BrainSpace Documentation

Release 0.1.4

Reinder Vos de Wael, Oualid Benkarim, Boris Bernhardt

Aug 01, 2022

Table of Contents:

BrainSpace is a lightweight cross-platform toolbox primarily intended for macroscale gradient mapping and analysis of neuroimaging and connectome level data. The current version of BrainSpace is available in Python and MATLAB, programming languages widely used by the neuroimaging and network neuroscience communities. The toolbox also contains several maps that allow for exploratory analysis of gradient correspondence with other brain-derived features, together with tools to generate spatial null models.

CHAPTER 1

Installation Guide

BrainSpace is available in Python and MATLAB.

1.1 Python installation

BrainSpace works on Python 3.5+, and probably with older versions of Python 3, although it is not tested.

1.1.1 Dependencies

To use BrainSpace, the following Python packages are required:

- numpy
- scipy
- scikit-learn
- vtk
- matplotlib
- nibabel
- nilearn

Nibabel is required for reading/writing Gifti surfaces. Matplotlib is only used for colormaps and we may remove this dependency in future releases.

1.1.2 Additional dependencies

To enable interactivity, some plotting functionality in IPython notebooks makes use of the panel package. PyQT is another dependency for background plotting. See PyVista for more on background plotting. The support of background rendering however is still experimental.

- panel
- pyqt

1.1.3 Installation

BrainSpace can be installed using `pip`:

```
pip install brainspace
```

Alternatively, you can install the package from Github as follows:

```
git clone https://github.com/MICA-MNI/BrainSpace.git
cd BrainSpace
python setup.py install
```

1.2 MATLAB installation

This toolbox has been tested with MATLAB versions R2018b, although we expect it to work with versions R2018a and newer. Operating systems used during testing were OSX Mojave (10.14.6) and Linux Xenial Xerus (16.04.6).

BrainSpace can be installed by downloading and unzipping the code from Github and running the following in MATLAB:

```
addpath(genpath('/path/to/BrainSpace/matlab/'))
```

If you want to load BrainSpace every time you start MATLAB, type `edit startup` and append the above line to the end of this file.

You can move the MATLAB directory to other locations. However, the example data loader functions used in our tutorials require the MATLAB and shared directories to both be in the same directory.

If you wish to open gifti files (necessary for the tutorial) you will also need to install the gifti library.

CHAPTER 2

Getting Started

2.1 Introduction to Gradients

Classically, many neuroimaging studies have attempted to parcellate the human brain into distinct areas based on anatomical or functional features. Whilst effective, the strict boundaries assumed by these methods are often unrealistic and there lacks a clear ordering between parcels. More recently, "gradient" approaches, which define the brain as a set of continuous scores along manifold axes, have gained popularity (see also *References*). Core to these techniques is the computation of an affinity matrix that captures inter-area similarity of a given feature followed by the application of dimensionality reduction techniques to identify a gradual ordering of the input matrix in a lower dimensional manifold space.

BrainSpace is a compact and flexible toolbox that implements a wide variety of approaches to build macroscale gradients from neuroimaging and connectome data. In this overview, we will show the the basic steps needed for the identi cation of gradients, and their visualization. The steps below will help you to get started and to build your first gradients. If you haven't done so yet, we recommend you install the package (*Installation Guide*) so you can follow along with the examples.

2.2 Basic Gradient Construction

The packages comes with the conte69 surface, and several cortical features and parcellations. Let's start by loading the conte69 surfaces:

Python

Matlab

```
>>> from brainspace.datasets import load_conte69

>>> # Load left and right hemispheres
>>> surf_lh, surf_rh = load_conte69()
>>> surf_lh.n_points
32492
```

(continues on next page)

(continued from previous page)

```
>>> surf_rh.n_points
32492
```

```
addpath(genpath('/path/to/micasoft/BrainSpace/matlab'));

% Load left and right hemispheres
[surf_lh, surf_rh] = load_conte69();
```

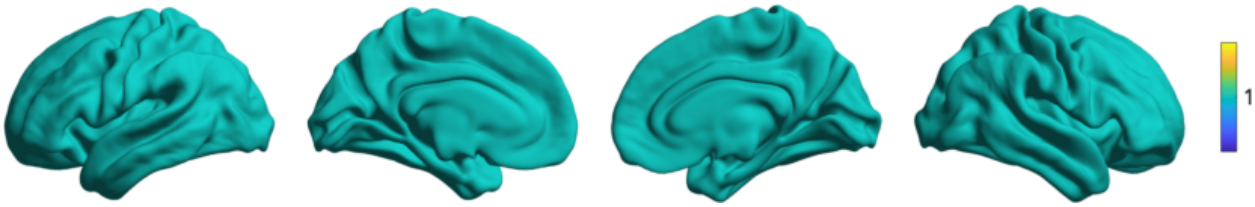
To load your own surfaces, you can use the *read_surface* function. BrainSpace also provides surface plotting functionality. We can plot the conte69 surfaces as follows:

Python

Matlab

```
>>> from brainspace.plotting import plot_hemispheres

>>> plot_hemispheres(surf_lh, surf_rh, size=(800, 200))
```

```
plot_hemispheres(ones(64984,1),{surf_lh,surf_rh});
```



Let's also load the mean connectivity matrix built from a subset of the human connectome project (HCP). The package comes with several example matrices, downsampled using the Schaefer parcellations (Schaefer et al., 2017). Let's load one of them.

Python

Matlab

```
>>> from brainspace.datasets import load_group_fc, load_parcellation

>>> labeling = load_parcellation('schaefer', scale=400, join=True)
>>> m = load_group_fc('schaefer', scale=400)
>>> m.shape
(400, 400)
```

```
labeling = load_parcellation('schaefer',400);
conn_matices = load_group_fc('schaefer',400);
m = conn_matices.schaefer_400;
```

To compute the gradients of our connectivity matrix *m* we create the *GradientMaps* object and fit the model to our data:

Python

Matlab

```
>>> from brainspace.gradient import GradientMaps
```

(continues on next page)

(continued from previous page)

```
>>> # Build gradients using diffusion maps and normalized angle
>>> gm = GradientMaps(n_components=2, approach='dm', kernel='normalized_angle')

>>> # and fit to the data
>>> gm.fit(m)
GradientMaps(alignment=None, approach='dm', kernel='normalized_angle',
             n_components=2, random_state=None)

>>> # The gradients are in
>>> gm.gradients_.shape
(400, 2)
```

```
% Build gradients using diffusion maps and normalized angle
gm = GradientMaps('kernel','na','approach','dm','n_components',2);

% and fit to the data
gm = gm.fit(m);
```

