2L, 30L, 3L,
  tzinfo=tz$gettz("US/Pacific"))

#  ---- Create NWB file object ------------------------
nwbfile <- pynwb$NWBFile(
  session_description="Mouse exploring a closed field",
  identifier=py_str(uuid$uuid4()),
  session_start_time=session_start_time,
  session_id="session_4321",
  experimenter=py_list(c("Baggins, Frodo")),
  lab="Bag End Laboratory",
  institution="University of Middle Earth at the Shire",
  experiment_description="Thank you Bilbo Baggins.",
  keywords=py_list(c("behavior", "exploration"))
)

# ---- Add subject -----------------------------------
subject <- pynwb$file$Subject(
  subject_id="001",
  age="P90D",
  description="mouse 5",
  species="Mus musculus",
  sex="M"
)

nwbfile$subject <- subject

nwbfile

# ---- Add TimeSeries ----------------------------------
data <- seq(100, 190, by = 10)
time_series_with_rate <- pynwb$TimeSeries(
  name="test_timeseries",
  description="an example time series",
  data=data,
  unit="m",
  starting_time=0.0,
  rate=1.0
)
time_series_with_rate

nwbfile$add_acquisition(time_series_with_rate)

# ---- New Spatial positions ---------------------------------
position_data <- cbind(
  seq(0, 10, length.out = 50),
  seq(0, 9, length.out = 50)
)
position_timestamps = seq(0, 49) / 200
```

```
spatial_series_obj = pynwb$behavior$SpatialSeries(
  name="SpatialSeries",
  description="(x,y) position in open field",
  data=position_data,
  timestamps=position_timestamps,
  reference_frame="(0,0) is bottom left corner",
)
spatial_series_obj

position_obj = pynwb$behavior$Position(
  spatial_series=spatial_series_obj)
position_obj

# ---- Behavior Processing Module ------------------------------
behavior_module <- nwbfile$create_processing_module(
  name="behavior", description="processed behavioral data"
)
behavior_module$add(position_obj)

nwbfile$processing$behavior

# omit some process

# ---- Write --------------------------------------------------
f <- normalizePath(tempfile(fileext = ".nwb"),
                   winslash = "/",
                   mustWork = FALSE)
io <- pynwb$NWBHDF5IO(f, mode = "w")
io$write(nwbfile)
io$close()




## End(Not run)
```

---

`resample_volume` *Down-sample or super-sample volume*

---

## Description

Using nearest-neighbor.

## Usage

```
resample_volume(x, new_dim, na_fill = NA)
```

## Arguments

| | |
|---|---|
| x | image volume |
| new_dim | new dimension |
| na_fill | value to fill if missing |

## Value

A new volume with desired shape

## Examples

```
# ---- Toy example ----------------------------

dm <- c(6, 6, 6)
arr <- array(seq_len(prod(dm)) + 0.5, dm)
orig <- as_ieegio_volume(
  arr, vox2ras = cbind(diag(1, nrow = 4, ncol = 3), c(-dm / 2, 1)))

# resample
downsampled <- resample_volume(orig, new_dim = c(3, 3, 3))
dim(downsampled)

# up-sample on coronal
upsampled <- resample_volume(orig, new_dim = c(20, 20, 24))
dim(upsampled)

par(mfrow = c(2, 2), mar = c(0, 0, 2.1, 0.1))
plot(orig, pixel_width = 0.5, zoom = 20, main = "Original")
plot(downsampled, pixel_width = 0.5, zoom = 20, main = "Down-sampled")
plot(upsampled, pixel_width = 0.5, zoom = 20, main = "Super-sampled")
plot(
  orig,
  main = "Overlay super-sample (diff)",
  col = c("black", "white"),
  pixel_width = 0.5, zoom = 20
)
plot(
  upsampled,
  add = TRUE,
  col = c("white", "black"),
  pixel_width = 0.5, zoom = 20,
  alpha = 0.5
)

