mripy

Release 0.6.21

herrlich10

Sep 22, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

`mripy` is a collection of handy small tools for analyzing neuroimaging data (esp. high resolution fMRI), which can be used both as a Python package and a set of command line tools. It is a useful augmentation to the AFNI tool chain.

CHAPTER

TWO

INSTALLATION (IN LINUX/MAC)

2.1 Install `AFNI` and `FreeSurfer`

Please follow the instructions in their official websites (AFNI, FreeSurfer).

2.2 Install `mripy`

Download and install the `mripy` package:

```
$ pip install mripy
```

Set `$PATH` for using the package scripts in the terminal. For bash, the commands look like:

```
$ vi ~/.bashrc
$ export PATH="path/to/mripy/scripts":$PATH
```

The path could be something like "~/anaconda3/lib/Python3.8/site-packages/mripy/scripts".

Download and install the dependencies:

```
$ pip install nibabel deepdish
```

2.3 Install `neuropythy` `docker` (for HCP retinotopy altas)

First install docker for your OS, then pull the `neuropythy` image:

```
# Pull a particular version
$ docker pull nben/neuropythy@sha256:
↪2541ee29a8d6bc676d9c3622ef4a38a258dd90e06c02534996a1c8354f9ac888

# Give it a tag
$ docker tag b38ebfcf6477 nben/neuropythy:mripy
```

More information about the HCP retinotopy altas and the neuropythy package can be found in Noah C. Benson's website.

CHAPTER

THREE

MRIPY PACKAGE

3.1 Subpackages

3.1.1 mripy.io package

Submodules

mripy.io.freesurfer module

mripy.io.freesurfer.**read_fs_annotation**(*fname*, *return_df=False*)

Returns
nodes

Return type
int array

References

https://github.com/fieldtrip/fieldtrip/blob/master/external/freesurfer/read_annotation.m

mripy.io.freesurfer.**read_fs_curv**(*fname*)

Read FreeSurfer surface data binary file (big endian).

Returns
curv – Per vertex data value (e.g., curvature, thickness, sulc, etc.).

Return type
float array

References

https://github.com/fieldtrip/fieldtrip/blob/master/external/freesurfer/read_curv.m http://www.grahamwideman.com/gw/brain/fs/surfacefileformats.htm

mripy.io.freesurfer.**read_fs_int32**(*fi*)

Read big endian 4-byte integer from opened binary file.

mripy.io.freesurfer.**read_fs_patch**(*fname*)

Read FreeSurfer surface patch binary file (big endian).

Returns

- **vtx** (*int array*) – The stored vtx value encoding both nodes and border information.
- **verts** (*Nx3 float array, [x, y, z]*)
- **nodes** (*int array*) – The node index of the patch vertices on the original inflated surface. The numbers are obviously non-contiguous.
- **border** (*bool array*) – Whether the vertex is on the border of the patch.

References

https://rdrr.io/cran/freesurferformats/src/R/read_fs_patch.R http://www.grahamwideman.com/gw/brain/fs/surfacefileformats.htm (obsoleted)

mripy.io.freesurfer.**read_fs_str**(*fi*)

mripy.io.freesurfer.**read_fs_surf**(*fname*)

Read FreeSurfer surface mesh binary file (big endian).

Returns

- **verts** (*Nx3 float array, [x, y, z]*)
- **faces** (*Nx3 int array, [v1, v2, v3]*)

References

https://github.com/fieldtrip/fieldtrip/blob/master/external/freesurfer/read_surf.m http://www.grahamwideman.com/gw/brain/fs/surfacefileformats.htm

mripy.io.freesurfer.**read_fs_uint24**(*fi*)

Read big endian 3-byte unsigned integer from opened binary file.

mripy.io.freesurfer.**write_fs_color_table**(*fname*, *color_table*)

Write FreeSurfer color table file (**.ctab*), similar to {subject}/label/aparc.annot.a2009s.ctab, or https://surfer.nmr.mgh.harvard.edu/fswiki/FsTutorial/AnatomicalROI/FreeSurferColorLUT

This file will be useful to generate a valid annot.niml.dset containing correct color table, which can then be used in **.spec* to display custom anatomical labels at SUMA crosshair location. E.g., >>> FSread_annot -input lh.HCP-MMP1.annot -FSversion 2009 -FScmap lh.HCP-MMP1.ctab -FScmaprange 0 180 -dset lh.HCP-MMP1.annot.niml.dset -overwrite

mripy.io.freesurfer.**write_fs_curv**(*fname*, *curv*)

Write FreeSurfer surface data binary file (big endian).

Parameters

- **curv** (*1D array like*) – Surface data, each vertex must have one and only one value.
- **version.** (*Only support writing in the new binary*) –

mripy.io.freesurfer.**write_fs_int32**(*fo*, *x*)

Write big endian 4-byte integer to opened binary file.

mripy.io.freesurfer.**write_fs_uint24**(*fo*, *x*)

Write big endian 3-byte unsigned integer to opened binary file.

mripy.io.gifti module

mripy.io.gifti.**read_gii_dset**(*fname*)

Read GIFTI surface data in XML format.

mripy.io.niml module

mripy.io.niml.**parse_attr**(*attr*)

Split a single attribute into key and value.

Notes

NIML attributes are in the general form "attname=string", separated by whitespace. XML allows whitespace to occur around the "=" that separates the attname from the string. NIML does not allow this whitespace; the next character after attname must be "=", and the next character after that must be a Name character or a quote character.

We don't need to deal with the quotes around value here, leaving it to xml.etree.TreeBuilder.

mripy.io.niml.**parse_data**(*between*, *fmt*)

Parse data stream from between-tag content (can be empty). If ni_type exists, the data is converted into a numpy array.

TODO: Handle escape sequence (<, >, ", &) in text data.

mripy.io.niml.**parse_data_format**(*attrs*)

Parse data stream format based on ni_form, ni_type, ni_dimen, etc. for both binary, base64, and text data.

mripy.io.niml.**parse_ni_type**(*ni_type*, *flatten=None*)

Parse ni_type into numpy dtype.

The method exploits the flexibilty of np.dtype() in a recursive manner. Assumptions which seem to be contradicting the NIML specification:

1. The standard type is not abbreviated, e.g., int
2. The multiple type is indicated by "*", e.g., 4*int
3. The compound type is separated by ",", e.g., 4*float,int,String

Parameters

- **ni_type** (*str*) –
- **flatten** (*bool*) – Limited to create non-hierarchical structured dtype, e.g., interpreting "4*float,int,String" as "float,float,float,float,int,String".

mripy.io.niml.**parse_niml**(*fname*)

Parse NIML file into Python xml.etree.Element using incremental event-driven parsing.

References

https://afni.nimh.nih.gov/pub/dist/src/niml/NIML_base.html

mripy.io.niml.**read_until**(*fi*, *end_token*, *batch_size=1024*)

Read until encountering a specific character.

Useful for event-driven parsing, especially for large binary file like NIML dset. This method is designed to work with both str and bytes.

Module contents

class mripy.io.**BallMask**(*master*, *c*, *r*)

Bases: *Mask*

class mripy.io.**CylinderMask**(*master*, *c*, *r*)

Bases: *Mask*

class mripy.io.**Mask**(*master=None*, *kind='mask'*)

Bases: object

ball(*c*, *r*, ***kwargs*)

compatible(*other*)

classmethod concat(*masks*)

constrain(*func*, *return_selector=False*, *inplace=False*)

Parameters

func (*callable*) – selector = func(x, y, z) is used to select a subset of self.index

cylinder(*c*, *r*, ***kwargs*)

The elongated axis is represented as nan

dump(*fname*, *dtype=None*)

classmethod from_dict(*d*)

classmethod from_expr(*expr=None*, ***kwargs*)

classmethod from_files(*files*, *combine='union'*, ***kwargs*)

property ijk

infer_selector(*smaller*)

near(*x*, *y*, *z*, *r*, ***kwargs*)

mm

pick(*selector*, *inplace=False*)

slab(*x1=None*, *x2=None*, *y1=None*, *y2=None*, *z1=None*, *z2=None*, ***kwargs*)

to_dict()

to_file(*fname*, *undump_value=False*)

undump(*prefix*, *x*, *method='nibabel'*, *space=None*)

property xyz

property xyz_nifti

class mripy.io.**MaskDumper**(*mask_file*)

Bases: object

dump(*fname*)

undump(*prefix*, *x*)

class mripy.io.**SlabMask**(*master*, *x1=None*, *x2=None*, *y1=None*, *y2=None*, *z1=None*, *z2=None*)

Bases: *Mask*

mripy.io.**change_dim_order**(*in_file*, *out_file=None*, *dim_order=None*, *method='afni'*)

Parameters

- **dim_order** (*1D array with 8 numbers*) –

```
>>> np.array([  5, 300, 300, 124,   1,   2,   1,   1], dtype=np.
↪int16) # for stats
>>> np.array([  4, 150, 150,  62, 158,   1,   1,   1], dtype=np.
↪int16) # for epi
```

- **method** (*str, 'afni' | 'nibabel'*) –

mripy.io.**change_space**(*in_file*, *out_file=None*, *space=None*, *method='nibabel'*)

```
>>> change_space('MNI152_2009_template.nii.gz', 'template.nii', space='ORIG')
>>> change_space('test+tlrc.HEAD') # -> test.nii as ORIG
```

mripy.io.**compress**(*in_file*, *out_file=None*)

mripy.io.**convert_dicom**(*dicom_dir*, *out_file=None*, *dicom_ext=None*, *interactive=False*, *extra_cmd=None*)

mripy.io.**convert_dicoms**(*dicom_dirs*, *out_dir=None*, *prefix=None*, *out_type='.nii'*, *dicom_ext='.IMA'*, ***kwargs*)

Parameters

- **dicom_dirs** (*list or str*) –
  1. A list of folders containing **.IMA* files
  2. It can also be a glob pattern that describes a list of folders, e.g., "raw_fmri/func??"
  3. Finally, it can be a root folder (e.g., "raw_fmri") containing multiple sub-folders of **.IMA* files, raw_fmri/anat, raw_fmri/func01, raw_fmri/func02, etc.
- **out_dir** (*str*) – Output directory for converted datasets, default is current directory. The output would look like:

```
out_dir/anat.nii, out_dir/func01.nii, out_dir/func02.nii, etc.
```

mripy.io.**decompress**(*in_file*, *out_file=None*)

mripy.io.**extract_physio**(*physio_file*, *dicom_file*, *TR=None*, *dummy=0*, *channels=['resp', 'puls']*, *verbose=1*)

mripy.io.**filter_cluster**(*in_file*, *out_file*, *top=None*, *neighbor=2*)

neighbor
[int] 1 : face touch 2 : edge touch (default, as in afni) 3 : corner touch

mripy.io.**filter_dicom_files**(*files*, *series_numbers=None*, *instance_numbers=None*, *series_pattern='.+?\\.(\\d{4})\\.(\\d{4}).+(\\d+)\\.\\d+'*)

mripy.io.**generate_afni_idcode**()

mripy.io.**get_dim_order**(*in_file*)

mripy.io.**get_ni_type**(*x*)

mripy.io.**get_space**(*in_file*)

mripy.io.**hms2dt**(*hms*, *date=None*, *timestamp=False*)
Convert time string in hms format to datetime object.

hms is like "102907.165000". This format is used in dicom header.

mripy.io.**match_physio_with_series**(*physio_infos*, *series_infos*, *channel=None*, *method='cover'*)

mripy.io.**mmn2dt**(*mmn*, *date=None*, *timestamp=False*)
Convert time string in mmn format to datetime object.

mmn is "msec since midnight", like "37747165". This format is used in physiological measurement log file.

mripy.io.**parse_dicom_header**(*fname*, *fields=None*)
Execute afni command *dicom_hdr* to readout most useful info from dicom header.

Parameters

- **fname** (*str*) –
- **fields** (*{field: (matcher, extracter(match))}*) – You can require additional fields in dicom header to be parsed. - field : e.g., 'ImageTime' - matcher : e.g. r'ID Image Time//(S+)' - extracter : e.g., lambda match: io.hms2dt(match.group(1), date='20170706', timestamp=True)

mripy.io.**parse_physio_file**(*fname*, *date=None*)

Notes

IMPLEMENTATION

1. The first 4 (ext, puls, resp) or 5 (ecg) values are parameters (of unknown meanings).
2. There can be multiple data lines, within which extra parameters is inclosed between 5002 and 6002, especially for ecg.
3. The footer is inclosed between 5003 and 6003, following physiological data (and that's why the final data value always appears to be 5003).
4. The MDH values are timestamps derived from the clock in the scanner (so do DICOM images), while the MPCU values are timestamps derived from the clock within the PMU recording system [1]. Use MDH time to synchronize physiological and imaging time series.
5. The trigger values (5000) are "inserted" into the data, and have to be stripped out from the time series [1]. This fact is double checked by looking at the smooth trend of the puls waveform.
6. The sampling rate is slightly (and consistently) slower than specified in the manual and in [1].