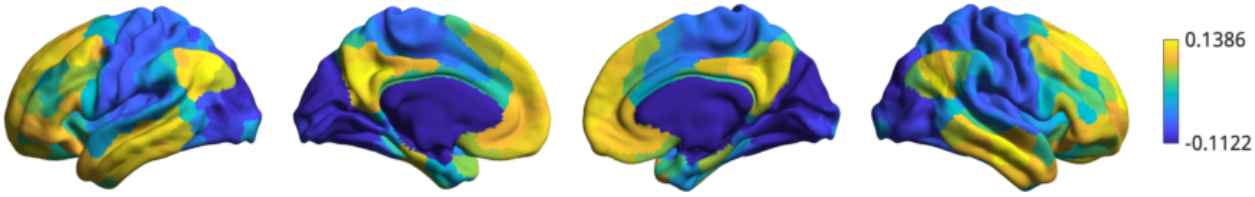
Now we can visually inspect the gradients. Let's plot the first gradient:

Python

Matlab

```
>>> import numpy as np
>>> from brainspace.utils.parcellation import map_to_labels

>>> # map to original size
>>> grad = map_to_labels(gm.gradients_[:, 0], labeling, mask=labeling != 0,
...                      fill=np.nan)

>>> # Plot first gradient on the cortical surface.
>>> plot_hemispheres(surf_lh, surf_rh, array_name=grad, size=(800, 200))
```

```
% Plot the first gradient on the cortical surface.
plot_hemispheres(gm.gradients{1}(:,1), {surf_lh,surf_rh}, ...
        'parcellation',labeling.schaefer_400);
```



As we can see, this gradient corresponds to those observed previously in the literature i.e. running from default mode to sensory areas.

That concludes this getting started section. For more full documentation and tutorials please see *MATLAB Package* and/or *Python Package*.

CHAPTER 3

Python Package

This page contains links to all related documents on BrainSpace python package. The *tutorials* is a good starting point to get familiar with the main functionality provided by BrainSpace: creation and alignment of *gradients*, and *null models*. With the tutorials you can also learn about the *data* that comes with the package, *plotting* and other functions. For more information, please refer to the API.

In the python package, surface functionality is built on top of the Visualization Toolkit (VTK). BrainSpace offers a *high-level interface* to work with VTK. In VTK wrapping, we introduce the wrapping scheme used in BrainSpace and some basic usage examples. Note, however, that this part is not a requirement to start using BrainSpace. In the *Mesh* module you can find most of the functionality you need to work with surfaces. A surface in BrainSpace is represented as `BSPolyData` object. Reading/writing of several formats is supported through the `read_surface` and `write_surface` functions.

3.1 Tutorials

3.1.1 Tutorial 0: Preparing your data for gradient analysis

In this example, we will introduce how to preprocess raw MRI data and how to prepare it for subsequent gradient analysis in the next tutorials.

Requirements

For this tutorial, you will need to install the Python package nilearn version 0.9.0 or above. You can do it using `pip`:

```
pip install "nilearn>=0.9.0"
```

Preprocessing

Begin with an MRI dataset that is organized in BIDS format. We recommend preprocessing your data using fmriprep, as described below, but any preprocessing pipeline will work.

Following is example code to run fmriprep using docker from the command line:

```
docker run -ti --rm \
  -v <local_BIDS_data_dir>:/data:ro \
  -v <local_output_dir>:/out poldracklab/fmriprep:latest \
  --output-spaces fsaverage5 \
  --fs-license-file license.txt \
  /data /out participant
```

Note: For this tutorial, it is crucial to output the data onto a cortical surface template space.

Import the dataset as timeseries

The timeseries should be a numpy array with the dimensions: nodes x timepoints

Following is an example for reading in data:

```
import nibabel as nib
import numpy as np

filename = 'filename.{}.mgz' # where {} will be replaced with 'lh' and 'rh'
timeseries = [None] * 2
for i, h in enumerate(['lh', 'rh']):
    timeseries[i] = nib.load(filename.format(h)).get_fdata().squeeze()
timeseries = np.vstack(timeseries)
```

As a **working example**, simply fetch timeseries:

```
from brainspace.datasets import fetch_timeseries_preprocessing
timeseries = fetch_timeseries_preprocessing()
```

Confound regression

To remove confound regressors from the output of the fmriprep pipeline, first extract the confound columns. For example:

```
from nilearn.interfaces.fmriprep import load_confounds_strategy
confounds_out = load_confounds_strategy("path/to/fmriprep/output/sub-<subject>_task-
↪<task>_space-<space>_desc-preproc_bold.nii.gz",
                                        denoise_strategy='simple')
```

As a **working example**, simply read in confounds

```
from brainspace.datasets import load_confounds_preprocessing
confounds_out = load_confounds_preprocessing()
```

Do the confound regression

```
from nilearn import signal
clean_ts = signal.clean(timeseries.T, confounds=confounds_out).T
```

And extract the cleaned timeseries onto a set of labels

```
import numpy as np
from nilearn import datasets
from brainspace.utils.parcellation import reduce_by_labels

# Fetch surface atlas
atlas = datasets.fetch_atlas_surf_destrieux()

# Remove non-cortex regions
regions = atlas['labels'].copy()
masked_regions = [b'Medial_wall', b'Unknown']
masked_labels = [regions.index(r) for r in masked_regions]
for r in masked_regions:
    regions.remove(r)

# Build Destrieux parcellation and mask
labeling = np.concatenate([atlas['map_left'], atlas['map_right']])
mask = ~np.isin(labeling, masked_labels)

# Distinct labels for left and right hemispheres
lab_lh = atlas['map_left']
labeling[lab_lh.size:] += lab_lh.max() + 1

# extract mean timeseries for each label
seed_ts = reduce_by_labels(clean_ts[mask], labeling[mask], axis=1, red_op='mean')
```