# ---- Real example ---------------------------
nifti_file <- "brain.demosubject.nii.gz"

if( ieegio_sample_data(nifti_file, test = TRUE) ) {

  orig <- read_volume(ieegio_sample_data(nifti_file))
  dim(orig)
```

```
  # resample
  downsampled <- resample_volume(orig, new_dim = c(30, 30, 30))
  dim(downsampled)

  # up-sample on coronal
  upsampled <- resample_volume(orig, new_dim = c(300, 300, 64))
  dim(upsampled)

  par(mfrow = c(2, 2), mar = c(0, 0, 2.1, 0.1))
  plot(orig, main = "Original")
  plot(downsampled, main = "Down-sampled")
  plot(upsampled, main = "Super-sampled")
  plot(
    orig,
    main = "Overlay super-sample",
    col = c("black", "white"),
    zoom = 2,
    vlim = c(0, 255)
  )
  plot(
    upsampled,
    add = TRUE,
    col = c("white", "black"),
    zoom = 2,
    alpha = 0.5,
    vlim = c(0, 255)
  )


}
```

---

`resolve_roi_as` *Resolve a region of interest into a chosen representation*

---

## Description

Applies the criteria recorded by `as_ieegio_roi` and converts the result into the requested geometry. Where `as_ieegio_roi` only describes *how* to derive a region, `resolve_roi_as` carries that description out and hands back an object that *is* the region.

## Usage

```
resolve_roi_as(
  x,
  mode = c("auto", "volume", "pointcloud", "streamlines", "surface"),
  ...
)
```

## Arguments

| | |
|---|---|
| x | a region of interest, or anything as_ieegio_roi accepts; non-ROI inputs are converted first, with no criteria attached |
| mode | representation to resolve into: "auto" (default, keep the representation the region already has), "volume", "pointcloud", "surface", or "streamlines" |
| ... | tuning arguments for the chosen mode; see 'Details' |

## Details

The resolved object is always in world ('RAS') coordinates: geometry transforms are applied to the coordinates and the stored transform is reset to the identity, so two resolved regions can be compared directly. The criteria are cleared from the "roi_info" attribute afterwards, since they have been spent and the data no longer holds the values they described.

Not every pair of representations is convertible. Streamlines carry tract connectivity that nothing else does, so mode = "streamlines" accepts only a streamlines region and fails otherwise. The remaining conversions all succeed, though some lose information: a mesh resolved as a point cloud keeps its vertices and drops its faces.

Arguments in ... depend on mode:

"volume" dim and vox2ras give the target grid explicitly; otherwise one is built along the world axes at resolution spacing (default 0.5) around the region. fill_surface fills the interior of a closed mesh rather than only marking the voxels it passes through, and needs the **ravetools** package.

"surface" lambda smooths the surface extracted from a point cloud or volume, which also needs **ravetools**. For streamlines, tube_radius and tube_sides set the thickness and the number of sides of the tube built around each tract.

"pointcloud", "streamlines" no tuning arguments.

A resolved point cloud gains a point_radius entry in its "roi_info", describing the ball to draw around each of its points as min, median, q95, and max. The last three say what radius covers at least half, at least ninety-five percent, and all of the original region; min instead is the radius at which balls on neighboring points begin to overlap. No other representation carries it.

## Value

The resolved region: an 'ieegio' volume, surface, or streamlines object carrying the "ieegio_roi" class and an "roi_info" attribute whose type reports the representation.

## See Also

as_ieegio_roi records the criteria that this function applies.

## Examples

```
# ---- A volume becomes the point cloud of its voxel centers -----------
vox2ras <- rbind(cbind(diag(2, 3), c(-10, -10, -10)), c(0, 0, 0, 1))
mask <- array(0, c(10, 10, 10))
mask[3:5, 3:5, 3:5] <- 1
```

```
roi <- as_ieegio_roi(mask, vox2ras = vox2ras, threshold_lb = 0.5)

# ROI is a volume, resolved as pointcloud
points <- resolve_roi_as(roi, "pointcloud")
print(points)

# covering the voxel reaches its corner, not merely half its spacing
point_radius <- attr(points, "roi_info")$point_radius

print(point_radius)

# Plot the point cloud as spheres with radius of `max`, the union of
# the spheres covers the entire mask
plot(points, cex = point_radius$max)