ABOUT TIMING

The scanner clock is slightly faster than the wall clock so that 2 sec in real time is recorded as ~2.008 sec in the scanner, affacting both dicom header and physiological footer, even though the actual TR is precisely 2 s (as measured by timing the s triggers with psychtoolbox) and the actual sampling rate of physiological data is precisely 50 Hz (as estimated by dividing the total number of samples by the corrected recording duration).

References

[1] https://cfn.upenn.edu/aguirre/wiki/public:pulse-oximetry_during_fmri_scanning

mripy.io.**parse_physio_files**(*fname*, *date=None*, *channels=None*)

mripy.io.**parse_series_info**(*fname*, *timestamp=False*, *shift_time=None*, *series_pattern='.+?\\.(\\d{4})\\.'*, *fields=None*, *parser=None*)

Potential bug: *dicom.parse_dicom_header* doesn't support *fields* as kwargs

mripy.io.**parse_slice_order**(*dicom_files*)

mripy.io.**read_affine**(*fname*, *sep=None*)

Returns
mat

Return type
3x4 or Nx3x4

mripy.io.**read_afni**(*fname*, *remove_nii=True*, *return_img=False*)

mripy.io.**read_asc**(*fname*, *dtype=None*)

Read FreeSurfer/SUMA surface (vertices and faces) in **.asc* format.

mripy.io.**read_gii**(*fname*, *return_img=False*)

mripy.io.**read_label**(*fname*)

Read FreeSurfer label

mripy.io.**read_nii**(*fname*, *return_img=False*)

mripy.io.**read_niml_bin_nodes**(*fname*)

Read "Node Bucket" (node indices and values) from niml (binary) dataset.

mripy.io.**read_niml_dset**(*fname*, *tags=None*, *as_asc=True*, *return_type='list'*)

mripy.io.**read_patch_asc**(*fname*, *dtype=None*, *index_type='multimap'*)

Read FreeSurfer/SUMA patch (noncontiguous vertices and faces) in **.asc* format.

index_type
[str]

- "raw" or "array"
- "map" or "dict"
- "multimap" or "func"

mripy.io.**read_register_dat**(*fname*)

mripy.io.**read_stim**(*fname*)

mripy.io.**read_surf_data**(*fname*)

mripy.io.**read_surf_info**(*fname*)

mripy.io.**read_surf_mesh**(*fname*, *return_img=False*, ***kwargs*)

mripy.io.**read_txt**(*fname*, *dtype=<class 'float'>*, *comment='#'*, *delimiter=None*, *skiprows=0*, *nrows=None*, *return_comments=False*)

Read numerical array from text file, much faster than np.loadtxt()

mripy.io.**read_vol**(*fname*, *return_img=False*)

mripy.io.**read_warp**(*fname*)

References

[1] https://afni.nimh.nih.gov/pub/dist/doc/program_help/3dQwarp.html
"An AFNI nonlinear warp dataset stores the displacements (in DICOM mm) from the base dataset grid to the source dataset grid. AFNI stores a 3D warp as a 3-volume dataset (NiFTI or AFNI format), with the voxel values being the displacements in mm (32-bit floats) needed to 'reach out' and bring (interpolate) another dataset into alignment – that is, 'pulling it back' to the grid defined in the warp dataset header."

mripy.io.**sort_dicom_series**(*folder*, *series_pattern='.+?\\.(\\d{4})\\.'*)

Parameters
folder (*string*) – Path to the folder containing all the **.IMA* files.

Returns
studies – [{'0001': [file0, file1, ...], '0002': [files], ...}, {study1}, ...]

Return type
list of dicts

class mripy.io.**unzipped**(*zip_file*)

Bases: object

mripy.io.**write_1D_nodes**(*fname*, *idx*, *val*)

mripy.io.**write_affine**(*fname*, *mat*, *oneline=True*, *sep=None*)

TODO: Not support multivolume affine yet

mripy.io.**write_afni**(*prefix*, *vol*, *base_img=None*)

mripy.io.**write_asc**(*fname*, *verts*, *faces*)

mripy.io.**write_gii**(*fname*, *verts*, *faces*)

mripy.io.**write_nii**(*fname*, *vol*, *base_img=None*, *space=None*, *dim=None*)

mripy.io.**write_niml_bin_nodes**(*fname*, *idx*, *val*)

Write "Node Bucket" (node indices and values) as niml (binary) dataset.

References

[1] https://afni.nimh.nih.gov/afni/community/board/read.php?1,60396,60399#msg-60399

[2] After some trial-and-error, the following components are required:
self_idcode, COLMS_RANGE, COLMS_TYPE (tell suma how to interpret val), no whitespace between opening tag and binary data.

mripy.io.**write_surf_data**(*fname*, *nodes*, *values*)

mripy.io.**write_surf_mesh**(*fname*, *verts*, *faces*, ***kwargs*)

mripy.io.**write_vol**(*fname*, *vol*, *base_img=None*)

3.1.2 mripy.scripts package

Submodules

mripy.scripts.afni_viewer module

mripy.scripts.extract_physio module

mripy.scripts.mripy_1dplot module

mripy.scripts.mripy_run_iglesias18 module

mripy.scripts.mripy_run_wang15 module

mripy.scripts.report_parameters module

mripy.scripts.script_utils module

mripy.scripts.sort_dicom module

Module contents

3.1.3 mripy.tests package

Submodules

mripy.tests.context module

mripy.tests.test_afni module

class mripy.tests.test_afni.**test_afni**(*methodName='runTest'*)

Bases: TestCase

test_get_affine()

test_get_prefix()

test_get_suma_spec()

test_substitute_hemi()

mripy.tests.test_io module

class mripy.tests.test_io.**test_io**(*methodName='runTest'*)

Bases: TestCase

test_Mask()

mripy.tests.test_timecourse module

class mripy.tests.test_timecourse.**test_Attributes**(*methodName='runTest'*)

Bases: TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

test_copy()

test_pick()

class mripy.tests.test_timecourse.**test_Epochs_Attributes**(*methodName='runTest'*)

Bases: TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

test_copy()

test_pick()

mripy.tests.test_utils module

class mripy.tests.test_utils.**test_utils**(*methodName='runTest'*)

Bases: TestCase

test_fname_with_ext()

mripy.tests.test_utils_slow module

class mripy.tests.test_utils_slow.**test_utils**(*methodName='runTest'*)

Bases: TestCase

test_SharedMemoryArray_CoW()

Test the speed of CoW if read-only

test_SharedMemoryArray_array()

Test the life saving container interface

test_SharedMemoryArray_memory()

Test shared-memory versus copy-on-write

Module contents

3.2 Submodules

3.3 mripy.afni module

mripy.afni.**add_colormap**(*cmap*, *name=None*, *cyclic=False*, *index=None*, *categorical=False*)

cmap : list of RGB colors | matplotlib.colors.LinearSegmentedColormap

mripy.afni.**call**(*cmd*)

mripy.afni.**check_output**(*cmd*, *tags=None*, *pattern=None*, *verbose=0*, ***kwargs*)

The syntax of subprocess.check_output(shell=False) is tedious for long cmd. But for security reason, we don't want to use shell=True for external cmd. This helper function allows you to execute a single cmd without shell=True.

Parameters

- **cmd** (*str*) – A single command string packed with all options (but no wildcard)
- ****kwargs** – Go to *subprocess.check_output(**kwargs)*

Returns

lines – Much easier to deal with compared with subprocess.check_output()

Return type

list of lines

mripy.afni.**filter_output**(*lines*, *tags=None*, *pattern=None*, *ex_tags=None*, *ex_pattern=None*)

Filter output lines according to their initial tags (++, +, *, etc.) and/or a regex search pattern.

Parameters

- **tags** (*list of tags*) – Default is [], which means all lines will pass the filter.
- **pattern** (*str*) –
- **ex_tags** (*list of tags to exclude*) –
- **ex_pattern** (*str*) –

mripy.afni.**generate_spec**(*fname*, *surfs*, *ext=None*, ***kwargs*)

mripy.afni.**get_DELTA**(*fname*)

mripy.afni.**get_DIMENSION**(*fname*)

[x, y, z, t, 0] Not work for bucket.

mripy.afni.**get_ORIENT**(*fname*, *format='str'*)

Parameters

format (*str, {'code', 'str', 'mat', 'sorter'}*) –

References

[1] https://afni.nimh.nih.gov/pub/dist/doc/program_help/README.attributes.html
#define ORI_R2L_TYPE 0 // Right to Left #define ORI_L2R_TYPE 1 // Left to Right #define ORI_P2A_TYPE 2 // Posterior to Anterior #define ORI_A2P_TYPE 3 // Anterior to Posterior #define ORI_I2S_TYPE 4 // Inferior to Superior #define ORI_S2I_TYPE 5 // Superior to Inferior

Thus "0 3 4" is standard DICOM Reference Coordinates System, i.e., RAI. The AFNI convention is also that R-L, A-P, and I-S are negative-to-positive, i.e., RAI.

[2] https://nipy.org/nibabel/nifti_images.html
On the other hand, NIFTI images have an affine relating the voxel coordinates to world coordinates in RAS+ space, or LPI in AFNI's term.

mripy.afni.**get_ORIGIN**(*fname*)

mripy.afni.**get_S2E_mat**(*fname*, *mat='S2E'*)

mripy.afni.**get_TR**(*fname*)

mripy.afni.**get_affine**(*fname*)

mripy.afni.**get_affine_nifti**(*fname*)

mripy.afni.**get_attribute**(*fname*, *name*, *type=None*)

mripy.afni.**get_brick_labels**(*fname*, *label2index=False*)

mripy.afni.**get_censor_from_X**(*fname*)

Return whether it is selected ("good" = not censored) for each volume.

Parameters

fname (*str*) – X.prefix.1D (design matrix) file as generated by 3dDeconvolve.

mripy.afni.**get_crop**(*fname*, *xyz*, *r*)

For AFNI driver "crop=x1:x2,y1:y2" See https://afni.nimh.nih.gov/pub/dist/doc/program_help/README.driver.html

Parameters

- **fname** (*str*) –
- **xyz** (*array-like*) – Cursor location [x, y, z] in mm (as shown in AFNI GUI).
- **r** (*float*) – Crop radius in mm.

mripy.afni.**get_dims**(*fname*)

Dimensions (number of voxels) of the data matrix. See also: get_head_dims

mripy.afni.**get_head_delta**(*fname*)

Resolution (voxel size) along R-L, A-P, I-S axes.

mripy.afni.**get_head_dims**(*fname*)

Dimensions (number of voxels) along R-L, A-P, I-S axes. See also: get_dims

mripy.afni.**get_head_extents**(*fname*)

Spatial extent along R, L, A, P, I and S.

mripy.afni.**get_hemi**(*fname*)

mripy.afni.**get_nifti_field**(*fname*, *name*, *type=None*)

mripy.afni.**get_prefix**(*fname*, *with_path=False*)

Return "dset" given "path/to/dset+orig.HEAD", "dset+orig.", "dset+tlrc", "dsets"

mripy.afni.**get_runs_from_X**(*fname*, *fmt='groups'*)

Return groups (run index started from 0) for each volume.

Parameters

- **fname** (*str*) – X.prefix.1D (design matrix) file as generated by 3dDeconvolve.
- **fmt** (str, 'starts'|'ends'|**<'groups'>|**'afni') – The return format about runs information. - 'starts': The start volume index for each run, e.g., [0, 3] - 'ends': [start, end] volume index for each run, e.g., [[0,2],[3,5]] - 'groups': For use with sklearn, e.g., [0,0,0,1,1,1] - 'afni': For use with afni subbrick selection, e.g., ["'[0..2]'","'[3..5]'"]

mripy.afni.**get_slice_timing**(*fname*)

mripy.afni.**get_subbrick_selector**(*good*)

Convert boolean indexing to AFNI's subbrick selection syntax. E.g., [True, False, True, True, True] -> "0,2..4"

mripy.afni.**get_suma_info**(*suma_dir*, *suma_spec=None*)

mripy.afni.**get_suma_spec**(*suma_spec*)

Infer other spec files from one spec file (either lh.spec, rh.spec, or both.spec).

