Package: raveio (via r-universe)

November 24, 2024

Type Package

Title File-System Toolbox for RAVE Project

Version 0.9.0.75

Language en-US

Description Includes multiple cross-platform read/write interfaces for 'RAVE' project. 'RAVE' stands for ``R analysis and visualization of human intracranial electroencephalography data''. The whole project aims at providing powerful free-source package that analyze brain recordings from patients with electrodes placed on the cortical surface or inserted into the brain. 'raveio' as part of this project provides tools to read/write neurophysiology data from/to 'RAVE' file structure, as well as several popular formats including 'EDF(+)', 'Matlab', 'BIDS-iEEG', and 'HDF5', etc. Documentation and examples about 'RAVE' project are provided at <`https://openwetware.org/wiki/RAVE`>, and the paper by John F. Magnotti, Zhengjia Wang, Michael S. Beauchamp (2020) <doi:10.1016/j.neuroimage.2020.117341>; see 'citation(``raveio'')' for details.

BugReports `https://github.com/beauchamplab/raveio/issues`

URL `https://beauchamplab.github.io/raveio/`

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.2

SystemRequirements little-endian platform

Imports utils, data.table, edfReader (>= 1.2.1), dipsaus, filearray (>= 0.1.3), fst (>= 0.9.2), glue, globals, hdf5r (>= 1.3.4), R.matlab (>= 3.6.2), R6, stringr (>= 1.4.0), yaml (>= 2.2.1), targets (>= 0.8.0), callr (>= 3.7.0), remotes (>= 2.1.2), promises (>= 1.2.0), threeBrain (>= 0.2.5), rpymat, ravetools, checkmate (>= 2.3.1)

Suggests jsonlite, visNetwork, testthat, knitr, rmarkdown, shiny, shinyWidgets, freesurferformats, rpyANTs, readNSx, later (>= 1.3.0)

Config/pak/sysreqs git libglpk-dev make libhdf5-dev libicu-dev libpng-dev libxml2-dev libssl-dev python3 libzmq3-dev zlib1g-dev

Repository https://rave-ieeg.r-universe.dev

RemoteUrl https://github.com/beauchamplab/raveio

RemoteRef HEAD

RemoteSha 99bc752473c7b1ff20d84de0ab0ccdadb7fa9626

Contents

`ants_coreg` *Register 'CT' or 'MR' images via* 'ANTs'

Description

`ants_coreg` aligns 'CT' to 'MR' images; `ants_mri_to_template` aligns native 'MR' images to group templates

Usage

```
ants_coreg(
  ct_path,
  mri_path,
  coreg_path = NULL,
  reg_type = c("DenseRigid", "Rigid", "SyN", "Affine", "TRSAA", "SyNCC", "SyNOnly"),
  aff_metric = c("mattes", "meansquares", "GC"),
  syn_metric = c("mattes", "meansquares", "demons", "CC"),
```

```
  verbose = TRUE,
  ...
)

cmd_run_ants_coreg(
  subject,
  ct_path,
  mri_path,
 reg_type = c("DenseRigid", "Rigid", "SyN", "Affine", "TRSAA", "SyNCC", "SyNOnly"),
  aff_metric = c("mattes", "meansquares", "GC"),
  syn_metric = c("mattes", "meansquares", "demons", "CC"),
  verbose = TRUE,
  dry_run = FALSE
)

ants_mri_to_template(
  subject,
  template_subject = getOption("threeBrain.template_subject", "N27"),
  preview = FALSE,
  verbose = TRUE,
  ...
)

cmd_run_ants_mri_to_template(
  subject,
  template_subject = getOption("threeBrain.template_subject", "N27"),
  verbose = TRUE,
  dry_run = FALSE
)

ants_morph_electrode(subject, preview = FALSE, dry_run = FALSE)
```

Arguments

`ct_path`, `mri_path`	absolute paths to 'CT' and 'MR' image files
`coreg_path`	registration path, where to save results; default is the parent folder of `ct_path`
`reg_type`	registration type, choices are `'DenseRigid'`, `'Rigid'`, `'Affine'`, `'SyN'`, `'TRSAA'`, `'SyNCC'`, `'SyNOnly'`, or other types; see `ants_registration`
`aff_metric`	cost function to use for linear or 'affine' transform
`syn_metric`	cost function to use for `'SyN'` transform
`verbose`	whether to verbose command; default is true
`...`	other arguments passed to `ants_registration`
`subject`	'RAVE' subject
`dry_run`	whether to dry-run the script and to print out the command instead of executing the code; default is false

`template_subject`	template to map 'MR' images
`preview`	whether to preview results; default is false

Value

Aligned 'CT' will be generated at the `coreg_path` path:

`'ct_in_t1.nii.gz'`	aligned 'CT' image; the image is also re-sampled into 'MRI' space
`'transform.yaml'`	transform settings and outputs
`'CT_IJK_to_MR_RAS.txt'`	transform matrix from volume 'IJK' space in the original 'CT' to the 'RAS' anatomical coordinate in 'MR' scanner; 'affine' transforms only
`'CT_RAS_to_MR_RAS.txt'`	transform matrix from scanner 'RAS' space in the original 'CT' to 'RAS' in 'MR' scanner space; 'affine' transforms only

---

`ants_preprocessing` *Process 'T1' weighted 'MRI' using* `ANTs`

Description

This function is soft-deprecated. Use `yael_preprocess` instead.

Usage

```
ants_preprocessing(
  work_path,
  image_path,
  resample = FALSE,
  verbose = TRUE,
  template_subject = raveio_getopt("threeBrain_template_subject")
)
```

Arguments

`work_path`	working directory, all intermediate images will be stored here
`image_path`	input image path
`resample`	whether to resample the input image before processing
`verbose`	whether to verbose the processing details
`template_subject`	template mapping, default is derived from `raveio_getopt`

Value

Nothing. All images are saved to `work_path`

`archive_subject` *Archive and share a subject*

Description

Archive and share a subject

Usage

```
archive_subject(
  subject,
  path,
 includes = c("orignal_signals", "processed_data", "rave_imaging", "pipelines", "notes",
    "user_generated"),
  config = list(),
  work_path = NULL,
  zip_flags = NULL
)
```

Arguments

`subject`	'RAVE' subject to archive
`path`	path to a zip file to store; if missing or empty, then the path will be automatically created
`includes`	data to include in the archive; default includes all (original raw signals, processed signals, imaging files, stored pipelines, notes, and user-generated exports)
`config`	a list of configurations, including changing subject code, project name, or to exclude cache data; see examples
`work_path`	temporary working path where files are copied; default is temporary path. Set this variable explicitly when temporary path is on external drives (for example, users have limited storage on local drives and cannot hold the entire subject)
`zip_flags`	`zip` flags

Examples

```
# This example requires you to install demo subject

## Not run:


# Basic usage
path <- archive_subject('demo/DemoSubject')

# clean up
unlink(path)
```

```
# Advanced usage: include all the original signals
# and processed data, no cache data, re-name to
# demo/DemoSubjectLite
path <- archive_subject(
  'demo/DemoSubject',
  includes = c("orignal_signals", "processed_data"),
  config = list(
    rename = list(
      project_name = "demo",
      subject_code = "DemoSubjectLite"
    ),
    orignal_signals = list(
      # include all raw signals
      include_all = TRUE
    ),
    processed_data = list(
      include_cache = FALSE
    )
  )
)

# Clean up temporary zip file
unlink(path)


## End(Not run)
```

`as_rave_project` *Convert character to* `RAVEProject` *instance*

Description

Convert character to `RAVEProject` instance

Usage

```
as_rave_project(project, ...)
```

Arguments

`project`	character project name
`...`	passed to other methods

Value

A `RAVEProject` instance

See Also

`RAVEProject`

`as_rave_subject` *Get `RAVESubject` instance from character*

Description

Get `RAVESubject` instance from character

Usage

```
as_rave_subject(subject_id, strict = TRUE, reload = TRUE)
```

Arguments

`subject_id`	character in format `"project/subject"`
`strict`	whether to check if subject directories exist or not
`reload`	whether to reload (update) subject information, default is true

Value

`RAVESubject` instance

See Also

`RAVESubject`

`as_rave_unit` *Convert numeric number into print-friendly format*

Description

Convert numeric number into print-friendly format

Usage

```
as_rave_unit(x, unit, label = "")
```

Arguments

`x`	numeric or numeric vector
`unit`	the unit of `x`
`label`	prefix when printing `x`

Value

Still numeric, but print-friendly class

Examples

```
sp <- as_rave_unit(1024, 'GB', 'Hard-disk space is ')
print(sp, digits = 0)

sp - 12

as.character(sp)

as.numeric(sp)

# Vectorize
sp <- as_rave_unit(c(500,200), 'MB/s', c('Writing: ', 'Reading: '))
print(sp, digits = 0, collapse = '\n')
```

`as_yael_process` *Create a 'YAEL' imaging processing instance*

Description

Image registration across different modals. Normalize brain 'T1'-weighted 'MRI' to template brain and generate subject-level atlas files.

Usage

```
as_yael_process(subject)
```

Arguments

`subject`	character (subject code, or project name with subject code), or `RAVESubject` instance.

Value

A processing instance, see `YAELProcess`

Examples

```
library(raveio)
process <- as_yael_process("testtest2")

# This example requires extra demo data & settings.
## Not run:

# Import and set original T1w MRI and CT
```

```
process$set_input_image("/path/to/T1w_MRI.nii", type = "T1w")
process$set_input_image("/path/to/CT.nii.gz", type = "CT")

# Co-register CT to MRI
process$register_to_T1w(image_type = "CT")

# Morph T1w MRI to 0.5 mm^3 MNI152 template
process$map_to_template(
  template_name = "mni_icbm152_nlin_asym_09b",
  native_type = "T1w"
)


## End(Not run)
```

`backup_file` *Back up and rename the file or directory*

Description

Back up and rename the file or directory

Usage

```
backup_file(path, remove = FALSE, quiet = FALSE)
```

Arguments

`path`	path to a file or a directory
`remove`	whether to remove the original path; default is false
`quiet`	whether not to verbose the messages; default is false

Value

`FALSE` if nothing to back up, or the back-up path if `path` exists

Examples

```
path <- tempfile()
file.create(path)

path2 <- backup_file(path, remove = TRUE)

file.exists(c(path, path2))
unlink(path2)
```

`BlackrockFile` *Class definition to load data from 'BlackRock' 'Micro-systems' files*

Description

Currently only supports minimum file specification version `2.3`. Please contact the package maintainer or 'RAVE' team if older specifications are needed

Value

absolute file path

absolute file paths

nothing

a data frame

a list of spike 'waveform' (without normalization)

a normalized numeric vector (analog signals with `'uV'` as the unit)

Public fields

`block`	character, session block ID

Active bindings

`base_path`	absolute base path to the file
`version`	'NEV' specification version
`electrode_table`	electrode table
`sample_rate_nev_timestamp`	sample rate of 'NEV' data packet time-stamps
`has_nsx`	named vector of 'NSx' availability
`recording_duration`	recording duration of each 'NSx'
`sample_rates`	sampling frequencies of each 'NSx' file

Methods

Public methods:

- `BlackrockFile$print()`
- `BlackrockFile$new()`
- `BlackrockFile$nev_path()`
- `BlackrockFile$nsx_paths()`
- `BlackrockFile$refresh_data()`
- `BlackrockFile$get_epoch()`
- `BlackrockFile$get_waveform()`
- `BlackrockFile$get_electrode()`

- `BlackrockFile$clone()`

Method `print()`**:** print user-friendly messages

Usage:

`BlackrockFile$print()`

Method `new()`**:** constructor

Usage:

`BlackrockFile$new(path, block, nev_data = TRUE)`

Arguments:

`path` the path to 'BlackRock' file, can be with or without file extensions

`block` session block ID; default is the file name

`nev_data` whether to load comments and 'waveforms'

Method `nev_path()`**:** get 'NEV' file path

Usage:

`BlackrockFile$nev_path()`

Method `nsx_paths()`**:** get 'NSx' file paths

Usage:

`BlackrockFile$nsx_paths(which = NULL)`

Arguments:

`which` which signal file to get, or `NULL` to return all available paths, default is `NULL`; must be integers

Method `refresh_data()`**:** refresh and load 'NSx' data

Usage:

`BlackrockFile$refresh_data(force = FALSE, verbose = TRUE, nev_data = FALSE)`

Arguments:

`force` whether to force reload data even if the data has been loaded and cached before

`verbose` whether to print out messages when loading

`nev_data` whether to refresh 'NEV' extended data; default is false

Method `get_epoch()`**:** get epoch table from the 'NEV' comment data packet

Usage:

`BlackrockFile$get_epoch()`

Method `get_waveform()`**:** get 'waveform' of the spike data

Usage:

`BlackrockFile$get_waveform()`

Method `get_electrode()`**:** get electrode data

Usage:

`BlackrockFile$get_electrode(electrode, nstype = NULL)`

Arguments:

`electrode` integer, must be a length of one

`nstype` which signal bank, for example, `'ns3'`, `'ns5'`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`cache_path` *Manipulate cached data on the file systems*

Description

Manipulate cached data on the file systems

Usage

```
cache_root(check = FALSE)

clear_cached_files(subject_code, quiet = FALSE)
```

Arguments

`check`	whether to ensure the cache root path
`subject_code`	subject code to remove; default is missing. If `subject_code` is provided, then only this subject-related cache files will be removed.
`quiet`	whether to suppress the message

Details

'RAVE' intensively uses cache files. If running on personal computers, the disk space might be filled up very quickly. These cache files are safe to remove if there is no 'RAVE' instance running. Function `clear_cached_files` is designed to remove these cache files. To run this function, please make sure that all 'RAVE' instances are shutdown.

Value

`cache_root` returns the root path that stores the 'RAVE' cache data; `clear_cached_files` returns nothing

Examples

```
cache_root()
```

`cache_to_filearray` *Generate and automatically cache a file array*

Description

Avoid repeating yourself

Usage

```
cache_to_filearray(
  fun,
  filebase,
  globals,
  dimension,
  type = "auto",
  partition_size = 1L,
  verbose = FALSE,
  ...
)
```

Arguments

`fun`	function that can be called with no mandatory arguments; the result should be in a matrix or an array
`filebase`	where to store the array
`globals`	names of variables such that any changes should trigger a new evaluation of `fun`. This argument is highly recommended to be set explicitly (with atomic variables) though the function automatically calculates the global variables
`dimension`	what is the supposed dimension, default is automatically calculated from array. If specified explicitly and the file array dimension is inconsistent, a new calculation will be triggered.
`type`	file array type, default is `"auto"`; can be explicitly specified; see `filearray_create`. Inconsistent type will trigger a new calculation.
`partition_size`	file array partition size; default is `1`; set it to `NA` to generate it automatically. Notice inconsistent partition size will not trigger calculation if the key variables remain the same
`verbose`	whether to verbose debug information
`...`	passed to `findGlobals`

Examples

```
c <- 2
b <- list(d = matrix(1:9,3))
filebase <- tempfile()
```

```
f <- function() {
  message("New calculation")
  re <- c + b$d
  dimnames(re) <- list(A=1:3, B = 11:13)

  # `extra` attribute will be saved
  attr(re, "extra") <- "extra meta data"
  re
}

# first time running
arr <- cache_to_filearray( f, filebase = filebase )

# cached, no re-run
arr <- cache_to_filearray( f, filebase = filebase )

# file array object
arr

# read into memory
arr[]

# read extra data
arr$get_header("extra")

# get digest results
arr$get_header("raveio::filearray_cache")

## Clean up this example
unlink(filebase, recursive = TRUE)
```

`catgl` *Print colored messages*

Description

Print colored messages

Usage

```
catgl(..., .envir = parent.frame(), level = "DEBUG", .pal, .capture = FALSE)
```

Arguments

`..., .envir`	passed to `glue`
`level`	passed to `cat2`
`.pal`	see `pal` in `cat2`
`.capture`	logical, whether to capture message and return it without printing

Details

The level has order that sorted from low to high: `"DEBUG"`, `"DEFAULT"`, `"INFO"`, `"WARNING"`, `"ERROR"`, `"FATAL"`. Each different level will display different colors and icons before the message.

You can suppress messages with certain levels by setting 'raveio' options via `raveio_setopt('verbose_level', <level>)`. Messages with levels lower than the threshold will be muffled. See examples.

Value

The message as characters

Examples

```
# ------------------ Basic Styles ---------------------

# Temporarily change verbose level for example
raveio_setopt('verbose_level', 'DEBUG', FALSE)

# debug
catgl('Debug message', level = 'DEBUG')

# default
catgl('Default message', level = 'DEFAULT')

# info
catgl('Info message', level = 'INFO')

# warning
catgl('Warning message', level = 'WARNING')

# error
catgl('Error message', level = 'ERROR')

try({
  # fatal, will call stop and raise error
  catgl('Error message', level = 'FATAL')
}, silent = TRUE)

# ------------------ Muffle messages ---------------------

# Temporarily change verbose level to 'WARNING'
raveio_setopt('verbose_level', 'WARNING', FALSE)

# default, muffled
catgl('Default message')

# message printed for level >= Warning
catgl('Default message', level = 'WARNING')
catgl('Default message', level = 'ERROR')
```

`cmd_run_3dAllineate` *External shell commands for 'RAVE'*

Description

These shell commands are for importing 'DICOM' images to 'Nifti' format, reconstructing cortical surfaces, and align' the CT' to 'MRI'. The commands are only tested on 'MacOS' and 'Linux'. On 'Windows' machines, please use the 'WSL2' system.

Usage

```
cmd_run_3dAllineate(
  subject,
  mri_path,
  ct_path,
  overwrite = FALSE,
  command_path = NULL,
  dry_run = FALSE,
  verbose = dry_run
)

cmd_execute(
  script,
  script_path,
  command = "bash",
  dry_run = FALSE,
  backup = TRUE,
  args = NULL,
  ...
)

cmd_run_r(
  expr,
  quoted = FALSE,
  verbose = TRUE,
  dry_run = FALSE,
  log_file = tempfile(),
  script_path = tempfile(),
  ...
)

cmd_run_dcm2niix(
  subject,
  src_path,
  type = c("MRI", "CT"),
  merge = c("Auto", "No", "Yes"),
  float = c("Yes", "No"),
```

```
  crop = c("No", "Yes", "Ignore"),
  overwrite = FALSE,
  command_path = NULL,
  dry_run = FALSE,
  verbose = dry_run
)

cmd_run_flirt(
  subject,
  mri_path,
  ct_path,
  dof = 6,
 cost = c("mutualinfo", "leastsq", "normcorr", "corratio", "normmi", "labeldiff", "bbr"),
  search = 90,
 searchcost = c("mutualinfo", "leastsq", "normcorr", "corratio", "normmi", "labeldiff",
    "bbr"),
  overwrite = FALSE,
  command_path = NULL,
  dry_run = FALSE,
  verbose = dry_run
)

cmd_run_recon_all(
  subject,
  mri_path,
 args = c("-all", "-autorecon1", "-autorecon2", "-autorecon3", "-autorecon2-cp",
    "-autorecon2-wm", "-autorecon2-pial"),
  work_path = NULL,
  overwrite = FALSE,
  command_path = NULL,
  dry_run = FALSE,
  verbose = dry_run
)

cmd_run_recon_all_clinical(
  subject,
  mri_path,
  work_path = NULL,
  overwrite = FALSE,
  command_path = NULL,
  dry_run = FALSE,
  verbose = dry_run,
  ...
)
```

Arguments

`subject`	characters or a `RAVESubject` instance

`mri_path`	the absolute to 'MRI' volume; must in 'Nifti' format
`ct_path`	the absolute to 'CT' volume; must in 'Nifti' format
`overwrite`	whether to overwrite existing files; default is false
`command_path`	command line path if 'RAVE' cannot find the command binary files
`dry_run`	whether to run in dry-run mode; under such mode, the shell command will not execute. This is useful for debugging scripts; default is false
`verbose`	whether to print out the command script; default is true under dry-run mode, and false otherwise
`script`	the shell script
`script_path`	path to run the script
`command`	which command to invoke; default is `'bash'`
`backup`	whether to back up the script file immediately; default is true
`args`	further arguments in the shell command, especially the 'FreeSurfer' reconstruction command
`...`	passed to `system2`, or additional arguments
`expr`	expression to run as command
`quoted`	whether `expr` is quoted; default is false
`log_file`	where should log file be stored
`src_path`	source of the 'DICOM' or 'Nifti' image (absolute path)
`type`	type of the 'DICOM' or 'Nifti' image; choices are `'MRI'` and `'CT'`
`merge`, `float`, `crop`	`'dcm2niix'` conversion arguments; ignored when the source is in 'Nifti' format
`dof`, `cost`, `search`, `searchcost`	parameters used by 'FSL' `'flirt'` command; see their documentation for details
`work_path`	work path for 'FreeSurfer' command;

Value

A list of data containing the script details:

`script`	script details
`script_path`	where the script should/will be saved
`dry_run`	whether dry-run mode is turned on
`log_file`	path to the log file
`execute`	a function to execute the script

`cmd_run_yael_preprocess`

Process brain images for `'YAEL'`

Description

Aligns 'T1w' with other image types; normalizes 'T1w' 'MRI' to 'MNI152' templates via symmetric non-linear morphs. Create brain custom atlases from templates.

Usage

```
cmd_run_yael_preprocess(
  subject_code,
  t1w_path = NULL,
  ct_path = NULL,
  t2w_path = NULL,
  fgatir_path = NULL,
  preopct_path = NULL,
  flair_path = NULL,
  t1w_contrast_path = NULL,
  register_reversed = FALSE,
 normalize_template = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09c"),
  run_recon_all = TRUE,
  dry_run = FALSE,
  verbose = TRUE
)

yael_preprocess(
  subject_code,
  t1w_path = NULL,
  ct_path = NULL,
  t2w_path = NULL,
  fgatir_path = NULL,
  preopct_path = NULL,
  flair_path = NULL,
  t1w_contrast_path = NULL,
  register_policy = c("auto", "all"),
  register_reversed = FALSE,
  normalize_template = "mni_icbm152_nlin_asym_09a",
  normalize_policy = c("auto", "all"),
 normalize_back = ifelse(length(normalize_template) >= 1, normalize_template[[1]], NA),
  atlases = list(),
  add_surfaces = FALSE,
  verbose = TRUE
)
```

Arguments

`subject_code`	'RAVE' subject code
`t1w_path`	(required) 'T1' weighted 'MRI' path
`ct_path`	(optional in general but mandatory for electrode localization) post-surgery 'CT' path
`t2w_path`	(optional) 'T2' weighted 'MRI' path
`fgatir_path`	(optional) 'fGATIR' (fast gray-matter acquisition 'T1' inversion recovery) image path
`preopct_path`	(optional) pre-surgery 'CT' path
`flair_path`	(optional) 'FLAIR' (fluid-attenuated inversion recovery) image path
`t1w_contrast_path`	(optional) 'T1' weighted 'MRI' with contrast (usually used to show the blood vessels)
`register_reversed`	direction of the registration; `FALSE` (default) registers other images (such as post-surgery 'CT' to 'T1'); set to `FALSE` if you would like the `'T1'` to be registered into other images. Since 'YAEL' does not re-sample the images, there is no essential difference on the final registration results
`normalize_template`	names of the templates which the native 'T1' images will be normalized into
`run_recon_all`	whether to run `'FreeSurfer'` reconstruction; default is true
`dry_run`	whether to dry-run the script and check if error exists before actually execute the scripts.
`verbose`	whether to print out the progress; default is `TRUE`
`register_policy`	whether images should be registered with `'T1w'` image; default is `"auto"`: automatically run registration algorithm if missing; alternative is `"all"`: force the registration algorithm even if mapping files exist
`normalize_policy`	normalization policy; similar to `register_policy` but is applied to normalization. Default is `"auto"`: automatically run normalization when the mapping is missing, and skip if exists; alternative is `"all"`: force to run the normalization.
`normalize_back`	length of one (select from `normalize_template`), which template is to be used to generate native brain mask and transform matrices
`atlases`	a named list: the names must be template names from `normalize_template` and the values must be directories of atlases of the corresponding templates (see 'Examples').
`add_surfaces`	Whether to add surfaces for the subject; default is `FALSE`. The surfaces are created by reversing the normalization from template brain, hence the results will not be accurate. Enable this option only if cortical surface estimation is not critical.