# ---- Points are sampled back onto a voxel grid -----------------------
box <- as.matrix(expand.grid(-3:3, -2:2, 0:4))
volume <- resolve_roi_as(as_ieegio_roi(box), "volume", resolution = 0.5)
dim(volume)
sum(volume[])

plot(volume, zoom = 10)

# ---- A tract has no thickness, so it is given a tube body ------------
tract <- cbind(seq(0, 10, by = 0.5), rnorm(21), rnorm(21))
roi <- as_ieegio_roi(tract, type = "streamlines")
plot(roi)

tube <- resolve_roi_as(roi, "surface", tube_radius = 0.2, tube_sides = 8)

plot(tube)

# ---- A threshold expression trims a mesh -----------------------------
vertices <- matrix(
  c(0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1),
  ncol = 3, byrow = TRUE
)
faces <- matrix(
  c(1, 2, 3, 1, 2, 4, 1, 3, 4, 2, 3, 4),
  ncol = 3, byrow = TRUE
)
surface <- as_ieegio_surface(
  vertices,
  faces = faces,
  measurements = data.frame(Curv = c(1, 1, 1, -1))
)
roi <- as_ieegio_roi(surface, threshold_expr = .m1 > 0)

# only the face whose three corners all survive is kept
trimmed <- resolve_roi_as(roi, "surface")
ncol(trimmed$geometry$faces)
```

```
# `auto` keeps whatever representation the region already has
attr(resolve_roi_as(roi, "auto"), "roi_info")$type
```

---

`SignalDataCache` *Class definition for signal cache*

---

## Description

This class is an internal abstract class

## Methods

### Public methods:

- `FileCache$get_header()`
- `FileCache$get_annotations()`
- `FileCache$get_channel_table()`
- `FileCache$get_channel()`
- `FileCache$delete()`

`FileCache$get_header()`: Get header information, often small list object

*Usage:*

```
FileCache$get_header(...)
```

*Arguments:*

`...` passed to child methods

`FileCache$get_annotations()`: Get annotation information, often a large table

*Usage:*

```
FileCache$get_annotations(...)
```

*Arguments:*

`...` passed to child methods

`FileCache$get_channel_table()`: Get channel table

*Usage:*

```
FileCache$get_channel_table(...)
```

*Arguments:*

`...` passed to child methods

`FileCache$get_channel()`: Get channel data

*Usage:*

```
FileCache$get_channel(x, ...)
```

*Arguments:*

`x` channel order or label

`...` passed to child methods

*Returns:* Channel signal with time-stamps inheriting class `'ieegio_get_channel'`

`FileCache$delete()`: Delete file cache

*Usage:*

```
FileCache$delete(...)
```

*Arguments:*

`...` passed to child methods

---

`surface_to_surface` *Transform surface between coordinate spaces*

---

## Description

Transforms surface vertex positions from one coordinate space or orientation to another, optionally applying an additional custom transform.

## Usage

```
surface_to_surface(surface, space_from = "", space_to = "", transform = NULL)
```

## Arguments

| | |
|---|---|
| `surface` | an `ieegio_surface` object or file path; see `as_ieegio_surface` for valid inputs |
| `space_from` | source coordinate space; either an `ieegio_space` object (from `new_space`) or a character string; default is empty string |
| `space_to` | target coordinate space; either an `ieegio_space` object or a character string; default is empty string |
| `transform` | optional 4x4 affine transformation matrix or `ieegio_transforms` object to apply; see `as_ieegio_transform` |

## Details

The function handles orientation changes (e.g., `"RAS"` to `"LPS"`) and optional custom transforms. It creates a transform chain consisting of: an affine (orientation alignment from source), the custom transform, and a post-affine (final orientation alignment to target).

If the provided transform has a `"passive"` interpretation, it is automatically converted to an `"active"` interpretation before being applied to the vertex coordinates.

## Value

A transformed `ieegio_surface` object with updated vertex positions and transform metadata

## See Also

as_ieegio_surface for creating surface objects, new_space for defining coordinate spaces, transform_orientation for orientation transforms, volume_to_surface for creating surfaces from volumes

## Examples