Out:

```
/home/oualid/Apps/anaconda3/envs/py3/lib/python3.7/site-packages/nilearn/datasets/__
↪init__.py:89: FutureWarning: Fetchers from the nilearn.datasets module will be␣
↪updated in version 0.9 to return python strings instead of bytes and Pandas␣
↪dataframes instead of Numpy arrays.
  "Numpy arrays.", FutureWarning)

Dataset created in /home/oualid/nilearn_data/destrieux_surface

Downloading data from https://www.nitrc.org/frs/download.php/9343/lh.aparc.a2009s.
↪annot ...
 ...done. (0 seconds, 0 min)
Downloading data from https://www.nitrc.org/frs/download.php/9342/rh.aparc.a2009s.
↪annot ...
 ...done. (0 seconds, 0 min)
```

Calculate functional connectivity matrix

The following example uses nilearn:

```
from nilearn.connectome import ConnectivityMeasure

correlation_measure = ConnectivityMeasure(kind='correlation')
correlation_matrix = correlation_measure.fit_transform([seed_ts.T])[0]
```

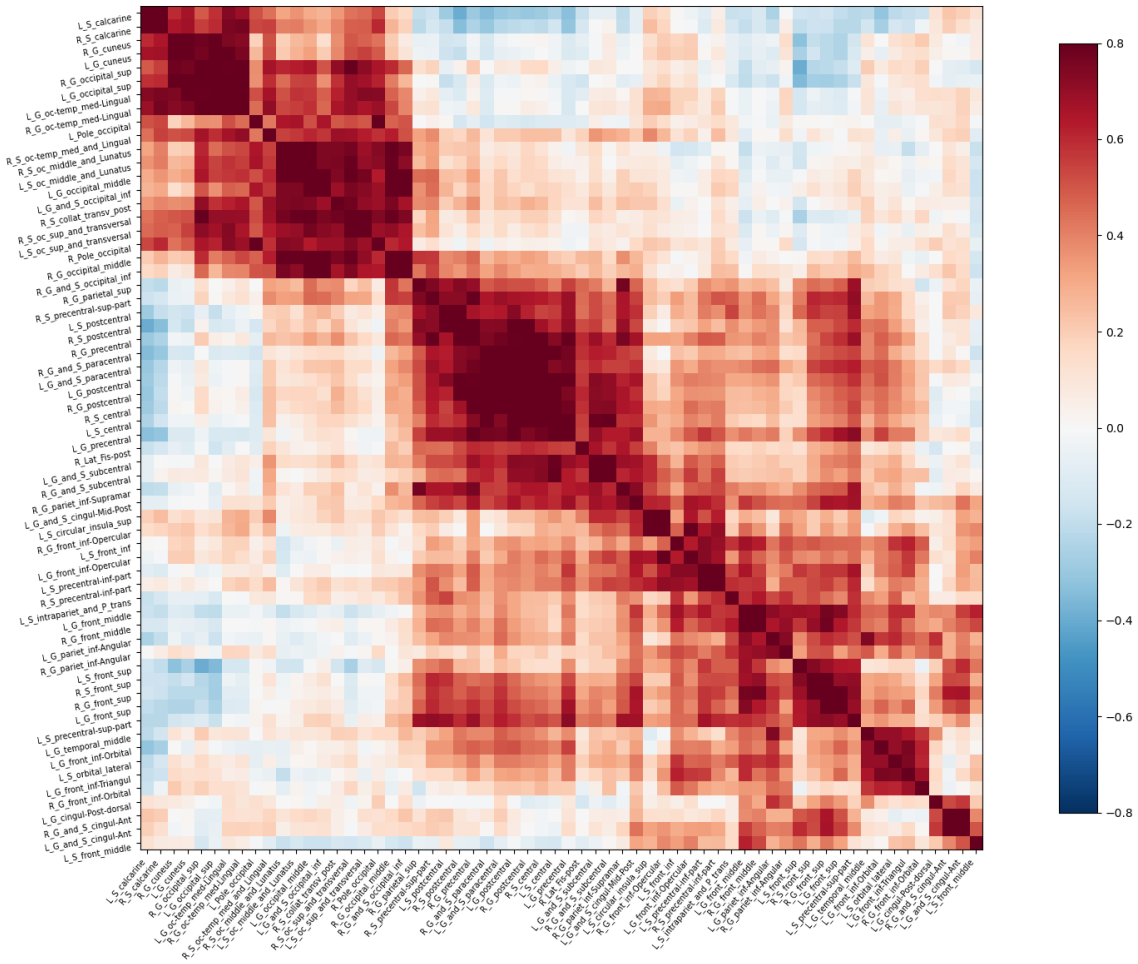
Plot the correlation matrix:

```
from nilearn import plotting
```

(continues on next page)

(continued from previous page)

```
# Reduce matrix size, only for visualization purposes
mat_mask = np.where(np.std(correlation_matrix, axis=1) > 0.2)[0]
c = correlation_matrix[mat_mask][:, mat_mask]

# Create corresponding region names
regions_list = ['%s_%s' % (h, r.decode()) for h in ['L', 'R'] for r in regions]
masked_regions = [regions_list[i] for i in mat_mask]


corr_plot = plotting.plot_matrix(c, figure=(15, 15), labels=masked_regions,
                                 vmax=0.8, vmin=-0.8, reorder=True)
```



Run gradient analysis and visualize

Run gradient analysis

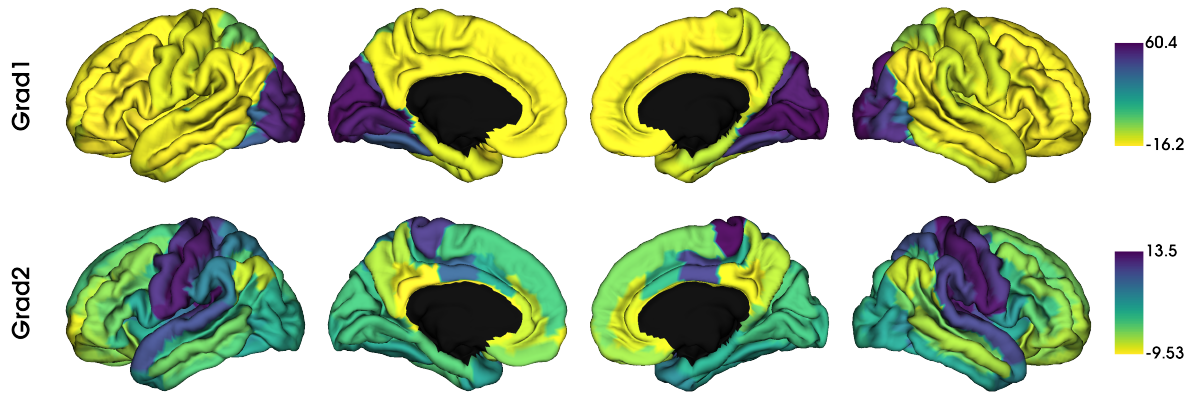
```
from brainspace.gradient import GradientMaps

gm = GradientMaps(n_components=2, random_state=0)
gm.fit(correlation_matrix)
```