Value

Nothing, a subject imaging folder will be created under 'RAVE' raw folder

Examples

```
## Not run:


# For T1 preprocessing only
yael_preprocess(
  subject_code = "patient01",
  t1w_path = "/path/to/T1.nii or T1.nii.gz",

  # normalize T1 to all 2009 MNI152-Asym brains (a,b,c)
  normalize_template = c(
    "mni_icbm152_nlin_asym_09a",
    "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"
  ),

  # only normalize if not exists
  normalize_policy = "auto",

  # use MNI152b to create native processing folder
  normalize_back = "mni_icbm152_nlin_asym_09b",

  # Atlases generated from different templates have different
  # coordinates, hence both folder path and template names must be
  # provided
  atlases = list(
    mni_icbm152_nlin_asym_09b = "/path/to/atlas/folder1",
    mni_icbm152_nlin_asym_09c = "/path/to/atlas/folder2"
  )

)

# For T1 and postop CT coregistration only
yael_preprocess(
  subject_code = "patient01",
  t1w_path = "/path/to/T1.nii or T1.nii.gz",
  ct_path = "/path/to/CT.nii or CT.nii.gz",

  # No normalization
  normalize_template = NULL,
  normalize_back = NA

)

# For both T1 and postop CT coregistration and T1 normalization
yael_preprocess(
  subject_code = "patient01",
  t1w_path = "/path/to/T1.nii or T1.nii.gz",
  ct_path = "/path/to/CT.nii or CT.nii.gz",

  normalize_template = c(
    "mni_icbm152_nlin_asym_09a",
```

```
    "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"
  ),

  normalize_policy = "auto",

  normalize_back = "mni_icbm152_nlin_asym_09b",

  atlases = list(
    mni_icbm152_nlin_asym_09b = "/path/to/atlas/folder1",
    mni_icbm152_nlin_asym_09c = "/path/to/atlas/folder2"
  )

)



## End(Not run)
```

`collapse2` *Collapse high-dimensional tensor array*

Description

Collapse high-dimensional tensor array

Usage

```
collapse2(x, keep, method = c("mean", "sum"), ...)

## S3 method for class 'FileArray'
collapse2(x, keep, method = c("mean", "sum"), ...)

## S3 method for class 'Tensor'
collapse2(x, keep, method = c("mean", "sum"), ...)

## S3 method for class 'array'
collapse2(x, keep, method = c("mean", "sum"), ...)
```

Arguments

`x`	R array, `FileArray-class`, or `Tensor` object
`keep`	integer vector, the margins to keep
`method`	character, calculates mean or sum of the array when collapsing
`...`	passed to other methods

Value

A collapsed array (or a vector or matrix), depending on keep

See Also

`collapse`

Examples

```
x <- array(1:16, rep(2, 4))

collapse2(x, c(3, 2))

# Alternative method, but slower when `x` is a large array
apply(x, c(3, 2), mean)

# filearray
y <- filearray::as_filearray(x)

collapse2(y, c(3, 2))

collapse2(y, c(3, 2), "sum")

# clean up
y$delete(force = TRUE)
```

`collapse_power` *Collapse power array with given analysis cubes*

Description

Collapse power array with given analysis cubes

Usage

```
collapse_power(x, analysis_index_cubes)

## S3 method for class 'array'
collapse_power(x, analysis_index_cubes)

## S3 method for class 'FileArray'
collapse_power(x, analysis_index_cubes)
```

Arguments

`x`	a `FileArray-class` array, must have 4 modes in the following sequence `Frequency`, `Time`, `Trial`, and `Electrode`
`analysis_index_cubes`	a list of analysis indices for each mode

Value

a list of collapsed (mean) results

`freq_trial_elec`	collapsed over time-points
`freq_time_elec`	collapsed over trials
`time_trial_elec`	collapsed over frequencies
`freq_time`	collapsed over trials and electrodes
`freq_elec`	collapsed over trials and time-points
`freq_trial`	collapsed over time-points and electrodes
`time_trial`	collapsed over frequencies and electrodes
`time_elec`	collapsed over frequencies and trials
`trial_elec`	collapsed over frequencies and time-points
`freq`	power per frequency, averaged over other modes
`time`	power per time-point, averaged over other modes
`trial`	power per trial, averaged over other modes

Examples

```
if(!is_on_cran()) {

# Generate a 4-mode tensor array
x <- filearray::filearray_create(
  tempfile(), dimension = c(16, 100, 20, 5),
  partition_size = 1
)
x[] <- rnorm(160000)
dnames <- list(
  Frequency = 1:16,
  Time = seq(0, 1, length.out = 100),
  Trial = 1:20,
  Electrode = 1:5
)
dimnames(x) <- dnames

# Collapse array
results <- collapse_power(x, list(
  overall = list(),
  A = list(Trial = 1:5, Frequency = 1:6),
  B = list(Trial = 6:10, Time = 1:50)
))

# Plot power over frequency and time
groupB_result <- results$B


image(t(groupB_result$freq_time),
      x = dnames$Time[groupB_result$cube_index$Time],
      y = dnames$Frequency[groupB_result$cube_index$Frequency],
```

```
      xlab = "Time (s)",
      ylab = "Frequency (Hz)",
      xlim = range(dnames$Time))

x$delete(force = TRUE)


}
```

`compose_channel` *Compose a "phantom" channel from existing electrodes*

Description

In some cases, for example, deep-brain stimulation ('DBS'), it is often needed to analyze averaged electrode channels from segmented 'DBS' leads, or create bipolar contrast between electrode channels, or to generate non-equally weighted channel averages for 'Laplacian' reference. `compose_channel` allows users to generate a phantom channel that does not physically exist, but is treated as a normal electrode channel in 'RAVE'.

Usage

```
compose_channel(
  subject,
  number,
  from,
  weights = rep(1/length(from), length(from)),
  normalize = FALSE,
  force = FALSE,
  label = sprintf("Composed-%s", number),
 signal_type = c("auto", "LFP", "Spike", "EKG", "Audio", "Photodiode", "Unknown")
)
```

Arguments

`subject`	'RAVE' subject
`number`	new channel number, must be positive integer, cannot be existing electrode channel numbers
`from`	a vector of electrode channels that is used to compose this new channel, must be non-empty; see `weights` if these channels are not equally weighted.
`weights`	numerical weights used on each `from` channels; the length of `weights` must equals to the length of `from`; default is equally weighted for each channel (mean of `from` channels).
`normalize`	whether to normalize the weights such that the composed channel has the same variance as `from` channels; default is false

`force`	whether to overwrite existing composed channel if it exists; default is false. By specifying `force=TRUE`, users are risking breaking the data integrity since any analysis based on the composed channel is no longer reproducible. Also users cannot overwrite original channels under any circumstances.
`label`	the label for the composed channel; will be stored at `'electrodes.csv'`
`signal_type`	signal type of the composed channel; default is `'auto'` (same as the first `from` channel); other choices see `SIGNAL_TYPES`

Value

Nothing

Examples

```
library(raveio)


# Make sure demo subject exists in this example, just want to make
# sure the example does not error out
if(
  interactive() && "demo" %in% get_projects() &&
  "DemoSubject" %in% as_rave_project('demo')$subjects() &&
  local({
    subject <- as_rave_subject("demo/DemoSubject")
    !100 %in% subject$electrodes
  })
) {

  # the actual example code:
  # new channel 100 = 2 x channel 14 - (channe 15 + 16)
  compose_channel(
    subject = "demo/DemoSubject",
    number = 100,
    from = c(14, 15, 16),
    weights = c(2, -1, -1),
    normalize = FALSE
  )

}
```

`convert-fst` *Convert 'fst' files to other formats*

Description

'HDF5', 'csv' are common file formats that can be easily 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.

`convert_blackrock` *Convert 'BlackRock' 'NEV/NSx' files*

Description

Convert 'BlackRock' 'NEV/NSx' files

Usage

```
convert_blackrock(
  file,
  block = NULL,
  subject = NULL,
  to = NULL,
  epoch = c("comment", "digital_inputs", "recording", "configuration", "log",
    "button_trigger", "tracking", "video_sync"),
  format = c("mat", "hdf5"),
  header_only = FALSE,
  ...
)
```

Arguments

`file`	path to any 'NEV/NSx' file
`block`	the block name, default is file name
`subject`	subject code to save the files; default is `NULL`

`to`	save to path, must be a directory; default is under the file path. If `subject` is provided, then the default is `subject` raw directory path
`epoch`	what type of events should be included in epoch file; default include comment, digital inputs, recording trigger, configuration change, log comment, button trigger, tracking, and video trigger.
`format`	output format, choices are `'mat'` or `'hdf5'`
`header_only`	whether just to generate channel and epoch table; default is false
`...`	ignored for enhanced backward compatibility

Value

The results will be stored in directory specified by `to`. Please read the output message carefully.

`convert_electrode_table_to_bids`

Convert electrode table

Description

Convert electrode table

Usage

```
convert_electrode_table_to_bids(
  subject,
  space = c("ScanRAS", "MNI305", "fsnative")
)
```

Arguments

`subject`	'RAVE' subject
`space`	suggested coordinate space, notice this argument might not be supported when `'FreeSurfer'` reconstruction is missing.

Value

A list of table in data frame and a list of meta information

`dir_create2` *Force creating directory with checks*

Description

Force creating directory with checks

Usage

```
dir_create2(x, showWarnings = FALSE, recursive = TRUE, check = TRUE, ...)
```

Arguments

`x`	path to create
`showWarnings, recursive, ...`	passed to `dir.create`
`check`	whether to check the directory after creation

Value

Normalized path

Examples

```
path <- file.path(tempfile(), 'a', 'b', 'c')

# The following are equivalent
dir.create(path, showWarnings = FALSE, recursive = TRUE)

dir_create2(path)
```

`ECoGTensor` *'iEEG/ECoG' Tensor class inherit from* `Tensor`

Description

Four-mode tensor (array) especially designed for 'iEEG/ECoG' data. The Dimension names are: `Trial`, `Frequency`, `Time`, and `Electrode`.

Value

a data frame with the dimension names as index columns and `value_name` as value column

an `ECoGTensor` instance

Super class

`raveio::Tensor` -> `ECoGTensor`

Methods

Public methods:

- `ECoGTensor$flatten()`
- `ECoGTensor$new()`

Method `flatten()`**:** converts tensor (array) to a table (data frame)

Usage:

```
ECoGTensor$flatten(include_index = TRUE, value_name = "value")
```

Arguments:

`include_index` logical, whether to include dimension names

`value_name` character, column name of the value

Method `new()`**:** constructor

Usage:

```
ECoGTensor$new(
  data,
  dim,
  dimnames,
  varnames,
  hybrid = FALSE,
  swap_file = temp_tensor_file(),
  temporary = TRUE,
  multi_files = FALSE,
  use_index = TRUE,
  ...
)
```

Arguments:

`data` array or vector

`dim` dimension of data, mush match with `data`

`dimnames` list of dimension names, equal length as `dim`

`varnames` names of `dimnames`, recommended names are: `Trial`, `Frequency`, `Time`, and `Electrode`

`hybrid` whether to enable hybrid mode to reduce RAM usage

`swap_file` if hybrid mode, where to store the data; default stores in `raveio_getopt('tensor_temp_path')`

`temporary` whether to clean up the space when exiting R session

`multi_files` logical, whether to use multiple files instead of one giant file to store data

`use_index` logical, when `multi_files` is true, whether use index dimension as partition number

`...` further passed to `Tensor` constructor

Author(s)

Zhengjia Wang

`export_table` *Export data frame to different common formats*

Description

Stores and load data in various of data format. See 'Details' for limitations.

Usage

```
export_table(
  x,
  file,
  format = c("auto", "csv", "csv.zip", "h5", "fst", "json", "rds", "yaml"),
  ...
)

import_table(
  file,
  format = c("auto", "csv", "csv.zip", "h5", "fst", "json", "rds", "yaml"),
  ...
)
```

Arguments

`x`	data table to be saved to `file`
`file`	file to store the data
`format`	data storage format, default is `'auto'` (infer from the file extension); other choices are `'csv'`, `'csv.zip'`, `'h5'`, `'fst'`, `'json'`, `'rds'`, `'yaml'`
`...`	parameters passed to other functions

Details

The format `'rds'`, `'h5'`, `'fst'`, `'json'`, and `'yaml'` try to preserve the first-level column attributes. Factors will be preserved in these formats. Such property does not exist in `'csv'`, `'csv.zip'` formats.

Open-data formats are `'h5'`, `'csv'`, `'csv.zip'`, `'json'`, `'yaml'`. These formats require the table elements to be native types (numeric, character, factor, etc.).

`'rds'`, `'h5'`, and `'fst'` can store large data sets. `'fst'` is the best choice is performance and file size are the major concerns. `'rds'` preserves all the properties of the table.

Value

The normalized path for `export_table`, and a `data.table` for `import_table`

Examples

```
x <- data.table::data.table(
  a = rnorm(10),
  b = letters[1:10],
  c = 1:10,
  d = factor(LETTERS[1:10])
)

f <- tempfile(fileext = ".csv.zip")

export_table(x = x, file = f)

y <- import_table(file = f)

str(x)
str(y)

# clean up
unlink(f)
```

`find_path` *Try to find path along the root directory*

Description

Try to find `path` under root directory even if the original path is missing; see examples.

Usage

```
find_path(path, root_dir, all = FALSE)
```

Arguments

`path`	file path
`root_dir`	top directory of the search path
`all`	return all possible paths, default is false

Details

When file is missing, `find_path` concatenates the root directory and path combined to find the file. For example, if path is `"a/b/c/d"`, the function first seek for existence of `"a/b/c/d"`. If failed, then `"b/c/d"`, and then `"~/c/d"` until reaching root directory. If `all=TRUE`, then all files/directories found along the search path will be returned

Value

The absolute path of file if exists, or `NULL` if missing/failed.

Examples

```
root <- tempdir()

# ------ Case 1: basic use case -------

# Create a path in root
dir_create2(file.path(root, 'a'))

# find path even it's missing. The search path will be
# root/ins/cd/a - missing
# root/cd/a     - missing
# root/a        - exists!
find_path('ins/cd/a', root)

# ------ Case 2: priority -------
# Create two paths in root
dir_create2(file.path(root, 'cc/a'))
dir_create2(file.path(root, 'a'))

# If two paths exist, return the first path found
# root/ins/cd/a - missing
# root/cd/a     - exists - returned
# root/a        - exists, but ignored
find_path('ins/cc/a', root)

# ------ Case 3: find all -------
# Create two paths in root
dir_create2(file.path(root, 'cc/a'))
dir_create2(file.path(root, 'a'))

# If two paths exist, return the first path found
# root/ins/cd/a - missing
# root/cd/a     - exists - returned
# root/a        - exists - returned
find_path('ins/cc/a', root, all = TRUE)
```

`generate_reference` *Generate common average reference signal for 'RAVE' subjects*

Description

To properly run this function, please install `ravetools` package.

Usage

```
generate_reference(subject, electrodes)
```

Arguments

`subject`	subject ID or `RAVESubject` instance
`electrodes`	electrodes to calculate the common average; these electrodes must run through 'Wavelet' first

Details

The goal of generating common average signals is to capture the common movement from all the channels and remove them out from electrode signals.

The common average signals will be stored at subject reference directories. Two exact same copies will be stored: one in 'HDF5' format such that the data can be read universally by other programming languages; one in `filearray` format that can be read in R with super fast speed.

Value

A reference instance returned by `new_reference` with signal type determined automatically.

`get_projects` *Get all possible projects in 'RAVE' directory*

Description

Get all possible projects in 'RAVE' directory

Usage

```
get_projects(refresh = TRUE)
```

Arguments

`refresh`	whether to refresh the cache; default is true

Value

characters of project names

`get_val2` *Get value or return default if invalid*

Description

Get value or return default if invalid

Usage

```
get_val2(x, key = NA, default = NULL, na = FALSE, min_len = 1L, ...)
```

Arguments

`x`	a list, or environment, or just any R object
`key`	the name to obtain from `x`. If `NA`, then return `x`. Default is `NA`
`default`	default value if
`na`, `min_len`, `...`	passed to `is_valid_ish`

Value

values of the keys or default is invalid

Examples

```
x <- list(a=1, b = NA, c = character(0))

# ------------------------ Basic usage ------------------------

# no key, returns x if x is valid
get_val2(x)

get_val2(x, 'a', default = 'invalid')



# get 'b', NA is not filtered out
get_val2(x, 'b', default = 'invalid')

# get 'b', NA is considered invalid
get_val2(x, 'b', default = 'invalid', na = TRUE)



# get 'c', length 0 is allowed
get_val2(x, 'c', default = 'invalid', min_len = 0)

# length 0 is forbidden
get_val2(x, 'c', default = 'invalid', min_len = 1)
```

`h5_names` *Returns all names contained in 'HDF5' file*

Description

Returns all names contained in 'HDF5' file

Usage

```
h5_names(file)
```

Arguments

`file`	'HDF5' file path

Value

characters, data set names

`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

```
h5_valid(file, mode = c("r", "w"), close_all = FALSE)
```

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

Value

logical whether the file can be opened.

Examples

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

# No data written to the file, hence invalid
h5_valid(f, 'r')

save_h5(x, f, 'dset')
h5_valid(f, 'w')

# Open the file and hold a connection
ptr <- hdf5r::H5File$new(filename = f, mode = 'w')

# Can read, but cannot write
h5_valid(f, 'r')  # TRUE
h5_valid(f, 'w')  # FALSE

# However, this can be reset via `close_all=TRUE`
h5_valid(f, 'r', close_all = TRUE)
h5_valid(f, 'w')  # TRUE

# Now the connection is no longer valid
ptr
```

`import_electrode_table` *Import electrode table into subject meta folder*

Description

Import electrode table into subject meta folder

Usage

```
import_electrode_table(path, subject, use_fs = NA, dry_run = FALSE, ...)
```

Arguments

`path`	path of table file, must be a `'csv'` file
`subject`	'RAVE' subject ID or instance
`use_fs`	whether to use 'FreeSurfer' files to calculate other coordinates
`dry_run`	whether to dry-run the process; if true, then the table will be generated but not saved to subject's meta folder
`...`	passed to `read.csv`

Value

Nothing, the electrode information will be written directly to the subject's meta directory

---

`install_modules` *Install 'RAVE' modules*

Description

Install 'RAVE' modules

Usage

```
install_modules(modules, dependencies = FALSE)
```

Arguments

`modules`	a vector of characters, repository names; default is to automatically determined from a public registry
`dependencies`	whether to update dependent packages; default is false

Value

nothing

`install_subject` *Install a subject from the internet, a zip file or a directory*

Description

Install a subject from the internet, a zip file or a directory

Usage

```
install_subject(
  path = ".",
  ask = interactive(),
  overwrite = FALSE,
  backup = TRUE,
  use_cache = TRUE,
  dry_run = FALSE,
  force_project = NA,
  force_subject = NA,
  ...
)
```

Arguments

`path`	path to subject archive, can be a path to directory, a zip file, or an internet address (must starts with `'http'`, or `'ftp'`)
`ask`	when `overwrite` is false, whether to ask the user if subject exists; default is true when running in interactive session; users will be prompt with choices; if `ask=FALSE` and `overwrite=FALSE`, then the process will end with a warning if the subject exists.
`overwrite`	whether to overwrite existing subject, see argument `ask` and `backup`
`backup`	whether to back-up the subject when overwriting the data; default is true, which will rename the old subject folders instead of removing; set to true to remove existing subject.
`use_cache`	whether to use cached extraction directory; default is true. Set it to `FALSE` if you want a clean installation.
`dry_run`	whether to dry-run the process instead of actually installing; this rehearsal can help you see the progress and prevent you from losing data
`force_project`, `force_subject`	force set the project or subject; will raise a warning as this might mess up some pipelines
`...`	passed to `download.file`

Examples

```
# Please run 2nd example of function archive_subject

## Not run:

install_subject(path)


## End(Not run)
```

`is.blank` *Check If Input Has Blank String*

Description

Check If Input Has Blank String

Usage

```
is.blank(x)
```

Arguments

`x`	input data: a vector or an array

Value

`x == ""`

`is.zerolenth` *Check If Input Has Zero Length*

Description

Check If Input Has Zero Length

Usage

```
is.zerolenth(x)
```

Arguments

`x`	input data: a vector, list, or array

Value

whether `x` has zero length

`is_on_cran` *Check if current session is on 'CRAN'*

Description

Use this function only for examples and test. The goal is to comply with the 'CRAN' policy. Do not use it in normal functions to cheat. Violating 'CRAN' policy will introduce instability to your code. Make sure reading Section 'Details' before using this function.

Usage

```
is_on_cran(if_interactive = FALSE, verbose = FALSE)
```

Arguments

`if_interactive`	whether interactive session will be considered as on 'CRAN'; default is `FALSE`
`verbose`	whether to print out reason of return; default is no

Details

According to 'CRAN' policy, package examples and test functions may only use maximum 2 'CPU' cores. Examples running too long should be suppressed. Normally package developers will use `interactive()` to avoid running examples or parallel code on 'CRAN'. However, when checked locally, these examples will be skipped too. Coding bug in those examples will not be reported.

The objective is to allow 'RAVE' package developers to write and test examples locally or on integrated development environment (such as 'Github'), while suppressing them on 'CRAN'. In such way, bugs in the examples will be revealed and fixed promptly.

Do not use this function inside of the package functions to cheat or slip illegal code under the eyes of 'CRAN' folks. This will increase their work load and introduce instability to your code. If I find it out, I will report your package to 'CRAN'. Only use this function to make your package more robust. If you are developing 'RAVE' module, this function is explicitly banned. I'll implement a check for this, sooner or later.

Value

A logical whether current environment should be considered as on 'CRAN'.

`is_valid_ish` *Check if data is close to "valid"*

Description

Check if data is close to "valid"

Usage

```
is_valid_ish(
  x,
  min_len = 1,
  max_len = Inf,
  mode = NA,
  na = TRUE,
  blank = FALSE,
  all = FALSE
)
```

Arguments

`x`	data to check
`min_len, max_len`	minimal and maximum length
`mode`	which storage mode (see `mode`) should `x` be considered valid. Default is `NA`: disabled.
`na`	whether `NA` values considered invalid?
`blank`	whether blank string considered invalid?
`all`	if `na` or `blank` is true, whether all element of `x` being invalid will result in failure?

Value

logicals whether input x is valid

Examples

```
# length checks
is_valid_ish(NULL)                     # FALSE
is_valid_ish(integer(0))               # FALSE
is_valid_ish(integer(0), min_len = 0)  # TRUE
is_valid_ish(1:10, max_len = 9)        # FALSE

# mode check
is_valid_ish(1:10)                     # TRUE
is_valid_ish(1:10, mode = 'numeric')   # TRUE
is_valid_ish(1:10, mode = 'character') # FALSE

# NA or blank checks
is_valid_ish(NA)                     # FALSE
is_valid_ish(c(1,2,NA), all = FALSE) # FALSE
is_valid_ish(c(1,2,NA), all = TRUE)  # TRUE as not all elements are NA

is_valid_ish(c('1',''), all = FALSE)  # TRUE
is_valid_ish(1:3, all = FALSE)        # TRUE as 1:3 are not characters
```

`join_tensors` *Join Multiple Tensors into One Tensor*

Description

Join Multiple Tensors into One Tensor

Usage

```
join_tensors(tensors, temporary = TRUE)
```

Arguments

`tensors`	list of `Tensor` instances
`temporary`	whether to garbage collect space when exiting R session

Details

Merges multiple tensors. Each tensor must share the same dimension with the last one dimension as 1, for example, `100x100x1`. Join 3 tensors like this will result in a `100x100x3` tensor. This function is handy when each sub-tensors are generated separately. However, it does no validation test. Use with cautions.

Value

A new `Tensor` instance with the last dimension

Author(s)

Zhengjia Wang

Examples

```
tensor1 <- Tensor$new(data = 1:9, c(3,3,1), dimnames = list(
A = 1:3, B = 1:3, C = 1
), varnames = c('A', 'B', 'C'))
tensor2 <- Tensor$new(data = 10:18, c(3,3,1), dimnames = list(
  A = 1:3, B = 1:3, C = 2
), varnames = c('A', 'B', 'C'))
merged <- join_tensors(list(tensor1, tensor2))
merged$get_data()
```

`lapply_async` *Run `lapply` in parallel*

Description

Uses `lapply_async2`, but allows better parallel scheduling via `with_future_parallel`. On 'Unix', the function will fork processes. On 'Windows', the function uses strategies specified by `on_failure`

Usage

```
lapply_async(
  x,
  FUN,
  FUN.args = list(),
  callback = NULL,
  ncores = NULL,
  on_failure = "multisession",
  ...
)
```

Arguments

`x`	iterative elements
`FUN`	function to apply to each element of `x`
`FUN.args`	named list that will be passed to `FUN` as arguments
`callback`	callback function or `NULL`. When passed as function, the function takes one argument (elements of `x`) as input, and it suppose to return one string character.

`ncores`	number of cores to use, constraint by the `max_worker` option (see `raveio_getopt`); default is the maximum number of workers available
`on_failure`	alternative strategy if fork process is disallowed (set by users or on 'Windows')
`...`	passed to `lapply_async2`

Examples

```
if(!is_on_cran()) {
library(raveio)

# ---- Basic example ----------------------------
lapply_async(1:16, function(x) {
  # function that takes long to fun
  Sys.sleep(1)
  x
})

# With callback
lapply_async(1:16, function(x){
  Sys.sleep(1)
  x + 1
}, callback = function(x) {
  sprintf("Calculating|%s", x)
})

# With ncores
pids <- lapply_async(1:16, function(x){
  Sys.sleep(0.5)
  Sys.getpid()
}, ncores = 2)

# Unique number of PIDs (cores)
unique(unlist(pids))

# ---- With scheduler ----------------------------
# Scheduler pre-initialize parallel workers and temporary
# switches parallel context. The workers ramp-up
# time can be saved by reusing the workers.
#
with_future_parallel({

  # lapply_async block 1
  pids <- lapply_async(1:16, function(x){
    Sys.sleep(1)
    Sys.getpid()
  }, callback = function(x) {
    sprintf("lapply_async without ncores|%s", x)
  })
  print(unique(unlist(pids)))

  # lapply_async block 2
```

```
  pids <- lapply_async(1:16, function(x){
    Sys.sleep(1)
    Sys.getpid()
  }, callback = function(x) {
    sprintf("lapply_async with ncores|%s", x)
  }, ncores = 4)
  print(unique(unlist(pids)))

})


}
```

`LazyFST` *R6 Class to Load 'fst' Files*

Description

provides hybrid data structure for 'fst' file

Value

none

none

none

vector, dimensions

subset of data

Methods

Public methods:

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

Method `open()`**:** to be compatible with `LazyH5`

Usage:

`LazyFST$open(...)`

Arguments:

`...` ignored

Method `close()`**:** close the connection

Usage:

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

Arguments:

`...` ignored

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

Method `save()`**:** to be compatible with `LazyH5`

Usage:

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

Arguments:

`...` ignored

Method `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

Method `get_dims()`**:** get data dimension

Usage:

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

Arguments:

`...` ignored

Method `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

Author(s)

Zhengjia Wang

Examples

```
if(!is_on_cran()){

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

# Save to local disk
f <- tempfile()
fst::write_fst(as.data.frame(x), path = 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)

# The reading of column is very fast
system.time(dat[,100])

# Reading rows might be slow
system.time(dat[1,])

}
```

`LazyH5` *Lazy 'HDF5' file loader*

Description

provides hybrid data structure for 'HDF5' file

Value

none

self instance

self instance

subset of data

dimension of the array

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

Public fields

`quiet` whether to suppress messages

Methods

Public methods:

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

Method `finalize()`**:** garbage collection method

Usage:

```
LazyH5$finalize()
```

Method `print()`**:** overrides print method

Usage:

```
LazyH5$print()
```

Method `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

Method `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

Method `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

`...` passed to `createDataSet` in `hdf5r` package

Method `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

Method `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

Method `get_dims()`**:** get data dimension

Usage:

`LazyH5$get_dims(stay_open = TRUE)`

Arguments:

`stay_open` whether to leave the connection opened

Method `get_type()`**:** get data type

Usage:

`LazyH5$get_type(stay_open = TRUE)`

Arguments:

`stay_open` whether to leave the connection opened

Author(s)

Zhengjia Wang

Examples

```
# Data to save
x <- array(rnorm(1000), c(10,10,10))

# Save to local disk
f <- tempfile()
save_h5(x, file = f, name = 'x', chunk = c(10,10,10), level = 0)

# Load via LazyFST
dat <- LazyH5$new(file_path = f, data_name = 'x', read_only = TRUE)

dat

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

# Read a slice of the data
system.time(dat[,10,])
```

`LFP_electrode` *Definitions of electrode with 'LFP' signal type*

Description

Please use a safer `new_electrode` function to create instances. This documentation is to describe the member methods of the electrode class `LFP_electrode`

Value

if the reference number if `NULL` or `'noref'`, then returns 0, otherwise returns a `FileArray-class`

If `simplify` is enabled, and only one block is loaded, then the result will be a vector (`type="voltage"`) or a matrix (others), otherwise the result will be a named list where the names are the blocks.

Super class

`raveio::RAVEAbstarctElectrode` -> `LFP_electrode`

Active bindings

`h5_fname` 'HDF5' file name

`valid` whether current electrode is valid: subject exists and contains current electrode or reference; subject electrode type matches with current electrode type

`raw_sample_rate` voltage sample rate

`power_sample_rate` power/phase sample rate

`preprocess_info` preprocess information

`power_file` path to power 'HDF5' file

`phase_file` path to phase 'HDF5' file

`voltage_file` path to voltage 'HDF5' file

Methods

Public methods:

- `LFP_electrode$print()`
- `LFP_electrode$set_reference()`
- `LFP_electrode$new()`
- `LFP_electrode$.load_noref_wavelet()`
- `LFP_electrode$.load_noref_voltage()`
- `LFP_electrode$.load_wavelet()`
- `LFP_electrode$.load_voltage()`
- `LFP_electrode$.load_raw_voltage()`
- `LFP_electrode$load_data()`
- `LFP_electrode$load_blocks()`
- `LFP_electrode$clear_cache()`
- `LFP_electrode$clear_memory()`
- `LFP_electrode$clone()`

Method `print()`**:** print electrode summary

Usage:

```
LFP_electrode$print()
```

Method `set_reference()`**:** set reference for current electrode

Usage:

```
LFP_electrode$set_reference(reference)
```

Arguments:

`reference` either `NULL` or `LFP_electrode` instance

Method `new()`**:** constructor

Usage:

```
LFP_electrode$new(subject, number, quiet = FALSE)
```

Arguments:

`subject, number, quiet` see constructor in `RAVEAbstarctElectrode`

Method `.load_noref_wavelet()`**:** load non-referenced wavelet coefficients (internally used)

Usage:

```
LFP_electrode$.load_noref_wavelet(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

Method **`.load_noref_voltage():`** load non-referenced voltage (internally used)

Usage:

```
LFP_electrode$.load_noref_voltage(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

`srate` voltage signal sample rate

Method **`.load_wavelet():`** load referenced wavelet coefficients (internally used)

Usage:

```
LFP_electrode$.load_wavelet(
  type = c("power", "phase", "wavelet-coefficient"),
  reload = FALSE
)
```

Arguments:

`type` type of data to load

`reload` whether to reload cache

Method **`.load_voltage():`** load referenced voltage (internally used)

Usage:

```
LFP_electrode$.load_voltage(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

Method **`.load_raw_voltage():`** load raw voltage (no process)

Usage:

```
LFP_electrode$.load_raw_voltage(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

Method **`load_data():`** method to load electrode data

Usage:

```
LFP_electrode$load_data(
  type = c("power", "phase", "voltage", "wavelet-coefficient", "raw-voltage")
)
```

Arguments:

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"wavelet-coefficient"`, and `"raw-voltage"`. For `"power"`, `"phase"`, and `"wavelet-coefficient"`, 'Wavelet' transforms are required. For `"voltage"`, 'Notch' filters must be applied. All these types except for `"raw-voltage"` will be referenced. For `"raw-voltage"`, no reference will be performed since the data will be the "raw" signal (no processing).

Method **`load_blocks():`** load electrode block-wise data (with no reference), useful when epoch is absent

Usage:

```
LFP_electrode$load_blocks(
  blocks,
  type = c("power", "phase", "voltage", "wavelet-coefficient", "raw-voltage"),
  simplify = TRUE
)
```

Arguments:

`blocks` session blocks

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"raw-voltage"` (with no filters applied, as-is from imported), `"wavelet-coefficient"`. Note that if type is `"raw-voltage"`, then the data only needs to be imported; for `"voltage"` data, 'Notch' filters must be applied; for all other types, 'Wavelet' transforms are required.

`simplify` whether to simplify the result

Method `clear_cache()`**:** method to clear cache on hard drive

Usage:

```
LFP_electrode$clear_cache(...)
```

Arguments:

`...` ignored

Method `clear_memory()`**:** method to clear memory

Usage:

```
LFP_electrode$clear_memory(...)
```

Arguments:

`...` ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Download subject demo/DemoSubject

subject <- as_rave_subject("demo/DemoSubject", strict = FALSE)

if(dir.exists(subject$path)) {

# Electrode 14 in demo/DemoSubject
e <- new_electrode(subject = subject, number = 14, signal_type = "LFP")

# Load CAR reference "ref_13-16,24"
ref <- new_reference(subject = subject, number = "ref_13-16,24",
                     signal_type = "LFP")
```

```
e$set_reference(ref)

# Set epoch
e$set_epoch(epoch = 'auditory_onset')

# Set loading window
e$trial_intervals <- list(c(-1, 2))

# Preview
print(e)

# Now epoch power
power <- e$load_data("power")
names(dimnames(power))

# Subset power
subset(power, Time ~ Time < 0, Electrode ~ Electrode == 14)

# Draw baseline
tempfile <- tempfile()
bl <- power_baseline(power, baseline_windows = c(-1, 0),
                     method = "decibel", filebase = tempfile)
collapsed_power <- collapse2(bl, keep = c(2,1))
# Visualize
dname <- dimnames(bl)
image(collapsed_power, x = dname$Time, y = dname$Frequency,
      xlab = "Time (s)", ylab = "Frequency (Hz)",
      main = "Mean power over trial (Baseline: -1~0 seconds)",
      sub = glue('Electrode {e$number} (Reference: {ref$number})'))
abline(v = 0, lty = 2, col = 'blue')
text(x = 0, y = 20, "Audio onset", col = "blue", cex = 0.6)

# clear cache on hard disk
e$clear_cache()
ref$clear_cache()

}
```

`LFP_reference` *Definitions of reference with 'LFP' signal type*

Description

Please use a safer `new_reference` function to create instances. This documentation is to describe the member methods of the electrode class `LFP_reference`

Value

if the reference number if `NULL` or `'noref'`, then returns 0, otherwise returns a `FileArray-class`

If `simplify` is enabled, and only one block is loaded, then the result will be a vector (`type="voltage"`) or a matrix (others), otherwise the result will be a named list where the names are the blocks.

Super class

`raveio::RAVEAbstarctElectrode` -> `LFP_reference`

Active bindings

`exists` whether electrode exists in subject

`h5_fname` 'HDF5' file name

`valid` whether current electrode is valid: subject exists and contains current electrode or reference; subject electrode type matches with current electrode type

`raw_sample_rate` voltage sample rate

`power_sample_rate` power/phase sample rate

`preprocess_info` preprocess information

`power_file` path to power 'HDF5' file

`phase_file` path to phase 'HDF5' file

`voltage_file` path to voltage 'HDF5' file

Methods

Public methods:

- `LFP_reference$print()`
- `LFP_reference$set_reference()`
- `LFP_reference$new()`
- `LFP_reference$.load_noref_wavelet()`
- `LFP_reference$.load_noref_voltage()`
- `LFP_reference$.load_wavelet()`
- `LFP_reference$.load_voltage()`
- `LFP_reference$load_data()`
- `LFP_reference$load_blocks()`
- `LFP_reference$clear_cache()`
- `LFP_reference$clear_memory()`
- `LFP_reference$clone()`

Method `print()`: print reference summary

Usage:

```
LFP_reference$print()
```

Method `set_reference()`: set reference for current electrode

Usage:

```
LFP_reference$set_reference(reference)
```

Arguments:

`reference` either `NULL` or `LFP_electrode` instance

Method `new():` constructor

Usage:

```
LFP_reference$new(subject, number, quiet = FALSE)
```

Arguments:

`subject`, `number`, `quiet` see constructor in `RAVEAbstarctElectrode`

Method `.load_noref_wavelet():` load non-referenced wavelet coefficients (internally used)

Usage:

```
LFP_reference$.load_noref_wavelet(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

Method `.load_noref_voltage():` load non-referenced voltage (internally used)

Usage:

```
LFP_reference$.load_noref_voltage(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

`srate` voltage signal sample rate

Method `.load_wavelet():` load referenced wavelet coefficients (internally used)

Usage:

```
LFP_reference$.load_wavelet(
  type = c("power", "phase", "wavelet-coefficient"),
  reload = FALSE
)
```

Arguments:

`type` type of data to load

`reload` whether to reload cache

Method `.load_voltage():` load referenced voltage (internally used)

Usage:

```
LFP_reference$.load_voltage(reload = FALSE)
```

Arguments:

`reload` whether to reload cache

Method `load_data():` method to load electrode data

Usage:

```
LFP_reference$load_data(
  type = c("power", "phase", "voltage", "wavelet-coefficient")
)
```

Arguments:

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"wavelet-coefficient"`.

Method `load_blocks()`**:** load electrode block-wise data (with reference), useful when epoch is absent

Usage:

```
LFP_reference$load_blocks(
  blocks,
  type = c("power", "phase", "voltage", "wavelet-coefficient"),
  simplify = TRUE
)
```

Arguments:

`blocks` session blocks

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"wavelet-coefficient"`. Note that if type is voltage, then 'Notch' filters must be applied; otherwise 'Wavelet' transforms are required.

`simplify` whether to simplify the result

Method `clear_cache()`**:** method to clear cache on hard drive

Usage:

```
LFP_reference$clear_cache(...)
```

Arguments:

`...` ignored

Method `clear_memory()`**:** method to clear memory

Usage:

```
LFP_reference$clear_memory(...)
```

Arguments:

`...` ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:

# Download subject demo/DemoSubject


subject <- as_rave_subject("demo/DemoSubject")

# Electrode 14 as reference electrode (Bipolar referencing)
e <- new_reference(subject = subject, number = "ref_14",
```

```
                   signal_type = "LFP")

# Reference "ref_13-16,24" (CAR or white-matter reference)
ref <- new_reference(subject = subject, number = "ref_13-16,24",
                     signal_type = "LFP")
ref

# Set epoch
e$set_epoch(epoch = 'auditory_onset')

# Set loading window
e$trial_intervals <- list(c(-1, 2))

# Preview
print(e)

# Now epoch power
power <- e$load_data("power")
names(dimnames(power))

# Subset power
subset(power, Time ~ Time < 0, Electrode ~ Electrode == 14)

# clear cache on hard disk
e$clear_cache()


## End(Not run)
```

`load_bids_ieeg_header` *Read in description files from 'BIDS-iEEG' format*

Description

Analyze file structures and import all `json` and `tsv` files. File specification can be found at `https://bids-specification.readthedocs.io/en/stable/`, chapter "Modality specific files", section "Intracranial Electroencephalography" (doi:10.1038/s4159701901057). Please note that this function has very limited support on BIDS format.

Usage

```
load_bids_ieeg_header(bids_root, project_name, subject_code, folder = "ieeg")
```

Arguments

`bids_root`	'BIDS' root directory
`project_name`	project folder name
`subject_code`	subject code, do not include `"sub-"` prefix

`folder`	folder name corresponding to 'iEEG' data. It's possible to analyze other folders. However, by default, the function is designed for `'ieeg'` folder.

Value

A list containing the information below:

`subject_code`	character, removed leading `"sub-"`
`project_name`	character, project name
`has_session`	whether session/block names are indicated by the file structure
`session_names`	session/block names indicated by file structure. If missing, then session name will be "default"
`paths`	a list containing path information
`stimuli_path`	stimuli path, not used for now
`sessions`	A named list containing meta information for each session/block. The names of the list is task name, and the items corresponding to the task contains events and channel information. Miscellaneous files are stored in "others" variable.

Examples

```
# Download https://github.com/bids-standard/bids-examples/
# extract to directory ~/rave_data/bids_dir/

bids_root <- '~/rave_data/bids_dir/'
project_name <- 'ieeg_visual'

if(dir.exists(bids_root) &&
   dir.exists(file.path(bids_root, project_name, 'sub-01'))){

  header <- load_bids_ieeg_header(bids_root, project_name, '01')

  print(header)

  # sessions
  names(header$sessions)

  # electrodes
  head(header$sessions$`01`$spaces$unknown_space$table)

  # visual task channel settings
  head(header$sessions$`01`$tasks$`01-visual-01`$channels)

  # event table
  head(header$sessions$`01`$tasks$`01-visual-01`$channels)
}
```

`load_fst_or_h5` *Function try to load 'fst' arrays, if not found, read 'HDF5' arrays*

Description

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

Usage

```
load_fst_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

---

`load_h5` *Lazy Load 'HDF5' File via* `hdf5r-package`

Description

Wrapper for class `LazyH5`, which load data with "lazy" mode - only read part of dataset when needed.

Usage

```
load_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

`save_h5`

Examples

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

# save x to file with name /group/dataset/1
save_h5(x, file, '/group/dataset/1', quiet = TRUE)

# read data
y <- load_h5(file, '/group/dataset/1', ram = TRUE)
class(y)   # array

z <- load_h5(file, '/group/dataset/1', ram = FALSE)
class(z)   # LazyH5

dim(z)
```

`load_meta2` *Load 'RAVE' subject meta data*

Description

Load 'RAVE' subject meta data

Usage

```
load_meta2(meta_type, project_name, subject_code, subject_id, meta_name)
```

Arguments

`meta_type`	electrodes, epochs, time_points, frequencies, references ...
`project_name`	project name
`subject_code`	subject code
`subject_id`	"project_name/subject_code"
`meta_name`	only used if meta_type is epochs or references

Value

A data frame of the specified meta type or `NULL` is no meta data is found.

`load_yaml` *A port to* `read_yaml`

Description

For more examples, see `save_yaml`.

Usage

```
load_yaml(file, ..., map = NULL)
```

Arguments

`file, ...`	passed to `read_yaml`
`map`	`fastmap2` instance or `NULL`

Value

A `fastmap2`. If `map` is provided then return map, otherwise return newly created one

See Also

`fastmap2`, `save_yaml`, `read_yaml`, `write_yaml`

`mgh_to_nii` *Convert 'FreeSurfer' `'mgh'` to 'Nifti'*

Description

Convert 'FreeSurfer' `'mgh'` to 'Nifti'

Usage

```
mgh_to_nii(from, to)
```

Arguments

`from`	path to 'FreeSurfer' `'mgh'` or `'mgz'` file
`to`	path to 'Nifti' file, must ends with `'nii'` or `'nii.gz'`

Value

Nothing; the file will be created to path specified by `to`

`module_add` *Add new 'RAVE' (2.0) module to current project*

Description

Add new 'RAVE' (2.0) module to current project

Usage

```
module_add(
  module_id,
  module_label,
  path = ".",
  type = c("default", "bare", "scheduler"),
  ...,
  pipeline_name = module_id,
  overwrite = FALSE
)
```

Arguments

`module_id`	module ID to create, must be unique
`module_label`	a friendly label to display in the dashboard
`path`	project root path; default is current directory
`type`	template to choose, options are `'default'` and `'bare'`
`...`	additional configurations to the module such as `'order'`, `'group'`, `'badge'`
`pipeline_name`	the pipeline name to create along with the module; default is identical to `module_id`
`overwrite`	whether to overwrite existing module if module with same ID exists; default is false

Value

Nothing.

`module_registry` *'RAVE' module registry*

Description

Create, view, or reserve the module registry

Usage

```
module_registry(
  title,
  repo,
  modules,
  authors,
  url = sprintf("https://github.com/%s", repo)
)

module_registry2(repo, description)

get_modules_registries(update = NA)

get_module_description(path)

add_module_registry(title, repo, modules, authors, url, dry_run = FALSE)
```

Arguments

`title`	title of the registry, usually identical to the description title in `'DESCRIPTION'` or `RAVE-CONFIG` file
`repo`	'Github' repository

`modules`	characters of module ID, must only contain letters, digits, underscore, dash; must not be duplicated with existing registered modules
`authors`	a list of module authors; there must be one and only one author with `'cre'` role (see `person`). This author will be considered maintainer, who will be in charge if editing the registry
`url`	the web address of the repository
`update`	whether to force updating the registry
`path, description`	path to `'DESCRIPTION'` or `RAVE-CONFIG` file
`dry_run`	whether to generate and preview message content instead of opening an email link

Details

A 'RAVE' registry contains the following data entries: repository title, name, 'URL', authors, and a list of module IDs. 'RAVE' requires that each module must use a unique module ID. It will cause an issue if two modules share the same ID. Therefore 'RAVE' maintains a public registry list such that the module maintainers can register their own module ID and prevent other people from using it.

To register your own module ID, please use `add_module_registry` to validate and send an email to the 'RAVE' development team.

Value

a registry object, or a list of registries

Examples

```
if(interactive()) {

library(raveio)

# get current registries
get_modules_registries(FALSE)

# create your own registry
module_registry(
  repo = "rave-ieeg/rave-pipelines",
  title = "A Collection of 'RAVE' Builtin Pipelines",
  authors = list(
    list("Zhengjia", "Wang", role = c("cre", "aut"),
         email = "dipterix@rave.wiki")
  ),
  modules = "brain_viewer"
)

# If your repository is on Github and RAVE-CONFIG file exists
module_registry2("rave-ieeg/rave-pipelines")
```

```
# send a request to add your registry

reg <- module_registry2("rave-ieeg/rave-pipelines")
add_module_registry(reg)

}
```

`new_constraints` *Create* `'RAVE'` *constrained variables*

Description

Create a variable that automatically validates

Usage

```
new_constraints(type, assertions = NULL)

new_constrained_variable(name, initial_value, constraints = NULL, ...)

new_constrained_binding(name, expr, quoted = FALSE, constraints = NULL, ...)
```

Arguments

`type`	variable type; `checkmate::assert_*` will be automatically applied if applicable
`assertions`	named list; each name stands for an assertion type, and the corresponding item can be one of the follows; please see 'Examples' for usages. **list of arguments or `NULL`** name of the assertion must be a valid assertion function in package `checkmate`. For example, `list(numeric=NULL)` will call `checkmate::assert_numeric` when value is validated **a function** name of the assertion can be arbitrary, users are in charge of the validation function. This function should take only one argument and return either `TRUE` if the validation passes, or a character of the error message.
`name`	`character(1)`, variable name
`initial_value`	initial value, if missing, then variable will be assigned with an empty list with class name `'key_missing'`
`constraints, ...`	when `constraints` is an instance of `RAVEVariableConstraints`, `...` will be ignored. When `constraints` is a string, then `constraints` will be passed to `new_constraints` (see argument `type`), and `...` will be packed as assertion parameters (see `assertions`)
`expr`	expression for binding
`quoted`	whether `expr` is quoted, default is false

Examples

```
# ---- Basic usage ----------------------------------------
analysis_range <- new_constrained_variable("Analysis range")

# Using checkmates::assert_numeric
analysis_range$use_constraints(
  constraints = "numeric",
  any.missing = FALSE,
  len = 2,
  sorted = TRUE,
  null.ok = FALSE
)

analysis_range$initialized # FALSE
print(analysis_range)

# set value
analysis_range$set_value(c(1, 2))

# get value
analysis_range$value   # or $get_value()

# ---- Fancy constraints ------------------------------------
# construct an analysis range between -1~1 or 4~10
time_window <- validate_time_window(c(-1, 1, 4, 10))
analysis_range <- new_constrained_variable("Analysis range")
analysis_range$use_constraints(
  constraints = new_constraints(
    type = "numeric",
    assertions = list(
      # validator 1
      "numeric" = list(
        any.missing = FALSE,
        len = 2,
        sorted = TRUE,
        null.ok = FALSE
      ),

      # validator 2
      "range" = quote({
        check <- FALSE
        if(length(.x) == 2) {
          check <- sapply(time_window, function(w) {
            if(
              .x[[1]] >= w[[1]] &&
              .x[[2]] <= w[[2]]
            ) { return (TRUE) }
            return( FALSE )
          })
        }
        if(any(check)) { return(TRUE) }
```

```
        valid_ranges <- paste(
          sapply(time_window, function(w) {
            paste(sprintf("%.2f", w), collapse = ",")
          }),
          collapse = "] or ["
        )
        return(sprintf("Invalid range: must be [%s]", valid_ranges))
      })
    )
  )
)

# validate and print out error messages
# remove `on_error` argument to stop on errors
analysis_range$validate(on_error = "message")

# Try with values (-2,1) instead of c(0,1)
analysis_range$value <- c(0, 1)

print(analysis_range)
analysis_range[]

# Change the context
time_window <- validate_time_window(c(0, 0.5))

# re-validate will error out
analysis_range$validate(on_error = "message")
```

`new_electrode` *Create new electrode channel instance or a reference signal instance*

Description

Create new electrode channel instance or a reference signal instance

Usage

```
new_electrode(subject, number, signal_type, ...)

new_reference(subject, number, signal_type, ...)
```

Arguments

`subject`	characters, or a `RAVESubject` instance
`number`	integer in `new_electrode`, or characters in `new_reference`; see 'Details' and 'Examples'

`signal_type`	signal type of the electrode or reference; can be automatically inferred, but it is highly recommended to specify a value; see `SIGNAL_TYPES`
`...`	other parameters passed to class constructors, respectively

Details

In `new_electrode`, `number` should be a positive valid integer indicating the electrode number. In `new_reference`, `number` can be one of the followings:

`'noref'`**, or** `NULL` no reference is needed

`'ref_X'` where `'X'` is a single number, then the reference is another existing electrode; this could occur in bipolar-reference cases

`'ref_XXX'` `'XXX'` is a combination of multiple electrodes that can be parsed by `parse_svec`. This could occur in common average reference, or white matter reference. One example is `'ref_13-16,24'`, meaning the reference signal is an average of electrode 13, 14, 15, 16, and 24.

Value

Electrode or reference instances that inherit `RAVEAbstarctElectrode` class

Examples

```
## Not run:

# Download subject demo/DemoSubject (~500 MB)

# Electrode 14 in demo/DemoSubject
subject <- as_rave_subject("demo/DemoSubject")
e <- new_electrode(subject = subject, number = 14, signal_type = "LFP")

# Load CAR reference "ref_13-16,24"
ref <- new_reference(subject = subject, number = "ref_13-16,24",
                     signal_type = "LFP")
e$set_reference(ref)


# Set epoch
e$set_epoch(epoch = 'auditory_onset')

# Set loading window
e$trial_intervals <- list(c(-1, 2))

# Preview
print(e)

# Now epoch power
power <- e$load_data("power")
names(dimnames(power))

# Subset power
```

```
subset(power, Time ~ Time < 0, Electrode ~ Electrode == 14)

# Draw baseline
tempfile <- tempfile()
bl <- power_baseline(power, baseline_windows = c(-1, 0),
                     method = "decibel", filebase = tempfile)
collapsed_power <- collapse2(bl, keep = c(2,1))
# Visualize
dname <- dimnames(bl)
image(collapsed_power, x = dname$Time, y = dname$Frequency,
      xlab = "Time (s)", ylab = "Frequency (Hz)",
      main = "Mean power over trial (Baseline: -1~0 seconds)",
      sub = glue('Electrode {e$number} (Reference: {ref$number})'))
abline(v = 0, lty = 2, col = 'blue')
text(x = 0, y = 20, "Audio onset", col = "blue", cex = 0.6)

# clear cache on hard disk
e$clear_cache()
ref$clear_cache()


## End(Not run)
```

`new_variable_collection` *Create a collection of constraint variables*

Description

Create a collection of constraint variables

Usage

```
new_variable_collection(name = "", explicit = TRUE, r6_def = NULL)
```

Arguments

`name`	collection name, default is empty
`explicit`	whether setting and getting variables should be explicit, default is `TRUE`, which means trying to get undefined variables will result in errors
`r6_def`	`R6` class generator; default is `RAVEVariableCollection`. This input is for class definitions that are child classes of `RAVEVariableCollection`.

Value

A `RAVEVariableCollectionWrapper` instance

Examples

```
collection <- new_variable_collection()

# Add unconstrained variables
collection$add_variable(id = "title", "Voltage traces")

# Add a variable with placeholder
collection$add_variable(id = "time_points")

# Add variable with constraints
collection$add_variable(
  id = "analysis_range",
  var = new_constrained_variable(
    name = "Analysis range",
    initial_value = c(0, 1),
    constraints = "numeric",
    any.missing = FALSE,
    len = 2,
    sorted = TRUE,
    null.ok = FALSE
  )
)

collection$use_constraints(quote({
  # `x` is the list of values
  time_range <- range(.x$time_points, na.rm = TRUE)
  if(
    .x$analysis_range[[1]] >= time_range[[1]] &&
    .x$analysis_range[[2]] <= time_range[[2]]
  ) {
    # valid
    re <- TRUE
  } else {

    # error message
    re <- sprintf(
      "Invalid analysis range, must be within [%.2f, %.2f]",
      time_range[[1]], time_range[[2]]
    )
  }

  re
}))


collection$set_value("time_points", seq(-1, 10, by = 0.5))

# validation will pass
collection$validate()

# Get variable values
collection$as_list()
```

```
collection[]

# get one variable
collection$get_value("analysis_range")

# get unregistered variable
collection$get_value("unregistered_variable")

# get partial variables with single `[`
collection["title", "analysis_range"]
collection[c("title", "analysis_range")]

collection$set_value("analysis_range", c(-2, 5))

## Not run:
collection$validate()

# error out when explicit, please either
# set explicit=TRUE or register the variable via $add_variable
collection$set_value("unregistered_variable", 1)


## End(Not run)

# turn off explicit variable option
collection$explicit <- FALSE
collection$set_value("unregistered_variable", 1)
collection$get_value("unregistered_variable")
```

`niftyreg_coreg` *Register 'CT' to 'MR' images via* 'NiftyReg'

Description

Supports 'Rigid', 'affine', or 'non-linear' transformation

Usage

```
niftyreg_coreg(
  ct_path,
  mri_path,
  coreg_path = NULL,
  reg_type = c("rigid", "affine", "nonlinear"),
  interp = c("trilinear", "cubic", "nearest"),
  verbose = TRUE,
  ...
)
```

```
cmd_run_niftyreg_coreg(
  subject,
  ct_path,
  mri_path,
  reg_type = c("rigid", "affine", "nonlinear"),
  interp = c("trilinear", "cubic", "nearest"),
  verbose = TRUE,
  dry_run = FALSE,
  ...
)
```

Arguments

`ct_path`, `mri_path`	absolute paths to 'CT' and 'MR' image files
`coreg_path`	registration path, where to save results; default is the parent folder of `ct_path`
`reg_type`	registration type, choices are `'rigid'`, `'affine'`, or `'nonlinear'`
`interp`	how to interpolate when sampling volumes, choices are `'trilinear'`, `'cubic'`, or `'nearest'`
`verbose`	whether to verbose command; default is true
`...`	other arguments passed to `register_volume`
`subject`	'RAVE' subject
`dry_run`	whether to dry-run the script and to print out the command instead of executing the code; default is false

Value

Nothing is returned from the function. However, several files will be generated at the 'CT' path:

`'ct_in_t1.nii'` aligned 'CT' image; the image is also re-sampled into 'MRI' space

`'CT_IJK_to_MR_RAS.txt'` transform matrix from volume 'IJK' space in the original 'CT' to the 'RAS' anatomical coordinate in 'MR' scanner

`'CT_RAS_to_MR_RAS.txt'` transform matrix from scanner 'RAS' space in the original 'CT' to 'RAS' in 'MR' scanner space

---

`pipeline` *Creates 'RAVE' pipeline instance*

Description

Set pipeline inputs, execute, and read pipeline outputs

Usage

```
pipeline(
  pipeline_name,
  settings_file = "settings.yaml",
  paths = pipeline_root(),
  temporary = FALSE
)

pipeline_from_path(path, settings_file = "settings.yaml")
```

Arguments

`pipeline_name`	the name of the pipeline, usually title field in the `'DESCRIPTION'` file, or the pipeline folder name (if description file is missing)
`settings_file`	the name of the settings file, usually stores user inputs
`paths`	the paths to search for the pipeline, usually the parent directory of the pipeline; default is `pipeline_root`, which only search for pipelines that are installed or in current working directory.
`temporary`	see `pipeline_root`
`path`	the pipeline folder

Value

A `PipelineTools` instance

Examples

```
if(!is_on_cran()) {

library(raveio)

# ------------ Set up a bare minimal example pipeline ---------------
pipeline_path <- pipeline_create_template(
  root_path = tempdir(), pipeline_name = "raveio_demo",
  overwrite = TRUE, activate = FALSE, template_type = "rmd-bare")

save_yaml(list(
  n = 100, pch = 16, col = "steelblue"
), file = file.path(pipeline_path, "settings.yaml"))

pipeline_build(pipeline_path)

rmarkdown::render(input = file.path(pipeline_path, "main.Rmd"),
                  output_dir = pipeline_path,
                  knit_root_dir = pipeline_path,
                  intermediates_dir = pipeline_path, quiet = TRUE)

utils::browseURL(file.path(pipeline_path, "main.html"))
```

```
# --------------------- Example starts ------------------------

pipeline <- pipeline("raveio_demo", paths = tempdir())

pipeline$run("plot_data")

# Run again and you will see some targets are skipped
pipeline$set_settings(pch = 2)
pipeline$run("plot_data")

head(pipeline$read("input_data"))

# or use
pipeline[c("n", "pch", "col")]
pipeline[-c("input_data")]

pipeline$target_table

pipeline$result_table

pipeline$progress("details")

# --------------------- Clean up ------------------------
unlink(pipeline_path, recursive = TRUE)

}
```

`pipeline-knitr-markdown`

Configure 'rmarkdown' *files to build 'RAVE' pipelines*

Description

Allows building 'RAVE' pipelines from 'rmarkdown' files. Please use it in 'rmarkdown' scripts only. Use `pipeline_create_template` to create an example.