```
library(ieegio)

# geometry
geom_file <- "gifti/GzipBase64/sujet01_Lwhite.surf.gii"

if(ieegio_sample_data(geom_file, test = TRUE)) {

surf_ras <- read_surface(ieegio_sample_data(geom_file))
plot(surf_ras)

# ---- Change axis orientation ------------------
# convert from RAS orientation to LPS
surf_lps <- surface_to_surface(
  surf_ras,
  space_from = new_space("", orientation = "RAS"),
  space_to = new_space("", orientation = "LPS")
)
plot(surf_lps)

# validate
lps_verts <- diag(c(-1, -1, 1, 1)) %*% surf_ras$geometry$vertices
range(surf_lps$geometry$vertices - lps_verts)

# ---- Apply transforms ------------------
transform <- matrix(
  byrow = TRUE, nrow = 4,
  c(
    0.5, 0, 0.3, 1,
    0, -1, 0.2, 2,
    0, 0.7, -0.5, 4,
    0, 0, 0, 1
  )
)
surf_stretch <- surface_to_surface(surf_ras, transform = transform)
plot(surf_stretch)

# validate
stretch_verts <- transform %*% surf_ras$geometry$vertices
range(surf_stretch$geometry$vertices - stretch_verts)


}
```

`transform_flirt2ras` *Convert FLIRT transform to world (RAS) coordinates*

## Description

Converts an FSL FLIRT matrix from FSL scaled-voxel coordinates to world (RAS) coordinates. Allows partial conversion by specifying only source, only reference, or both images.

## Usage

```
transform_flirt2ras(transform, source = NULL, reference = NULL)
```

## Arguments

| | |
|---|---|
| `transform` | an `ieegio_transforms` object with FSL orientation (typically from `io_read_flirt_transform`), or a 4x4 matrix |
| `source` | source (moving) image used in FLIRT registration. Can be:<br>• A file path to a NIfTI image<br>• An `ieegio_volume` object<br>• `NULL` to skip source-side conversion |
| `reference` | reference (fixed) image used in FLIRT registration. Can be:<br>• A file path to a NIfTI image<br>• An `ieegio_volume` object<br>• `NULL` to skip reference-side conversion |

## Details

FSL FLIRT matrices operate in a scaled-voxel coordinate system that depends on the image geometry. The conversion to world coordinates uses:

`world_transform` = `ref_vox2ras` %*% `ref_fsl2vox` %*% `flirt` %*% `src_vox2fsl` %*% `src_ras2vox`

Where:

- `src_ras2vox`: Inverse of source image's voxel-to-RAS matrix
- `src_vox2fsl`: Source voxel-to-FSL coordinate transform
- `flirt`: The original FLIRT matrix
- `ref_fsl2vox`: Inverse of reference voxel-to-FSL transform
- `ref_vox2ras`: Reference image's voxel-to-RAS matrix

The FSL coordinate system uses scaled voxels with possible X-axis flip depending on the image's `sform` determinant sign.

### Value

An ieegio_transforms object with updated orientations:

- Both images provided: RAS -> RAS transform
- Source only: RAS -> FSL transform (source side converted)
- Reference only: FSL -> RAS transform (reference side converted)
- Neither: FSL -> FSL transform (unchanged)

### See Also

io_read_flirt_transform for reading FLIRT matrices transform_orientation for general orientation transforms

### Examples

```
## Not run:
# Read FLIRT matrix
xfm <- io_read_flirt_transform("source_to_reference.mat")

# Full conversion to RAS coordinates
xfm_ras <- transform_flirt2ras(xfm,
                               source = "source.nii.gz",
                               reference = "reference.nii.gz")

# Partial conversion (reference side only)
xfm_partial <- transform_flirt2ras(xfm, reference = "reference.nii.gz")

# Using ieegio_volume objects
src_vol <- read_volume("source.nii.gz", header_only = TRUE)
ref_vol <- read_volume("reference.nii.gz", header_only = TRUE)
xfm_ras <- transform_flirt2ras(xfm, source = src_vol, reference = ref_vol)