Parameters

suma_spec (*str*) – Either a .spec file or the suma_dir.

mripy.afni.**get_suma_subj**(*suma_dir*)

Infer SUMA subject given path to SUMA folder.

mripy.afni.**get_surf_type**(*suma_dir*)

Infer SUMA surface mesh file type (.gii vs .asc).

mripy.afni.**get_surf_vol**(*suma_dir*)

Infer SUMA SurfVol filename with full path (agnostic about file type: .nii vs +orig.HEAD/BRIK).

mripy.afni.**infer_surf_dset_variants**(*fname*, *hemis=['lh', 'rh', 'both', 'mh', 'bh']*)

```
>>> infer_surf_dset_variants('data.niml.dset')
{'lh': 'lh.data.niml.dset', 'rh': 'rh.data.niml.dset', 'both': 'both.data.niml.dset
↪', mh': 'mh.data.niml.dset'}
>>> infer_surf_dset_variants('lh.data.niml.dset')
{'lh': 'lh.data.niml.dset'}
```

Parameters

fname (*str, list, or dict*) –

mripy.afni.**insert_suffix**(*fname*, *suffix*)

mripy.afni.**parse_patch**(*patch*)

Notes

1. Each replacement is started with one or more comment lines. The last comment line is treated as replacement target, which may contain an optional replacement directive at the end: # This is an example <replace command="1"/>

   Possible directives for replacing the original scripts includes:

   1) command="n": replace n commands
   2) line="n": replace n lines
   3) until="regexp": replace until a specific line (the regexp is the last line to be replaced)

2. Each replacement must end with two consecutive newlines.

mripy.afni.**patch_afni_proc**(*original*, *patch*, *inplace=True*)

mripy.afni.**set_attribute**(*fname*, *name*, *value*, *type=None*)

mripy.afni.**set_brick_labels**(*fname*, *labels*)

mripy.afni.**set_nifti_field**(*fname*, *name*, *value*, *out_file=None*)

mripy.afni.**set_slice_timing**(*fname*, *times*, *TR*)

We have to provide a TR because we don't know whether the default value TR=1.0 is valid.

mripy.afni.**split_out_file**(*out_file*, *split_path=False*, *trailing_slash=False*)

Ensure that path.join(out_dir, prefix, ext) can be checked by path.exists().

```
>>> split_out_file('dset.nii')
('dset', '.nii')
>>> split_out_file('dset.1D')
('dset', '.1D')
>>> split_out_file('folder/dset')
('folder/dset', '+orig.HEAD')
>>> split_out_file('folder/dset+orig', split_path=True)
('folder', 'dset', '+orig.HEAD')
>>> split_out_file('dset+orig.', split_path=True)
('', 'dset', '+orig.HEAD')
>>> split_out_file('folder/dset+orig.HEAD', split_path=True, trailing_slash=True)
('folder/', 'dset', '+orig.HEAD')
>>> split_out_file('dset+tlrc.BRIK', split_path=True, trailing_slash=True)
('', 'dset', '+tlrc.HEAD')
```

mripy.afni.**substitute_hemi**(*fname*, *hemi='{0}'*)

mripy.afni.**update_afnirc**(***kwargs*)

mripy.afni.**write_colorscale_file**(*fname*, *pal_name*, *colors*, *locations=None*, *interp=None*)

Parameters

- **fname** (**.pal* file name) –
- **pal_name** (*palette name (or title)*) –
- **colors** (*if you fill the colorscale file with a lot of*) – first color (bottom) -> last color (top)

- **locations** (*locations of the breakpoints where colors are defined*) – 0 (bottom) -> 1 (top)
- **interp** (*'linear'|'nearest'*) –
- **colorscale."** (*AFNI document says "There are exactly 128 color locations on an AFNI*) –
- **details** (*For*) –
- **https** (*see*) –
- **fact** (*But in*) –
- **colors** –
- **used.** (*only the first 256 colors will be*) –

mripy.afni.**xyz2ijk**(*fname*, *xyz*)

Convert RAI=dicom xyz to data matrix indices ijk

3.4 mripy.dcm module

class mripy.dcm.**ExtendedCircle**(*width*, *pad=0.3*)

Bases: Circle

An extended Circle BoxStyle that keeps a minimum predefined width parameter.

References

https://stackoverflow.com/questions/40796117/how-do-i-make-the-width-of-the-title-box-span-the-entire-plot
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/patches.py

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

x0 and y0 are the lower left corner of original bbox. They are set automatically by matplotlib.

class mripy.dcm.**ExtendedSimple**(*head_length=0.5*, *head_width=0.5*, *tail_width=0.2*)

Bases: _Base

A simple arrow. Only works with a quadratic Bezier curve.

transmute(*path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the path object along which the arrow will be drawn, and the mutation_size, with which the arrow head etc. will be scaled. The linewidth may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a Path instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of a same length.

mripy.dcm.**draw_circle_arrow**(*xy*, *radius*, *patchA=None*, *ax=None*)

References

https://stackoverflow.com/questions/37512502/how-to-make-arrow-that-loops-in-matplotlib/38224201

mripy.dcm.**plot_dcm**(*rois*, *inputs*, *A*, *B*, *C*, *PA=None*, *PB=None*, *PC=None*, *roi_order=None*, *rads=None*, *ax=None*)

Visualize a DCM (as well as estimated A, B, C parameters).

Parameters

- **A** (*[n_rois, n_rois]*) – Baseline connectivity, dest <- src
- **B** (*[n_rois, n_rois, n_inputs]*) – Modulation on connectivity (non-zero for modulatory inputs only)
- **C** (*[n_rois, n_inputs]*) – Driving on nodes (non-zero for driving inputs only)

References

https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.annotate.html https://matplotlib.org/stable/gallery/userdemo/annotate_simple_coord01.html

https://matplotlib.org/3.3.4/gallery/lines_bars_and_markers/gradient_bar.html (gradient fill)

mripy.dcm.**plot_peb_bma**(*GCM*, ***kwargs*)

3.5 mripy.decoding module

class mripy.decoding.**Demeaner**

Bases: TransformerMixin, BaseEstimator

Remove the mean of each sample individually.

For fMRI, this removes mean activation within ROI for each sample.

fit(*X*, *y=None*)

Do nothing and return the estimator unchanged. This method is just there to implement the usual API and hence work in pipelines. :param X: The data to estimate the demean parameters. :type X: {array-like, sparse matrix} of shape (n_samples, n_features) :param y: Not used, present here for API consistency by convention. :type y: Ignored

Returns
self – Fitted transformer.

Return type
object

transform(*X*)

Subtract the mean of each row in X. :param X: The data to demean, row by row. :type X: {array-like, sparse matrix} of shape (n_samples, n_features)

Returns
X_tr – Transformed array.

Return type
{ndarray, sparse matrix} of shape (n_samples, n_features)

mripy.decoding.**compute_critical_value**(*x*, *y*, *permute='permute'*, *data=None*, *alpha=0.05*, *tail=2*)

Get critical values based on permutation distribution, and account for multiple comparisons using extreme statistics.

Parameters

- **x** (*str, list of str*) – Columns along which multiple comparisons occur (e.g., roi, time).
- **y** (*str*) – Column for performance measurement (e.g., test_accuracy, PC, RT).
- **data** (*pd.DataFrame(x, y, permute)*) – permute == 0 is originally observed data, >= 1 is permutation data.

mripy.decoding.**cross_validate_ext**(*model*, *X*, *y*, *groups=None*, *cv=None*, *pred_kws=None*, *method=None*)

mripy.decoding.**cross_validate_with_permutation**(*model*, *X*, *y*, *groups*, *rois=None*, *n_permutations=1000*, *scoring=None*, *cv=None*)

mripy.decoding.**permute_within_group**(*y*, *groups*)

mripy.decoding.**standardize_within_group**(*X*, *groups*, *with_mean=True*, *with_std=True*)

This is an extension of the "mean centering" method proposed in [1]. Can be used as a replacement for the training-set-wise standardization. Both with_mean and with_std may provide some extra performance.

References

[1] Lee, S., & Kable, J. W. (2018). Simple but robust improvement in multivoxel pattern classification. PloS One, 13(11), e0207083.

3.6 mripy.dicom module

mripy.dicom.**parse_SQ_data_element**(*fi*)

We only support Data Element with Explicit VR at present (Table 7.1-2). We don't support nested Item at present (2018-09-26).

References

[1] http://dicom.nema.org/Dicom/2013/output/chtml/part05/chapter_7.html

mripy.dicom.**parse_Siemens_CSA**(*b*)

mripy.dicom.**parse_Siemens_CSA2**(*b*)

mripy.dicom.**parse_dicom_header**(*fname*, *search_for_tags=None*, ***kwargs*)

Parameters

- **fname** (*str*) –
- **search_for_tags** (*set*) – Search for specific dicom tags, and stop file scanning early if all tags of interest are seen. e.g., search_for_tags={'0020,0011', '0020,0013'} will search for SeriesNumber and InstanceNumber. This will save you some time, esp. when the remote file is accessed via slow data link.
- ****kwargs** – This is only for backward compatibility.

Notes

"Implicit and Explicit VR Data Elements shall not coexist in a Data Set and Data Sets nested within it (see Section 7.5). Whether a Data Set uses Explicit or Implicit VR, among other characteristics, is determined by the negotiated Transfer Syntax (see Section 10 and Annex A)." [1]

References

[1] http://dicom.nema.org/Dicom/2013/output/chtml/part05/chapter_7.html [2] https://stackoverflow.com/questions/119684/parse-dicom-files-in-native-python

mripy.dicom.**parse_series_info**(*dicom_files*, *dicom_ext=None*, *parser=None*, *return_headers=False*)

Parameters
dicom_files (*list or str*) – A list of dicom files (e.g., as provided by sort_dicom_series), or a folder that contains a single series (e.g., "../raw_fmri/func01"), or a single dicom file.

mripy.dicom.**sort_dicom_series**(*folder*)

Parameters
folder (*string*) – Path to the folder containing all the dicom files.

Returns
studies – [{'0001': [file0, file1, ...], '0002': [files], ...}, {study1}, ...]

Return type
list of dicts

3.7 mripy.dicom_report module

mripy.dicom_report.**inspect_mp2rage**(*data_dir*, *subdir_pattern='T1??'*)

mripy.dicom_report.**pares_patent_age**(*age*)

mripy.dicom_report.**print_subject_info**(*dir_pattern*, *exclude=None*)

mripy.dicom_report.**remove_duplicate_parts**(*folders*)

mripy.dicom_report.**report_parameters**(*dicom_folder*, *preset=None*, *return_preset=False*, *return_info=False*)

3.8 mripy.encoding module

class mripy.encoding.**BaseModel**

Bases: *Savable2*

from_dict(*d*)

get_params(*deep=True*)

set_params(***parameters*)

to_dict()

class mripy.encoding.**BayesianChannelModel**(*n_channels='required'*, *basis_func='required'*, *stimulus_domain='required'*, *circular=False*, *stimulus_prior=None*, *global_search=False*, *verbose=2*)

Bases: *BaseModel*

property Omega_

property Omega_inv_

bayesian_inversion(*X*, *stimulus_domain=None*, *stimulus_prior=None*, *density=True*)

Parameters

- **X** (*2D array, n_trials * n_voxels*) –
- **stimulus_domain** (*1D array, n_domain*) –
- **stimulus_prior** (*2D array, n_trials * n_domain (or 1D array, n_domain)*) – None for a flat stimulus prior, same for all trials.