Usage

```
configure_knitr(languages = c("R", "python"))

pipeline_setup_rmd(
  module_id,
  env = parent.frame(),
  collapse = TRUE,
  comment = "#>",
  languages = c("R", "python"),
  project_path = dipsaus::rs_active_project(child_ok = TRUE, shiny_ok = TRUE)
)
```

Arguments

`languages`	one or more programming languages to support; options are `'R'` and `'python'`
`module_id`	the module ID, usually the name of direct parent folder containing the pipeline file
`env`	environment to set up the pipeline translator
`collapse, comment`	passed to `set` method of `opts_chunk`
`project_path`	the project path containing all the pipeline folders, usually the active project folder

Value

A function that is supposed to be called later that builds the pipeline scripts

---

`PipelineCollections` *Connect and schedule pipelines*

Description

Connect and schedule pipelines

Connect and schedule pipelines

Value

A list containing

`id` the pipeline ID that can be used by `deps`

`pipeline` forked pipeline instance

`target_names` copy of `names`

`depend_on` copy of `deps`

`cue` copy of `cue`

`standalone` copy of `standalone`

Public fields

`verbose` whether to verbose the build

Active bindings

`root_path` path to the directory that contains pipelines and scheduler

`collection_path` path to the pipeline collections

`pipeline_ids` pipeline ID codes

Methods

Public methods:

- `PipelineCollections$new()`
- `PipelineCollections$add_pipeline()`
- `PipelineCollections$build_pipelines()`
- `PipelineCollections$run()`
- `PipelineCollections$get_scheduler()`

Method `new()`**:** Constructor

Usage:

```
PipelineCollections$new(root_path = NULL, overwrite = FALSE)
```

Arguments:

`root_path` where to store the pipelines and intermediate results

`overwrite` whether to overwrite if `root_path` exists

Method `add_pipeline()`**:** Add pipeline into the collection

Usage:

```
PipelineCollections$add_pipeline(
  x,
  names = NULL,
  deps = NULL,
  pre_hook = NULL,
  post_hook = NULL,
  cue = c("always", "thorough", "never"),
  search_paths = pipeline_root(),
  standalone = TRUE,
  hook_envir = parent.frame()
)
```

Arguments:

`x` a pipeline name (can be found via `pipeline_list`), or a `PipelineTools`

`names` pipeline targets to execute

`deps` pipeline IDs to depend on; see ’Values’ below

`pre_hook` function to run before the pipeline; the function needs two arguments: input map (can be edit in-place), and path to a directory that allows to store temporary files

`post_hook` function to run after the pipeline; the function needs two arguments: pipeline object, and path to a directory that allows to store intermediate results

`cue` whether to always run dependence

`search_paths` where to search for pipeline if `x` is a character; ignored when `x` is a pipeline object

`standalone` whether the pipeline should be standalone, set to `TRUE` if the same pipeline added multiple times should run independently; default is true

`hook_envir` where to look for global environments if `pre_hook` or `post_hook` contains global variables; default is the calling environment

Method `build_pipelines()`**:** Build pipelines and visualize

Usage:

```
PipelineCollections$build_pipelines(visualize = TRUE)
```

Arguments:

`visualize` whether to visualize the pipeline; default is true

Method `run()`**:** Run the collection of pipelines

Usage:

```
PipelineCollections$run(
  error = c("error", "warning", "ignore"),
  .scheduler = c("none", "future", "clustermq"),
  .type = c("callr", "smart", "vanilla"),
  .as_promise = FALSE,
  .async = FALSE,
  rebuild = NA,
  ...
)
```

Arguments:

`error` what to do when error occurs; default is `'error'` throwing errors; other choices are `'warning'` and `'ignore'`

`.scheduler, .type, .as_promise, .async, ...` passed to `pipeline_run`

`rebuild` whether to re-build the pipeline; default is `NA` (if the pipeline has been built before, then do not rebuild)

Method `get_scheduler()`**:** Get `scheduler` object

Usage:

```
PipelineCollections$get_scheduler()
```

`PipelineResult` *Pipeline result object*

Description

Pipeline result object

Pipeline result object

Value

`TRUE` if the target is finished, or `FALSE` if timeout is reached

Public fields

`progressor` progress bar object, usually generated from `progress2`

`promise` a `promise` instance that monitors the pipeline progress

`verbose` whether to print warning messages

`names` names of the pipeline to build

`async_callback` function callback to call in each check loop; only used when the pipeline is running in `async=TRUE` mode

`check_interval` used when `async=TRUE` in `pipeline_run`, interval in seconds to check the progress

Active bindings

`variables` target variables of the pipeline

`variable_descriptions` readable descriptions of the target variables

`valid` logical true or false whether the result instance hasn't been invalidated

`status` result status, possible status are `'initialize'`, `'running'`, `'finished'`, `'canceled'`, and `'errored'`. Note that `'finished'` only means the pipeline process has been finished.

`process` (read-only) process object if the pipeline is running in `'async'` mode, or `NULL`; see `r_bg`.

Methods

Public methods:

- `PipelineResult$validate()`
- `PipelineResult$invalidate()`
- `PipelineResult$get_progress()`
- `PipelineResult$new()`
- `PipelineResult$run()`
- `PipelineResult$await()`
- `PipelineResult$print()`
- `PipelineResult$get_values()`
- `PipelineResult$clone()`

Method `validate()`: check if result is valid, raises errors when invalidated

Usage:

```
PipelineResult$validate()
```

Method `invalidate()`: invalidate the pipeline result

Usage:

```
PipelineResult$invalidate()
```

Method `get_progress()`: get pipeline progress

Usage:

```
PipelineResult$get_progress()
```

Method `new()`: constructor (internal)

Usage:

```
PipelineResult$new(path = character(0L), verbose = FALSE)
```

Arguments:

`path` pipeline path

`verbose` whether to print warnings

Method `run()`: run pipeline (internal)

Usage:

```
PipelineResult$run(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  async = FALSE,
  process = NULL
)
```

Arguments:

`expr` expression to evaluate

`env` environment of `expr`

`quoted` whether `expr` has been quoted

`async` whether the process runs in other sessions

`process` the process object inherits `process`, will be inferred from `expr` if `process=NULL`, and will raise errors if cannot be found

Method `await()`: wait until some targets get finished

Usage:

```
PipelineResult$await(names = NULL, timeout = Inf)
```

Arguments:

`names` target names to wait, default is `NULL`, i.e. to wait for all targets that have been scheduled

`timeout` maximum waiting time in seconds

Method `print()`: print method

Usage:

```
PipelineResult$print()
```

Method `get_values()`: get results

Usage:

```
PipelineResult$get_values(names = NULL, ...)
```

Arguments:

`names` the target names to read

`...` passed to `pipeline_read`

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

Usage:

`PipelineResult$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`PipelineTools` *Class definition for pipeline tools*

Description

Class definition for pipeline tools

Class definition for pipeline tools

Value

The value of the inputs, or a list if `key` is missing

The values of the targets

A `PipelineResult` instance if `as_promise` or `async` is true; otherwise a list of values for input `names`

An environment of shared variables

See `type`

A table of the progress

Nothing

ancestor target names (including `names`)

A new pipeline object based on the path given

A new pipeline object based on the path given

the saved file path

the data if file is found or a default value

A list of key-value pairs

A list of the preferences. If `simplify` is true and length if keys is 1, then returns the value of that preference

logical whether the keys exist

Active bindings

`description` pipeline description

`settings_path` absolute path to the settings file

`extdata_path` absolute path to the user-defined pipeline data folder

`preference_path` directory to the pipeline preference folder

`target_table` table of target names and their descriptions

`result_table` summary of the results, including signatures of data and commands

`pipeline_path` the absolute path of the pipeline

`pipeline_name` the code name of the pipeline

Methods

Public methods:

- `PipelineTools$new()`
- `PipelineTools$set_settings()`
- `PipelineTools$get_settings()`
- `PipelineTools$read()`
- `PipelineTools$run()`
- `PipelineTools$eval()`
- `PipelineTools$shared_env()`
- `PipelineTools$python_module()`
- `PipelineTools$progress()`
- `PipelineTools$attach()`
- `PipelineTools$visualize()`
- `PipelineTools$target_ancestors()`
- `PipelineTools$fork()`
- `PipelineTools$fork_to_subject()`
- `PipelineTools$with_activated()`
- `PipelineTools$clean()`
- `PipelineTools$save_data()`
- `PipelineTools$load_data()`
- `PipelineTools$set_preferences()`
- `PipelineTools$get_preferences()`
- `PipelineTools$has_preferences()`
- `PipelineTools$clone()`

Method `new()`**:** construction function

Usage:

```
PipelineTools$new(
  pipeline_name,
  settings_file = "settings.yaml",
  paths = pipeline_root(),
  temporary = FALSE
)
```

Arguments:

`pipeline_name` name of the pipeline, usually in the pipeline `'DESCRIPTION'` file, or pipeline folder name

`settings_file` the file name of the settings file, where the user inputs are stored

`paths` the paths to find the pipeline, usually the parent folder of the pipeline; default is `pipeline_root()`

`temporary` whether not to save `paths` to current pipeline root registry. Set this to `TRUE` when importing pipelines from subject pipeline folders

Method `set_settings()`**:** set inputs

Usage:

```
PipelineTools$set_settings(..., .list = NULL)
```

Arguments:

`..., .list` named list of inputs; all inputs should be named, otherwise errors will be raised

Method **`get_settings()`:** get current inputs

Usage:

```
PipelineTools$get_settings(key, default = NULL, constraint)
```

Arguments:

`key` the input name; default is missing, i.e., to get all the settings

`default` default value if not found

`constraint` the constraint of the results; if input value is not from `constraint`, then only the first element of `constraint` will be returned.

Method **`read()`:** read intermediate variables

Usage:

```
PipelineTools$read(var_names, ifnotfound = NULL, ...)
```

Arguments:

`var_names` the target names, can be obtained via `x$target_table` member; default is missing, i.e., to read all the intermediate variables

`ifnotfound` variable default value if not found

`...` other parameters passing to `pipeline_read`

Method **`run()`:** run the pipeline

Usage:

```
PipelineTools$run(
  names = NULL,
  async = FALSE,
  as_promise = async,
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  return_values = TRUE,
  ...
)
```

Arguments:

`names` pipeline variable names to calculate; default is to calculate all the targets

`async` whether to run asynchronous in another process

`as_promise` whether to return a `PipelineResult` instance

`scheduler, type, envir, callr_function, return_values, ...` passed to `pipeline_run` if `as_promise` is true, otherwise these arguments will be passed to `pipeline_run_bare`

Method **`eval()`:** run the pipeline in order; unlike `$run()`, this method does not use the `targets` infrastructure, hence the pipeline results will not be stored, and the order of `names` will be respected.

Usage:

```
PipelineTools$eval(
  names,
  env = parent.frame(),
  shortcut = FALSE,
  clean = TRUE,
  ...
)
```

Arguments:

`names` pipeline variable names to calculate; must be specified

`env` environment to evaluate and store the results

`shortcut` logical or characters; default is `FALSE`, meaning `names` and all the dependencies (if missing from `env`) will be evaluated; set to `TRUE` if only `names` are to be evaluated. When `shortcut` is a character vector, it should be a list of targets (including their ancestors) whose values can be assumed to be up-to-date, and the evaluation of those targets can be skipped.

`clean` whether to evaluate without polluting `env`

`...` passed to `pipeline_eval`

Method `shared_env()`**:** run the pipeline shared library in scripts starting with path `R/shared`

Usage:

```
PipelineTools$shared_env(callr_function = callr::r)
```

Arguments:

`callr_function` either `callr::r` or `NULL`; when `callr::r`, the environment will be loaded in isolated R session and serialized back to the main session to avoid contaminating the main session environment; when `NULL`, the code will be sourced directly in current environment.

Method `python_module()`**:** get 'Python' module embedded in the pipeline

Usage:

```
PipelineTools$python_module(
  type = c("info", "module", "shared", "exist"),
  must_work = TRUE
)
```

Arguments:

`type` return type, choices are `'info'` (get basic information such as module path, default), `'module'` (load module and return it), `'shared'` (load a shared sub-module from the module, which is shared also in report script), and `'exist'` (returns true or false on whether the module exists or not)

`must_work` whether the module needs to be existed or not. If `TRUE`, the raise errors when the module does not exist; default is `TRUE`, ignored when `type` is `'exist'`.

Method `progress()`**:** get progress of the pipeline

Usage:

```
PipelineTools$progress(method = c("summary", "details"))
```

Arguments:

`method` either `'summary'` or `'details'`

Method `attach()`: attach pipeline tool to environment (internally used)

Usage:

```
PipelineTools$attach(env)
```

Arguments:

`env` an environment

Method `visualize()`: visualize pipeline target dependency graph

Usage:

```
PipelineTools$visualize(
  glimpse = FALSE,
  aspect_ratio = 2,
  node_size = 30,
  label_size = 40,
  ...
)
```

Arguments:

`glimpse` whether to glimpse the graph network or render the state

`aspect_ratio` controls node spacing

`node_size`, `label_size` size of nodes and node labels

`...` passed to `pipeline_visualize`

Method `target_ancestors()`: a helper function to get target ancestors

Usage:

```
PipelineTools$target_ancestors(names, skip_names = NULL)
```

Arguments:

`names` targets whose ancestor targets need to be queried

`skip_names` targets that are assumed to be up-to-date, hence will be excluded, notice this exclusion is recursive, that means not only `skip_names` are excluded, but also their ancestors will be excluded from the result.

Method `fork()`: fork (copy) the current pipeline to a new directory

Usage:

```
PipelineTools$fork(path, policy = "default")
```

Arguments:

`path` path to the new pipeline, a folder will be created there

`policy` fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying `main.html` and `shared` folder

Method `fork_to_subject()`: fork (copy) the current pipeline to a 'RAVE' subject

Usage:

```
PipelineTools$fork_to_subject(
  subject,
  label = "NA",
  policy = "default",
  delete_old = FALSE,
  sanitize = TRUE
)
```

Arguments:

`subject` subject ID or instance in which pipeline will be saved

`label` pipeline label describing the pipeline

`policy` fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying `main.html` and `shared` folder

`delete_old` whether to delete old pipelines with the same label default is false

`sanitize` whether to sanitize the registry at save. This will remove missing folders and import manually copied pipelines to the registry (only for the pipelines with the same name)

Method `with_activated()`**:** run code with pipeline activated, some environment variables and function behaviors might change under such condition (for example, `targets` package functions)

Usage:

```
PipelineTools$with_activated(expr, quoted = FALSE, env = parent.frame())
```

Arguments:

`expr` expression to evaluate

`quoted` whether `expr` is quoted; default is false

`env` environment to run `expr`

Method `clean()`**:** clean all or part of the data store

Usage:

```
PipelineTools$clean(
 destroy = c("all", "cloud", "local", "meta", "process", "preferences", "progress",
    "objects", "scratch", "workspaces"),
  ask = FALSE
)
```

Arguments:

`destroy, ask` see `tar_destroy`

Method `save_data()`**:** save data to pipeline data folder

Usage:

```
PipelineTools$save_data(
  data,
  name,
  format = c("json", "yaml", "csv", "fst", "rds"),
  overwrite = FALSE,
  ...
)
```

Arguments:

`data` R object

`name` the name of the data to save, must start with letters

`format` serialize format, choices are `'json'`, `'yaml'`, `'csv'`, `'fst'`, `'rds'`; default is `'json'`. To save arbitrary objects such as functions or environments, use `'rds'`

`overwrite` whether to overwrite existing files; default is no

`...` passed to saver functions

Method **`load_data()`:** load data from pipeline data folder

Usage:

```
PipelineTools$load_data(
  name,
  error_if_missing = TRUE,
  default_if_missing = NULL,
  format = c("auto", "json", "yaml", "csv", "fst", "rds"),
  ...
)
```

Arguments:

`name` the name of the data

`error_if_missing` whether to raise errors if the name is missing

`default_if_missing` default values to return if the name is missing

`format` the format of the data, default is automatically obtained from the file extension

`...` passed to loader functions

Method **`set_preferences()`:** set persistent preferences from the pipeline. The preferences should not affect how pipeline is working, hence usually stores minor variables such as graphic options. Changing preferences will not invalidate pipeline cache.

Usage:

```
PipelineTools$set_preferences(..., .list = NULL)
```

Arguments:

`..., .list` key-value pairs of initial preference values. The keys must start with 'global' or the module ID, followed by dot and preference type and names. For example `'global.graphics.continuous_palette` for setting palette colors for continuous heat-map; "global" means the settings should be applied to all 'RAVE' modules. The module-level preference, `'power_explorer.export.default_format'` sets the default format for power-explorer export dialogue.

`name` preference name, must contain only letters, digits, underscore, and hyphen, will be coerced to lower case (case-insensitive)

Method **`get_preferences()`:** get persistent preferences from the pipeline.

Usage:

```
PipelineTools$get_preferences(
  keys,
  simplify = TRUE,
  ifnotfound = NULL,
  validator = NULL,
  ...
)
```

Arguments:

`keys` characters to get the preferences

`simplify` whether to simplify the results when length of key is 1; default is true; set to false to always return a list of preferences

`ifnotfound` default value when the key is missing

`validator` `NULL` or function to validate the values; see 'Examples'

`...` passed to `validator` if `validator` is a function

Examples:

```
library(raveio)
if(interactive() && length(pipeline_list()) > 0) {
  pipeline <- pipeline("power_explorer")

  # set dummy preference
  pipeline$set_preferences("global.example.dummy_preference" = 1:3)

  # get preference
  pipeline$get_preferences("global.example.dummy_preference")

  # get preference with validator to ensure the value length to be 1
  pipeline$get_preferences(
    "global.example.dummy_preference",
    validator = function(value) {
      stopifnot(length(value) == 1)
    },
    ifnotfound = 100
  )

  pipeline$has_preferences("global.example.dummy_preference")
}
```

Method `has_preferences()`**:** whether pipeline has preference keys

Usage:

`PipelineTools$has_preferences(keys, ...)`

Arguments:

`keys` characters name of the preferences

`...` passed to internal methods

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`PipelineTools$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

`pipeline`

Examples

```
## ------------------------------------------------
## Method `PipelineTools$get_preferences`
## ------------------------------------------------


library(raveio)
if(interactive() && length(pipeline_list()) > 0) {
  pipeline <- pipeline("power_explorer")

  # set dummy preference
  pipeline$set_preferences("global.example.dummy_preference" = 1:3)

  # get preference
  pipeline$get_preferences("global.example.dummy_preference")

  # get preference with validator to ensure the value length to be 1
  pipeline$get_preferences(
    "global.example.dummy_preference",
    validator = function(value) {
      stopifnot(length(value) == 1)
    },
    ifnotfound = 100
  )

  pipeline$has_preferences("global.example.dummy_preference")
}
```

`pipeline_collection` *Combine and execute pipelines*

Description

Combine and execute pipelines

Usage

```
pipeline_collection(root_path = NULL, overwrite = FALSE)
```

Arguments

`root_path`	directory to store pipelines and results
`overwrite`	whether to overwrite if `root_path` exists; default is false, and raises an error when `root_path` exists

Value

A `PipelineCollections` instance

`pipeline_install` *Install 'RAVE' pipelines*

Description

Install 'RAVE' pipelines

Usage

```
pipeline_install_local(
  src,
  to = c("default", "custom", "workdir", "tempdir"),
  upgrade = FALSE,
  force = FALSE,
  set_default = NA,
  ...
)

pipeline_install_github(
  repo,
  to = c("default", "custom", "workdir", "tempdir"),
  upgrade = FALSE,
  force = FALSE,
  set_default = NA,
  ...
)
```

Arguments

`src`	pipeline directory
`to`	installation path; choices are `'default'`, `'custom'`, `'workdir'`, and `'tempdir'`. Please specify pipeline root path via `pipeline_root` when `'custom'` is used.
`upgrade`	whether to upgrade the dependence; default is `FALSE` for stability, however, it is highly recommended to upgrade your dependencies
`force`	whether to force installing the pipelines
`set_default`	whether to set current pipeline module folder as the default, will be automatically set when the pipeline is from the official 'Github' repository.
`...`	other parameters not used
`repo`	'Github' repository in user-repository combination, for example, `'rave-ieeg/rave-pipeline'`

Value

nothing

`pipeline_settings_get_set` *Get or change pipeline input parameter settings*

Description

Get or change pipeline input parameter settings

Usage

```
pipeline_settings_set(
  ...,
  pipeline_path = Sys.getenv("RAVE_PIPELINE", "."),
  pipeline_settings_path = file.path(pipeline_path, "settings.yaml")
)

pipeline_settings_get(
  key,
  default = NULL,
  constraint = NULL,
  pipeline_path = Sys.getenv("RAVE_PIPELINE", "."),
  pipeline_settings_path = file.path(pipeline_path, "settings.yaml")
)
```

Arguments

`pipeline_path`	the root directory of the pipeline
`pipeline_settings_path`	the settings file of the pipeline, must be a 'yaml' file; default is `'settings.yaml'` in the current pipeline
`key, ...`	the character key(s) to get or set
`default`	the default value is key is missing
`constraint`	the constraint of the resulting value; if not `NULL`, then result must be within the `constraint` values, otherwise the first element of `constraint` will be returned. This is useful to make sure the results stay within given options

Value

`pipeline_settings_set` returns a list of all the settings. `pipeline_settings_get` returns the value of given key.

`power_baseline` *Calculate power baseline*

Description

Calculate power baseline

Usage

```
power_baseline(
  x,
  baseline_windows,
  method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore"),
  units = c("Trial", "Frequency", "Electrode"),
  ...
)

## S3 method for class 'rave_prepare_power'
power_baseline(
  x,
  baseline_windows,
  method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore"),
  units = c("Frequency", "Trial", "Electrode"),
  electrodes,
  ...
)

## S3 method for class 'FileArray'
power_baseline(
  x,
  baseline_windows,
  method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore"),
  units = c("Frequency", "Trial", "Electrode"),
  filebase = NULL,
  ...
)

## S3 method for class 'array'
power_baseline(
  x,
  baseline_windows,
  method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore"),
  units = c("Trial", "Frequency", "Electrode"),
  ...
)

## S3 method for class 'ECoGTensor'
```

```
power_baseline(
  x,
  baseline_windows,
 method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore"),
  units = c("Trial", "Frequency", "Electrode"),
  filebase = NULL,
  hybrid = TRUE,
  ...
)
```

Arguments

`x`	R array, `filearray`, `ECoGTensor`, or `'rave_prepare_power'` object created by `prepare_subject_power`.
`baseline_windows`	list of baseline window (intervals)
`method`	baseline method; choices are `'percentage'`, `'sqrt_percentage'`, `'decibel'`, `'zscore'`, `'sqrt_zscore'`; see ’Details’ in `baseline_array`
`units`	the unit of the baseline; see ’Details’
`...`	passed to other methods
`electrodes`	the electrodes to be included in baseline calculation; for power repository object produced by `prepare_subject_power` only; default is all available electrodes in each of `signal_types`
`filebase`	where to store the output; default is `NULL` and is automatically determined
`hybrid`	whether the array will be

Details

The arrays must be four-mode tensor and must have valid named `dimnames`. The dimension names must be `'Trial'`, `'Frequency'`, `'Time'`, `'Electrode'`, case sensitive.

The `baseline_windows` determines the baseline windows that are used to calculate time-points of baseline to be included. This can be one or more intervals and must pass the validation function `validate_time_window`.

The `units` determines the unit of the baseline. It can be one or more of `'Trial'`, `'Frequency'`, `'Electrode'`. The default value is all of them, i.e., baseline for each combination of trial, frequency, and electrode. To share the baseline across trials, please remove `'Trial'` from `units`. To calculate baseline that should be shared across electrodes (e.g. in some mini-electrodes), remove `'Electrode'` from the `units`.

Value

Usually the same type as the input: for arrays, `filearray`, or `ECoGTensor`, the outputs are also the same type with the same dimensions; for `'rave_prepare_power'` repositories, the results will be stored in its `'baselined'` element; see ’Examples’.