## End(Not run)
```

---

## transform_orientation *Create transform between coordinate orientations*

---

### Description

Generates an affine transformation to convert coordinates or coordinate frames between different anatomical orientation conventions (e.g., RAS to LPS). Supports all 48 possible 3D orientations including axis permutations, plus FSL scaled-voxel coordinates (which require an image for conversion).

## Usage

```
transform_orientation(
  space_from,
  orientation_from,
  orientation_to,
  interpretation = c("active", "passive"),
  image = NULL
)
```

## Arguments

| | |
|---|---|
| space_from | either an ieegio_space object or a character string identifying the source space. If provided, orientation_from must be omitted (orientation is extracted from the space object). |
| orientation_from | character string specifying the source orientation (e.g., "RAS", "LPS", "FSL"). Only used if space_from is missing. Must be one of the 48 valid orientation codes plus "FSL". |
| orientation_to | character string specifying the target orientation. Must be one of the 48 valid orientation codes plus "FSL". |
| interpretation | character string specifying transform interpretation:<br>• "active" (default): Point transform - transforms point coordinates from one orientation to another. Use this when you have coordinates in the source orientation and want to convert them.<br>• "passive": Axis transform - transforms the coordinate frame/basis vectors. This is the transpose of the active transform. Use this when transforming reference frames or basis vectors. |
| image | optional image for FSL coordinate conversion. Required when either orientation_from or orientation_to is "FSL" (but not both). Can be a file path or an ieegio_volume object. |

## Details

The function creates orthogonal transformations (rotations and reflections) to convert between different anatomical coordinate conventions. For active transforms, the matrix can be directly applied to homogeneous point coordinates. For passive transforms, the matrix transforms coordinate axes/frames instead.

Common orientation codes (first 8):

- RAS, LAS, LPS, RPS, LPI, RPI, LAI, RAI (standard axis order)

Extended orientations (40 more) include axis permutations like:

- PIR, AIL, SAR, etc. (permuted axes)

**FSL Coordinates:** When "FSL" orientation is involved, an image is required for conversion because FSL coordinates are image-dependent (scaled voxels with possible X-flip). Three scenarios are supported:

- FSL -> FSL: Identity transform (no image needed)
- FSL -> RAS/other: Uses vox2ras %*% fsl2vox from image
- RAS/other -> FSL: Uses vox2fsl %*% ras2vox from image

The relationship between active and passive interpretations: passive_matrix = t(active_matrix) for orthogonal transforms.

## Value

An ieegio_transforms object containing a 4x4 affine transformation matrix

## Examples

```
# Active transform: convert point coordinates from RAS to LPS
trans <- transform_orientation(orientation_from = "RAS",
                               orientation_to = "LPS",
                               interpretation = "active")
trans$data[[1]]  # diag(-1, -1, 1, 1)

# Apply to a point
point_ras <- c(10, 20, 30, 1)
point_lps <- trans$data[[1]] %*% point_ras

# Using a space object
space <- new_space("scanner", orientation = "RAS")
trans <- transform_orientation(space_from = space,
                               orientation_to = "LPS")

# Passive transform: transform coordinate axes
trans_passive <- transform_orientation(orientation_from = "RAS",
                                       orientation_to = "LPS",
                                       interpretation = "passive")

# With axis permutation
trans <- transform_orientation(orientation_from = "RAS",
                               orientation_to = "PIR")

## Not run:
# FSL to RAS conversion (requires image)
trans_fsl2ras <- transform_orientation(orientation_from = "FSL",
                                       orientation_to = "RAS",
                                       image = "brain.nii.gz")

## End(Not run)
```

`volume_to_surface` *Create smooth surface from volume mask or data*

## Description

Create smooth surface from volume mask or data

## Usage