Returns
posterior

Return type
2D array, n_trials * n_domain

fit(*X*, *y*)

Parameters

- **X** (*2D array*) – n_trials * n_voxels BOLD response pattern (e.g., beta for each trial, or delayed and detrended time points within block plateau)
- **y** (*1D array*) – n_trials stimulus value (e.g., orientation, color)

get_params(*deep=True*)

loglikelihood(*X*, *y*)

predict(*X*, *stimulus_domain=None*, *stimulus_prior=None*, *return_all=False*)

class mripy.encoding.**ChannelEncodingModel**(*n_channels*, *basis_func*, *stimulus_domain*, *circular=False*, *verbose=2*)

Bases: *BaseModel*

channel_inversion(*X*, *stimulus_domain=None*)

correlation_inversion(*X*, *stimulus_domain=None*)

fit(*X*, *y*)

Parameters

- **X** (*2D array*) – n_trials * n_voxels BOLD response pattern (e.g., beta for each trial, or delayed and detrended time points within block plateau)
- **y** (*1D array*) – n_trials stimulus value (e.g., orientation, color)

get_params(*deep=True*)

inverted_encoding(*X*)

pRF(*stimulus_domain=None*, *method='ols'*, *X=None*, *y=None*)

predict(*X*, *stimulus_domain=None*, *return_all=False*)

voxel_inversion(*X*, *stimulus_domain=None*)

class mripy.encoding.**EnsembleModel**(*n_ensemble=10*, *base_model='required'*, *pred_method=None*, *pred_options=None*)

Bases: *BaseModel*

fit(*X*, *y*)

from_dict(*d*)

get_params(*deep=True*)

predict(*X*, *method=None*, *options=None*, *return_all=False*, *pred_kws=None*)

to_dict()

mripy.encoding.**basis_Sprague2013**(*s*, *n_channels=6*, *spacing=2*, *center=None*, *size=None*, *power=7*, *dim=1*, *intercept=False*)

Parameters
s (*2D array, n_trials * dim | 1D array, n_trials*) –

Returns
fs

Return type
2D array, n_channels * n_trials

References

{Sprague2013}

mripy.encoding.**basis_vanBergen2015**(*s*, *n_channels=8*, *power=5*)

Parameters
s (*1D array, n_trials*) –

Returns
fs

Return type
2D array, n_channels * n_trials

References

{van Bergen2015}

mripy.encoding.**circular_correct**(*y_true*, *y_pred*, *domain=None*, *n_targets=None*, *tolerance=None*, *return_dist=False*)

mripy.encoding.**discretize_prediction**(*y_pred*, *targets*, *circular_domain=None*)

Discretize continous prediction to the nearest target. Can handle irregular target grid and also circular domain.

E.g., y_pred = encoding.discretize_prediction(y_hat, arange(8)/8*pi, circular_domain=[0, pi]) correct = encoding.circular_correct(y_true, y_hat, domain=[0, pi], n_targets=8) assert(allclose(mean(y_pred==y_true), mean(correct)))

mripy.encoding.**shift_distribution**(*d*, *stimulus_domain*, *center_on=None*, *circular=True*)

Parameters
d (*n_trials * n_domain*) –

3.9 mripy.evaluation module

mripy.evaluation.**afni_costs**(*base_file*, *in_file*, *mask=None*)

In general, the alignment algorithm want to **minimize** the cost.

Some metrics in their canonical form are maximized when perfectly aligned (e.g., mi and hel). These are usually negated when output by *-allcost*, so that ALL cost (EXCEPT lss) values will decrease when alignment improves (given that the base/source contrasts are valid as required by the metric, e.g., lpc is ONLY meaningful if the base and source have opposite contrast). This behavior can be verified by manually applying a transform to a image.

BASED ON GLOBAL CORRELATION COEFFICIENT ls : 1 - abs(Pearson corrcoef), near zero sp : 1 - abs(Spearman corrcoef), near zero lss (maximized): Pearson corrcoef, near one (the ONLY exception)

BASED ON 2D JOINT HISTOGRAM mi (negated) : (negative) mutual information, large negative value nmi : 1/normalized MI, small positive value je : joint entropy, small positive value hel (negated) : (negative) Hellinger distance, large negative value crM : near zero crA : near zero crU : near zero

BASED ON LOCAL (A FEW MM) PEARSON CORRELATION COEFFICIENT (SIMILAR TO BOUNDARY-BASED METHODS) lpc : sum(w[i]*pc[i]*abs(pc[i])) / sum(w[i]), large negative value (FOR opposite contrast ONLY) lpa : 1 - abs(lpc), large negative value (FOR same contrast ONLY) lpc+ : lpc + hel*0.4 + crA*0.4 + nmi*0.2 + mi*0.2 + ov*0.4 lpa+ : lpa + hel*1.0 + crA*0.4 + nmi*0.2 + mi*0.0 + ov*0.0

References

[1] **https://afni.nimh.nih.gov/pub/dist/doc/program_help/3dAllineate.html**
See “Cost functional descriptions (for use with -allcost output)” section

[2] https://www.youtube.com/watch?v=PaZinetFKGY&list=PL_CD549H9kgqJ1GDXAs1BWkgEimAHZeNX

mripy.evaluation.**residual_motion**(*files*, *print_table=False*)

3.10 mripy.math module

class mripy.math.**DomainMapper**(*domain=None*)

Bases: object

from2pi(*y*)

to2pi(*x*)

mripy.math.**LPI2RAI_affine**(*mat*)

mripy.math.**RAI2LPI_affine**(*mat*)

mripy.math.**apply_affine**(*mat*, *xyz*)

xyz : 3xN array

mripy.math.**argsort_rows**(*rows*, *cols=None*)

mripy.math.**array2dataframe**(*a*, *factors*, *var_name='value'*)

a : array factors : dict (factor->levels)

mripy.math.**circular_corrcoef**(*x1*, *x2*, *domain=None*, *n_perm=1000*, *ci=0.95*, *n_boot=None*)

The complex corrcoef method used here is fundamentally different from the (Fisher & Lee, 1983) method which is implemented by pycircstat.corrcc(). For more details, read my note https://docs.google.com/document/d/1sl39YH3g3TFQu1zX-Ax477NULEyJE–MHMXg2gg0cyk/edit

mripy.math.**circular_mean**(*x*, *domain=None*, *weight=None*, *axis=None*)

Circular mean for values from arbitary circular domain (not necessarily angles).

mripy.math.**circular_std**(*x*, *domain=None*, *weight=None*, *axis=None*)

Following scipy.stats.circstd()'s definition of circular standard deviation that in the limit of small angles returns the 'linear' standard deviation. scipy.stats.circstd() doesn't support weight. pycircstat.std() doesn't support domain. astropy.stats.circvar() follows another definition.

mripy.math.**concat_affine**(*mat2*, *mat1*)

mat @ v = mat2 @ mat1 @ v

For afni's "fetching" convention, the first transform goes first. mat1 = io.read_affine('SurfVoltoT1.aff12.1D') # Fetching SurfVol from T1 grid mat2 = io.read_affine('T1toEPI.aff12.1D') # Fetching T1 from EPI grid mat = math.concat_affine(mat1, mat2) # Note the order! The combined transform needs to fetch SurfVol from EPI io.write_affine('SurfVoltoEPI.aff12.1D', mat) # Fetching SurfVol from EPI

mripy.math.**corrcoef_along_axis**(*x*, *y*, *axis=None*, *norm=True*)

mripy.math.**gaussian_logpdf**(*x*, *mean*, *cov*, *cov_inv=None*, *axis=-1*)

More efficient multivariate normal distribution log pdf than stats.multivariate_normal.logpdf()

mripy.math.**invert_affine**(*mat*)

mripy.math.**median_argmax**(*x*, *axis=-1*)

mripy.math.**nearest**(*x*, *parity='odd'*, *round=<function round_>*)

mripy.math.**normalize_logP**(*logP*, *axis=None*)

mripy.math.**pinv**(*x*)

mripy.math.**polyfit3d**(*x*, *y*, *z*, *f*, *deg*, *method=None*)

mripy.math.**tsarray2dataframe**(*tsarray*, *t=None*, *ts_name='value'*, *t_name='time'*, *trial_name='trial'*, *trial_df=None*)

Parameters
tsarray (*ndarray, n_trials * n_times*) –

Returns
df

Return type
DataFrame

3.11 mripy.paraproc module

class mripy.paraproc.**ArrayWrapper**(*name*, *bases*, *dct*)

Bases: type

This is the metaclass for classes that wrap an np.ndarray and delegate non-reimplemented operators (among other magic functions) to the wrapped array.

class mripy.paraproc.**PooledCaller**(*pool_size=None*, *verbose=1*)

Bases: object

Execute multiple command line programs, as well as python callables, asynchronously and parallelly across a pool of processes.

all_successful(*jobs=None*, *verbose=None*)

dispatch()

idss(*total*, *batch_size=None*)

run(*cmd*, **args*, *_depends=None*, *_retry=None*, *_dispatch=False*, *_error_pattern=None*, *_error_whitelist=None*, *_suppress_warning=False*, *_block=False*, ***kwargs*)

Asynchronously run command or callable (queued execution, return immediately).

See subprocess.Popen() for more information about the arguments.

Multiple commands can be separated with ";" and executed sequentially within a single subprocess in linux/mac, only if shell=True.

Python callable can also be executed in parallel via multiprocessing. Note that although return values of the callable are retrieved via PIPE, sometimes it could be advantageous to directly save the computation results into a shared file (e.g., an HDF5 file), esp. when they're large. In the later case, a proper lock mechanism via multiprocessing.Lock() is required.

Parameters

- **cmd** (*list, str, or callable*) – Computation in command line programs is handled with subprocess. Computation in python callable is handled with multiprocessing.
- **shell** (*bool*) – If provided, must be a keyword argument. If shell is True, the command will be executed through the shell.
- ***args** – If cmd is a callable, **args* are passed to the callable as its arguments.
- ****kwargs** – If cmd is a callable, ***kwargs* are passed to the callable as its keyword arguments. If cmd is a list or str, ***kwargs* are passed to subprocess.Popen().
- **_depends** (*list*) – A list of jobs (identified by their uuid) that have to be done before this job can be scheduled.
- **_retry** (*int*) – Number of retry before accepting failure (if detecting non-zero return code).
- **_dispatch** (*bool*) – Dispatch the job immediately, which will run in the background without blocking.
- **_error_pattern** (*str*) –
- **_suppress_warning** (*bool*) –
- **_block** (*bool*) – if True, call wait() internally and block.

Returns
_uuid – The uuid of current job (which can be used as future jobs' dependency)

Return type
str

run1(*cmd*, **args*, *_error_pattern=None*, *_error_whitelist=None*, *_suppress_warning=False*, ***kwargs*)

wait(*pool_size=None*, *return_codes=False*, *return_jobs=False*, *raise_when_failed=True*)
Wait for all jobs in the queue to finish.

Returns

- **return_values** (*list*) – Return values of executed python callable. Always *None* for command.
- **codes** (*list (only when return_codes=True)*) – The return code of the child process for each job.
- **jobs** (*list (only when return_jobs=True)*) – Detailed information about each child process, including captured stdout and stderr.

class mripy.paraproc.**SharedMemoryArray**(*dtype*, *shape*, *initializer=None*, *lock=True*)
Bases: object

This class can be used as a usual np.ndarray, but its data buffer is allocated in shared memory (under Cached Files in memory monitor), and can be passed across processes without any data copy/duplication, even when write access happens (which is lock-synchronized).

The idea is to allocate memory using multiprocessing.Array, and access it from current or another process via a numpy.ndarray view, without actually copying the data. So it is both convenient and efficient when used with multiprocessing.

This implementation also demonstrates the power of composition + metaclass, as opposed to the canonical multiple inheritance.

```
dtype2ctypes = {<class 'bool'>: <class 'ctypes.c_bool'>, <class 'int'>: <class
'ctypes.c_long'>, <class 'float'>: <class 'ctypes.c_double'>, dtype('bool'): <class
'ctypes.c_bool'>, dtype('int64'): <class 'ctypes.c_long'>, dtype('int32'): <class
'ctypes.c_int'>, dtype('int16'): <class 'ctypes.c_short'>, dtype('int8'): <class
'ctypes.c_byte'>, dtype('uint64'): <class 'ctypes.c_ulong'>, dtype('uint32'): <class
'ctypes.c_uint'>, dtype('uint16'): <class 'ctypes.c_ushort'>, dtype('uint8'): <class
'ctypes.c_ubyte'>, dtype('float64'): <class 'ctypes.c_double'>, dtype('float32'):
<class 'ctypes.c_float'>}
```

classmethod from_array(*arr*, *lock=True*)
Initialize a new shared-memory array with an existing array.

classmethod zeros(*shape*, *dtype=<class 'float'>*, *lock=True*)
Return a new array of given shape and dtype, filled with zeros.