Examples

```
## Not run:
# The following code need to download additional demo data
# Please see https://rave.wiki/ for more details

library(raveio)
repo <- prepare_subject_power(
  subject = "demo/DemoSubject",
  time_windows = c(-1, 3),
  electrodes = c(14, 15))

##### Direct baseline on the repository
power_baseline(x = repo, method = "decibel",
               baseline_windows = list(c(-1, 0), c(2, 3)))
power_mean <- repo$power$baselined$collapse(
  keep = c(2,1), method = "mean")
image(power_mean, x = repo$time_points, y = repo$frequency,
      xlab = "Time (s)", ylab = "Frequency (Hz)",
      main = "Mean power over trial (Baseline: -1~0 & 2~3)")
abline(v = 0, lty = 2, col = 'blue')
text(x = 0, y = 20, "Aud-Onset", col = "blue", cex = 0.6)

##### Alternatively, baseline on electrode instances
baselined <- lapply(repo$power$data_list, function(inst) {
  re <- power_baseline(inst, method = "decibel",
                       baseline_windows = list(c(-1, 0), c(2, 3)))
  collapse2(re, keep = c(2,1), method = "mean")
})
power_mean2 <- (baselined[[1]] + baselined[[2]]) / 2

# Same with precision difference
max(abs(power_mean2 - power_mean)) < 1e-6



## End(Not run)
```

`prepare_subject_bare0` *Prepare 'RAVE' single-subject data*

Description

Prepare 'RAVE' single-subject data

Usage

```
prepare_subject_bare0(
  subject,
```

```
    electrodes,
    reference_name,
    ...,
    quiet = TRUE,
    repository_id = NULL
  )

  prepare_subject_bare(
    subject,
    electrodes,
    reference_name,
    ...,
    repository_id = NULL
  )

  prepare_subject_with_epoch(
    subject,
    electrodes,
    reference_name,
    epoch_name,
    time_windows,
    env = parent.frame(),
    ...
  )

  prepare_subject_with_blocks(
    subject,
    electrodes,
    reference_name,
    blocks,
    raw = FALSE,
    signal_type = "LFP",
    time_frequency = (!raw && signal_type == "LFP"),
    quiet = raw,
    env = parent.frame(),
    repository_id = NULL,
    ...
  )

  prepare_subject_phase(
    subject,
    electrodes,
    reference_name,
    epoch_name,
    time_windows,
    signal_type = c("LFP"),
    env = parent.frame(),
    verbose = TRUE,
```

```
  ...
)

prepare_subject_power(
  subject,
  electrodes,
  reference_name,
  epoch_name,
  time_windows,
  signal_type = c("LFP"),
  env = parent.frame(),
  verbose = TRUE,
  ...
)

prepare_subject_wavelet(
  subject,
  electrodes,
  reference_name,
  epoch_name,
  time_windows,
  signal_type = c("LFP"),
  env = parent.frame(),
  verbose = TRUE,
  ...
)

prepare_subject_raw_voltage_with_epoch(
  subject,
  electrodes,
  epoch_name,
  time_windows,
  ...,
  quiet = TRUE,
  repository_id = NULL
)

prepare_subject_voltage_with_epoch(
  subject,
  electrodes,
  epoch_name,
  time_windows,
  reference_name,
  ...,
  quiet = TRUE,
  repository_id = NULL
)
```

Arguments

`subject`	character of project and subject, such as `"demo/YAB"`, or `RAVESubject` instance
`electrodes`	integer vector of electrodes, or a character that can be parsed by `parse_svec`
`reference_name`	reference name to be loaded
`...`	ignored
`quiet`	whether to quietly load the data
`repository_id`	used internally
`epoch_name`	epoch name to be loaded, or a `RAVEEpoch` instance
`time_windows`	a list of time windows that are relative to epoch onset time; need to pass the validation `validate_time_window`
`env`	environment to evaluate
`blocks`	one or more session blocks to load
`raw`	whether to load from original (before processing) data; if true, then time-frequency data will not be loaded.
`signal_type`	electrode signal type (length of one) to be considered; default is 'LFP'. This option rarely needs to change unless you really want to check the power data from other types. For other signal types, check `SIGNAL_TYPES`
`time_frequency`	whether to load time-frequency data when preparing block data
`verbose`	whether to show progress

Value

A `fastmap2` (basically a list) of objects. Depending on the functions called, the following items may exist in the list:

`subject` A `RAVESubject` instance

`epoch_name` Same as input `epoch_name`

`epoch` A `RAVEEpoch` instance

`reference_name` Same as input `reference_name`

`reference_table` A data frame of reference

`electrode_table` A data frame of electrode information

`frequency` A vector of frequencies

`time_points` A vector of time-points

`power_list` A list of power data of the electrodes

`power_dimnames` A list of trial indices, frequencies, time points, and electrodes that are loaded

---

`progress_with_logger` *Enhanced progress with logger message*

Description

For best performance, please install `'ravedash'`. This function can replace `progress2`.

Usage

```
progress_with_logger(
  title,
  max = 1,
  ...,
  quiet = FALSE,
  session = shiny::getDefaultReactiveDomain(),
  shiny_auto_close = FALSE,
  outputId = NULL,
  log
)
```

Arguments

`title`, `max`, `...`, `quiet`, `session`, `shiny_auto_close`	see `progress2`
`outputId`	will be used if package `'shidashi'` is installed, otherwise will be ignored
`log`	function, `NULL`, or missing; default is missing, which will use `logger` function in the package `'ravedash'`, or `cat2` if `'ravedash'` is not installed. If `log=NULL`, then the message will be suppressed in 'shiny' applications. If a function provided, then the function will be called.

Value

A list, see `progress2`

`py_nipy_coreg` *Register 'CT' to 'MR' images via '`nipy`' script*

Description

Align 'CT' using `nipy.algorithms.registration.histogram_registration`.

Usage

```
py_nipy_coreg(
  ct_path,
  mri_path,
  clean_source = TRUE,
  inverse_target = TRUE,
  precenter_source = TRUE,
  smooth = 0,
  reg_type = c("rigid", "affine"),
  interp = c("pv", "tri"),
  similarity = c("crl1", "cc", "cr", "mi", "nmi", "slr"),
  optimizer = c("powell", "steepest", "cg", "bfgs", "simplex"),
  tol = 1e-04,
  dry_run = FALSE
)

cmd_run_nipy_coreg(
  subject,
  ct_path,
  mri_path,
  clean_source = TRUE,
  inverse_target = TRUE,
  precenter_source = TRUE,
  reg_type = c("rigid", "affine"),
  interp = c("pv", "tri"),
  similarity = c("crl1", "cc", "cr", "mi", "nmi", "slr"),
  optimizer = c("powell", "steepest", "cg", "bfgs", "simplex"),
  dry_run = FALSE,
  verbose = FALSE
)
```

Arguments

`ct_path, mri_path`	absolute paths to 'CT' and 'MR' image files
`clean_source`	whether to replace negative 'CT' values with zeros; default is true
`inverse_target`	whether to inverse 'MRI' color intensity; default is true
`precenter_source`	whether to adjust the 'CT' transform matrix before alignment, such that the origin of 'CT' is at the center of the volume; default is true. This option may avoid the case that 'CT' is too far-away from the 'MR' volume at the beginning of the optimization
`smooth, interp, optimizer, tol`	optimization parameters, see `'nipy'` documentation for details.
`reg_type`	registration type, choices are `'rigid'` or `'affine'`
`similarity`	the cost function of the alignment; choices are `'crl1'` ('L1' regularized correlation), `'cc'` (correlation coefficient), `'cr'` (correlation), `'mi'` (mutual infor-

	mation), `'nmi'` (normalized mutual information), `'slr'` (likelihood ratio). In reality I personally find `'crl1'` works best in most cases, though many tutorials suggest `'nmi'`.
`dry_run`	whether to dry-run the script and to print out the command instead of executing the code; default is false
`subject`	'RAVE' subject
`verbose`	whether to verbose command; default is false

Value

Nothing is returned from the function. However, several files will be generated at the 'CT' path:

`'ct_in_t1.nii'` aligned 'CT' image; the image is also re-sampled into 'MRI' space

`'CT_IJK_to_MR_RAS.txt'` transform matrix from volume 'IJK' space in the original 'CT' to the 'RAS' anatomical coordinate in 'MR' scanner

`'CT_RAS_to_MR_RAS.txt'` transform matrix from scanner 'RAS' space in the original 'CT' to 'RAS' in 'MR' scanner space

---

`rave-pipeline` *'RAVE' pipeline functions*

Description

Utility functions for 'RAVE' pipelines, currently designed for internal development use. The infrastructure will be deployed to 'RAVE' in the future to facilitate the "self-expanding" aim. Please check the official 'RAVE' website.

Usage

```
pipeline_root(root_path, temporary = FALSE)

pipeline_list(root_path = pipeline_root())

pipeline_find(name, root_path = pipeline_root())

pipeline_attach(name, root_path = pipeline_root())

pipeline_run(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  names = NULL,
  async = FALSE,
  check_interval = 0.5,
```

```
  progress_quiet = !async,
  progress_max = NA,
  progress_title = "Running pipeline",
  return_values = TRUE,
  ...
)

pipeline_clean(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  destroy = c("all", "cloud", "local", "meta", "process", "preferences", "progress",
    "objects", "scratch", "workspaces"),
  ask = FALSE
)

pipeline_run_bare(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  names = NULL,
  return_values = TRUE,
  ...
)

load_targets(..., env = NULL)

pipeline_target_names(pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))

pipeline_debug(
  quick = TRUE,
  env = parent.frame(),
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  skip_names
)

pipeline_dep_targets(
  names,
  skip_names = NULL,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", ".")
)

pipeline_eval(
  names,
  env = new.env(parent = parent.frame()),
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  settings_path = file.path(pipe_dir, "settings.yaml"),
  shortcut = FALSE
```

```
)

pipeline_visualize(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  glimpse = FALSE,
  targets_only = TRUE,
  shortcut = FALSE,
  zoom_speed = 0.1,
  ...
)

pipeline_progress(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  method = c("summary", "details", "custom"),
  func = targets::tar_progress_summary
)

pipeline_fork(
  src = Sys.getenv("RAVE_PIPELINE", "."),
  dest = tempfile(pattern = "rave_pipeline_"),
  policy = "default",
  activate = FALSE,
  ...
)

pipeline_build(pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))

pipeline_read(
  var_names,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  branches = NULL,
  ifnotfound = NULL,
  dependencies = c("none", "ancestors_only", "all"),
  simplify = TRUE,
  ...
)

pipeline_vartable(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  targets_only = TRUE,
  complete_only = FALSE,
  ...
)

pipeline_hasname(var_names, pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))

pipeline_watch(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
```

```
  targets_only = TRUE,
  ...
)

pipeline_create_template(
  root_path,
  pipeline_name,
  overwrite = FALSE,
  activate = TRUE,
  template_type = c("rmd", "r", "rmd-bare", "rmd-scheduler")
)

pipeline_create_subject_pipeline(
  subject,
  pipeline_name,
  overwrite = FALSE,
  activate = TRUE,
  template_type = c("rmd", "r")
)

pipeline_description(file)

pipeline_load_extdata(
  name,
  format = c("auto", "json", "yaml", "csv", "fst", "rds"),
  error_if_missing = TRUE,
  default_if_missing = NULL,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  ...
)

pipeline_save_extdata(
  data,
  name,
  format = c("json", "yaml", "csv", "fst", "rds"),
  overwrite = FALSE,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  ...
)

pipeline_shared(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  callr_function = callr::r
)

pipeline_set_preferences(
  ...,
  .list = NULL,
```

```
  .pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  .preference_instance = NULL
)

pipeline_get_preferences(
  keys,
  simplify = TRUE,
  ifnotfound = NULL,
  validator = NULL,
  ...,
  .preference_instance = NULL
)

pipeline_has_preferences(keys, ..., .preference_instance = NULL)
```

Arguments

`root_path`	the root directory for pipeline templates
`temporary`	whether not to save `paths` to current pipeline root registry. Set this to `TRUE` when importing pipelines from subject pipeline folders
`name`, `pipeline_name`	the pipeline name to create; usually also the folder
`pipe_dir`, `.pipe_dir`	where the pipeline directory is; can be set via system environment `Sys.setenv("RAVE_PIPELINE"=...)`
`scheduler`	how to schedule the target jobs: default is `'none'`, which is sequential. If you have multiple heavy-weighted jobs that can be scheduled at the same time, you can choose `'future'` or `'clustermq'`
`type`	how the pipeline should be executed; current choices are `"smart"` to enable 'future' package if possible, `'callr'` to use `r`, or `'vanilla'` to run everything sequentially in the main session.
`callr_function`	function that will be passed to `tar_make`; will be forced to be `NULL` if `type='vanilla'`, or `r` if `type='callr'`
`names`	the names of pipeline targets that are to be executed; default is `NULL`, which runs all targets; use `pipeline_target_names` to check all your available target names.
`async`	whether to run pipeline without blocking the main session
`check_interval`	when running in background (non-blocking mode), how often to check the pipeline
`progress_title`, `progress_max`, `progress_quiet`	control the progress, see `progress2`.
`return_values`	whether to return pipeline target values; default is true; only works in `pipeline_run_bare` and will be ignored by `pipeline_run`
`...`, `.list`	other parameters, targets, etc.
`destroy`	what part of data repository needs to be cleaned
`ask`	whether to ask
`env`, `envir`	environment to execute the pipeline

`quick`	whether to skip finished targets to save time
`skip_names`	hint of target names to fast skip provided they are up-to-date; only used when `quick=TRUE`. If missing, then `skip_names` will be automatically determined
`settings_path`	path to settings file name within subject's pipeline path
`shortcut`	whether to display shortcut targets
`glimpse`	whether to hide network status when visualizing the pipelines
`targets_only`	whether to return the variable table for targets only; default is true
`zoom_speed`	zoom speed when visualizing the pipeline dependence
`method`	how the progress should be presented; choices are `"summary"`, `"details"`, `"custom"`. If custom method is chosen, then `func` will be called
`func`	function to call when reading customized pipeline progress; default is `tar_progress_summary`
`src`, `dest`	pipeline folder to copy the pipeline script from and to
`policy`	fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying `main.html` and `shared` folder
`activate`	whether to activate the new pipeline folder from `dest`; default is false
`var_names`	variable name to fetch or to check
`branches`	branch to read from; see `tar_read`
`ifnotfound`	default values to return if variable is not found
`dependencies`	whether to load dependent targets, choices are `'none'` (default, only load targets specified by `names`), `'ancestors_only'` (load all but the ancestors targets), and `'all'` (both targets and ancestors)
`simplify`	whether to simplify the output
`complete_only`	whether only to show completed and up-to-date target variables; default is false
`overwrite`	whether to overwrite existing pipeline; default is false so users can double-check; if true, then existing pipeline, including the data will be erased
`template_type`	which template type to create; choices are `'r'` or `'rmd'`
`subject`	character indicating valid 'RAVE' subject ID, or `RAVESubject` instance
`file`	path to the 'DESCRIPTION' file under the pipeline folder, or pipeline collection folder that contains the pipeline information, structures, dependencies, etc.
`format`	format of the extended data, default is `'json'`, other choices are `'yaml'`, `'fst'`, `'csv'`, `'rds'`
`error_if_missing`, `default_if_missing`	what to do if the extended data is not found
`data`	extended data to be saved
`.preference_instance`	internally used
`keys`	preference keys
`validator`	`NULL` or function to validate values

Value

`pipeline_root`	the root directories of the pipelines
`pipeline_list`	the available pipeline names under `pipeline_root`
`pipeline_find`	the path to the pipeline
`pipeline_run`	a `PipelineResult` instance
`load_targets`	a list of targets to build
`pipeline_target_names`	a vector of characters indicating the pipeline target names
`pipeline_visualize`	a widget visualizing the target dependence structure
`pipeline_progress`	a table of building progress
`pipeline_fork`	a normalized path of the forked pipeline directory
`pipeline_read`	the value of corresponding `var_names`, or a named list if `var_names` has more than one element
`pipeline_vartable`	a table of summaries of the variables; can raise errors if pipeline has never been executed
`pipeline_hasname`	logical, whether the pipeline has variable built
`pipeline_watch`	a basic shiny application to monitor the progress
`pipeline_description`	the list of descriptions of the pipeline or pipeline collection

---

`rave-raw-validation` *Validate raw files in 'rave' directory*

Description

Validate subjects and returns whether the subject can be imported into 'rave'

Usage

```
validate_raw_file(
  subject_code,
  blocks,
  electrodes,
  format,
  data_type = c("continuous"),
  ...
)

IMPORT_FORMATS
```

Arguments

`subject_code`	subject code, direct folder under 'rave' raw data path
`blocks`	block character, direct folder under subject folder. For raw files following 'BIDS' convention, see details
`electrodes`	electrodes to verify
`format`	integer or character. For characters, run `names(IMPORT_FORMATS)`
`data_type`	currently only support continuous type of signals
`...`	other parameters used if validating `'BIDS'` format; see details.

Format

An object of class `list` of length 7.

Details

Six types of raw file structures are supported. They can be basically classified into two categories: 'rave' native raw structure and 'BIDS-iEEG' structure.

In 'rave' native structure, subject folders are stored within the root directory, which can be obtained via `raveio_getopt('raw_data_dir')`. Subject directory is the subject code. Inside of subject folder are block files. In 'rave', term 'block' is the combination of session, task, and run. Within each block, there should be 'iEEG' data files.

In 'BIDS-iEEG' format, the root directory can be obtained via `raveio_getopt('bids_data_dir')`. 'BIDS' root folder contains project folders. This is unlike 'rave' native raw data format. Subject folders are stored within the project directories. The subject folders start with `'sub-'`. Within subject folder, there are session folders with prefix `'ses-'`. Session folders are optional. 'iEEG' data is stored in `'ieeg'` folder under the session/subject folder. `'ieeg'` folder should contain at least

electrodes.tsv `sub-<label>*_electrodes.tsv`

'iEEG' description `sub-<label>*_task-<label>_run-<index>_ieeg.json`

'iEEG' data file `sub-<label>*_task-<label>_run-<index>_ieeg.<ext>`, in current 'rave', only extensions `'.vhdr+.eeg/.dat'` ('BrainVision') or 'EDF' (or plus) are supported.

When format is 'BIDS', `project_name` must be specified.

The following formats are supported:

`'.mat/.h5 file per electrode per block'` 'rave' native raw format, each block folder contains multiple 'Matlab' or 'HDF5' files. Each file corresponds to a channel/electrode. File names should follow `'xxx001.mat'` or `'xxx001.h5'`. The numbers before the extension are channel numbers.

`'Single .mat/.h5 file per block'` 'rave' native raw format, each block folder contains **only** one 'Matlab' or 'HDF5' file. The file name can be arbitrary, but extension must be either `'.mat'` or `'.h5'`. Within the file there should be a matrix containing all the data. The short dimension of the matrix will be channels, and larger side of the dimension corresponds to the time points.

`'Single EDF(+) file per block'` 'rave' native raw format, each block folder contains **only** one `'.edf'` file.

`'Single BrainVision file (.vhdr+.eeg, .vhdr+.dat) per block'` 'rave' native raw format, each block folder contains **only** two files. The first file is header `'.vhdr'` file. It contains all meta information. The second is either `'.eeg'` or `'.dat'` file containing the body, i.e. signal entries.

`'BIDS & EDF(+)'` 'BIDS' format. The data file should have `'.edf'` extension

`'BIDS & BrainVision (.vhdr+.eeg, .vhdr+.dat)'` 'BIDS' format. The data file should have `'.vhdr'+'.eeg/.dat'` extensions

Value

logical true or false whether the directory is valid. Attributes containing error reasons or snapshot of the data. The attributes might be:

`snapshot`	description of data found if passing the validation
`valid_run_names`	For 'BIDS' format, valid `session+task+run` name if passing the validation
`reason`	named list where the names are the reason why validation fails and values are corresponding sessions or electrodes or both.

`rave-server` *Install and configure 'RAVE' server as background service using shiny-server*

Description

Works on 'Linux' and 'Mac' only.

Usage

```
rave_server_install(
 url = "https://github.com/rstudio/shiny-server/archive/refs/tags/v1.5.18.987.zip"
)

rave_server_configure(
  ports = 17283,
  user = Sys.info()[["user"]],
  rave_version = c("1", "2")
)
```

Arguments

`url`	'URL' to shiny-server 'ZIP' file to download
`ports`	integer vectors or character, indicating the port numbers to host 'RAVE' instances a valid port must be within the range from 1024 to 65535.
`user`	user to run the service as; default is the login user
`rave_version`	internally used; might be deprecated in the future

Value

nothing

Examples

```
## Not run:

# OS-specific. Please install R package `rpymat` first

# Install rave-server
rave_server_install()

# Let port 17283-17290 to host RAVE instance
rave_server_configure(ports = "17283-17290")


## End(Not run)
```

`rave-snippet` *'RAVE' code snippets*

Description

Run snippet code

Usage

```
update_local_snippet(force = TRUE)

load_snippet(topic, local = TRUE)
```

Arguments

`force`	whether to force updating the snippets; default is true
`topic`	snippet topic
`local`	whether to use local snippets first before requesting online repository

Value

'load_snippet' returns snippet as a function, others return nothing

Examples

```
if(!is_on_cran()) {

  update_local_snippet()
  snippet <- load_snippet("dummy-snippet")

  # Read snippet documentation
  print(snippet)

  # Run snippet as a function
  snippet("this is an input")
}
```

`RAVEAbstarctElectrode` *Abstract definition of electrode class in RAVE*

Description

This class is not intended for direct use. Please create new child classes and implement some key methods.

Value

If `simplify` is enabled, and only one block is loaded, then the result will be a vector (`type="voltage"`) or a matrix (others), otherwise the result will be a named list where the names are the blocks.

Public fields

`subject` subject instance (`RAVESubject`)

`number` integer stands for electrode number or reference ID

`reference` reference electrode, either `NULL` for no reference or an electrode instance inherits `RAVEAbstarctElectrode`

`epoch` a `RAVEEpoch` instance

Active bindings

`type` signal type of the electrode, such as 'LFP', 'Spike', and 'EKG'; default is 'Unknown'

`power_enabled` whether the electrode can be used in power analyses such as frequency, or frequency-time analyses; this usually requires transforming the electrode raw voltage signals using signal processing methods such as 'Fourier', 'wavelet', 'Hilbert', 'multi-taper', etc. If an electrode has power data, then it's power data can be loaded via `prepare_subject_power` method.

`is_reference` whether this instance is a reference electrode

`location` location type of the electrode, see `LOCATION_TYPES` for details

`exists` whether electrode exists in subject

`preprocess_file` path to preprocess 'HDF5' file `power_file` path to power 'HDF5' file

`phase_file` path to phase 'HDF5' file

`voltage_file` path to voltage 'HDF5' file

`reference_name` reference electrode name

`epoch_name` current epoch name

`cache_root` run-time cache path; `NA` if epoch or trial intervals are missing

`trial_intervals` trial intervals relative to epoch onset

Methods

Public methods:

- `RAVEAbstarctElectrode$new()`
- `RAVEAbstarctElectrode$set_reference()`
- `RAVEAbstarctElectrode$set_epoch()`
- `RAVEAbstarctElectrode$clear_cache()`
- `RAVEAbstarctElectrode$clear_memory()`
- `RAVEAbstarctElectrode$load_data()`
- `RAVEAbstarctElectrode$load_blocks()`
- `RAVEAbstarctElectrode$clone()`

Method `new()`**:** constructor

Usage:

```
RAVEAbstarctElectrode$new(subject, number, quiet = FALSE)
```

Arguments:

`subject` character or `RAVESubject` instance

`number` current electrode number or reference ID

`quiet` reserved, whether to suppress warning messages

Method `set_reference()`**:** set reference for instance

Usage:

```
RAVEAbstarctElectrode$set_reference(reference)
```

Arguments:

`reference` `NULL` or `RAVEAbstarctElectrode` instance instance

Method `set_epoch()`**:** set epoch instance for the electrode

Usage:

```
RAVEAbstarctElectrode$set_epoch(epoch)
```

Arguments:

`epoch` characters or `RAVEEpoch` instance. For characters, make sure `"epoch_<name>.csv"` is in meta folder.

Method `clear_cache()`**:** method to clear cache on hard drive

Usage:

`RAVEAbstarctElectrode$clear_cache(...)`

Arguments:

`...` implemented by child instances

Method `clear_memory()`**:** method to clear memory

Usage:

`RAVEAbstarctElectrode$clear_memory(...)`

Arguments:

`...` implemented by child instances

Method `load_data()`**:** method to load electrode data

Usage:

`RAVEAbstarctElectrode$load_data(type)`

Arguments:

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"wavelet-coefficient"`, or others depending on child class implementations

Method `load_blocks()`**:** load electrode block-wise data (with reference), useful when epoch is absent

Usage:

`RAVEAbstarctElectrode$load_blocks(blocks, type, simplify = TRUE)`

Arguments:

`blocks` session blocks

`type` data type such as `"power"`, `"phase"`, `"voltage"`, `"wavelet-coefficient"`.

`simplify` whether to simplify the result

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`RAVEAbstarctElectrode$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:

# To run this example, please download demo subject (~700 MB) from
# https://github.com/beauchamplab/rave/releases/tag/v0.1.9-beta

generator <- RAVEAbstarctElectrode

# load demo subject electrode 14
e <- generator$new("demo/DemoSubject", number = 14)
```

```
# set epoch
e$subject$epoch_names
e$set_epoch("auditory_onset")
head(e$epoch$table)

# set epoch range (-1 to 2 seconds relative to onset)
e$trial_intervals <- c(-1,2)
# or to set multiple ranges
e$trial_intervals <- list(c(-2,-1), c(0, 2))

# set reference
e$subject$reference_names
reference_table <- e$subject$meta_data(
  meta_type = "reference",
  meta_name = "default")
ref_name <- subset(reference_table, Electrode == 14)[["Reference"]]

# the reference is CAR type, mean of electrode 13-16,24
ref_name

# load & set reference
ref <- generator$new(e$subject, ref_name)
e$set_reference(ref)


## End(Not run)
```

`RAVEEpoch` *Definition for epoch class*

Description

Trial epoch, contains the following information: `Block` experiment block/session string; `Time` trial onset within that block; `Trial` trial number; `Condition` trial condition. Other optional columns are `Event_xxx` (starts with "Event"). See https://openwetware.org/wiki/RAVE:Epoching or more details.

Value

`self$table`

If event is one of `"trial onset"`, `"default"`, `""`, or `NULL`, then the result will be `"Time"` column; if the event is found, then return will be the corresponding event column. When the event is not found and `missing` is `"error"`, error will be raised; default is to return `"Time"` column, as it's trial onset and is mandatory.

If `condition_type` is one of `"default"`, `""`, or `NULL`, then the result will be `"Condition"` column; if the condition type is found, then return will be the corresponding condition type column. When the condition type is not found and `missing` is `"error"`, error will be raised; default is to return `"Condition"` column, as it's the default and is mandatory.

Public fields

`name` epoch name, character

`subject` `RAVESubject` instance

`data` a list of trial information, internally used

`table` trial epoch table

`.columns` epoch column names, internally used

Active bindings

`columns` columns of trial table

`n_trials` total number of trials

`trials` trial numbers

`available_events` available events other than trial onset

`available_condition_type` available condition type other than the default

Methods

Public methods:

- `RAVEEpoch$new()`
- `RAVEEpoch$trial_at()`
- `RAVEEpoch$update_table()`
- `RAVEEpoch$set_trial()`
- `RAVEEpoch$get_event_colname()`
- `RAVEEpoch$get_condition_colname()`
- `RAVEEpoch$clone()`

Method `new()`**:** constructor

Usage:

```
RAVEEpoch$new(subject, name)
```

Arguments:

`subject` `RAVESubject` instance or character

`name` character, make sure `"epoch_<name>.csv"` is in meta folder

Method `trial_at()`**:** get `i`th trial

Usage:

```
RAVEEpoch$trial_at(i, df = TRUE)
```

Arguments:

`i` trial number

`df` whether to return as data frame or a list

Method `update_table()`**:** manually update table field

Usage:

```
RAVEEpoch$update_table()
```

Method `set_trial()`**:** set one trial

Usage:

```
RAVEEpoch$set_trial(Block, Time, Trial, Condition, ...)
```

Arguments:

`Block` block string

`Time` time in second

`Trial` positive integer, trial number

`Condition` character, trial condition

`...` other key-value pairs corresponding to other optional columns

Method `get_event_colname()`**:** Get epoch column name that represents the desired event

Usage:

```
RAVEEpoch$get_event_colname(
  event = "",
  missing = c("warning", "error", "none")
)
```

Arguments:

`event` a character string of the event, see `$available_events` for all available events; set to `"trial onset"`, `"default"`, or blank to use the default

`missing` what to do if event is missing; default is to warn

Method `get_condition_colname()`**:** Get condition column name that represents the desired condition type

Usage:

```
RAVEEpoch$get_condition_colname(
  condition_type,
  missing = c("warning", "error", "none")
)
```

Arguments:

`condition_type` a character string of the condition type, see `$available_condition_type` for all available condition types; set to `"default"` or blank to use the default

`missing` what to do if condition type is missing; default is to warn if the condition column is not found.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Please download DemoSubject ~700MB from
# https://github.com/beauchamplab/rave/releases/tag/v0.1.9-beta

## Not run:

# Load meta/epoch_auditory_onset.csv from subject demo/DemoSubject
epoch <-RAVEEpoch$new(subject = 'demo/DemoSubject',
                      name = 'auditory_onset')

# first several trials
head(epoch$table)

# query specific trial
old_trial1 <- epoch$trial_at(1)

# Create new trial or change existing trial
epoch$set_trial(Block = '008', Time = 10,
                Trial = 1, Condition = 'AknownVmeant')
new_trial1 <- epoch$trial_at(1)

# Compare new and old trial 1
rbind(old_trial1, new_trial1)

# To get updated trial table, must update first
epoch$update_table()
head(epoch$table)


## End(Not run)
```

---

raveio-constants *The constant variables*

Description

The constant variables

Usage

```
SIGNAL_TYPES

LOCATION_TYPES

MNI305_to_MNI152

PIPELINE_FORK_PATTERN
```

Format

An object of class `character` of length 6.