```
volume_to_surface(
  volume,
  lambda = 0.2,
  degree = 2,
  threshold_lb = 0.5,
  threshold_ub = NA,
  smooth_method = c("implicit", "explicit", "none"),
  smooth_iterations = 10L,
  max_vertices = 5e+05,
  ...
)
```

## Arguments

| | |
|---|---|
| `volume` | volume object or path to the NIfTI volume files, see `as_ieegio_volume` for details |
| `lambda, degree` | smooth parameters; see `vcg_smooth_implicit` for details. To disable smoothing, set `lambda` to negative or `NA` |
| `threshold_lb, threshold_ub` | threshold of volume, see `vcg_isosurface`; default is any voxel value above 0.5. Voxels strictly between the two thresholds form the mask; voxels that are `NA`, `NaN`, or infinite are invalid and never part of it |
| `smooth_method` | `"implicit"` (default) smooths with `vcg_smooth_implicit` using `lambda` and `degree`; `"explicit"` smooths with `mris_smooth` instead, repeated neighbor averaging whose memory grows only linearly with the surface (with a **ravetools** version that does not have `mris_smooth`, the `"laplace"` type of `vcg_smooth_explicit` is used); `"none"` returns the surface without smoothing |
| `smooth_iterations` | number of averaging rounds when `smooth_method` is `"explicit"`; default is `10` |
| `max_vertices` | used only when `smooth_method` is `"implicit"`, whose memory grows quickly with the surface size: surfaces with more vertices than this are reduced to about this many with `ravetools::vcg_decimate()` before smoothing, which removes vertices from flat regions first and keeps the shape; default is `500000`. Because the smoothing works in mesh steps, the same `lambda` and `degree` smooth a reduced surface more; use a larger value or `Inf` to smooth at full resolution. With a **ravetools** version that does not have `vcg_decimate`, surfaces with more than `20000` vertices are not smoothed, since the implicit smoothing of those versions can crash on large surfaces |

| | |
|---|---|
| ... | passed to as_ieegio_volume |

## Value

A as_ieegio_surface object; the surface is transformed into anatomical space defined by the volume. When no valid voxel lies within the thresholds, the surface has a single vertex at the origin and no face.

## Examples

```
# toy example; in practice, use tha path to the volume
volume <- array(0, dim = rep(30, 3))
volume[11:20, 11:20, 3:28] <- 1
volume[3:28, 11:20, 11:20] <- 1
volume[11:20, 3:28, 11:20] <- 1
vox2ras <- diag(1, 4)

surf <- volume_to_surface(volume, vox2ras = vox2ras)

if(interactive()) {
  plot(surf)
}
```

---

# write_colormap *Write a color map to file*

---

## Description

Write an ieegio_colormap or ieegio_colortable to disk. The default format is **threeBrain**/RAVE JSON, which preserves all information and is readable by both ieegio and **threeBrain**.

## Usage

```
write_colormap(
  x,
  con,
  format = "threebrain",
  gtype = c("auto", "surface", "volume"),
  ...
)
```

## Arguments

| | |
|---|---|
| x | an ieegio_colormap or ieegio_colortable |
| con | file path to write |

| | |
|---|---|
| format | output format; currently only "threebrain" (default) and "fs_lut" are supported |
| gtype | geometry type for the **threeBrain** format: "auto" (default; discrete maps use "volume", continuous maps use "surface"), "surface", or "volume"; ignored for other formats |
| ... | additional arguments |

## Examples

```
# Build a discrete colormap with region labels
ct <- as_ieegio_colortable(data.frame(
  Key = 1:3,
  R = c(255L, 0L, 0L), G = c(0L, 200L, 0L),
  B = c(0L, 0L, 180L), A = c(255L, 255L, 255L)
))
lut <- as_ieegio_lookup(data.frame(
  Key   = 1:3,
  Label = c("Cortex", "White matter", "CSF")
))
cm <- as_ieegio_colormap(ct, lookup = lut)

# Write as threeBrain JSON (default)
tmp_json <- tempfile(fileext = ".json")
write_colormap(cm, tmp_json)

# Write as FreeSurfer LUT
tmp_lut <- tempfile(fileext = ".txt")
write_colormap(cm, tmp_lut, format = "fs_lut")

unlink(c(tmp_json, tmp_lut))
```

---

# write_edf *Write to 'EDF' format*

---

## Description

Currently supports continuous 'EDF+' format with annotations

## Usage