This is the preferred usage, which avoids holding two copies of the potentially very large data simultaneously in the memory.

class mripy.paraproc.**TeeOut**(*err=False*, *tee=True*)
Bases: StringIO

write(*s*)
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

mripy.paraproc.**add_metaclass**(*metaclass*)

Class decorator for creating a class with a metaclass.

mripy.paraproc.**check_output_for_errors**(*output*, *error_pattern=None*, *error_whitelist=None*, *verbose=1*, *label=''*)

User can skip error checking by setting error_pattern=”

mripy.paraproc.**check_output_for_goal**(*output*, *goal_pattern=None*)

mripy.paraproc.**cmd_for_disp**(*cmd*)

Format cmd for printing.

Parameters
cmd (*str, list, or callable*) –

mripy.paraproc.**cmd_for_exec**(*cmd*, *shell=False*)

Format cmd appropriately for execution according to whether shell=True.

Split a cmd string into a list, if shell=False. Join a cmd list into a string, if shell=True. Do nothing to callable.

Parameters

- **cmd** (*str, list, or callable*) –
- **shell** (*bool*) –

mripy.paraproc.**format_duration**(*duration*, *format='standard'*)

Format duration (in seconds) in a more human friendly way.

mripy.paraproc.**run**(*cmd*, *check=True*, *error_pattern=None*, *error_whitelist=None*, *goal_pattern=None*, *shell=False*, *verbose=2*)

Run an external command line.

This function is similar to subprocess.run introduced in Python 3.5, but provides a slightly simpler and perhaps more convenient API.

Parameters
cmd (*str or list*) –

3.12 mripy.preprocess module

class mripy.preprocess.**ANTsTransform**(*transforms*, *base_file=None*, *source_file=None*)

Bases: *Transform*

apply(*in_file*, *out_file*, *base_file=None*, *interp=None*, ***kwargs*)

For volumes, forward transform (from input/moving to base/fixed)

apply_inverse(*in_file*, *out_file*, *base_file=None*, *interp=None*, ***kwargs*)

For volumes, inverse transform (from base/fixed to input/moving)

apply_inverse_to_points(*in_file*, *out_file*)

For list of points, inverse transform (from base/fixed to input/moving)

Parameters

- **in_file** (**.csv* file with “x,y,z,t” header line.) –
- **out_file** (**.csv* file with “x,y,z,t” header line.) –

apply_inverse_to_xyz(*xyz*, *convention='DICOM'*)

Parameters

- **xyz** (*Nx3 array*) –
- **convention** (*'DICOM' | 'NIFTI'*) –

apply_to_points(*in_file*, *out_file*)

For list of points, forward transform (from input/moving to base/fixed)

Parameters

- **in_file** (**.csv* file with "x,y,z,t" header line.) –
- **out_file** (**.csv* file with "x,y,z,t" header line.) –

apply_to_xyz(*xyz*, *convention='DICOM'*)

Parameters

- **xyz** (*Nx3 array*) –
- **convention** (*'DICOM' | 'NIFTI'*) –

classmethod from_align_ants(*outputs*)

class mripy.preprocess.**Transform**(*transforms*, *base_file=None*, *source_file=None*)

Bases: object

apply(*in_file*, *out_file*, *base_file=None*, *interp=None*, ***kwargs*)

For volumes, forward transform (from input/moving to base/fixed)

apply_inverse(*in_file*, *out_file*, *base_file=None*, *interp=None*, ***kwargs*)

For volumes, inverse transform (from base/fixed to input/moving)

classmethod from_json(*fname*, *replace_path=False*)

Parameters

replace_path (*bool*) – Replace path of the transform files according to json file path

inverse()

rebase(*base_file*)

replace_path(*p*)

to_json(*fname*)

mripy.preprocess.**afni2ants_affine**(*afni_affine*, *ants_affine*)

mripy.preprocess.**afni2ants_warp**(*afni_warp*, *ants_warp*)

mripy.preprocess.**align_S2E**(*base_file*, *suma_dir*, *out_file=None*, ***kwargs*)

mripy.preprocess.**align_anat**(*base_file*, *in_file*, *out_file*, *strip=None*, *N4=None*, *init_shift=None*, *init_rotate=None*, *init_xform=None*, *method=None*, *cost=None*, *n_params=None*, *interp=None*, *max_rotate=None*, *max_shift=None*, *emask=None*, *save_weight=None*)

emask

[fname] Mask to exclude from analysis.

mripy.preprocess.**align_anat2epi**(*anat_file*, *epi_file*, *out_file*, *base_file=None*, *init_oblique=None*, *init_epi_rotate=None*, *init_anat_rotate=None*)

Different init methods are mutual exclusive (i.e., at most one init method could be used at a time).

mripy.preprocess.**align_ants**(*base_file*, *in_file*, *out_file*, *strip=None*, *base_mask=None*, *in_mask=None*, *base_mask_SyN=None*, *in_mask_SyN=None*, *init_transform=None*, *preset=None*, *n_jobs=None*)

Nonlinearly align *in_file* to *base_file* using ANTs' SyN method via antsRegistration.

The default preset can be used for both within- and cross-modality nonlinear registration, i.e., for both T1-to-T1 and EPI-to-T1 (more precisely, T2*-to-T1) deformable alignment.

Examples

1. **Align MNI template to T1_al using default preset. Skullstrip T1_al.**
   Apply inversed mask (1-mask) to MNI template at the nonlinear (SyN) stage. >>> prep.align_ants("T1_al.nii", "MNI152_2009_template.nii.gz", "MNI_al.nii", strip="base", in_mask_SyN="mask.nii -I")
2. **Align T1_ns_al.nii to MNI template using "test" preset.**
   For a quick test of the parameters in a few minutes. The result will not be good, but should not be weird. >>> prep.align_ants("MNI152_2009_template.nii.gz", "T1_ns_al.nii", "T1_ns_MNI.nii", preset="test")

Parameters

- **strip** (*str*) – 'base' | 'input' | 'both'
- **preset** (*str*) – None | 'default' | 'test' | 'path/to/my/preset.json' For production, just leave it as None or use the 'default' presest (est. time: 3 hr). For quick test, use the 'test' presest (est. time: 3 min).

Returns

outputs –

outputs['transform']
[ANTsTransform object] You can apply the forward or inverse transform to other volumes. E.g., >>> outputs['transform'].apply(in_file, out_file) >>> outputs['transform'].apply_inverse(in_file, out_file)

Return type
dict

mripy.preprocess.**align_center**(*base_file*, *in_file*, *out_file*)

mripy.preprocess.**align_epi**(*in_files*, *out_files*, *best_reverse=None*, *blip_results=None*, *blip_kws=None*, *volreg_kws=None*, *template=None*, *template_pool=None*, *template_candidate_runs=None*, *final_resample=True*, *final_res=None*)

Parameters
blip_results (*list or dict*) – E.g., 1) {'warp_file': *'.volreg.warp.nii', 'blip_file': '*.volreg.blip.nii'} 2) {'warf_file': 'epi01.for2mid.warp.nii'}

mripy.preprocess.**all_finished**(*outputs*)

mripy.preprocess.**ants2afni_affine**(*ants_affine*, *afni_affine*)

mripy.preprocess.**ants2afni_warp**(*ants_warp*, *afni_warp*)

mripy.preprocess.**apply_ants**(*transforms*, *base_file*, *in_file*, *out_file*, *interp=None*, *dim=None*, *image_type=None*, *n_jobs=None*)

Parameters

- **transforms** (*list of file names*) – Online matrix inversion is supported as **_0GenericAffine.mat -I*.
- **base_file** (*str*) – If None, apply transforms to point list using *antsApplyTransformsToPoints*, and *in_file* is expected to be a **.csv* file. Otherwise, apply transforms to image using *antsApplyTransforms*.
- **interp** (*str*) – 'LanczosWindowedSinc', 'NearestNeighbor', 'Linear', 'BSpline[<order=3>]', etc.
- **volumes** (*Note that for*) –
- **grid)** (*last transform applies first (pulling from base*) –

:param : :param as in AFNI: :param as well as in ANTs command line.: :param However for point lists: :param FIRST transform applies first (pushing input points): :param : :param and INVERSE transforms should be used compared with volume case: :param as in ANTs.:

mripy.preprocess.**apply_transforms**(*transforms*, *base_file*, *in_file*, *out_file*, *interp=None*, *res=None*, *save_xform=None*)

Note that last transform applys first, as in AFNI. Transforms can be modified as "xform.aff12.1D -I" for inversion.

mripy.preprocess.**assign_mp2rage_labels**(*T1s*, *dicom_dirs*, *dicom_ext='.IMA'*)

mripy.preprocess.**average_anat**(*T1s*, *out_file*, *template_idx=0*, *T1s_ns=None*, *weight=None*)

mripy.preprocess.**blip_unwarp**(*forward_file*, *reverse_file*, *reverse_loc*, *out_file*, *PE_axis='AP'*, *min_patch=None*, *pre_blur=None*)

abin/unWarpEPI.py do this through: unwarp estimate (before tshift) > tshift > unwarp apply > motion correction > concat apply

mripy.preprocess.**calculate_min_patch**(*fname*, *warp_res=10*)

warp_res
[float] Effective "warp resolution" in mm.

mripy.preprocess.**clusterize**(*in_file*, *out_file*, *cluster_size*, *neighbor=2*)

mripy.preprocess.**collect_perblock_stats**(*mask_file*, *stats_file*, *n_conditions*, *n_repeats*, *n_runs=None*, *n_stats=2*, *stat_index=0*, *skip_first=1*)

mripy.preprocess.**combine_affine_transforms**(*transforms*, *out_file=None*)

Note that:

1. Each transform can be modified by -I, -S, -P, e.g., transforms=['T1_al.aff12.1D -I']
2. An oneline transform could contain multiple lines, one for each time point.

mripy.preprocess.**combine_censors**(*censor_files*, *out_file*, *method='intersect'*)

Combine multiple censor files: 1=keep, 0=discard

Parameters
method (*str, {'intersect', 'concatinate'}*) –

mripy.preprocess.**copy_S2E_mat**(*src*, *dst*)

mripy.preprocess.**copy_dset**(*in_file*, *out_file*)

mripy.preprocess.**correct_bias_field**(*in_file*, *out_file*, *n_jobs=None*)

mripy.preprocess.**correct_motion**(*base_file*, *in_file*, *out_file*, *algorithm='3dvolreg'*, *mode='rigid'*)

algorithm : {'3dvolreg', '3dAllineate'}

mripy.preprocess.**count_outliers**(*in_file*, *out_file=None*, *censor_file=None*, *th=0.1*)

Compute outlier voxel fraction for each volume.

Censor (=0, excluded from further analysis) a volume when too many voxels (default>10%) within the automask are outliers.

mripy.preprocess.**create_brain_mask**(*in_files*, *out_file*, *clfrac=None*)

clfrac

[float, between 0.1 and 0.9, [default=0.5]] A small 'clip level fraction' will tend to make the mask larger.

mripy.preprocess.**create_extent_mask**(*transforms*, *base_file*, *in_files*, *out_file*)

mripy.preprocess.**create_hcp_retinotopic_atlas**(*subj_dir*, *suma='SUMA'*, *NIFTI=True*)

Create HCP retinotopic atlas (benson14 template) using docker, and convert the volume and surface datasets into SUMA format.

mripy.preprocess.**create_mp2rage_SNR_mask**(*T1s*, *out_file*)

Need to call prep.assign_mp2rage_labels() first.

mripy.preprocess.**create_suma_dir**(*subj_dir*, *suma_dir=None*, *NIFTI=True*)

Notes

ABOUT NIFTI

If NIFTI=True, @SUMA_Make_Spec_FS will use mri_convert to convert $surf_dir/orig.mgz (which is averaged and hires) to SurfVol.nii. Volumes will be in .nii and surfaces will be in .gii format. This is the preferable way to go.

If NIFTI=False, @SUMA_Make_Spec_FS will use to3d to convert COR- files to SurfVol+orig.HEAD which is 1x1x1. Volumes will be in +orig.HEAD and surfaces will be in .asc format. This is only provided for backward compatibility.

mripy.preprocess.**create_suma_script**(*spec_file*, *surf_vol*, *out_file*, *use_relpath=False*)

mripy.preprocess.**create_vessel_mask_BOLD**(*beta*, *out_file*, *th=10*)

mripy.preprocess.**create_vessel_mask_EPI**(*mean_epi*, *ribbon*, *out_file*, *corr_file=None*, *th=None*)

Find vessel voxels based on bias-corrected EPI intensity (Kay's method). The result may also highlight misalignment between EPI and T1, as well as errors in pial surface reconstruction.