Out:

```
GradientMaps(n_components=2, random_state=0)
```

Visualize results

```
from brainspace.datasets import load_fsa5
from brainspace.plotting import plot_hemispheres
from brainspace.utils.parcellation import map_to_labels

# Map gradients to original parcels
grad = [None] * 2
for i, g in enumerate(gm.gradients_.T):
    grad[i] = map_to_labels(g, labeling, mask=mask, fill=np.nan)


# Load fsaverage5 surfaces
surf_lh, surf_rh = load_fsa5()

plot_hemispheres(surf_lh, surf_rh, array_name=grad, size=(1200, 400), cmap='viridis_r
↪',
                 color_bar=True, label_text=['Grad1', 'Grad2'], zoom=1.5)
```



This concludes the setup tutorial. The following tutorials can be run using either the output generated here or the example data.

Total running time of the script: (0 minutes 3.871 seconds)

3.1.2 Tutorial 1: Building your first gradient

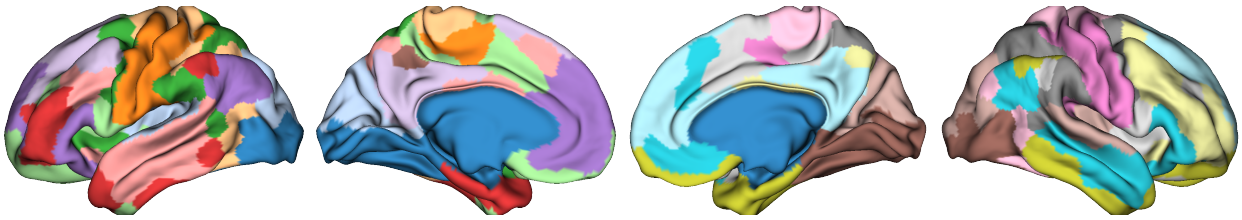
In this example, we will derive a gradient and do some basic inspections to determine which gradients may be of interest and what the multidimensional organization of the gradients looks like.

We'll first start by loading some sample data. Note that we're using parcellated data for computational efficiency.

```
from brainspace.datasets import load_group_fc, load_parcellation, load_conte69

# First load mean connectivity matrix and Schaefer parcellation
conn_matrix = load_group_fc('schaefer', scale=400)
labeling = load_parcellation('schaefer', scale=400, join=True)

# and load the conte69 surfaces
surf_lh, surf_rh = load_conte69()
```

Let's first look at the parcellation scheme we're using.

```
from brainspace.plotting import plot_hemispheres

plot_hemispheres(surf_lh, surf_rh, array_name=labeling, size=(1200, 200),
                 cmap='tab20', zoom=1.85)
```



and let's construct our gradients.

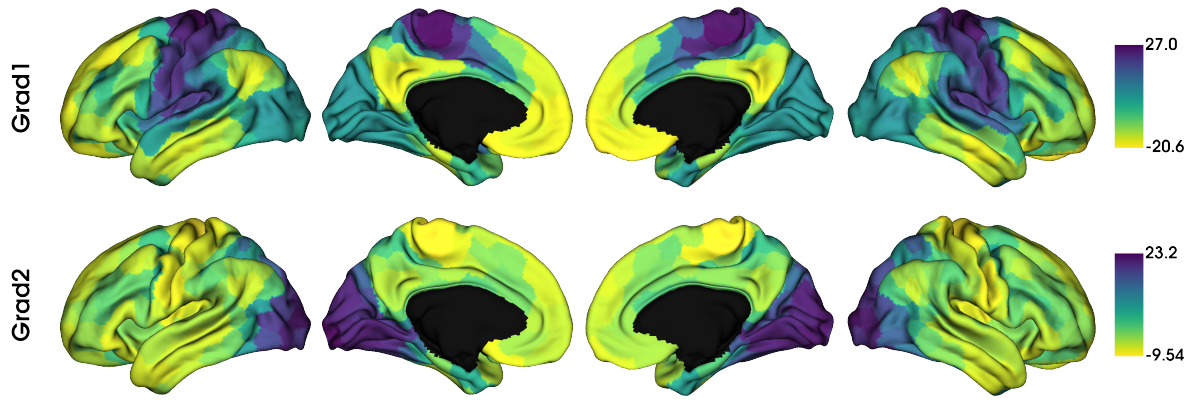
```
from brainspace.gradient import GradientMaps

# Ask for 10 gradients (default)
gm = GradientMaps(n_components=10, random_state=0)
gm.fit(conn_matrix)
```

Out:

```
GradientMaps(random_state=0)
```

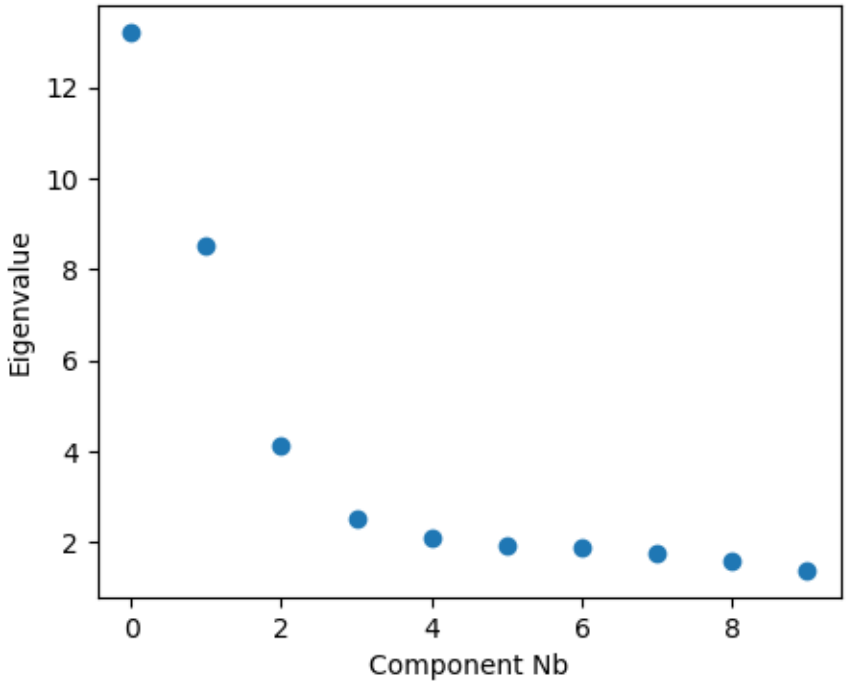
Note that the default parameters are diffusion embedding approach, 10 components, and no kernel (use raw data). Once you have your gradients, a good first step is to simply inspect what they look like. Let's have a look at the first two gradients.

```
import numpy as np

from brainspace.utils.parcellation import map_to_labels

mask = labeling != 0

grad = [None] * 2
for i in range(2):
    # map the gradient to the parcels
    grad[i] = map_to_labels(gm.gradients_[:, i], labeling, mask=mask, fill=np.nan)

plot_hemispheres(surf_lh, surf_rh, array_name=grad, size=(1200, 400), cmap='viridis_r
↪',
                 color_bar=True, label_text=['Grad1', 'Grad2'], zoom=1.55)
```