```
as_edf_channel(
  x,
  channel_num,
  sample_rate,
  label = sprintf("Ch%d", channel_num),
  physical_min = NA,
  physical_max = NA,
  is_annotation = NA,
```

```
  transducer_type = "",
  unit = "uV",
  filter = "",
  comment = ""
)

write_edf(
  channels,
  con,
  patient_id = "anomymous",
  recording_id = NULL,
  record_duration = NA,
  physical_min = NA,
  physical_max = NA,
  start_time = Sys.time()
)
```

## Arguments

| | |
|---|---|
| x | channel signals or annotations; for signals, x is a numeric vector; for annotations, x is a data frame with 'timestamp', 'comments', and optionally 'duration' (case sensitive) columns |
| channel_num | channel number, integer |
| sample_rate | sampling frequency |
| label | channel label, default is 'Ch' followed by the channel number for signal channels, or "EDF Annotation" for annotations |
| physical_min, physical_max | range of the channel values when converting from physical unit to digital; default is the range of x |
| is_annotation | whether the channel is annotation |
| transducer_type | transducer type |
| unit | physical unit or dimension; default is 'uV' |
| filter | preliminary filters applied to the signals |
| comment | additional comments (maximum 32 bytes) |
| channels | list of channel data, each element should be generated from as_edf_channel |
| con | file path or binary connection |
| patient_id | patient identifier; default is 'anomymous' |
| recording_id | recording identifier |
| record_duration | duration of each recording chunk: 'EDF' format slices the data into equal-length chunks and writes the data (interleave channels) to file; this is the duration for each chunk, not the entire recording length |
| start_time | start time of the recording; see as.POSIXct |

## Value

`as_edf_channel` returns a channel wrapper (with metadata); `write_edf` writes to the connection and returns nothing

## Examples

```
signal <- sin(seq(0, 10, 0.01))

channels <- list(

  # signal
  as_edf_channel(channel_num = 1, signal,
                 sample_rate = 375.5),

  as_edf_channel(channel_num = 2, signal,
                 sample_rate = 200),

  # annotation
  as_edf_channel(channel_num = 3, data.frame(
    timestamp = c(0, 1, 2),
    comments = c("Start", "half to go", "Finish!")
  ))

)

# write to file
path <- tempfile(fileext = ".edf")
write_edf(con = path, channels = channels)

edf <- read_edf(con = path, extract_path = tempdir())

annot <- edf$get_annotations()
annot


ch1 <- edf$get_channel(1)

# around 1e-5 due to digitization
range(ch1$value[seq_along(signal)] - signal)

ch2 <- edf$get_channel(2)
range(ch2$value[seq_along(signal)] - signal)


plot(ch1$time, ch1$value, type = "l",
     main = "Toy-example")
lines(ch2$time, ch2$value, col = "red")
abline(v = annot$timestamp, col = "gray", lty = 2)