An object of class `character` of length 5.

An object of class `matrix` (inherits from `array`) with 4 rows and 4 columns.

An object of class `character` of length 1.

Details

`SIGNAL_TYPES` has the following options: `'LFP'`, `'Spike'`, `'EKG'`, `'Audio'`, `'Photodiode'`, or `'Unknown'`. As of 'raveio' `0.0.6`, only `'LFP'` (see `LFP_electrode`) signal type is supported.

`LOCATION_TYPES` is a list of the electrode location types: `'iEEG'` (this includes the next two), `'sEEG'` (stereo), `'ECoG'` (surface), `'EEG'` (scalp), `'Others'`. See field `'location'` in `RAVEAbstarctElectrode`

`MNI305_to_MNI152` is a 4-by-4 matrix converting `'MNI305'` coordinates to `'MNI152'` space. The difference of these two spaces is: `'MNI305'` is an average of 305 human subjects, while `'MNI152'` is the average of 152 people. These two coordinates differs slightly. While most of the 'MNI' coordinates reported by 'RAVE' and 'FreeSurfer' are in the `'MNI305'` space, many other programs are expecting `'MNI152'` coordinates.

`raveio-option` *Set/Get 'raveio' option*

Description

Persist settings on local configuration file

Usage

```
raveio_setopt(key, value, .save = TRUE)

raveio_resetopt(all = FALSE)

raveio_getopt(key, default = NA, temp = TRUE)

raveio_confpath(cfile = "settings.yaml")
```

Arguments

`key`	character, option name
`value`	character or logical of length 1, option value
`.save`	whether to save to local drive, internally used to temporary change option. Not recommended to use it directly.
`all`	whether to reset all non-default keys
`default`	is key not found, return default value

`temp`	when saving, whether the key-value pair should be considered temporary, a temporary settings will be ignored when saving; when getting options, setting `temp` to false will reveal the actual settings.
`cfile`	file name in configuration path

Details

`raveio_setopt` stores key-value pair in local path. The values are persistent and shared across multiple sessions. There are some read-only keys such as `"session_string"`. Trying to set those keys will result in error.

The following keys are reserved by 'RAVE':

- `data_dir` Directory path, where processed data are stored; default is at home directory, folder `~/rave_data/data_dir`
- `raw_data_dir` Directory path, where raw data files are stored, mainly the original signal files and imaging files; default is at home directory, folder `~/rave_data/raw_dir`
- `max_worker` Maximum number of CPU cores to use; default is one less than the total number of CPU cores
- `mni_template_root` Directory path, where 'MNI' templates are stored

`raveio_getopt` returns value corresponding to the keys. If key is missing, the whole option will be returned.

If set `all=TRUE`, `raveio_resetopt` resets all keys including non-standard ones. However `"session_string"` will never reset.

Value

`raveio_setopt` returns modified `value`; `raveio_resetopt` returns current settings as a list; `raveio_confpath` returns absolute path for the settings file; `raveio_getopt` returns the settings value to the given key, or `default` if not found.

See Also

`R_user_dir`

`RAVEMetaSubject` *Defines 'RAVE' subject class for meta analyses*

Description

`R6` class definition

Value

data frame

Super class

`raveio::RAVESubject` -> `RAVEMetaSubject`

Active bindings

`project` project instance of current subject; see `RAVEProject`

`project_name` character string of project name

`subject_code` character string of subject code

`subject_id` subject ID: `"project/subject"`

`path` subject root path

`rave_path` 'rave' directory under subject root path

`meta_path` meta data directory for current subject

`freesurfer_path` 'FreeSurfer' directory for current subject. If no path exists, values will be `NA`

`preprocess_path` preprocess directory under subject 'rave' path

`data_path` data directory under subject 'rave' path

`cache_path` path to 'FST' copies under subject 'data' path

`pipeline_path` path to pipeline scripts under subject's folder

`note_path` path that stores 'RAVE' related subject notes

`epoch_names` possible epoch names

`reference_names` possible reference names

`reference_path` reference path under 'rave' folder

`preprocess_settings` preprocess instance; see `RAVEPreprocessSettings`

`blocks` subject experiment blocks in current project

`electrodes` all electrodes, no matter excluded or not

`raw_sample_rates` voltage sample rate

`power_sample_rate` power spectrum sample rate

`has_wavelet` whether electrodes have wavelet transforms

`notch_filtered` whether electrodes are Notch-filtered

`electrode_types` electrode signal types

Methods

Public methods:

- `RAVEMetaSubject$print()`
- `RAVEMetaSubject$new()`
- `RAVEMetaSubject$meta_data()`
- `RAVEMetaSubject$clone()`

Method `print()`**:** override print method

Usage:

```
RAVEMetaSubject$print(...)
```

Arguments:

`...` ignored

Method `new()`**:** constructor

Usage:

```
RAVEMetaSubject$new(project_name, subject_code = NULL, strict = FALSE)
```

Arguments:

`project_name` character project name

`subject_code` character subject code

`strict` whether to check if subject folders exist

Method `meta_data()`**:** get subject meta data located in `"meta/"` folder

Usage:

```
RAVEMetaSubject$meta_data(
  meta_type = c("electrodes", "frequencies", "time_points", "epoch", "references"),
   meta_name = "default"
)
```

Arguments:

`meta_type` choices are 'electrodes', 'frequencies', 'time_points', 'epoch', 'references'

`meta_name` if `meta_type='epoch'`, read in `'epoch_<meta_name>.csv'`; if `meta_type='references'`, read in `'reference_<meta_name>.csv'`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`load_meta2`

`RAVEPreprocessSettings`

Defines preprocess configurations

Description

`R6` class definition

Value

list of electrode type, number, etc.

`NULL` when no channel is composed. When `flat` is `TRUE`, a data frame of weights with the columns composing electrode channel numbers, composed channel number, and corresponding weights; if `flat` is `FALSE`, then a weight matrix;

Public fields

`current_version` current configuration setting version

`path` settings file path

`backup_path` alternative back up path for redundancy checks

`data` list of raw configurations, internally used only

`subject` `RAVESubject` instance

`read_only` whether the configuration should be read-only, not yet implemented

Active bindings

`version` configure version of currently stored files

`old_version` whether settings file is old format

`blocks` experiment blocks

`electrodes` electrode numbers

`sample_rates` voltage data sample rate

`notch_filtered` whether electrodes are notch filtered

`has_wavelet` whether each electrode has wavelet transforms

`data_imported` whether electrodes are imported

`data_locked` whether electrode, blocks and sample rate are locked? usually when an electrode is imported into 'rave', that electrode is locked

`electrode_locked` whether electrode is imported and locked

`electrode_composed` composed electrode channels, not actual physically contacts, but is generated from those physically ones

`wavelet_params` wavelet parameters

`notch_params` Notch filter parameters

`electrode_types` electrode signal types

`@freeze_blocks` whether to free block, internally used

`@freeze_lfp_ecog` whether to freeze electrodes that record 'LFP' signals, internally used

`@lfp_ecog_sample_rate` 'LFP' sample rates, internally used

`all_blocks` characters, all possible blocks even not included in some projects

`raw_path` raw data path

`raw_path_type` raw data path type, 'native' or 'bids'

Methods

Public methods:

- `RAVEPreprocessSettings$new()`
- `RAVEPreprocessSettings$valid()`
- `RAVEPreprocessSettings$has_raw()`
- `RAVEPreprocessSettings$set_blocks()`
- `RAVEPreprocessSettings$set_electrodes()`
- `RAVEPreprocessSettings$set_sample_rates()`
- `RAVEPreprocessSettings$migrate()`
- `RAVEPreprocessSettings$electrode_info()`
- `RAVEPreprocessSettings$save()`
- `RAVEPreprocessSettings$get_compose_weights()`

Method `new()`**:** constructor

Usage:

```
RAVEPreprocessSettings$new(subject, read_only = TRUE)
```

Arguments:

`subject` character or `RAVESubject` instance

`read_only` whether subject should be read-only (not yet implemented)

Method `valid()`**:** whether configuration is valid or not

Usage:

```
RAVEPreprocessSettings$valid()
```

Method `has_raw()`**:** whether raw data folder exists

Usage:

```
RAVEPreprocessSettings$has_raw()
```

Method `set_blocks()`**:** set blocks

Usage:

```
RAVEPreprocessSettings$set_blocks(blocks, force = FALSE)
```

Arguments:

`blocks` character, combination of session task and run

`force` whether to ignore checking. Only used when data structure is not native, for example, 'BIDS' format

Method `set_electrodes()`**:** set electrodes

Usage:

```
RAVEPreprocessSettings$set_electrodes(
  electrodes,
  type = SIGNAL_TYPES,
  add = FALSE
)
```

Arguments:

`electrodes` integer vectors

`type` signal type of electrodes, see `SIGNAL_TYPES`

`add` whether to add to current settings

Method `set_sample_rates()`: set sample frequency

Usage:

```
RAVEPreprocessSettings$set_sample_rates(srate, type = SIGNAL_TYPES)
```

Arguments:

`srate` sample rate, must be positive number

`type` electrode type to set sample rate. In 'rave', all electrodes with the same signal type must have the same sample rate.

Method `migrate()`: convert old format to new formats

Usage:

```
RAVEPreprocessSettings$migrate(force = FALSE)
```

Arguments:

`force` whether to force migrate and save settings

Method `electrode_info()`: get electrode information

Usage:

```
RAVEPreprocessSettings$electrode_info(electrode)
```

Arguments:

`electrode` integer

Method `save()`: save settings to hard disk

Usage:

```
RAVEPreprocessSettings$save()
```

Method `get_compose_weights()`: get weights of each composed channels

Usage:

```
RAVEPreprocessSettings$get_compose_weights(flat = TRUE)
```

Arguments:

`flat` whether to flatten the data frame; default is true

Examples

```
# The following example require downloading demo subject (~700 MB) from
# https://github.com/beauchamplab/rave/releases/tag/v0.1.9-beta

## Not run:

conf <- RAVEPreprocessSettings$new(subject = 'demo/DemoSubject')
conf$blocks  # "008" "010" "011" "012"
```

```
conf$electrodes   # 5 electrodes

# Electrode 14 information
conf$electrode_info(electrode = 14)

conf$data_imported # All 5 electrodes are imported

conf$data_locked   # Whether block, sample rates should be locked


## End(Not run)
```

`RAVEProject` *Definition for 'RAVE' project class*

Description

Definition for 'RAVE' project class

Definition for 'RAVE' project class

Value

character vector

true or false whether subject is in the project

A data table of pipeline time-stamps and directories

Active bindings

`path` project folder, absolute path

`name` project name, character

`pipeline_path` path to pipeline scripts under project's folder

Methods

Public methods:

- `RAVEProject$print()`
- `RAVEProject$new()`
- `RAVEProject$subjects()`
- `RAVEProject$has_subject()`
- `RAVEProject$group_path()`
- `RAVEProject$subject_pipelines()`
- `RAVEProject$clone()`

Method `print()`**:** override print method

Usage:

```
RAVEProject$print(...)
```

Arguments:

`...` ignored

Method `new()`: constructor

Usage:

```
RAVEProject$new(project_name, strict = TRUE)
```

Arguments:

`project_name` character

`strict` whether to check project path

Method `subjects()`: get all imported subjects within project

Usage:

```
RAVEProject$subjects()
```

Method `has_subject()`: whether a specific subject exists in this project

Usage:

```
RAVEProject$has_subject(subject_code)
```

Arguments:

`subject_code` character, subject name

Method `group_path()`: get group data path for 'rave' module

Usage:

```
RAVEProject$group_path(module_id, must_work = FALSE)
```

Arguments:

`module_id` character, 'rave' module ID

`must_work` whether the directory must exist; if not exists, should a new one be created?

Method `subject_pipelines()`: list saved pipelines

Usage:

```
RAVEProject$subject_pipelines(
  pipeline_name,
  cache = FALSE,
  check = TRUE,
  all = FALSE
)
```

Arguments:

`pipeline_name` name of the pipeline

`cache` whether to use cached registry

`check` whether to check if the pipelines exist as directories

`all` whether to list all pipelines; default is false; pipelines with the same label but older timestamps will be hidden

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`RAVEProject$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`RAVESubject` *Defines 'RAVE' subject class*

Description

`R6` class definition

Value

data frame

integer vector of valid electrodes

The same as `value`

A named list of key-value pairs, or if one key is specified and `simplify=TRUE`, then only the value will be returned.

A data frame with four columns: `'namespace'` for the group name of the entry (entries within the same namespace usually share same module), `'timestamp'` for when the entry was registered. `'entry_name'` is the name of the entry. If `include_history` is true, then multiple entries with the same `'entry_name'` might appear since the obsolete entries are included. `'entry_value'` is the value of the corresponding entry.

If `as_table` is `FALSE`, then returns as `RAVEEpoch` instance; otherwise returns epoch table; will raise errors when file is missing or the epoch is invalid.

If `simplify` is true, returns a vector of reference electrode names, otherwise returns the whole table; will raise errors when file is missing or the reference is invalid.

If `simplify` is true, returns a vector of electrodes that are valid (or won't be excluded) under given reference; otherwise returns a table. If `subset` is true, then the table will be subset and only rows with electrodes to be loaded will be kept.

If `simplify` is true, returns a vector of frequencies; otherwise returns a table.

A table of pipeline registry

A `PipelineTools` instance

Active bindings

`project` project instance of current subject; see `RAVEProject`

`project_name` character string of project name

`subject_code` character string of subject code

`subject_id` subject ID: `"project/subject"` `path` subject root path

`rave_path` 'rave' directory under subject root path

`meta_path` meta data directory for current subject

`imaging_path` root path to imaging processing folder

`freesurfer_path` 'FreeSurfer' directory for current subject. If no path exists, values will be `NA`

`preprocess_path` preprocess directory under subject 'rave' path

`data_path` data directory under subject 'rave' path

`cache_path` path to 'FST' copies under subject 'data' path

`pipeline_path` path to pipeline scripts under subject's folder

`note_path` path that stores 'RAVE' related subject notes

`epoch_names` possible epoch names

`reference_names` possible reference names

`reference_path` reference path under 'rave' folder

`preprocess_settings` preprocess instance; see `RAVEPreprocessSettings`

`blocks` subject experiment blocks in current project

`electrodes` all electrodes, no matter excluded or not

`raw_sample_rates` voltage sample rate

`power_sample_rate` power spectrum sample rate

`has_wavelet` whether electrodes have wavelet transforms

`notch_filtered` whether electrodes are Notch-filtered

`electrode_types` electrode signal types

`electrode_composed` composed electrode channels, not actual physically contacts, but is generated from those physically ones

Methods

Public methods:

- `RAVESubject$print()`
- `RAVESubject$new()`
- `RAVESubject$meta_data()`
- `RAVESubject$valid_electrodes()`
- `RAVESubject$initialize_paths()`
- `RAVESubject$set_default()`
- `RAVESubject$get_default()`
- `RAVESubject$get_note_summary()`
- `RAVESubject$get_epoch()`
- `RAVESubject$get_reference()`
- `RAVESubject$get_electrode_table()`
- `RAVESubject$get_frequency()`
- `RAVESubject$list_pipelines()`

- `RAVESubject$load_pipeline()`
- `RAVESubject$clone()`

Method `print()`**:** override print method

Usage:

```
RAVESubject$print(...)
```

Arguments:

`...` ignored

Method `new()`**:** constructor

Usage:

```
RAVESubject$new(project_name, subject_code = NULL, strict = TRUE)
```

Arguments:

`project_name` character project name

`subject_code` character subject code

`strict` whether to check if subject folders exist

Method `meta_data()`**:** get subject meta data located in `"meta/"` folder

Usage:

```
RAVESubject$meta_data(
  meta_type = c("electrodes", "frequencies", "time_points", "epoch", "references"),
   meta_name = "default"
)
```

Arguments:

`meta_type` choices are 'electrodes', 'frequencies', 'time_points', 'epoch', 'references'

`meta_name` if `meta_type='epoch'`, read in `'epoch_<meta_name>.csv'`; if `meta_type='references'`, read in `'reference_<meta_name>.csv'`.

Method `valid_electrodes()`**:** get valid electrode numbers

Usage:

```
RAVESubject$valid_electrodes(reference_name, refresh = FALSE)
```

Arguments:

`reference_name` character, reference name, see `meta_name` in `self$meta_data` or `load_meta2` when `meta_type` is 'reference'

`refresh` whether to reload reference table before obtaining data, default is false

Method `initialize_paths()`**:** create subject's directories on hard disk

Usage:

```
RAVESubject$initialize_paths(include_freesurfer = TRUE)
```

Arguments:

`include_freesurfer` whether to create 'FreeSurfer' path

Method `set_default()`**:** set default key-value pair for the subject, used by 'RAVE' modules

Usage:

```
RAVESubject$set_default(key, value, namespace = "default")
```

Arguments:

`key` character

`value` value of the key

`namespace` file name of the note (without post-fix)

Method `get_default():` get default key-value pairs for the subject, used by 'RAVE' modules

Usage:

```
RAVESubject$get_default(
  ...,
  default_if_missing = NULL,
  simplify = TRUE,
  namespace = "default"
)
```

Arguments:

`...` single key, or a vector of character keys

`default_if_missing` default value is any key is missing

`simplify` whether to simplify the results if there is only one key to fetch; default is `TRUE`

`namespace` file name of the note (without post-fix)

Method `get_note_summary():` get summary table of all the key-value pairs used by 'RAVE' modules for the subject

Usage:

```
RAVESubject$get_note_summary(namespaces, include_history = FALSE)
```

Arguments:

`namespaces` namespaces for the entries; see method `get_default` or `set_default`. Default is all possible namespaces

`include_history` whether to include history entries; default is false

Method `get_epoch():` check and get subject's epoch information

Usage:

```
RAVESubject$get_epoch(epoch_name, as_table = FALSE, trial_starts = 0)
```

Arguments:

`epoch_name` epoch name, depending on the subject's meta files

`as_table` whether to convert to `data.frame`; default is false

`trial_starts` the start of the trial relative to epoch time; default is 0

Method `get_reference():` check and get subject's reference information

Usage:

```
RAVESubject$get_reference(reference_name, simplify = FALSE)
```

Arguments:

`reference_name` reference name, depending on the subject's meta file settings `simplify` whether to only return the reference column

Method `get_electrode_table():` check and get subject's electrode table with electrodes that are load-able

Usage:

```
RAVESubject$get_electrode_table(
  electrodes,
  reference_name,
  subset = FALSE,
  simplify = FALSE
)
```

Arguments:

`electrodes` characters indicating integers such as `"1-14,20-30"`, or integer vector of electrode numbers

`reference_name` see method `get_reference`

`subset` whether to subset the resulting data table

`simplify` whether to only return electrodes

Method `get_frequency():` check and get subject's frequency table, time-frequency decomposition is needed.

Usage:

```
RAVESubject$get_frequency(simplify = TRUE)
```

Arguments:

`simplify` whether to simplify as vector

Method `list_pipelines():` list saved pipelines

Usage:

```
RAVESubject$list_pipelines(
  pipeline_name,
  cache = FALSE,
  check = TRUE,
  all = FALSE
)
```

Arguments:

`pipeline_name` pipeline ID

`cache` whether to use cache registry to speed up

`check` whether to check if the pipelines exist

`all` whether to list all pipelines; default is false; pipelines with the same label but older timestamps will be hidden

Method `load_pipeline():` load saved pipeline

Usage:

```
RAVESubject$load_pipeline(directory)
```

Arguments:

`directory` pipeline directory name

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`load_meta2`

`RAVEVariable` *Class definition of 'RAVE' constrained variable*

Description

See `new_constrained_variable` for constructor function.

Value

Formatted characters

Self instance

Self

Current value

See `RAVEVariableConstraints`

`TRUE` if valid, otherwise returns the error message

A list of constraint data that can be passed into `$restore` method

`RAVEVariable` instance

Public fields

`name` Description of the variable

Active bindings

`constraints` instance of `RAVEVariableConstraints`, used to validate the input

`isRAVEVariable` always true

`type` constraint type

`value` value of the variable

`initialized` whether value is missing (value might not be valid)

`generator` class definition

Methods

Public methods:

- `RAVEVariable$new()`
- `RAVEVariable$format()`
- `RAVEVariable$use_constraints()`
- `RAVEVariable$set_value()`
- `RAVEVariable$get_value()`
- `RAVEVariable$validate()`
- `RAVEVariable$check()`
- `RAVEVariable$store()`
- `RAVEVariable$restore()`
- `RAVEVariable$clone()`

Method `new()`**:** Constructor function

Usage:

```
RAVEVariable$new(name = "Unnamed", initial_value)
```

Arguments:

`name` description of the variable

`initial_value` initial value; default is an empty list of class `"key_missing"`

Method `format()`**:** Format method

Usage:

```
RAVEVariable$format(prefix = NULL, ...)
```

Arguments:

`prefix` prefix of the string

`...` ignored

Method `use_constraints()`**:** Set variable validation

Usage:

```
RAVEVariable$use_constraints(constraints, .i, ...)
```

Arguments:

`constraints` either a `character(1)` or a `RAVEVariableConstraints` instance. When `constraints` is a string, the value will be the type of the constraint (see `new_constraints`)

`.i, ...` used when `constraints` is a string, either `.i` is an expression, or `list(.i,...)` forms a list of control parameters; see `assertions` in `new_constraints`.

Method `set_value()`**:** Set value

Usage:

```
RAVEVariable$set_value(
  x,
  env = parent.frame(),
  validate = TRUE,
  on_error = NULL
)
```

Arguments:

`x` value of the variable

`env` environment in which the validations will be evaluated

`validate` whether to validate if `x` is legit; if set to `TRUE` and `x` is invalid, then the values will not be set.

`on_error` a function takes two arguments: the error instance and old value; the returned value will be used to re-validate. Default is `NULL`, which is identical to returning the old value and stop on error.

Method `get_value()`**:** Get value

Usage:

```
RAVEVariable$get_value(...)
```

Arguments:

`...` ignored

Method `validate()`**:** Check if the value is valid

Usage:

```
RAVEVariable$validate(
  env = parent.frame(),
  on_error = c("error", "warning", "message", "muffle")
)
```

Arguments:

`env, on_error` passed to `RAVEVariableConstraints$assert`.

Method `check()`**:** Check if the value is valid with no error raised

Usage:

```
RAVEVariable$check(env = parent.frame())
```

Arguments:

`env` environment to evaluate validation expressions

Method `store()`**:** Convert constraint to atomic list, used for serializing

Usage:

```
RAVEVariable$store(...)
```

Arguments:

`...` ignored

Method `restore()`**:** Restores from atomic list generated by `$store()`

Usage:

```
RAVEVariable$restore(x, env = parent.frame(), ...)
```

Arguments:

`x` atomic list

`env` environment where to query the class definitions

`...` ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`RAVEVariable$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`RAVEVariableCollection`

Class definition of 'RAVE' constrained variable collection

Description

See `new_variable_collection` for construction

Value

Formatted characters

Self

The removed variable

`TRUE` if found, otherwise `FALSE`

Self

The variable value if variable if found and `get_definition` is false; or the variable definition if variable is found and is `RAVEVariable` or `RAVEVariableCollection`; or `ifnotfound` if the variable does not exist

The variable values in list

Nothing

`TRUE` if valid, or raises errors by default

`TRUE` if valid, otherwise returns the error message

A list of constraint data that can be passed into `$restore` method

`RAVEVariableCollection` instance

Public fields

`explicit` whether getting and setting values should be explicit. If true, then all variables must be defined (see `$add_variable`) before used.