But which gradients should you keep for your analysis? In some cases you may have an a priori interest in some previously defined set of gradients. When you do not have a pre-defined set, you can instead look at the lambdas (eigenvalues) of each component in a scree plot. Higher eigenvalues (or lower in Laplacian eigenmaps) are more important, so one can choose a cut-off based on a scree plot.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, figsize=(5, 4))
ax.scatter(range(gm.lambdas_.size), gm.lambdas_)
ax.set_xlabel('Component Nb')
ax.set_ylabel('Eigenvalue')

plt.show()
```



This concludes the first tutorial. In the next tutorial we will have a look at how to customize the methods of gradient estimation, as well as gradient alignments.

Total running time of the script: (0 minutes 1.397 seconds)

3.1.3 Tutorial 2: Customizing and aligning gradients

In this tutorial you'll learn about the methods available within the GradientMaps class. The flexible usage of this class allows for the customization of gradient computation with different kernels and dimensionality reductions, as well as aligning gradients from different datasets. This tutorial will only show you how to apply these techniques.

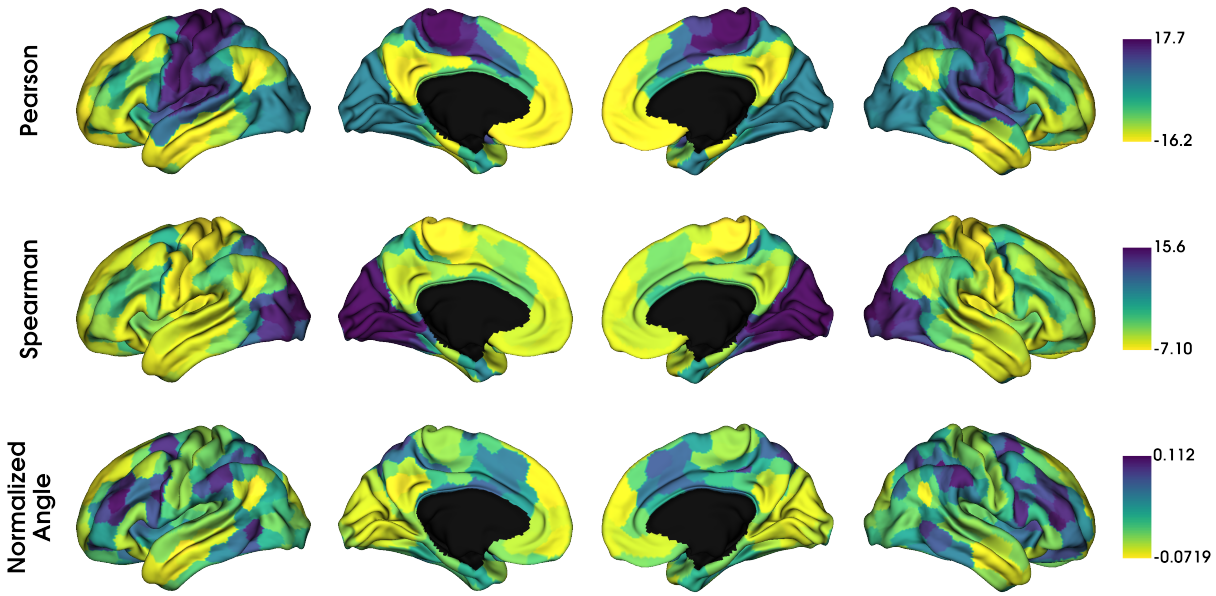
Customizing gradient computation

As before, we'll start by loading the sample data.

```
from brainspace.datasets import load_group_fc, load_parcellation, load_conte69

# First load mean connectivity matrix and Schaefer parcellation
conn_matrix = load_group_fc('schaefer', scale=400)
labeling = load_parcellation('schaefer', scale=400, join=True)

mask = labeling != 0

# and load the conte69 hemisphere surfaces
surf_lh, surf_rh = load_conte69()
```

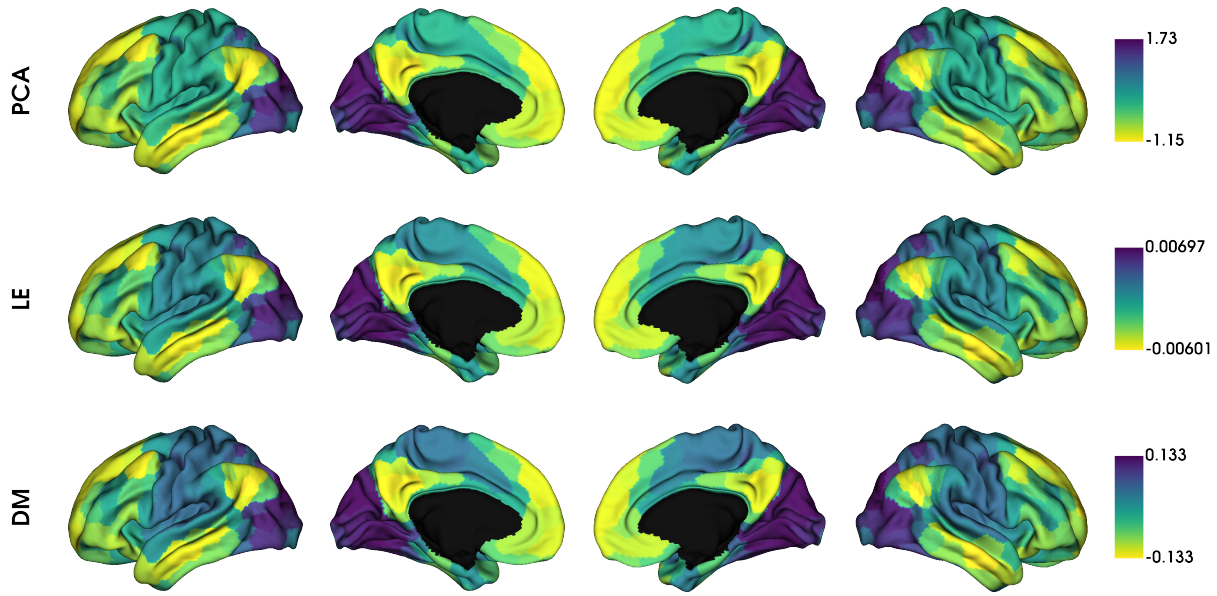
The GradientMaps object allows for many different kernels and dimensionality reduction techniques. Let's have a look at three different kernels.