edf$delete()
unlink(path)
```

# Index


as.POSIXct, *99*
as_ANTsImage, *33*
as_edf_channel *(*write_edf*)*, 98
as_ieegio_colormap, 3
as_ieegio_colortable, 4
as_ieegio_lookup, 5
as_ieegio_roi, 6, *24, 25*, *86, 87*
as_ieegio_streamlines, *8*, *35*
as_ieegio_streamlines *(*imaging-streamlines*)*, 27
as_ieegio_surface, *8*, 10, *39*, *58*, *90, 91*
as_ieegio_transform, 13, *43–45*, *90*
as_ieegio_volume, *8*, 14, *18, 19*
as_nifti_header, 16
axis, *69*

burn_curve, 17
burn_volume, *17*, 19

calculate_color, 20
catmull_rom_3d, *17*
colorspace_from_rgb, 22
colorspace_to_rgb *(*colorspace_from_rgb*)*, 22
configure_conda, *75*
convert-fst, 23
convert_fst_to_csv *(*convert-fst*)*, 23
convert_fst_to_hdf5 *(*convert-fst*)*, 23

data.frame, *55*
data.table, *55*
detect_roi_overlap, 23
drop, *51*, *54*

fst, *55*

ieegio_sample_data, 26
image, *74*
imaging-streamlines, 27
imaging-surface, 29
imaging-volume, 31
install_pynwb *(*pynwb_module*)*, 75
io-trk, 35
io-tt, 38
io-vtk-polys, 38
io-vtk-streamlines, 40
io_h5_names *(*io_h5_valid*)*, 41
io_h5_valid, 41
io_read_ants_transform, 42, *45*
io_read_flirt_transform, 44, *92, 93*
io_read_fs *(*imaging-surface*)*, 29
io_read_fst *(*low-level-read-write*)*, 54
io_read_fstarray_or_h5, 46
io_read_gii *(*imaging-surface*)*, 29
io_read_h5, 47, *49*, *52*
io_read_ini *(*low-level-read-write*)*, 54
io_read_json *(*low-level-read-write*)*, 54
io_read_mat *(*low-level-read-write*)*, 54
io_read_mgz *(*imaging-volume*)*, 31
io_read_nii *(*imaging-volume*)*, 31
io_read_niml, *30*
io_read_niml *(*niml_find*)*, 63
io_read_tck *(*io-trk*)*, 35
io_read_trk *(*io-trk*)*, 35
io_read_tt *(*io-tt*)*, 38
io_read_vtk_polys, *30*
io_read_vtk_polys *(*io-vtk-polys*)*, 38
io_read_vtk_streamlines, *38*
io_read_vtk_streamlines *(*io-vtk-streamlines*)*, 40
io_read_yaml *(*low-level-read-write*)*, 54
io_write_fst *(*low-level-read-write*)*, 54
io_write_gii *(*imaging-surface*)*, 29
io_write_h5, *47*, 48, *52*
io_write_json *(*low-level-read-write*)*, 54
io_write_mat *(*low-level-read-write*)*, 54
io_write_mgz *(*imaging-volume*)*, 31
io_write_nii *(*imaging-volume*)*, 31
io_write_tck *(*io-trk*)*, 35
io_write_trk *(*io-trk*)*, 35

io_write_vtk_polys *(io-vtk-polys)*, 38
io_write_vtk_streamlines
*(io-vtk-streamlines)*, 40
io_write_yaml *(low-level-read-write)*, 54

LazyFST, *47*, 49
LazyH5, *47*, *50*, 52
low-level-read-write, 54

merge.ieegio_surface, 58
merge.ieegio_volume, 60
mode, *53*
mris_smooth, *96*

new_space, 62, *90, 91*
niml_as_surface *(niml_find)*, 63
niml_find, 63
NWBHDF5IO, 65, *81*

plot.ieegio_colormap_continuous, 69
plot.ieegio_colormap_discrete
*(plot.ieegio_colormap_continuous)*,
69
plot.ieegio_surface, 70
plot.ieegio_volume, 73
print.ieegio_roi *(as_ieegio_roi)*, 6
print.ieegio_roi_overlap_result
*(detect_roi_overlap)*, 23
pynwb_module, 75

read.fs.curv, *30*
read.fs.surface, *30*
read_bci2000, 76
read_brainvis, 77
read_colormap, *63*, 78
read_edf, 79
read_nsx, 80
read_nwb, *65*, 81
read_streamlines, *35*
read_streamlines *(imaging-streamlines)*,
27
read_surface, *12*, *58*, *63*, *71*
read_surface *(imaging-surface)*, 29
read_volume, *73*
read_volume *(imaging-volume)*, 31
readMat, *55*
readNIfTI, *33*
readNifti, *33*
resample_volume, 84
resolve_roi_as, *8, 9*, *24, 25*, 86

SignalDataCache, *76, 77*, *80, 81*, 89
surface_to_surface, 90

transform_flirt2ras, *45*, 92
transform_orientation, *44*, *91*, 93, *93*

vcg_detect_collision, *25*
vcg_isosurface, *96*
vcg_smooth_explicit, *96*
vcg_smooth_implicit, *96*
volume_to_surface, *91*, 96

write_colormap, 97
write_edf, 98
write_streamlines
*(imaging-streamlines)*, 27
write_surface *(imaging-surface)*, 29
write_volume *(imaging-volume)*, 31