Active bindings

`.wrapper` wrapper instance of current variable collection

`generator` class definition

`isRAVEVariableCollection` always true

`variables` map containing the variable definitions

`varnames` variable names

`name` descriptive name of the collection

Methods

Public methods:

- `RAVEVariableCollection$new()`
- `RAVEVariableCollection$format()`
- `RAVEVariableCollection$add_variable()`
- `RAVEVariableCollection$remove_variable()`
- `RAVEVariableCollection$has_variable()`
- `RAVEVariableCollection$set_value()`
- `RAVEVariableCollection$get_value()`
- `RAVEVariableCollection$as_list()`
- `RAVEVariableCollection$use_constraints()`
- `RAVEVariableCollection$validate()`
- `RAVEVariableCollection$check()`
- `RAVEVariableCollection$store()`
- `RAVEVariableCollection$restore()`
- `RAVEVariableCollection$clone()`

Method `new()`**:** Constructor

Usage:

```
RAVEVariableCollection$new(name = "", explicit = TRUE)
```

Arguments:

`name` descriptive name of the collection

`explicit` see field `explicit`

Method `format()`**:** Format method

Usage:

```
RAVEVariableCollection$format(...)
```

Arguments:

`...` ignored

Method `add_variable()`**:** Registers a variable, must run if the collection is explicit

Usage:

```
RAVEVariableCollection$add_variable(id, var)
```

Arguments:

`id` variable 'ID'

`var` a `RAVEVariable` or `RAVEVariableCollection` instance if the variable is bounded, or simply normal R object (then the variable will have no constraint)

Method `remove_variable()`**:** Remove a variable

Usage:

```
RAVEVariableCollection$remove_variable(id)
```

Arguments:

`id` variable 'ID'

Method `has_variable()`**:** Check whether a variable exists

Usage:

```
RAVEVariableCollection$has_variable(id)
```

Arguments:

`id` variable 'ID'

Method `set_value()`**:** Set value of a variable

Usage:

```
RAVEVariableCollection$set_value(id, value, env = parent.frame(), ...)
```

Arguments:

`id` variable 'ID'

`value` the value to be set

`env, ...` passed to `RAVEVariable$set_value`

Method `get_value()`**:** Get value of a variable

Usage:

```
RAVEVariableCollection$get_value(
  id,
  env = parent.frame(),
  get_definition = FALSE,
  ifnotfound = NULL
)
```

Arguments:

`id` variable 'ID'

`env` environment of evaluation

`get_definition` whether to return the variable definition instance (`RAVEVariable` or `RAVEVariableCollection`) instead of the value; default is false

`ifnotfound` default value if not found; default is `NULL`

Method `as_list()`**:** Convert to list

Usage:

```
RAVEVariableCollection$as_list(env = parent.frame())
```

Arguments:

`env` environment of evaluation

Method `use_constraints()`**:** Set collection validation

Usage:

```
RAVEVariableCollection$use_constraints(x)
```

Arguments:

`x` either a `NULL` or an expression with global variables `x`, `self`, `private`, and `defs` Mainly used to validate the values of multiple variables (some variables are dependent or bounded by other variables)

Method `validate()`**:** Run validation

Usage:

```
RAVEVariableCollection$validate(
  env = parent.frame(),
  on_error = c("error", "warning", "message", "muffle")
)
```

Arguments:

`env` environment to evaluate validation expressions

`on_error` character, error handler

Method `check()`**:** Check if the value is valid with no error raised

Usage:

```
RAVEVariableCollection$check(env = parent.frame())
```

Arguments:

`env` environment to evaluate validation expressions

Method `store()`**:** Convert constraint to atomic list, used for serializing

Usage:

```
RAVEVariableCollection$store(...)
```

Arguments:

`...` ignored

Method `restore()`**:** Restores from atomic list generated by `$store()`

Usage:

```
RAVEVariableCollection$restore(x, env = parent.frame(), clear = FALSE, ...)
```

Arguments:

`x` atomic list

`env` environment where to query the class definitions

`clear` whether to clear the current variables; default is false

`...` ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`RAVEVariable`

`RAVEVariableConstraints` *Class definition for 'RAVE' variable constraints*

Description

See `new_constraints` for constructor function.

Value

Initialized instance

Formatted characters

Either `TRUE` if passed or a collection of assertion failures (or errors)

`TRUE` if valid, otherwise returns the error message

A list of constraint data that can be passed into `$restore` method

`RAVEVariableConstraints` instance

Public fields

`type` `character(1)`, type indicator

Active bindings

`n_validators` Number of validators

`isRAVEVariableConstraints` always true

`generator` class definition

Methods

Public methods:

- `RAVEVariableConstraints$new()`
- `RAVEVariableConstraints$format()`
- `RAVEVariableConstraints$assert()`
- `RAVEVariableConstraints$check()`
- `RAVEVariableConstraints$store()`
- `RAVEVariableConstraints$restore()`
- `RAVEVariableConstraints$clone()`

Method `new()`**:** Constructor method

Usage:

```
RAVEVariableConstraints$new(type = "UnboundedConstraint", assertions = NULL)
```

Arguments:

`type` type of the variable; default is `'UnboundedConstraint'` `assertions` named list of the constraint parameters. The names of `assertions` will be used to indicate the constraint type, and the values are the constraint parameters.

Method `format()`**:** Format method

Usage:

```
RAVEVariableConstraints$format(...)
```

Arguments:

`...` ignored

Method `assert()`**:** Validate the constraints

Usage:

```
RAVEVariableConstraints$assert(
  x,
  .var.name = checkmate::vname(x),
  on_error = c("error", "warning", "message", "muffle"),
  env = parent.frame(),
  data = NULL
)
```

Arguments:

`x` value to validate

`.var.name` descriptive name of `x`

`on_error` error handler, default is `'error'`: stop on first validation error

`env` environment of validation (used when assertions are expressions)

`data` named list of additional data to be used for evaluation if constraint is an expression

Method `check()`**:** Check if the value is valid with no error raised

Usage:

```
RAVEVariableConstraints$check(x, env = parent.frame(), data = NULL)
```

Arguments:

`x` valid to be validated

`env` environment to evaluate validation expressions

`data` named list of additional data to be used for evaluation if constraint is an expression

Method `store()`**:** Convert constraint to atomic list, used for serializing

Usage:

```
RAVEVariableConstraints$store(...)
```

Arguments:

`...` ignored

Method `restore()`**:** Restores from atomic list generated by `$store()`

Usage:

```
RAVEVariableConstraints$restore(x, ...)
```

Arguments:

`x` atomic list

`...` ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`rave_brain` *Load 'FreeSurfer' or 'AFNI/SUMA' brain from 'RAVE'*

Description

Create 3D visualization of the brain and visualize with modern web browsers

Usage

```
rave_brain(
  subject,
  surfaces = "pial",
  use_141 = TRUE,
  recache = FALSE,
  clean_before_cache = FALSE,
  compute_template = FALSE,
  usetemplateifmissing = FALSE,
  include_electrodes = TRUE
)
```

Arguments

`subject`	character, list, or `RAVESubject` instance; for list or other objects, make sure `subject$subject_id` is a valid 'RAVE' subject 'ID'
`surfaces`	one or more brain surface types from `"pial"`, `"white"`, `"smoothwm"`, `"pial-outer-smoothed"`, etc.; check `freesurfer_brain2`
`use_141`	whether to use 'AFNI/SUMA' standard 141 brain
`recache`	whether to re-calculate cache; only should be used when the original 'FreeSurfer' or 'AFNI/SUMA' files are changed; such as new files are added
`clean_before_cache`	whether to clean the original cache before `recache`; only set it to be true if original cached files are corrupted
`compute_template`	whether to compute template mappings; useful when template mapping with multiple subjects are needed

`usetemplateifmissing`	whether to use template brain when the subject brain files are missing. If set to true, then a template (usually 'N27') brain will be displayed as an alternative solution, and electrodes will be rendered according to their `'MNI305'` coordinates, or `'VertexNumber'` if given.
`include_electrodes`	whether to include electrode in the model; default is true

Value

A `'threeBrain'` instance if brain is found or `usetemplateifmissing` is set to true; otherwise returns `NULL`

Examples

```
# Please make sure DemoSubject is correctly installed
# The subject is ~1GB from Github

if(interactive()){
  brain <- rave_brain("demo/DemoSubject")

  if( !is.null(brain) ) { brain$plot() }

}
```

`rave_command_line_path` *Find and execute external command-line tools*

Description

Find and execute external command-line tools

Usage

```
normalize_commandline_path(
  path,
  type = c("dcm2niix", "freesurfer", "fsl", "afni", "others"),
  unset = NA
)

cmd_dcm2niix(error_on_missing = TRUE, unset = NA)

cmd_freesurfer_home(error_on_missing = TRUE, unset = NA)
```

```
cmd_fsl_home(error_on_missing = TRUE, unset = NA)

cmd_afni_home(error_on_missing = TRUE, unset = NA)

cmd_homebrew(error_on_missing = TRUE, unset = NA)

is_dry_run()
```

Arguments

`path`	path to normalize
`type`	type of command
`unset`	default to return if the command is not found
`error_on_missing`	whether to raise errors if command is missing

Value

Normalized path to the command, or `unset` if command is missing.

`rave_directories` *Returns a list of 'RAVE' directories*

Description

This function is internally used and should not be called directly.

Usage

```
rave_directories(
  subject_code,
  project_name,
  blocks = NULL,
  .force_format = c("", "native", "BIDS")
)
```

Arguments

`subject_code`	'RAVE' subject code
`project_name`	'RAVE' project name
`blocks`	session or block names, optional
`.force_format`	format of the data, default is automatically detected.

Value

A list of directories

`rave_export` *Export 'RAVE' data*

Description

Export portable data for custom analyses.

Usage

```
rave_export(x, path, ...)

## Default S3 method:
rave_export(x, path, format = c("rds", "yaml", "json"), ...)

## S3 method for class 'rave_prepare_subject_raw_voltage_with_epoch'
rave_export(x, path, zip = FALSE, ...)

## S3 method for class 'rave_prepare_subject_voltage_with_epoch'
rave_export(x, path, zip = FALSE, ...)

## S3 method for class 'rave_prepare_power'
rave_export(x, path, zip = FALSE, ...)
```

Arguments

`x`	R object or 'RAVE' repositories
`path`	path to save to
`...`	passed to other methods
`format`	export format
`zip`	whether to zip the files

Value

Exported data path

Examples

```
x <- "my data"
path <- tempfile()
rave_export(x, path)

readRDS(path)

## Not run:
  # Needs demo subject
  path <- tempfile()
  x <- prepare_subject_power("demo/DemoSubject")
```

```
  # Export power data to path
  rave_export(x, path)

## End(Not run)
```

`rave_import` *Import data into 'rave' projects*

Description

Import files with predefined structures. Supported file formats include 'Matlab', 'HDF5', 'EDF(+)', 'BrainVision' (`'.eeg/.dat/.vhdr'`). Supported file structures include 'rave' native structure and 'BIDS' (very limited) format. Please see https://openwetware.org/wiki/RAVE:ravepreprocess for tutorials.

Usage

```
rave_import(
  project_name,
  subject_code,
  blocks,
  electrodes,
  format,
  sample_rate,
  conversion = NA,
  data_type = "LFP",
  task_runs = NULL,
  add = FALSE,
  ...
)
```

Arguments

`project_name`	project name, for 'rave' native structure, this can be any character; for 'BIDS' format, this must be consistent with 'BIDS' project name. For subjects with multiple tasks, see Section "'RAVE' Project"
`subject_code`	subject code in character. For 'rave' native structure, this is a folder name under raw directory. For 'BIDS', this is subject label without `"sub-"` prefix
`blocks`	characters, for 'rave' native format, this is the folder names subject directory; for 'BIDS', this is session name with `"ses-"`. Section "Block vs. Session" for different meaning of "blocks" in 'rave' and 'BIDS'
`electrodes`	integers electrode numbers
`format`	integer from 1 to 6, or character. For characters, you can get options by running `names(IMPORT_FORMATS)`

`sample_rate`	sample frequency, must be positive
`conversion`	physical unit conversion, choices are `NA`, `V`, `mV`, `uV`
`data_type`	electrode signal type; see `SIGNAL_TYPES`
`task_runs`	for 'BIDS' formats only, see Section "Block vs. Session"
`add`	whether to add electrodes. If set to true, then only new electrodes are allowed to be imported, blocks will be ignored and trying to import electrodes that have been imported will still result in error.
`...`	other parameters

Value

None

'RAVE' Project

A 'rave' project can be very flexible. A project can refer to a task, a research objective, or "arbitrarily" as long as you find common research interests among subjects. One subject can appear in multiple projects with different blocks, hence `project_name` should be objective-based. There is no concept of "project" in 'rave' raw directory. When importing data, you choose subset of blocks from subjects forming a project.

When importing 'BIDS' data into 'rave', `project_name` must be consistent with 'BIDS' project name as a compromise. Once imported, you may change the project folder name in imported rave data directory to other names. Because once raw traces are imported, 'rave' data will become self-contained and 'BIDS' data are no longer required for analysis. This naming inconsistency will also be ignored.

Block vs. Session

'rave' and 'BIDS' have different definitions for a "chunk" of signals. In 'rave', we use "block". it means combination of session (days), task, and run, i.e. a block of continuous signals captured. Raw data files are supposed to be stored in file hierarchy of `<raw-root>/<subject_code>/<block>/<datafiles>`. In 'BIDS', sessions, tasks, and runs are separated, and only session names are indicated under subject folder. Because some previous compatibility issues, argument `'block'` refers to direct folder names under subject directories. This means when importing data from 'BIDS' format, `block` argument needs to be session names to comply with `'subject/block'` structure, and there is an additional mandatory argument `task_runs` especially designed for 'BIDS' format.

For 'rave' native raw data format, `block` will be as-is once imported.
For 'BIDS' format, `task_runs` will be treated as blocks once imported.

File Formats

Following file structure. Here use project `"demo"` and subject `"YAB"` and block `"008"`), electrode `14` as an example.

`format=1`**, or** `".mat/.h5 file per electrode per block"` folder `<raw>/YAB/008` contains 'Matlab' or 'HDF5' files per electrode. Data file name should look like `"xxx_14.mat"` `format=2`**, or** `"Single .mat/.h5 file per block"` `<raw>/YAB/008` contains only one 'Matlab' or 'HDF5' file. Data within the file should be a 2-dimensional matrix, where the column 14 is signal recorded from electrode 14

`format=3`**, or** `"Single EDF(+) file per block"` `<raw>/YAB/008` contains only one `'edf'` file

`format=4`**, or** `"Single BrainVision file (.vhdr+.eeg, .vhdr+.dat) per block"` `<raw>/YAB/008` contains only one `'vhdr'` file, and the data file must be inferred from the header file

`format=5`**, or** `"BIDS & EDF(+)"` `<bids>/demo/sub-YAB/ses-008/` must contains `*_electrodes.tsv`, each run must have channel file. The channel files and electrode file must be consistent in names.
Argument `task_runs` is mandatory, characters, combination of session, task name, and run number. For example, a task header file in BIDS with name `'sub-YAB_ses-008_task-visual_run-01_ieeg.edf'` has `task_runs` name as `'008-visual-01'`, where the first `'008'` refers to session, `'visual'` is task name, and the second `'01'` is run number.

`format=6`**, or** `"BIDS & BrainVision (.vhdr+.eeg, .vhdr+.dat)"` Same as previous format `"BIDS & EDF(+)"`, but data files have 'BrainVision' formats.

`rave_subject_format_conversion` *Compatibility support for 'RAVE' 1.0 format*

Description

Convert 'RAVE' subject generated by 2.0 pipeline such that 1.0 modules can use the data. The subject must have valid electrodes. The data must be imported, with time-frequency transformed to pass the validation before converting.

Usage

```
rave_subject_format_conversion(subject, verbose = TRUE, ...)
```

Arguments

`subject`	'RAVE' subject characters, such as `'demo/YAB'`, or a subject instance generated from `RAVESubject`
`verbose`	whether to verbose the messages
`...`	ignored, reserved for future use

Value

Nothing

`read-brainvision-eeg` *Load from 'BrainVision' file*

Description

Read in `'eeg'` or `'ieeg'` data from 'BrainVision' files with `.eeg` or `.dat` extensions.

Usage

```
read_eeg_header(file)

read_eeg_marker(file)

read_eeg_data(header, path = NULL)
```

Arguments

`file`	path to `'vhdr'` header file
`header`	header object returned by `read_eeg_header`
`path`	optional, path to data file if original data file is missing or renamed; must be absolute path.

Details

A 'BrainVision' dataset is usually stored separately in header file (`.vhdr`), marker file (`.vmrk`, optional) and data file (`.eeg` or `.dat`). These files must store under a same folder to be read into R.

Header data contains channel information. Data "channel" contains channel name, reference, resolution and physical unit. "resolution" times digital data values is the physical value of the recorded data. `read_eeg_data` makes this conversion internally . "unit" is the physical unit of recordings. By default `'uV'` means micro-volts.

Marker file that ends with `.vmrk` is optional. If the file is indicated by header file and exists, then a marker table will be included when reading headers. A marker table contains six columns: marker number, type, description, start position (in data point), size (duration in data points), and target channel (0 means applied for all channels).

Signal file name is usually contained within header file. Therefore it is desired that the signal file name never changed once created. However, in some cases when the signal files are renamed and cannot be indexed by header files, please specify `path` to force load signals from a different file.

Value

`read_eeg_header` returns a list containing information below:

`raw`	raw header contents
`common`	a list of descriptors of header
`channels`	table of channels, including number, reference, resolution and unit

`sample_rate`	sampling frequency
`root_path`	directory to where the data is stored
`channel_counts`	total channel counts
`markers`	`NULL` if marker file is missing, or list of marker description and table containing 6 columns.

`read_eeg_data` returns header, signal data and data description:

`data`	a matrix of signal values. Each row is a channel and each column is a time point.

Examples

```
header_file <- 'sub-01_ses-01_task-visual_run-01_ieeg.vhdr'

if( file.exists(header_file) ){
  # load a subject header
  header <- read_eeg_header(header_file)

  # load entire signal
  data <- read_eeg_data(header)

  data$description
}
```

`read-write-fst` *Read a 'fst' file*

Description

Read a 'fst' file

Usage

```
save_fst(x, path, ...)

load_fst(path, ..., as.data.table = TRUE)
```

Arguments

`x`	data frame to write to path
`path`	path to 'fst' file: must not be connection.
`...`	passed to `read_fst` or `write_fst`
`as.data.table`	passed to `read_fst` in `fst` package

`read_csv_ieeg` *Read comma separated value file and ignore headers*

Description

Resolved some irregular 'iEEG' format where the header could be missing.

Usage

```
read_csv_ieeg(file, nrows = Inf, drop = NULL)
```

Arguments

`file`	comma separated value file to read from. The file must contains all numerical values
`nrows`	number of rows to read
`drop`	passed to `fread`

Details

The function checks the first two rows of comma separated value file If the first row has different `storage.mode` than the second row, then the first row is considered header, otherwise header is treated missing. Note `file` must have at least two rows.

`read_edf_header` *Read 'EDF(+)' or 'BDF(+)' file headers*

Description

Wrapper of `readEdfHeader`, but added some information

Usage

```
read_edf_header(path)
```

Arguments

`path`	file path, passed to `readEdfHeader`

Details

The added names are: `isAnnot2`, `sampleRate2`, and `unit2`. To avoid conflict with other names, there is a "2" appended to each names. `isAnnot2` indicates whether each channel is annotation channel or recorded signals. `sampleRate2` is a vector of sample rates for each channels. `unit2` is physical unit of recorded signals. For 'iEEG' data, this is electric potential unit, and choices are `'V'` for volt, `'mV'` for millivolt, and `'uV'` for micro-volt. For more details, see `https://www.edfplus.info/specs/edftexts.html`

Value

A list is header information of an 'EDF/BDF' file.

See Also

`readEdfHeader`

`read_edf_signal` *Read 'EDF(+)' or 'BDF(+)' file signals*

Description

Read 'EDF(+)' or 'BDF(+)' file signals

Usage

```
read_edf_signal(
  path,
  signal_numbers = NULL,
  convert_volt = c("NA", "V", "mV", "uV")
)
```

Arguments

`path`	file path, passed to `readEdfHeader`
`signal_numbers`	channel/electrode numbers
`convert_volt`	convert voltage (electric potential) to a new unit, `NA` means no conversion, other choices are `'V'`, `'mV'`, and `'uV'`.

Value

A list containing header information, signal lists, and channel/electrode names. If `signal_numbers` is specified, the corresponding names should appear as `selected_signal_names`. `get_signal()` can get physical signals after unit conversion.

`read_mat` *Read 'Matlab' files*

Description

A compatible reader that can read both 'Matlab' files prior and after version 6.0

Usage

```
read_mat(file, ram = TRUE, engine = c("r", "py"))

read_mat2(file, ram = TRUE, engine = c("r", "py"))
```

Arguments

`file`	path to a 'Matlab' file
`ram`	whether to load data into memory. Only available when the file is in 'HDF5' format. Default is false and will load arrays, if set to true, then lazy-load data. This is useful when array is very large.
`engine`	method to read the file, choices are `'r'` and `'py'` ('Python'); if `'py'` is chosen, make sure `configure_conda` is configured.

Details

`readMat` can only read 'Matlab' files prior to version 6. After version 6, 'Matlab' uses 'HDF5' format to store its data, and `read_mat` can handle both cases.

The performance of `read_mat` can be limited when the file is too big or has many datasets as it reads all the data contained in 'Matlab' file into memory.

Value

A list of All the data stored in the file

See Also

`readMat`, `load_h5`

Examples

```
# Matlab .mat <= v7.3
x <- matrix(1:16, 4)
f <- tempfile()
R.matlab::writeMat(con = f, x = x)

read_mat(f)

# Matlab .mat >= v7.3, using hdf5
# Make sure you have installed hdf5r
```

```
if( dipsaus::package_installed('hdf5r') ){

f <- tempfile()
save_h5(x, file = f, name = 'x')

read_mat(f)

# For v7.3, you don't have to load all data into RAM
dat <- read_mat(f, ram = FALSE)
dat

dat$x[]

}
```

`read_nsx_nev` *Read 'BlackRock' event and signal files*

Description

Current implementation supports minimum 2.3 file specification version. Please contact the package maintainer to add specification configurations if you want us to support older versions.

Usage

```
read_nsx_nev(
  paths,
  nev_path = NULL,
  header_only = FALSE,
  nev_data = TRUE,
  verbose = TRUE,
  ram = FALSE,
  force_update = FALSE,
  temp_path = file.path(tempdir(), "blackrock-temp")
)
```

Arguments

`paths`	'NSx' signal files, usually with file extensions such as `'.ns1'`, `'.ns2'`, `'.ns3'`, `'.ns4'`, `'.ns5'`.
`nev_path`	'NEV' event files, with file extension `'.nev'`
`header_only`	whether to load header information only and avoid reading signal arrays
`nev_data`	whether to load `'.nev'` comments and 'waveforms'
`verbose`	whether to print out progress when loading signal array

`ram`	whether to load signals into the memory rather than storing with `filearray`; default is false
`force_update`	force updating the channel data even if the headers haven't changed
`temp_path`	temporary directory to store the channel data

`safe_read_csv` *Read comma separated value files with given column classes*

Description

Read comma separated value files with given column classes

Usage

```
safe_read_csv(
  file,
  header = TRUE,
  sep = ",",
  colClasses = NA,
  skip = 0,
  quote = "\"",
  ...,
  stringsAsFactors = FALSE
)
```

Arguments

`file`, `header`, `sep`, `colClasses`, `skip`, `quote`, `stringsAsFactors`, `...`	passed to `read.csv`

Details

Reading a comma separated value file using builtin function `read.csv` might result in some unexpected behavior. `safe_read_csv` does some preprocessing on the format so that it take cares of the following cases.

1. If `skip` exceeds the maximum rows of the data, return a blank data frame instead of raising error.

2. If row names are included in the file, `colClasses` automatically skip that column and starts from the second column

3. If length of `colClasses` does not equal to the number of columns, instead of cycling the class types, we set those columns to be `NA` type and let `read.csv` decide the default types.

4. `stringsAsFactors` is by default `FALSE` to be consistent with R 4.0, if the function is called in R 3.x.

Value

A data frame

Examples

```
f <- tempfile()
x <- data.frame(a = letters[1:10], b = 1:10, c = 2:11)

# ------------------ Auto-detect row names ------------------
# Write with rownames
utils::write.csv(x, f, row.names = LETTERS[2:11])

# read csv with base library utils
table1 <- utils::read.csv(f, colClasses = c('character', 'character'))

# 4 columns including row names
str(table1)

# read csv via safe_read_csv
table2 <- safe_read_csv(f, colClasses = c('character', 'character'))

# row names are automatically detected, hence 3 columns
# Only first columns are characters, the third column is auto
# detected as numeric
str(table2)

# read table without row names
utils::write.csv(x, f, row.names = FALSE)
table2 <- safe_read_csv(f, colClasses = c('character', 'character'))

# still 3 columns, and row names are 1:nrow
str(table2)

# --------------- Blank data frame when nrow too large ---------------
# instead of raising errors, return blank data frame
safe_read_csv(f, skip = 1000)
```

`safe_write_csv` *Save data to comma separated value files with backups*

Description

Save comma separated value files, if file exists, backup original file.