Parameters

- **mean_epi** (*str*) –
- **ribbon** (*str*) – Mask for gray matter which is aligned with mean_epi.
- **th** (*float or 'auto', default 0.75 (as in [1])*) –

References

[1] Kay, K., Jamison, K. W., Vizioli, L., Zhang, R., Margalit, E., & Ugurbil, K. (2019). A critical assessment of data quality and venous effects in sub-millimeter fMRI. NeuroImage, 189, 847–869.

mripy.preprocess.**create_vessel_mask_PDGRE**(*PD*, *GRE*, *out_file*, *PD_div_GRE=None*, *ath=300*, *rth=5*, *nath=100*, *nrth=-5*, *base_file=None*, *strip_PD=True*, *strip_GRE=True*)

Find vessel voxels based on PD/GRE contrast.

Parameters

- **PD** (*str, aligned with EXP*) –
- **GRE** (*str, aligned with PD*) –
- **ath** (*float*) – Global absolute threshold for PD/GRE
- **rth** (*float*) – Local (within neighborhood) relative deviation for PD/GRE

mripy.preprocess.**deoblique**(*in_file*, *out_file=None*, *template=None*)

mripy.preprocess.**detrend**(*motion_file*, *in_file*, *out_file*, *censor=True*, *motion_th=0.3*, *censor_file=None*, *regressor_file=None*)

mripy.preprocess.**find_best_reverse**(*seq_info*, *forward='func'*, *reverse='reverse'*, *for_out='epi{run}.nii'*, *rev_out='reverse{run}.nii'*, *return_single_best=False*)

Find best forward-reverse pair (as a mapping dict) according to temporal proximity, assuming that head motion is usually smaller if the two images are closer in time.

seq_info : DataFrame

mripy.preprocess.**find_epi_dark_voxel_threshold**(*v*, *method='Liu2020'*)

Find EPI intensity threshold for blood vessels on surface dset,
produced by Surface.vol2surf(func='min').

Parameters

method (*str, 'Liu2020' | 'Marquardt2018' | 'Kay2019'*) – Surprisingly, the three methods usually produce very similar results. The mixture of Gaussian method ('Kay2019') is robust with 'N4' or 'Kay2019' unified EPI data (using preprocess.unifize_epi), but may fail with '3dUnifize' or raw EPI data.

mripy.preprocess.**fs_recon**(*T1s*, *out_dir*, *T2=None*, *FLAIR=None*, *NIFTI=True*, *hires=True*, *fs_ver=None*, *V1=True*, *HCP_atlas=True*, *n_jobs=None*)

Parameters

- **T1s** (*list of str | 'brainmask_edit' | 'wm_edit'*) –
- **fs_ver** (*{'v6', 'v6.hcp', 'skip'}*) –

mripy.preprocess.**glm**(*in_files*, *out_file*, *design*, *model='BLOCK'*, *contrasts=None*, *TR=None*, *pick_runs=None*, *motion_files=None*, *censor=True*, *motion_th=0.3*, *censor_files=None*, *mask=None*, *regressor_file=None*, *poly=None*, *fitts=True*, *errts=True*, *REML=True*, *perblock=False*, *FDR=None*, *check_TR=True*)

Parameters

- **design** (*OrderedDict(L=('../stimuli/L.txt', 24), R="../stimuli/R.txt 'BLOCK(24,1)'")*) –

- **model** (*str*) –
- **contrasts** (*OrderedDict([('L+R', '+0.5*L +0.5*R'), ...])*) –
- **pick_runs** (*array-like*) – Indices of the runs you want to pick for analysis (start from zero)

Examples

1. **Canonical GLM**
   design = OrderedDict() design['L'] = ('../stimuli/L_localizer.txt', 24) design['R'] = ('../stimuli/R_localizer.txt', 24) contrasts = OrderedDict() contrasts['L+R'] = '+0.5*L +0.5*R' contrasts['L-R'] = '+L -R' model = 'BLOCK'
2. **FIR estimation**
   design = OrderedDict() design['L'] = f"{stim_dir}/L_{condition}_deconv.txt 'CSPLINzero(0,24,11)'" design['R'] = f"{stim_dir}/R_{condition}_deconv.txt 'CSPLINzero(0,24,11)'" contrasts = None model = 'BLOCK' # This is not used for TENT/CSPLIN

mripy.preprocess.**irregular_resample**(*transforms*, *xyz*, *in_file*, *order=3*)

Parameters

- **transforms** (*list of str*) – In order to project raw EPI in volume (in RAI) to surface coordinates (in LPI aka RAS+), i.e., surf_xyz = [dicom2nifti, exp2surf, volreg2template, unwarp2template, ijk2xyz] @ ijk, what we really do is mapping backward from xyz (assumed already in RAI): xyz -> E2A storing inv(exp2surf) -> ... -> inv(MAT).
- **xyz** (*Nx3 array, assumed in DICOM RAI as with AFNI volumes.*) – Note that FreeSurfur surface vertices are in NIFTI LPI aka RAS+, whereas AFNI uses DICOM RAI internally.

mripy.preprocess.**is_affine_transform**(*fname*)

mripy.preprocess.**manual_transform**(*in_file*, *out_file*, *shift=None*, *rotate=None*, *scale=None*, *shear=None*, *interp=None*)

shift : [x, y, z] in mm rotate : [I, R, A] in degrees (i.e., -z, -x, -y axes), right-hand rule

For example, 'shift=[1,0,0]' shifts in_file to the left by 1 mm (not necessarily 1 voxel). The acutal applied param will be negated. Because if the source is shifted to the left compared with the base, the resulted param will be x=1 indicating that source is shifted to the left, and apply that param will actually cause the source to shift rightward by 1 to match the base. So to shift in_file leftward, the actual param will be x=-1.

The matrix is specified in DICOM-ordered (RAI) coordinates (x=-R+L,y=-A+P,z=-I+S). By default the shift is applied AFTER matrix transform, as in augmented 4x4 affine. By default the shear matrix is LOWER triangle.

For '-1Dmatrix_save' and '-1Dmatrix_apply', the matrix specifies coordinate transformation from base to source DICOM coordinates. In other words, with the estimated matrix at hand, you are ready to build the transformed image by filling the base grid with source data.

Refer to "DEFINITION OF AFFINE TRANSFORMATION PARAMETERS"@3dAllineate for details.

mripy.preprocess.**nudge_cmd2mat**(*nudge_cmd*, *in_file*, *return_inverse=False*)

Refer to "Example 4"@SUMA_AlignToExperiment for details.

The 1st return (MAT) is what you want to write as 'src2base.aff12.1D' and use apply_transforms to get the same effect as your manual nudge. Remember, the AFNI way is to store inverse matrix under forward name (to "pull" data from src). So the mathematical map from moveable to tempalte is in the 2nd return (INV).

mripy.preprocess.**parse_seq_info**(*raw_dir*, *return_dataframe=True*)

Assuming your rawdata folder hierarchy is like:
raw_fmri ├— func01 ├— func02 ├— reverse01 ├— reverse02 ├— T1

mripy.preprocess.**prep_mp2rage**(*dicom_dirs*, *out_file='T1.nii'*, *unwarp=False*, *dicom_ext='.IMA'*)

Convert dicom files and remove the noise pattern outside the brain.

dicom_dirs
[list or str] A list of dicom file folders, e.g., [‘T101’, ‘T102’, . . .], or a glob pattern like ‘raw_fmri/T1??’

mripy.preprocess.**prep_mp2rages**(*data_dir*, *sessions=None*, *subdir_pattern='T1??'*, *unwarp=True*, ***kwargs*)

mripy.preprocess.**resample**(*in_file*, *out_file*, *res=None*, *base_file=None*, *interp=None*)

mripy.preprocess.**resample_to_surface**(*transforms*, *surfaces*, *in_file*, *out_files=None*, *mask_file=None*, *n_jobs=1*, ***kwargs*)

Parameters
mask_file (*str, dict*) – Surface mask. Either a file name or dict(lh=’lh.mask.niml.dset’, rh=’rh.mask.niml.dset’). This will generate a partial surface dataset.

Examples

resample_to_surface(transforms=[f’SurfVol_Alnd_Exp.E2A.1D’, f’epi{run}.volreg.aff12.1D’, f’epi{run}.volreg.warp.nii’],
surfaces=[f’{suma_dir}/lh.pial.asc’, f’{suma_dir}/rh.smoothwm.asc’], in_file=f’epi{run}.tshift.nii’)

mripy.preprocess.**retrieve_mp2rage_labels**(*dicom_dirs*, *dicom_ext='.IMA'*)

Retrieve mp2rage subvolume labels like UNI, ND, etc.

Parameters
dicom_dirs (*list or str*) – A list of dicom file folders, e.g., [‘T101’, ‘T102’, . . .], or a glob pattern like ‘raw_fmri/T1??’

Returns
label2dicom_dir – label -> (index, dicom_dir)

Return type
OrderedDict

mripy.preprocess.**scale**(*in_file*, *out_file*, *mask_file=None*, *dtype=None*)

mripy.preprocess.**skullstrip**(*in_file*, *out_file=None*)

mripy.preprocess.**unifize_epi**(*in_file*, *out_file*, *method='N4'*, *ribbon=None*)

Remove spatial trend/inhomogeneity in (mean) EPI data to better identify vessels using find_epi_dark_voxel_threshold().

The default ‘N4’ method requires you have ANTs installed.

Only ‘Kay2019’ method need an additional cortical ribbon mask that is aligned with the EPI volume.

Parameters
method (*str, 'N4' | 'Kay2019' | '3dUnifize'*) –

mripy.preprocess.**zscore**(*in_file*, *out_file*)

3.13 mripy.six module

Utilities for writing code that runs on Python 2 and 3

class mripy.six.**Module_six_moves_urllib**(*name*, *doc=None*)

Bases: module

Create a six.moves.urllib namespace that resembles the Python 3 namespace

error = <module 'mripy.six.moves.urllib.error'>

parse = <module 'mripy.six.moves.urllib_parse'>

request = <module 'mripy.six.moves.urllib.request'>

response = <module 'mripy.six.moves.urllib.response'>

robotparser = <module 'mripy.six.moves.urllib.robotparser'>

class mripy.six.**Module_six_moves_urllib_error**(*name*)

Bases: _LazyModule

Lazy loading of moved objects in six.moves.urllib_error

ContentTooShortError

HTTPError

URLError

class mripy.six.**Module_six_moves_urllib_parse**(*name*)

Bases: _LazyModule

Lazy loading of moved objects in six.moves.urllib_parse

ParseResult

SplitResult

parse_qs

parse_qsl

quote

quote_plus

splitquery

splittag

splituser

splitvalue

unquote

unquote_plus

unquote_to_bytes

urldefrag

urlencode

urljoin

urlparse

urlsplit

urlunparse

urlunsplit

uses_fragment

uses_netloc

uses_params

uses_query

uses_relative

class mripy.six.Module_six_moves_urllib_request(*name*)

Bases: _LazyModule

Lazy loading of moved objects in six.moves.urllib_request

AbstractBasicAuthHandler

AbstractDigestAuthHandler

BaseHandler

CacheFTPHandler

FTPHandler

FancyURLopener

FileHandler

HTTPBasicAuthHandler

HTTPCookieProcessor

HTTPDefaultErrorHandler

HTTPDigestAuthHandler

HTTPErrorProcessor

HTTPHandler

HTTPPasswordMgr

HTTPPasswordMgrWithDefaultRealm

HTTPRedirectHandler

HTTPSHandler

OpenerDirector

ProxyBasicAuthHandler

ProxyDigestAuthHandler

ProxyHandler

Request

URLopener

UnknownHandler

build_opener

getproxies

install_opener

pathname2url

proxy_bypass

url2pathname

urlcleanup

urlopen

urlretrieve

class mripy.six.Module_six_moves_urllib_response(*name*)

Bases: _LazyModule

Lazy loading of moved objects in six.moves.urllib_response

addbase

addclosehook

addinfo

addinfourl

class mripy.six.Module_six_moves_urllib_robotparser(*name*)

Bases: _LazyModule

Lazy loading of moved objects in six.moves.urllib_robotparser

RobotFileParser

class mripy.six.MovedAttribute(*name*, *old_mod*, *new_mod*, *old_attr=None*, *new_attr=None*)

Bases: _LazyDescr

class mripy.six.MovedModule(*name*, *old*, *new=None*)

Bases: _LazyDescr

mripy.six.**add_metaclass**(*metaclass*)

Class decorator for creating a class with a metaclass.

mripy.six.**add_move**(*move*)

Add an item to six.moves.

mripy.six.**assertCountEqual**(*self*, **args*, ***kwargs*)

mripy.six.**assertRaisesRegex**(*self*, **args*, ***kwargs*)

mripy.six.**assertRegex**(*self*, **args*, ***kwargs*)

mripy.six.**b**(*s*)

Byte literal

mripy.six.**create_unbound_method**(*func*, *cls*)

mripy.six.**get_unbound_function**(*unbound*)

Get the function out of a possibly unbound function

mripy.six.**int2byte**()

S.pack(v1, v2, ...) -> bytes

Return a bytes object containing values v1, v2, ... packed according to the format string S.format. See help(struct) for more on format strings.

mripy.six.**iteritems**(*d*, ***kw*)

Return an iterator over the (key, value) pairs of a dictionary.

mripy.six.**iterkeys**(*d*, ***kw*)

Return an iterator over the keys of a dictionary.

mripy.six.**iterlists**(*d*, ***kw*)

Return an iterator over the (key, [values]) pairs of a dictionary.

mripy.six.**itervalues**(*d*, ***kw*)

Return an iterator over the values of a dictionary.

mripy.six.**python_2_unicode_compatible**(*klass*)

A decorator that defines __unicode__ and __str__ methods under Python 2. Under Python 3 it does nothing.