```
import numpy as np

from brainspace.gradient import GradientMaps
from brainspace.plotting import plot_hemispheres
from brainspace.utils.parcellation import map_to_labels

kernels = ['pearson', 'spearman', 'normalized_angle']

gradients_kernel = [None] * len(kernels)
for i, k in enumerate(kernels):
    gm = GradientMaps(kernel=k, approach='dm', random_state=0)
    gm.fit(conn_matrix)

    gradients_kernel[i] = map_to_labels(gm.gradients_[:, i], labeling, mask=mask,
                                        fill=np.nan)


label_text = ['Pearson', 'Spearman', 'Normalized\nAngle']
plot_hemispheres(surf_lh, surf_rh, array_name=gradients_kernel, size=(1200, 600),
                 cmap='viridis_r', color_bar=True, label_text=label_text, zoom=1.45)
```



It seems the gradients provided by these kernels are quite similar although their scaling is quite different. Do note that the gradients are in arbitrary units, so the smaller/larger axes across kernels do not imply anything. Similar to using different kernels, we can also use different dimensionality reduction techniques.

```
# PCA, Laplacian eigenmaps and diffusion mapping
embeddings = ['pca', 'le', 'dm']

gradients_embedding = [None] * len(embeddings)
for i, emb in enumerate(embeddings):
    gm = GradientMaps(kernel='normalized_angle', approach=emb, random_state=0)
    gm.fit(conn_matrix)

    gradients_embedding[i] = map_to_labels(gm.gradients_[:, 0], labeling, mask=mask,
                                           fill=np.nan)


label_text = ['PCA', 'LE', 'DM']
plot_hemispheres(surf_lh, surf_rh, array_name=gradients_embedding, size=(1200, 600),
                 cmap='viridis_r', color_bar=True, label_text=label_text, zoom=1.45)
```



Gradient alignment

A more principled way of increasing comparability across gradients are alignment techniques. BrainSpace provides two alignment techniques: Procrustes analysis, and joint alignment. For this example we will load functional connectivity data of a second subject group and align it with the first group.

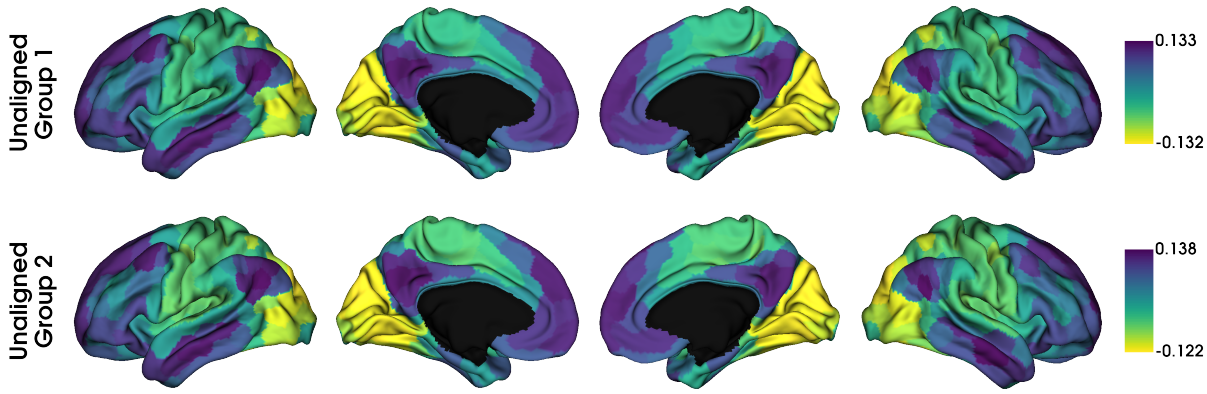
```
conn_matrix2 = load_group_fc('schaefer', scale=400, group='holdout')
gp = GradientMaps(kernel='normalized_angle', alignment='procrustes')
gj = GradientMaps(kernel='normalized_angle', alignment='joint')