Usage

```
safe_write_csv(x, file, ..., quiet = FALSE)
```

Arguments

x, `file`, `...`	pass to `write.csv`
`quiet`	whether to suppress overwrite message

Value

Normalized path of `file`

Examples

```
f <- tempfile()
x <- data.frame(a = 1:10)

# File not exists, same as write file, returns normalized `f`
safe_write_csv(x, f)

# Check whether file exists
file.exists(f)

# write again, and the old file will be copied
safe_write_csv(x, f)
```

---

save_h5 *Save objects to 'HDF5' file without trivial checks*

Description

Save objects to 'HDF5' file without trivial checks

Usage

```
save_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
`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

`load_h5`

Examples

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

# save x to file with name /group/dataset/1
save_h5(x, file, '/group/dataset/1', chunk = dim(x))

# read data
y <- load_h5(file, '/group/dataset/1')
y[]
```

`save_json` *Save or load R object in 'JSON' format*

Description

Save or load R object in 'JSON' format

Usage

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

load_json(con, ..., map = NULL)
```

Arguments

`x`	R object to save
`con`	file or connection
`...`	other parameters to pass into `toJSON` or `fromJSON`
`digits`	number of digits to save
`pretty`	whether the output should be pretty
`serialize`	whether to save a serialized version of `x`; see 'Examples'.
`map`	a map to save the results

Value

`save_json` returns nothing; `load_json` returns an R object.

Examples

```
# Serialize
save_json(list(a = 1, b = function(){}))

# use toJSON
save_json(list(a = 1, b = function(){}), serialize = FALSE)


# Demo of using serializer
f1 <- tempfile(fileext = ".json")
save_json(x ~ y + 1, f1)

load_json(f1)

unlink(f1)
```

`save_meta2` *Function to save meta data to 'RAVE' subject*

Description

Function to save meta data to 'RAVE' subject

Usage

```
save_meta2(data, meta_type, project_name, subject_code)
```

Arguments

`data`	data table
`meta_type`	see load meta
`project_name`	project name
`subject_code`	subject code

Value

Either none if no meta matched or the absolute path of file saved.

`save_yaml` *Write named list to file*

Description

Write named list to file

Usage

```
save_yaml(x, file, ..., sorted = FALSE)
```

Arguments

`x`	a named list, `fastmap2`, or anything that can be transformed into named list via `as.list`
`file, ...`	passed to `write_yaml`
`sorted`	whether to sort the results by name; default is false

Value

Normalized file path

See Also

`fastmap2`, `load_yaml`, `read_yaml`, `write_yaml`

Examples

```
x <- list(a = 1, b = 2)
f <- tempfile()

save_yaml(x, f)

load_yaml(f)

map <- dipsaus::fastmap2(missing_default = NA)
```

```
map$c <- 'lol'
load_yaml(f, map = map)

map$a
map$d
```

`Tensor` *R6 Class for large Tensor (Array) in Hybrid Mode*

Description

can store on hard drive, and read slices of GB-level data in seconds

Value

self

the sliced data

a data frame with the dimension names as index columns and `value_name` as value column

original array

the collapsed data

Public fields

`dim` dimension of the array

`dimnames` dimension names of the array

`use_index` whether to use one dimension as index when storing data as multiple files

`hybrid` whether to allow data to be written to disk

`last_used` timestamp of the object was read

`temporary` whether to remove the files once garbage collected

Active bindings

`varnames` dimension names (read-only)

`read_only` whether to protect the swap files from being changed

`swap_file` file or files to save data to

Methods

Public methods:

- `Tensor$finalize()`
- `Tensor$print()`
- `Tensor$.use_multi_files()`
- `Tensor$new()`
- `Tensor$subset()`
- `Tensor$flatten()`
- `Tensor$to_swap()`
- `Tensor$to_swap_now()`
- `Tensor$get_data()`
- `Tensor$set_data()`
- `Tensor$collapse()`
- `Tensor$operate()`

Method `finalize()`**:** release resource and remove files for temporary instances

Usage:

```
Tensor$finalize()
```

Method `print()`**:** print out the data dimensions and snapshot

Usage:

```
Tensor$print(...)
```

Arguments:

`...` ignored

Method `.use_multi_files()`**:** Internally used, whether to use multiple files to cache data instead of one

Usage:

```
Tensor$.use_multi_files(mult)
```

Arguments:

`mult` logical

Method `new()`**:** constructor

Usage:

```
Tensor$new(
  data,
  dim,
  dimnames,
  varnames,
  hybrid = FALSE,
  use_index = FALSE,
  swap_file = temp_tensor_file(),
  temporary = TRUE,
  multi_files = FALSE
)
```

Arguments:

`data` numeric array

`dim` dimension of the array

`dimnames` dimension names of the array

`varnames` characters, names of `dimnames`

`hybrid` whether to enable hybrid mode

`use_index` whether to use the last dimension for indexing

`swap_file` where to store the data in hybrid mode files to save data by index; default stores in `raveio_getopt('tensor_temp_path')`

`temporary` whether to remove temporary files when existing

`multi_files` if `use_index` is true, whether to use multiple

Method `subset()`**:** subset tensor

Usage:

```
Tensor$subset(..., drop = FALSE, data_only = FALSE, .env = parent.frame())
```

Arguments:

`...` dimension slices

`drop` whether to apply `drop` on subset data

`data_only` whether just return the data value, or wrap them as a `Tensor` instance

`.env` environment where `...` is evaluated

Method `flatten()`**:** converts tensor (array) to a table (data frame)

Usage:

```
Tensor$flatten(include_index = FALSE, value_name = "value")
```

Arguments:

`include_index` logical, whether to include dimension names

`value_name` character, column name of the value

Method `to_swap()`**:** Serialize tensor to a file and store it via `write_fst`

Usage:

```
Tensor$to_swap(use_index = FALSE, delay = 0)
```

Arguments:

`use_index` whether to use one of the dimension as index for faster loading

`delay` if greater than 0, then check when last used, if not long ago, then do not swap to hard drive. If the difference of time is greater than `delay` in seconds, then swap immediately.

Method `to_swap_now()`**:** Serialize tensor to a file and store it via `write_fst` immediately

Usage:

```
Tensor$to_swap_now(use_index = FALSE)
```

Arguments:

`use_index` whether to use one of the dimension as index for faster loading

Method `get_data()`**:** restore data from hard drive to memory

Usage:

```
Tensor$get_data(drop = FALSE, gc_delay = 3)
```

Arguments:

`drop` whether to apply `drop` to the data

`gc_delay` seconds to delay the garbage collection

Method `set_data()`**:** set/replace data with given array

Usage:

```
Tensor$set_data(v)
```

Arguments:

`v` the value to replace the old one, must have the same dimension

`notice` the a tensor is an environment. If you change at one place, the data from all other places will change. So use it carefully.

Method `collapse()`**:** apply mean, sum, or median to collapse data

Usage:

```
Tensor$collapse(keep, method = "mean")
```

Arguments:

`keep` which dimensions to keep

`method` `"mean"`, `"sum"`, or `"median"`

Method `operate()`**:** apply the tensor by anything along given dimension

Usage:

```
Tensor$operate(
  by,
  fun = .Primitive("/"),
  match_dim,
  mem_optimize = FALSE,
  same_dimension = FALSE
)
```

Arguments:

`by` R object

`fun` function to apply

`match_dim` which dimensions to match with the data

`mem_optimize` optimize memory

`same_dimension` whether the return value has the same dimension as the original instance

Examples

```
if(!is_on_cran()){

# Create a tensor
ts <- Tensor$new(
  data = 1:18000000, c(3000,300,20),
  dimnames = list(A = 1:3000, B = 1:300, C = 1:20),
```

```
  varnames = c('A', 'B', 'C'))

# Size of tensor when in memory is usually large
# `lobstr::obj_size(ts)` -> 8.02 MB

# Enable hybrid mode
ts$to_swap_now()

# Hybrid mode, usually less than 1 MB
# `lobstr::obj_size(ts)` -> 814 kB

# Subset data
start1 <- Sys.time()
subset(ts, C ~ C < 10 & C > 5, A ~ A < 10)
#> Dimension:  9 x 300 x 4
#> - A: 1, 2, 3, 4, 5, 6,...
#> - B: 1, 2, 3, 4, 5, 6,...
#> - C: 6, 7, 8, 9
end1 <- Sys.time(); end1 - start1
#> Time difference of 0.188035 secs

# Join tensors
ts <- lapply(1:20, function(ii){
  Tensor$new(
    data = 1:9000, c(30,300,1),
    dimnames = list(A = 1:30, B = 1:300, C = ii),
    varnames = c('A', 'B', 'C'), use_index = 2)
})
ts <- join_tensors(ts, temporary = TRUE)

}
```

`test_hdspeed` *Simple hard disk speed test*

Description

Simple hard disk speed test

Usage

```
test_hdspeed(
  path = tempdir(),
  file_size = 1e+06,
  quiet = FALSE,
  abort_if_slow = TRUE,
  use_cache = FALSE
)
```

Arguments

`path`	an existing directory where to test speed, default is temporary local directory.
`file_size`	in bytes, default is 1 MB.
`quiet`	should verbose messages be suppressed?
`abort_if_slow`	abort test if hard drive is too slow. This usually happens when the hard drive is connected via slow internet: if the write speed is less than 0.1 MB per second.
`use_cache`	if hard drive speed was tested before, abort testing and return cached results or not; default is false.

Value

A vector of two: writing and reading speed in MB per seconds.

`time_diff2` *Calculate time difference in seconds*

Description

Calculate time difference in seconds

Usage

```
time_diff2(start, end, units = "secs", label = "")
```

Arguments

`start, end`	start and end of timer
`units`	passed to `time_delta`
`label`	`rave-units` label for display purpose.

Value

A number inherits `rave-units` class.

See Also

`as_rave_unit`

Examples

```
start <- Sys.time()
Sys.sleep(0.1)
end <- Sys.time()
dif <- time_diff2(start, end, label = 'Running ')
print(dif, digits = 4)

is.numeric(dif)

dif + 1
```

`url_neurosynth` *Get 'Neurosynth' website address using 'MNI152' coordinates*

Description

Get 'Neurosynth' website address using 'MNI152' coordinates

Usage

```
url_neurosynth(x, y, z)
```

Arguments

`x, y, z`	numerical values: the right-anterior-superior 'RAS' coordinates in `'MNI152'` space

Value

'Neurosynth' website address

`validate_subject` *Validate subject data integrity*

Description

Check against existence, validity, and consistency

Arguments

`subject`	subject ID (character), or `RAVESubject` instance
`method`	validation method, choices are `'normal'` (default) or `'basic'` for fast checks; if set to `'normal'`, four additional validation parts will be tested (see parts with `*` in Section 'Value').
`verbose`	whether to print out the validation messages
`version`	data version, choices are `1` for 'RAVE' 1.0 data format, and `2` ('RAVE' 2.0 data format); default is `2`

Value

A list of nested validation results. The validation process consists of the following parts in order:

Data paths (`paths`)

`path` the subject's root folder

`path` the subject's 'RAVE' folder (the `'rave'` folder under the root directory)

`raw_path` the subject's raw data folder

`data_path` a directory storing all the voltage, power, phase data (before reference)

`meta_path` meta directory containing all the electrode coordinates, reference table, epoch information, etc.

`reference_path` a directory storing calculated reference signals

`preprocess_path` a directory storing all the preprocessing information

`cache_path` **(low priority)** data caching path

`freesurfer_path` **(low priority)** subject's 'FreeSurfer' directory

`note_path` **(low priority)** subject's notes

`pipeline_path` **(low priority)** a folder containing all saved pipelines for this subject

Preprocessing information (`preprocess`)

`electrodes_set` whether the subject has a non-empty electrode set

`blocks_set` whether the session block length is non-zero

`sample_rate_set` whether the raw sampling frequency is set to a valid, proper positive number

`data_imported` whether all the assigning electrodes have been imported

`notch_filtered` whether all the 'LFP' and 'EKG' signals have been 'Notch' filtered

`has_wavelet` whether all the 'LFP' signals are wavelet-transformed

`has_reference` at least one reference has been generated in the meta folder

`has_epoch` at least one epoch file has been generated in the meta folder

`has_electrode_file` meta folder has `electrodes.csv` file

Meta information (`meta`)

`meta_data_valid` this item only exists when the previous preprocess validation is failed or incomplete

`meta_electrode_table` the `electrodes.csv` file in the meta folder has correct format and consistent electrodes numbers to the preprocess information

`meta_reference_xxx` (`xxx` will be replaced with actual reference names) checks whether the reference table contains all electrodes and whether each reference data exists

`meta_epoch_xxx` (`xxx` will be replaced with actual epoch names) checks whether the epoch table has the correct formats and whether there are missing blocks indicated in the epoch files

Voltage data (`voltage_data*`)

`voltage_preprocessing` whether the raw preprocessing voltage data are valid. This includes data lengths are the same within the same blocks for each signal type

`voltage_data` whether the voltage data (after 'Notch' filters) exist and readable. Besides, the lengths of the data must be consistent with the raw signals

Spectral power and phase (`power_phase_data*`)

`power_data` whether the power data exists for all 'LFP' signals. Besides, to pass the validation process, the frequency and time-point lengths must be consistent with the preprocess record

`power_data` same as `power_data` but for the phase data

Epoch table (`epoch_tables*`) One or more sub-items depending on the number of epoch tables. To pass the validation, the event time for each session block must not exceed the actual signal duration. For example, if one session lasts for 200 seconds, it will invalidate the result if a trial onset time is later than 200 seconds.

Reference table (`reference_tables*`) One or more sub-items depending on the number of reference tables. To pass the validation, the reference data must be valid. The inconsistencies, for example, missing file, wrong frequency size, invalid time-point lengths will result in failure

---

`validate_time_window` *Validate time windows to be used*

Description

Make sure the time windows are valid intervals and returns a reshaped window list

Usage

```
validate_time_window(time_windows)
```

Arguments

`time_windows`	vectors or a list of time intervals

Value

A list of time intervals (ordered, length of 2)

Examples

```

# Simple time window
validate_time_window(c(-1, 2))

# Multiple windows
validate_time_window(c(-1, 2, 3, 5))

# alternatively
validate_time_window(list(c(-1, 2), c(3, 5)))
validate_time_window(list(list(-1, 2), list(3, 5)))


## Not run:

# Incorrect usage (will raise errors)
```

```
  # Invalid interval (length must be two for each intervals)
  validate_time_window(list(c(-1, 2, 3, 5)))

  # Time intervals must be in ascending order
  validate_time_window(c(2, 1))


## End(Not run)
```

`voltage_baseline` *Calculate voltage baseline*

Description

Calculate voltage baseline

Usage

```
voltage_baseline(
  x,
  baseline_windows,
  method = c("percentage", "zscore", "subtract_mean"),
  units = c("Trial", "Electrode"),
  ...
)

## S3 method for class 'rave_prepare_subject_raw_voltage_with_epoch'
voltage_baseline(
  x,
  baseline_windows,
  method = c("percentage", "zscore", "subtract_mean"),
  units = c("Trial", "Electrode"),
  electrodes,
  baseline_mean,
  baseline_sd,
  ...
)

## S3 method for class 'rave_prepare_subject_voltage_with_epoch'
voltage_baseline(
  x,
  baseline_windows,
  method = c("percentage", "zscore", "subtract_mean"),
  units = c("Trial", "Electrode"),
  electrodes,
```

```
  baseline_mean,
  baseline_sd,
  ...
)

## S3 method for class 'FileArray'
voltage_baseline(
  x,
  baseline_windows,
  method = c("percentage", "zscore", "subtract_mean"),
  units = c("Trial", "Electrode"),
  filebase = NULL,
  ...
)

## S3 method for class 'array'
voltage_baseline(
  x,
  baseline_windows,
  method = c("percentage", "zscore", "subtract_mean"),
  units = c("Trial", "Electrode"),
  ...
)
```

Arguments

`x`	R array, `filearray`, or `'rave_prepare_power'` object created by `prepare_subject_raw_voltage_wit`
`baseline_windows`	list of baseline window (intervals)
`method`	baseline method; choices are `'percentage'` and `'zscore'`; see 'Details' in `baseline_array`
`units`	the unit of the baseline; see 'Details'
`...`	passed to other methods
`electrodes`	the electrodes to be included in baseline calculation; for power repository object produced by `prepare_subject_power` only; default is all available electrodes in each of `signal_types`
`baseline_mean, baseline_sd`	internally used by 'RAVE' repository, provided baseline is not contained in the data. This is useful for calculating the baseline with data from other blocks.
`filebase`	where to store the output; default is `NULL` and is automatically determined

Details

The arrays must be three-mode tensor and must have valid named `dimnames`. The dimension names must be `'Trial'`, `'Time'`, `'Electrode'`, case sensitive.

The `baseline_windows` determines the baseline windows that are used to calculate time-points of baseline to be included. This can be one or more intervals and must pass the validation function `validate_time_window`.

The `units` determines the unit of the baseline. It can be either or both of `'Trial'`, `'Electrode'`. The default value is both, i.e., baseline for each combination of trial and electrode.

Value

The same type as the inputs

Examples

```
## Not run:
# The following code need to download additional demo data
# Please see https://rave.wiki/ for more details

library(raveio)
repo <- prepare_subject_raw_voltage_with_epoch(
  subject = "demo/DemoSubject",
  time_windows = c(-1, 3),
  electrodes = c(14, 15))

##### Direct baseline on repository
voltage_baseline(
  x = repo, method = "zscore",
  baseline_windows = list(c(-1, 0), c(2, 3))
)

voltage_mean <- repo$raw_voltage$baselined$collapse(
  keep = c(1,3), method = "mean")
matplot(voltage_mean, type = "l", lty = 1,
        x = repo$raw_voltage$dimnames$Time,
        xlab = "Time (s)", ylab = "Voltage (z-scored)",
        main = "Mean coltage over trial (Baseline: -1~0 & 2~3)")
abline(v = 0, lty = 2, col = 'darkgreen')
text(x = 0, y = -0.5, "Aud-Onset ", col = "darkgreen", cex = 0.6, adj = c(1,1))

##### Alternatively, baseline on each electrode channel
voltage_mean2 <- sapply(repo$raw_voltage$data_list, function(inst) {
  re <- voltage_baseline(
    x = inst, method = "zscore",
    baseline_windows = list(c(-1, 0), c(2, 3)))
  rowMeans(re[])
})

# Same with floating difference
max(abs(voltage_mean - voltage_mean2)) < 1e-8



## End(Not run)
```

`with_future_parallel` *Enable parallel computing provided by ’future’ package within the context*

Description

Enable parallel computing provided by ’future’ package within the context

Usage

```
with_future_parallel(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  on_failure = "multisession",
  max_workers = NA,
  ...
)
```

Arguments

`expr`	the expression to be evaluated
`env`	environment of the `expr`
`quoted`	whether `expr` has been quoted; default is false
`on_failure`	alternative ’future’ plan to use if forking a process is disallowed; this usually occurs on ’Windows’ machines; see details.
`max_workers`	maximum of workers; default is automatically set by `raveio_getopt("max_worker",1L)`
`...`	additional parameters passing into `make_forked_clusters`

Details

Some ’RAVE’ functions such as `prepare_subject_power` support parallel computing to speed up. However, the parallel computing is optional. You can enable it by wrapping the function calls within `with_future_parallel` (see examples).

The default plan is to use ’forked’ R sessions. This is a convenient, fast, and relative simple way to create multiple R processes that share the same memories. However, on some machines such as ’Windows’ the support has not yet been implemented. In such cases, the plan fall backs to a back-up specified by `on_failure`. By default, `on_failure` is `'multisession'`, a heavier implementation than forking the process, and slightly longer ramp-up time. However, the difference should be marginal for most of the functions.

When parallel computing is enabled, the number of parallel workers is specified by the option `raveio_getopt("max_worker", 1L)`.

Value

The evaluation results of `expr`

Examples

```
library(raveio)

demo_subject <- as_rave_subject("demo/DemoSubject", strict = FALSE)

if(dir.exists(demo_subject$path)) {
  with_future_parallel({
    prepare_subject_power("demo/DemoSubject")
  })
}
```

`YAELProcess` *Class definition of* `'YAEL'` *image pipeline*

Description

Rigid-registration across multiple types of images, non-linear normalization from native brain to common templates, and map template atlas or 'ROI' back to native brain. See examples at `as_yael_process`

Value

whether the image has been set (or replaced)

Absolute path if the image

'RAVE' subject instance

Nothing

A list of moving and fixing images, with rigid transformations from different formats.

See method `get_template_mapping`

A list of input, output images, with forward and inverse transform files (usually two `'Affine'` with one displacement field)

transformed image in 'ANTs' format

transformed image in 'ANTs' format

Nothing

A matrix of 3 columns, each row is a transformed points (invalid rows will be filled with `NA`)

A matrix of 3 columns, each row is a transformed points (invalid rows will be filled with `NA`)

Active bindings

`subject_code` 'RAVE' subject code

`work_path` Working directory ('RAVE' imaging path)

Methods

Public methods:

- `YAELProcess$new()`
- `YAELProcess$set_input_image()`
- `YAELProcess$get_input_image()`
- `YAELProcess$get_subject()`
- `YAELProcess$register_to_T1w()`
- `YAELProcess$get_native_mapping()`
- `YAELProcess$map_to_template()`
- `YAELProcess$get_template_mapping()`
- `YAELProcess$transform_image_from_template()`
- `YAELProcess$transform_image_to_template()`
- `YAELProcess$generate_atlas_from_template()`
- `YAELProcess$transform_points_to_template()`
- `YAELProcess$transform_points_from_template()`
- `YAELProcess$construct_ants_folder_from_template()`
- `YAELProcess$get_brain()`
- `YAELProcess$clone()`

Method `new()`**:** Constructor to instantiate the class

Usage:

```
YAELProcess$new(subject_code)
```

Arguments:

`subject_code` character code representing the subject

Method `set_input_image()`**:** Set the raw input for different image types

Usage:

```
YAELProcess$set_input_image(
  path,
  type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  overwrite = FALSE,
  on_error = c("warning", "error", "ignore")
)
```

Arguments:

`path` path to the image files in `'NIfTI'` format

`type` type of the image

`overwrite` whether to overwrite existing images if the same type has been imported before; default is false

`on_error` when the file exists and `overwrite` is false, how should this error be reported; choices are `'warning'` (default), `'error'` (throw error and abort), or `'ignore'`.

Method `get_input_image()`**:** Get image path

Usage:

```
YAELProcess$get_input_image(
  type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR")
)
```

Arguments:

`type` type of the image

Method `get_subject()`**:** Get 'RAVE' subject instance

Usage:

```
YAELProcess$get_subject(project_name = "YAEL", strict = FALSE)
```

Arguments:

`project_name` project name; default is `'YAEL'`

`strict` passed to `as_rave_subject`

Method `register_to_T1w()`**:** Register other images to 'T1' weighted 'MRI'

Usage:

```
YAELProcess$register_to_T1w(
  image_type = c("CT", "T2w", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  reverse = FALSE,
  verbose = TRUE
)
```

Arguments:

`image_type` type of the image to register, must be set via `process$set_input_image` first.

`reverse` whether to reverse the registration; default is false, meaning the fixed (reference) image is the `'T1'`. When setting to true, then the `'T1'` 'MRI' will become the moving image

`verbose` whether to print out the process; default is true

Method `get_native_mapping()`**:** Get the mapping configurations used by `register_to_T1w`

Usage:

```
YAELProcess$get_native_mapping(
  image_type = c("CT", "T2w", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  relative = FALSE
)
```

Arguments:

`image_type` type of the image registered to 'T1' weighted 'MRI'

`relative` whether to use relative path (to the `work_path` field)

Method `map_to_template()`**:** Normalize native brain to `'MNI152'` template

Usage:

```
YAELProcess$map_to_template(
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
 native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  verbose = TRUE
)
```

Arguments:

`template_name` which template to use, choices are `'mni_icbm152_nlin_asym_09a'`, `'mni_icbm152_nlin_asym_09b'` `'mni_icbm152_nlin_asym_09c'`.

`native_type` which type of image should be used to map to template; default is `'T1w'`

`verbose` whether to print out the process; default is true

Method `get_template_mapping()`**:** Get configurations used for normalization

Usage:

```
YAELProcess$get_template_mapping(
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  relative = FALSE
)
```

Arguments:

`template_name` which template is used

`native_type` which native image is mapped to template

`relative` whether the paths should be relative or absolute; default is false (absolute paths)

Method `transform_image_from_template()`**:** Apply transform from images (usually an atlas or 'ROI') on template to native space

Usage:

```
YAELProcess$transform_image_from_template(
  template_roi_path,
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  interpolator = c("auto", "nearestNeighbor", "linear", "gaussian", "bSpline",
    "cosineWindowedSinc", "welchWindowedSinc", "hammingWindowedSinc",
    "lanczosWindowedSinc", "genericLabel"),
  verbose = TRUE
)
```

Arguments:

`template_roi_path` path to the template image file which will be transformed into individuals' image

`template_name` templates to use

`native_type` which type of native image to use for calculating the coordinates (default `'T1w'`)

`interpolator` how to interpolate the `'voxels'`; default is `"auto"`: `'linear'` for probabilistic map and `'nearestNeighbor'` otherwise.

`verbose` whether the print out the progress

Method `transform_image_to_template()`**:** Apply transform to images (usually an atlas or 'ROI') from native space to template

Usage:

```
YAELProcess$transform_image_to_template(
  native_roi_path,
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  interpolator = c("auto", "nearestNeighbor", "linear", "gaussian", "bSpline",
    "cosineWindowedSinc", "welchWindowedSinc", "hammingWindowedSinc",
    "lanczosWindowedSinc", "genericLabel"),
  verbose = TRUE
)
```

Arguments:

`native_roi_path` path to the native image file that will be transformed into template

`template_name` templates to use

`native_type` which type of native image to use for calculating the coordinates (default `'T1w'`)

`interpolator` how to interpolate the `'voxels'`; default is `"auto"`: `'linear'` for probabilistic map and `'nearestNeighbor'` otherwise.

`verbose` whether the print out the progress

Method `generate_atlas_from_template()`**:** Generate atlas maps from template and morph to native brain

Usage:

```
YAELProcess$generate_atlas_from_template(
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  atlas_folder = NULL,
  surfaces = NA,
  verbose = TRUE
)
```

Arguments:

`template_name` which template to use

`atlas_folder` path to the atlas folder (that contains the atlas files)

`surfaces` whether to generate surfaces (triangle mesh); default is `NA` (generate if not existed). Other choices are `TRUE` for always generating and overwriting surface files, or `FALSE` to disable this function. The generated surfaces will stay in native `'T1'` space.

`verbose` whether the print out the progress

Method `transform_points_to_template()`**:** Transform points from native images to template

Usage:

```
YAELProcess$transform_points_to_template(
  native_ras,
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  verbose = TRUE
)
```

Arguments:

`native_ras` matrix or data frame with 3 columns indicating points sitting on native images in right-anterior-superior (`'RAS'`) coordinate system.

`template_name` template to use for mapping

`native_type` native image type where the points sit on

`verbose` whether the print out the progress

Method `transform_points_from_template()`: Transform points from template images to native

Usage:

```
YAELProcess$transform_points_from_template(
  template_ras,
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  native_type = c("T1w", "T2w", "CT", "FLAIR", "preopCT", "T1wContrast", "fGATIR"),
  verbose = TRUE
)
```

Arguments:

`template_ras` matrix or data frame with 3 columns indicating points sitting on template images in right-anterior-superior (`'RAS'`) coordinate system.

`template_name` template to use for mapping

`native_type` native image type where the points sit on

`verbose` whether the print out the progress

Method `construct_ants_folder_from_template()`: Create a reconstruction folder (as an alternative option) that is generated from template brain to facilitate the '3D' viewer. Please make sure method `map_to_template` is called before using this method (or the program will fail)

Usage:

```
YAELProcess$construct_ants_folder_from_template(
  template_name = c("mni_icbm152_nlin_asym_09a", "mni_icbm152_nlin_asym_09b",
    "mni_icbm152_nlin_asym_09c"),
  add_surfaces = TRUE
)
```

Arguments:

`template_name` template to use for mapping

`add_surfaces` whether to create surfaces that is morphed from template to local; default is `TRUE`. Please enable this option only if the cortical surfaces are not critical (for example, you are studying the deep brain structures). Always use `'FreeSurfer'` if cortical information is used.

Method `get_brain()`: Get '3D' brain model

Usage:

```
YAELProcess$get_brain(
  electrodes = TRUE,
  project_name = "YAEL",
```

```
  coord_sys = c("scannerRAS", "tkrRAS", "MNI152", "MNI305"),
  ...
)
```

Arguments:

`electrodes` whether to add electrodes to the viewers; can be logical, data frame, or a character (path to electrode table). When the value is `TRUE`, the electrode file under `project_name` will be loaded; when `electrodes` is a `data.frame`, or path to a `'csv'` file, then please specify `coord_sys` on what is the coordinate system used for columns `"x"`, `"y"`, and `"z"`.

`project_name` project name under which the electrode table should be queried, if `electrodes=TRUE`

`coord_sys` coordinate system if `electrodes` is a data frame with columns `"x"`, `"y"`, and `"z"`, available choices are `'scannerRAS'` (defined by 'T1' weighted native 'MRI' image), `'tkrRAS'` (`'FreeSurfer'` defined native 'TK-registered'), `'MNI152'` (template 'MNI' coordinate system averaged over 152 subjects; this is the common "'MNI' coordinate space" we often refer to), and `'MNI305'` (template 'MNI' coordinate system averaged over 305 subjects; this coordinate system used by templates such as `'fsaverage'`)

`...` passed to `threeBrain`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Index