To support Python 2 and 3 with a single code base, define a __str__ method returning text and apply this decorator to the class.

mripy.six.**raise_from**(*value*, *from_value*)

mripy.six.**remove_move**(*name*)

Remove item from six.moves.

mripy.six.**reraise**(*tp*, *value*, *tb=None*)

Reraise an exception.

mripy.six.**u**(*s*)

Text literal

mripy.six.**with_metaclass**(*meta*, **bases*)

Create a base class with a metaclass.

3.14 mripy.surface module

3.15 mripy.timecourse module

class mripy.timecourse.**Attributes**(*shape*)

Bases: object

add(*name*, *value*, *axis*)

classmethod concatinate(*attributes_list*, *axis*)

drop(*name*)

drop_all_with_axis(*axis*)

classmethod from_dict(*d*)

names_with_axis(*axis*)

pick(*index*, *axis*)

property shape

to_dict()

class mripy.timecourse.**Epochs**(*raw*, *events*, *event_id=None*, *tmin=-5*, *tmax=15*, *baseline=(-2, 0)*, *dt=0.1*, *interp='linear'*, *hamm=None*, *conditions=None*)

Bases: Savable, object

add_event_attr(*name*, *value*)

add_feature_attr(*name*, *value*)

aggregate(*event=True*, *feature=False*, *time=False*, *method=<function nanmean>*, *keepdims=<no value>*, *return_index=False*)

apply_baseline(*baseline*)

average(*feature=True*, *time=False*, *method=<function nanmean>*, *error='bootstrap'*, *ci=95*, *n_boot=1000*, *condition=None*)

Average data over event (and optionally feature and/or time) dimensions, and return an Evoked object.

copy()

drop_events(*ids*)

classmethod from_array(*data*, *TR=None*, *tmin=None*, *baseline=(-2, 0)*, *events=None*, *event_id=None*, *conditions=None*)

classmethod from_dict(*d*)

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish, or a proper initialization requires additional operations performed in __init__ but omitted here.

property n_events

property n_features

property n_times

pick(*event=None*, *feature=None*, *time=None*)

plot(*hue=None*, *style=None*, *row=None*, *col=None*, *hue_order=None*, *style_order=None*, *row_order=None*, *col_order=None*, *palette=None*, *dashes=None*, *figsize=None*, *legend=True*, *bbox_to_anchor=None*, *subplots_kws=None*, *average_kws=None*, *axs=None*, ***kwargs*)

property shape

summary(*event=False*, *feature=True*, *time=False*, *method=<function nanmean>*, *attributes=None*)

Summary data as a pandas DataFrame.

to_dict()

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish (i.e., other than int, float, str, list, tuple, dict, numpy array, pandas dataframe).

transform(*feature_name*, *feature_values*, *transformer*)

Transform the data into a new Epochs object. E.g., epochs.transform('depth', depth, tc.wcutter(depth, linspace(0, 1, 20), win_size=0.2, win_func='gaussian', exclude_outer=True))

class mripy.timecourse.**Evoked**(*info*, *data*, *nave*, *times*, *error=None*, *error_type=None*, *condition=None*)

Bases: object

plot(*color=None*, *error=True*, *info=True*, *error_kws=None*, *show_n='info'*, ***kwargs*)

property shape

class mripy.timecourse.**Raw**(*fname*, *mask=None*, *TR=None*)

Bases: *Savable*, object

property TR

copy()

classmethod from_array(*data*, *TR*)

data : 2D array, [n_features, n_times] TR : in sec

classmethod from_dict(*d*)

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish, or a proper initialization requires additional operations performed in __init__ but omitted here.

property n_features

property n_times

plot(*events=None*, *event_id=None*, *color=None*, *palette=None*, *figsize=None*, *event_kws=None*, ***kwargs*)

property shape

to_dict()

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish (i.e., other than int, float, str, list, tuple, dict, numpy array, pandas dataframe).

class mripy.timecourse.**RawCache**(*fnames*, *mask*, *TR=None*, *cache_file=None*, *force_redo=False*)

Bases: *Savable*, object

classmethod from_dict(*d*)

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish, or a proper initialization requires additional operations performed in __init__ but omitted here.

get_epochs(*mask*, *events*, *event_id*, *ids=None*, *cache_file=None*, ***kwargs*)

get_raws(*mask*, *ids=None*)

property n_runs

subset(*mask*, *cache_file=None*)

to_dict()

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish (i.e., other than int, float, str, list, tuple, dict, numpy array, pandas dataframe).

mripy.timecourse.**concatinate_epochs**(*epochs_list*, *axis=0*)

mripy.timecourse.**convolve_HRF**(*starts*, *lens*, *TR=2*, *scan_time=None*, *HRF=None*)

mripy.timecourse.**create_ERP**(*t*, *x*, *events*, *tmin=-8*, *tmax=16*, *dt=0.1*, *baseline=[-2, 0]*, *interp='linear'*)

t : time for each data point (can be non-contiguous) x : [feature, time] events : event onset time (can be on non-integer time point)

mripy.timecourse.**create_base_corr_func**(*times*, *baseline=None*, *method=None*)

Parameters

- **time** (*array-like*) – Sampling time for your data.
- **baseline** (*'none', 'all', or (tmin, tmax)*) – Baseline time interval.

mripy.timecourse.**create_ideal**(*stimuli*, *lens*, ***kwargs*)

Parameters

stimuli (*list of fname*) –

mripy.timecourse.**create_times**(*tmin*, *tmax*, *dt*)

mripy.timecourse.**cut**(*data*, *val*, *bins*, ***kwargs*)

mripy.timecourse.**cutter**(*val*, *bins*)

mripy.timecourse.**events_from_dataframe**(*df*, *time*, *conditions*, *duration=None*, *run=None*, *n_runs=None*, *event_id=None*)

Parameters

- **df** (*dataframe*) –
- **time** (*str*) –
- **conditions** (*list (names) or OrderedDict (name->levels)*) –
- **duration** (*str*) –
- **run** (*str*) –
- **event_id** (*dict (name->index)*) –

Returns

- **events** (*array*)

- **event_id** (*dict (name->index)*)

mripy.timecourse.**events_to_dataframe**(*events_list*, *event_id*, *conditions*)

mripy.timecourse.**first_peak**(*t*, *x*, *after=2*, *trough='before'*, *sg_win=41*, *sg_order=3*, *visualize=False*, *ax=None*)

Difference between first peak after *after* and first trough before it.

mripy.timecourse.**get_HRF**(*name*, *TR=2*, *duration=32*, *normalize='area'*, ***kwargs*)

Get hemodynamic response function.

Parameters

- **name** (*str*) – Name of the hemodynamic response function model. - 'canonical': difference of two gamma distribution functions [1]

  hrf = gamma.pdf(t, a1, scale=b1) - c*stats.gamma.pdf(t, a2, scale=b2) Acceptable kwargs are a1=6, b1=1, a2=16, b2=1, c=1/6. a1, a2 and b1, b2 are the shape and scale parameter of the two gammas, and c is the relative amplitude of the 2nd (negative) gamma. The default result matches the default spm_hrf result in SPM12.

  - **'SPM': difference of two gamma, parameterized as in spm_hrf (SPM12) [2]**
    Acceptable kwargs are P=[6,16,1,1,6,0,32], T=16. See spm_hrf for more. The conversion between 'canonical' and 'SPM' parameterizations: In scipy convention, larger scale parameter inplies wider distribution. So for gamma distribution parameterized as

    beta^alpha * x^(alpha-1) * np.exp(-beta*x) / special.gamma(alpha)

    we have scale = 1/beta, i.e., beta = 1/scale And since u (=0,1,2,...) is x * 1/dt, to keep the scale, beta should be multiplied by dt. So we have beta = dt/scale, and p3, p4 are actually the scale (disperse) parameter of gamma. p1, p2 are (standardized) shape parameters in units of scale parameter, p1 = alpha*p3, i.e., alpha = p1/p3. Note that the mean of the Gamma distribution is shape*scale and its mode is (shape-1)*scale. This means the positive and negative peaks of the hrf are approximately p1-1 and p2-1. Finally, p5 is the response/undershoot ratio, i.e., 1/c as in the 'canonical' parameterization. In summary, the following two expressions are equivalent:

    get_HRF('canonical', a1=a1, a2=a2, b1=b1, b2=b2, c=c) get_HRF('SPM', P=[a1*b1, a2*b2, b1, b2, 1/c, 0, 32])

  - **'GAM': one gamma (no undershoot), from AFNI 3dDeconvolve [3]**

    hrf = (t/(p*q))^p * exp(p-t/q)

    Acceptable kwargs are p=8.6, q=0.547. The peak of 'GAM(p,q)' is at time p*q after the stimulus. The FWHM is about 2.35*sqrt(p)*q.

  - **'TDM-early' and 'TDM-late': two-gamma model for the early and late**
    HRF components in temporal decomposition through manifold fitting [4] Note that the two HRF components should be fitted together in a GLM, although they have high correlation (~0.8).

- **TR** (*float*) – Sampling duration in seconds.
- **duration** (*float*) – Length of kernel in seconds.
- **normalize** (*str*) –
  - 'area' : normalize the (signed) area under the curve to one.
  - 'height' : normalize the max respones to one.

- ****kwargs** – Additional parameters adjustable for the HRF model. See *name* for details.

Returns

- **t** (*array*) – The time vector [0:TR:duration).
- **hrf** (*array*) – The HRF evaluated at each point of the time vector.

References

[1] Lindquist, M. A., Meng Loh, J., Atlas, L. Y., & Wager, T. D. (2009).
Modeling the hemodynamic response function in fMRI: Efficiency, bias and mis-modeling. NeuroImage, 45(1, Supplement 1), S187–S198.

[2] https://github.com/spm/spm/blob/main/spm_hrf.m [3] https://afni.nimh.nih.gov/pub/dist/doc/program_help/3dDeconvolve.html [4] Kay, K., Jamison, K. W., Zhang, R.-Y., & Uğurbil, K. (2020). A temporal

decomposition method for identifying venous effects in task-based fMRI. Nature Methods.

Notes

Gamma distribution function and gamma function are different things.

mripy.timecourse.**group_epochs**(*epochs_list*, *pool_feature=True*)

mripy.timecourse.**qcut**(*data*, *val*, *q*, ***kwargs*)

mripy.timecourse.**qcutter**(*val*, *q*)

mripy.timecourse.**read_events**(*event_files*)

Read events from AFNI style (each row is a run, and each element is an occurance) stimulus timing files.

Parameters

event_files (*dict*) – e.g., OrderedDict((‘Physical/Left’, ‘../stimuli/phy_left.txt’), (‘Physical/Right’, ‘../stimuli/phy_right.txt’), . . .)