gp.fit([conn_matrix, conn_matrix2])
gj.fit([conn_matrix, conn_matrix2])
```

Out:

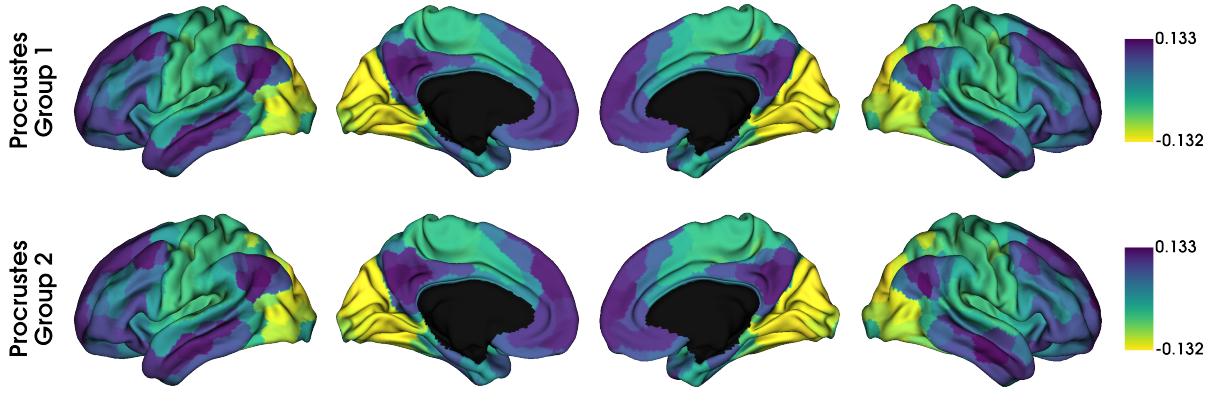
```
GradientMaps(alignment='joint', kernel='normalized_angle')
```

Here, *gp* contains the Procrustes aligned data and *gj* contains the joint aligned data. Let's plot them, but in separate figures to keep things organized.

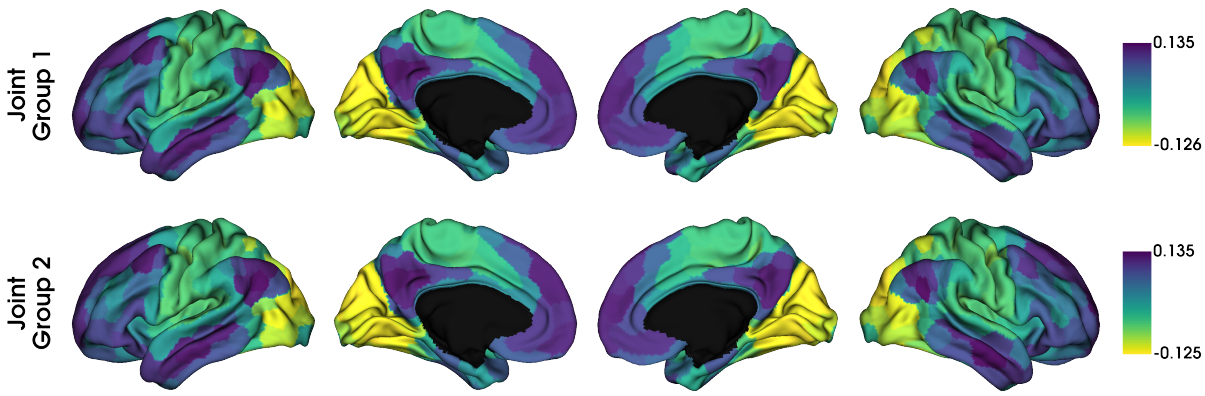
```
# First gradient from original and holdout data, without alignment
gradients_unaligned = [None] * 2
for i in range(2):
    gradients_unaligned[i] = map_to_labels(gp.gradients_[i][:, 0], labeling,
                                           mask=mask, fill=np.nan)

label_text = ['Unaligned\nGroup 1', 'Unaligned\nGroup 2']
plot_hemispheres(surf_lh, surf_rh, array_name=gradients_unaligned, size=(1200, 400),
                 cmap='viridis_r', color_bar=True, label_text=label_text, zoom=1.5)
```




```
# With procrustes alignment
gradients_procrustes = [None] * 2
for i in range(2):
    gradients_procrustes[i] = map_to_labels(gp.aligned_[i][:, 0], labeling, mask=mask,
                                            fill=np.nan)

label_text = ['Procrustes\nGroup 1', 'Procrustes\nGroup 2']
plot_hemispheres(surf_lh, surf_rh, array_name=gradients_procrustes, size=(1200, 400),
                 cmap='viridis_r', color_bar=True, label_text=label_text, zoom=1.5)
```




```
# With joint alignment
gradients_joint = [None] * 2
for i in range(2):
    gradients_joint[i] = map_to_labels(gj.aligned_[i][:, 0], labeling, mask=mask,
                                       fill=np.nan)

label_text = ['Joint\nGroup 1', 'Joint\nGroup 2']
plot_hemispheres(surf_lh, surf_rh, array_name=gradients_joint, size=(1200, 400),
                 cmap='viridis_r', color_bar=True, label_text=label_text, zoom=1.5)
```



Although in this example, we don't see any big differences, if the input data was less similar, alignments may also resolve changes in the order of the gradients. However, you should always inspect the output of an alignment; if the input data are sufficiently dissimilar then the alignment may produce odd results.

In some instances, you may want to align gradients to an out-of-sample gradient, for example when aligning individuals to a hold-out group gradient. When performing a Procrustes alignemnt, a 'reference' can be specified. The first alignment iteration will then be to the reference. For purposes of this example, we will use the gradient of the hold-out group as the reference.

```
gref = GradientMaps(kernel='normalized_angle', approach='le')
gref.fit(conn_matrix2)

galign = GradientMaps(kernel='normalized_angle', approach='le', alignment='procrustes
↪')
galign.fit(conn_matrix, reference=gref.gradients_)
```

Out:

```
GradientMaps(alignment='procrustes', approach='le', kernel='normalized_angle')
```

The gradients in *galign.aligned_* are now aligned to the reference gradients.

Gradient fusion

We can also fuse data across multiple modalities and build mutli-modal gradients. In this case we only look at one set of output gradients, rather than one per modality.

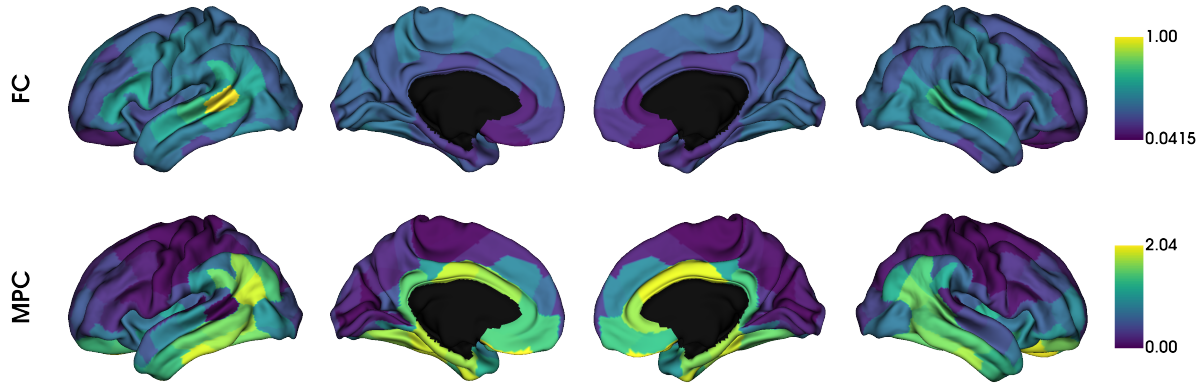
First, let's load the example data of microstructural profile covariance (Paquola et al., 2019) and functional connectivity.