Returns

- **events_list** (*list of n_events-by-3 arrays*) – The three columns are [start_time(sec), reserved, event_id(int)]
- **event_id** (*dict*)

mripy.timecourse.**wcut**(*data*, *val*, *v*, *win_size*, *win_func=None*, *exclude_outer=False*, ***kwargs*)

mripy.timecourse.**wcutter**(*val*, *v*, *win_size*, *win_func=None*, *exclude_outer=False*)

3.16 mripy.utils module

class mripy.utils.**CacheManager**(*persistent_file=None*, *ignore_init_run=True*)

Bases: object

exists(*fname*, *watch_files=None*, *force_redo=False*, ***kwargs*)

kwargs_updated(*fname*, *kwargs*)

load_contexts()

save_contexts()

watch_files_updated(*fname*, *watch_files*)

class mripy.utils.**FilenameManager**(*fmt*, ***kwargs*)

Bases: object

classmethod fmt2kws(*fmt*, ***kwargs*)

format(*fmt=None*, *keepdims=False*, ***kwargs*)

classmethod from_glob(*fmt*, ***kwargs*)

glob(***kwargs*)

parse(*files*, *multi_value='list'*)

files : list of filenames multi_value : {'wildcard', 'list'}

class mripy.utils.**ParallelCaller**

Bases: object

check_call(*cmd*, ***kwargs*)

Asynchronous check_call (launch and return immediately). Multiple commands can be stitched sequentially with ";" in linux/mac only if shell=True.

wait()

Wail for all parallel processes to finish (= wait for the slowest one).

class mripy.utils.**Savable**

Bases: object

classmethod from_dict(*d*)

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish, or a proper initialization requires additional operations performed in __init__ but omitted here.

classmethod load(*fname*)

save(*fname*)

to_dict()

Subclasses are expected to override this method if the data type of some attributes are not supported by deepdish (i.e., other than int, float, str, list, tuple, dict, numpy array, pandas dataframe).

class mripy.utils.**Savable2**

Bases: object

load(*fname*)

save(*fname*)

mripy.utils.**cd**(*d*)

mripy.utils.**contain_wildcard**(*fname*)

mripy.utils.**exists**(*fname*, *force_redo=False*)

Lazy evaluation the body only if fname not already exists.

mripy.utils.**expand_index_list**(*index_list*, *format=None*)

Expand a list of indices like: 1 3-5 7:10:2 8..10(2)

mripy.utils.**factorize**(*n*)

mripy.utils.**fname_with_ext**(*fname*, *ext*)

mripy.utils.**has_N4**()

mripy.utils.**has_ants**()

mripy.utils.**has_hcp_retino_docker**()

mripy.utils.**iterable**(*x*)

mripy.utils.**parallel_1D**(*cmd*, *in_file*, *prefix*, *n_jobs=1*, *combine_output=True*, ***kwargs*)

Parameters

cmd (*str*) – The command for parallel execution, must contain two placeholders {prefix} and {in_file}, which will be substituded with ‘’.format(). As expected, other {} must be escaped as {{}}. For example, >>> parallel_1D(“3dTcat -prefix {prefix} -overwrite {in_file}’{{0:9}}’”, ‘xyz_list.1D’, ‘test’, n_jobs=4)

mripy.utils.**parallel_3D**(*cmd*, *in_file*, *prefix*, *n_jobs=1*, *schema=None*, *fname_mapper=None*, *combine_output=True*, ***kwargs*)

Parameters

fname_mapper (*dict or callable*) –

mripy.utils.**select_and_replace_affix**(*glob_pattern*, *old_affix*, *new_affix*)

mripy.utils.**temp_folder**(*parent=None*)

mripy.utils.**temp_prefix**(*prefix='tmp_'*, *n=4*, *suffix='.'*)

3.17 mripy.vis module

mripy.vis.**draw_color_circle**(*cmap*, *res=512*)

Draw circular colorbar (color index from 0 to 1) clockwise from 0 to 12 o’clock

mripy.vis.**get_color_list**(*cmap*)

This function is still wrong!

mripy.vis.**plot_volreg**(*dfiles*, *convention=None*)

The six columns from left to right are:

- For 3dvolreg: roll(z), pitch(x), yaw(y), dS(z), dL(x), dP(y) Note that the interpretation for roll and yaw in afni is opposite from SPM (and common sense…) This program will follow the SPM convention.
- For 3dAllineate shift_rotate: x-shift y-shift z-shift$ z-angle x-angle$ y-angle$ Note that the dollar signs in the end indicate parameters that are fixed.

3.18 Module contents

CHAPTER

FOUR

GLM ANALYSIS FOR FMRI

We can use *mripy.preprocess.glm()* to analyze our fMRI data using a General Linear Model.

```
import numpy as np
import matplotlib.pyplot as plt
from mripy import io, preprocess as prep

data_dir = 'path/to/my/experiment/data'
subject = 'sub-01'
res_dir = f"{data_dir}/{subject}/results"
stim_dir = f"{data_dir}/{subject}/stimuli"
runs = [f"{k:02d}" for k in [1, 2, 3, 4]] # ['01', '02', '03', '04']
TR = 2 # s
```

4.1 Block design

4.1.1 Fixed duration blocks

This is the most basic design, especially for functional localizers.

In the following example, we have Face and House blocks, and each block lasts 16 s. The model can be specified as `BLOCK(16)`.

```
# Perform GLM using "BLOCK" model
design = OrderedDict()
design['Face'] = f"{stim_dir}/Face_block_onset.txt 'BLOCK(16)'"
design['House'] = f"{stim_dir}/House_block_onset.txt 'BLOCK(16)'"
contrasts = OrderedDict()
contrasts['F-H'] = '+Face -House'
prep.glm(in_files=[f"{res_dir}/epi{run}.scale.nii" for run in runs],
    out_file=f"{res_dir}/localizer.nii", design=design, contrasts=contrasts, TR=TR,
    motion_files=[f"{res_dir}/epi{run}.volreg.param.1D" for run in runs])

# The above method will generate the following files:
# - stats.localizer_REML.nii   # This is the usual stats file with F, beta, t values for␣
↪each regressor
```

4.1.2 Variable duration blocks

This is useful for modelling spontaneous change in states, e.g., binocular rivalry.

In this case, we can use the "dmUBLOCK" model in AFNI. "dm" stands for "duration modulated".

```
# Prepare event timing files with both block onset and block duration
# t['L'] and d['L'] contain the onset and duration of "Left eye dominant" event for all runs
t = {'L': [
            [1.2, 7.5, 15.7, 20.3, ...], # run01 "Left eye dominant" event onset
            [3.9, 9.4, 19.2, 25.5, ...], # run02 "Left eye dominant" event onset
            ...,
        ],
    'R': [...] # "Right eye dominant" event onset
}
d = {'L': [
            [4.3, 5.1, 2.6, 7.8, ...], # run01 "Left eye dominant" event duration
            [2.9, 6.2, 2.2, 5.3, ...], # run02 "Left eye dominant" event duration
            ...,
        ],
    'R': [...] # "Right eye dominant" event duration
}
for event in ['L', 'R']: # Left eye dominant vs Right eye dominant
    with open(f"{stim_dir}/{event}_rivalry.txt", 'w') as fout:
        # `tt` and `dd` contain the onset and duration for all events in a run
        # `ttt` and `ddd` are the onset and duration for each event (i.e., button press)
        for tt, dd in zip(t[event], d[event]):
            fout.write(' '.join([f"{ttt:8.3f}:{ddd:<8.3f}" for ttt, ddd in zip(tt, dd)])␣
↪+ '\n')

# Perform GLM using "dmUBLOCK" model
design = OrderedDict()
design['L'] = f"{stim_dir}/L_rivalry.txt 'dmUBLOCK'"
design['R'] = f"{stim_dir}/R_rivalry.txt 'dmUBLOCK'"
contrasts = OrderedDict()
contrasts['L+R'] = '+0.5*L +0.5*R'
contrasts['L-R'] = '+L -R'
prep.glm(in_files=[f"{res_dir}/epi{run}.scale.nii" for run in runs],
    out_file=f"{res_dir}/rivalry.nii", design=design, contrasts=contrasts, TR=TR,
    motion_files=[f"{res_dir}/epi{run}.volreg.param.1D" for run in runs])

# The above method will generate the following files:
# - stats.rivalry_REML.nii   # This is the usual stats file with F, beta, t values for␣
↪each regressor
```

4.2 Event-related design

4.2.1 Assume a particular shape for the HRF

We can assume the evoked fMRI response takes a particular shape of the HRF (Haemodynamic Response Function), with only one free parameter that we may adjust to fit our data, which is the amplitude of the peak response. By convention, we call this free parameter β.

The particular shape of HRF has many variants: GAM, SPM1, SPM2, SPM3, etc.

In the following example, we have two events: A and B.

```
# Perform GLM using "GAM" model
design = OrderedDict()
design['A'] = f"{stim_dir}/EventA_onset_time.txt 'GAM'"
design['B'] = f"{stim_dir}/EventB_onset_time.txt 'GAM'"
prep.glm(in_files=[f"{res_dir}/epi{run}.scale.nii" for run in runs],
    out_file=f"{res_dir}/ER.nii", design=design, TR=TR,
    motion_files=[f"{res_dir}/epi{run}.volreg.param.1D" for run in runs])

# The above method will generate the following files:
# - stats.ER_REML.nii   # This is the usual stats file with F, beta, t values for each␣
↪regressor
```

4.2.2 No assumption about the shape of HRF

This is referred to as TENT or CSPLIN model in AFNI, and FIR model in SPM.

TENT (n parameter tent function) models the evoked fMRI response by each event as a piecewise linear function.

CSPLIN (n parameter cubic spline function, n>=4) is a drop-in upgrade of TENT to a differentiable (i.e., smooth) set of functions. And this is our default choice.

Since we now have more parameters (i.e., beta values) to estimate for each brain location, we need more data to get a result with reasonably low variance. This can be achieved either by acquiring more time points (averaging over time), or by pooling all voxels in your ROI before running the GLM (averaging over space).

In the following example, we have two events: A and B. We first average all voxels in our ROI into a single time series, and then perform deconvolution to estimate the brain respones to event A and event B.

We assume the response starts at 0 s after the event onset, lasting 24 s, and we want to sample the response every 2 s (which does not need to be equal to the TR). This results in 13 samples over the 24 s period. We can express the model as `CSPLIN(0,24,13)`. If we further assume the response starts from 0 at 0 s, and has already been settled to 0 at 24 s after the event onset, the resulting model can be written as `CSPLINzero(0,24,13)`, although we now only have 11 free parameters to estimate.

```
# Average all voxels in the ROI into a single time series
mask = io.Mask(f"{res_dir}/V1.nii")
for run in runs:
    data = mask.dump(f"{res_dir}/epi{run}.scale.nii") # Dump data within the ROI -> [n_
↪voxels, n_times]
    data = data.mean(axis=0, keepdims=True) # Average across voxels
    # Note that the data must be 2D: [n_time_series, n_times], while n_time_series can␣
↪be 1
    # This allow you to save data from multiple ROIs or multiple conditions in a single␣
```

(continues on next page)

(continued from previous page)

```
↪*.1D file
    np.savetxt(f"{res_dir}/epi{run}.1D", data, fmt='%.6f') # Save as *.1D file (plain␣
↪text)

# Perform GLM using "CSPLINzero" model
design = OrderedDict()
design['A'] = f"{stim_dir}/EventA_onset_time.txt 'CSPLINzero(0,24,13)'"
design['B'] = f"{stim_dir}/EventB_onset_time.txt 'CSPLINzero(0,24,13)'"
prep.glm(in_files=[f"{res_dir}/epi{run}.1D" for run in runs],
    out_file=f"{res_dir}/V1_resp.1D", design=design, TR=TR,
    motion_files=[f"{res_dir}/epi{run}.volreg.param.1D" for run in runs])

# The above method will generate the following files:
# - stats.V1_resp_REML.1D   # This is the usual stats file with F, beta, t values for␣
↪each regressor
# - irp.A.V1_resp_REML.1D   # The impulse response of event A \
# - irp.B.V1_resp_REML.1D   # The impulse response of event B - These two are our␣
↪estimated impulse responses
irf = np.zeros([2,13]) # Impulse response function: [n_events, n_times]
for k, event in enumerate(['A', 'B']):
    # Remember that the first (0th) and the last (12th) element are zero by construction
    irf[k,1:-1] = io.read_txt(f"{res_dir}/irp.{event}.V1_resp_REML.1D")

# Plot the estimated evoked fMRI responses for event A and B
t = np.linspace(0, 24, 13) # Time after event onset in seconds
plt.plot(t, irf.T)
```

4.3 Need more flexibility in doing GLM?

Calling 3dDeconvolve directly allows you to access more types of models, and control the behavior of the estimation process in more details.

If you need more help or details about the underlying algorithm, the ultimate source of reference is the AFNI documentation about its 3dDeconvolve command.

CHAPTER

FIVE

MRIPY

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z