```
from brainspace.datasets import load_group_mpc

# First load mean connectivity matrix and parcellation
fc = load_group_fc('vosdewael', scale=200)
mpc = load_group_mpc('vosdewael', scale=200)

labeling = load_parcellation('vosdewael', scale=200, join=True)
mask = labeling != 0

seeds = [None] * 2
seeds[0] = map_to_labels(fc[0], labeling, mask=mask, fill=np.nan)
seeds[1] = map_to_labels(mpc[0], labeling, mask=mask, fill=np.nan)
```

(continues on next page)

(continued from previous page)

```
# visualise the features from a seed region (seed 0)
plot_hemispheres(surf_lh, surf_rh, array_name=seeds, label_text=['FC', 'MPC'],
                 size=(1200, 400), color_bar=True, cmap='viridis', zoom=1.45)
```



In order to fuse the matrices, we simply pass the matrices to the fusion command which will rescale and horizontally concatenate the matrices.

```
# Negative numbers are not allowed in fusion.
fc[fc < 0] = 0


def fusion(*args):
    from scipy.stats import rankdata
    from sklearn.preprocessing import minmax_scale

    max_rk = [None] * len(args)
    masks = [None] * len(args)
    for j, a in enumerate(args):
        m = masks[j] = a != 0
        a[m] = rankdata(a[m])
        max_rk[j] = a[m].max()

    max_rk = min(max_rk)
    for j, a in enumerate(args):
        m = masks[j]
        a[m] = minmax_scale(a[m], feature_range=(1, max_rk))

    return np.hstack(args)


# fuse the matrices
fused_matrix = fusion(fc, mpc)
```

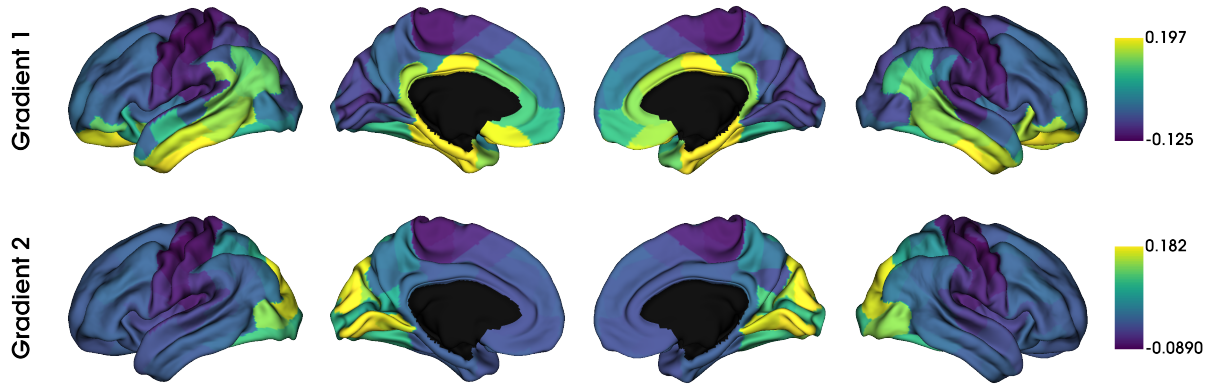
We then use this output in the fit function. This will convert the long horizontal array into a square affinity matrix, and then perform embedding.

```
gm = GradientMaps(n_components=2, kernel='normalized_angle')
gm.fit(fused_matrix)


gradients_fused = [None] * 2
```

(continues on next page)

(continued from previous page)

```
for i in range(2):
    gradients_fused[i] = map_to_labels(gm.gradients_[:, i], labeling, mask=mask,
                                       fill=np.nan)

plot_hemispheres(surf_lh, surf_rh, array_name=gradients_fused,
                 label_text=['Gradient 1', 'Gradient 2'], size=(1200, 400),
                 color_bar=True, cmap='viridis', zoom=1.45)
```



Note: The mpc matrix presented here matches the subject cohort of (Paquola et al., 2019). Other matrices in this package match the subject groups used by (Vos de Wael et al., 2018). We make direct comparisons in our tutorial for didactic purposes only.

That concludes the second tutorial. In the third tutorial we will consider null hypothesis testing of comparisons between gradients and other markers.

Total running time of the script: (0 minutes 6.461 seconds)

3.1.4 Tutorial 3: Null models for gradient significance

In this tutorial we assess the significance of correlations between the first canonical gradient and data from other modalities (curvature, cortical thickness and T1w/T2w image intensity). A normal test of the significance of the correlation cannot be used, because the spatial auto-correlation in MRI data may bias the test statistic. In this tutorial we will show three approaches for null hypothesis testing: spin permutations, Moran spectral randomization, and autocorrelation-preserving surrogates based on variogram matching.

Note: When using either approach to compare gradients to non-gradient markers, we recommend randomizing the non-gradient markers as these randomizations need not maintain the statistical independence between gradients.

Spin Permutations

Here, we use the spin permutations approach previously proposed in (Alexander-Bloch et al., 2018), which preserves the auto-correlation of the permuted feature(s) by rotating the feature data on the spherical domain. We will start by loading the conte69 surfaces for left and right hemispheres, their corresponding spheres, midline mask, and t1w/t2w intensity as well as cortical thickness data, and a template functional gradient.

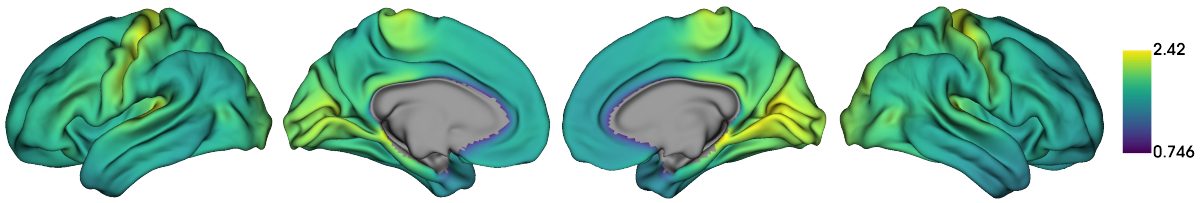
```
import numpy as np
from brainspace.datasets import load_gradient, load_marker, load_conte69

# load the conte69 hemisphere surfaces and spheres
surf_lh, surf_rh = load_conte69()
sphere_lh, sphere_rh = load_conte69(as_sphere=True)

# Load the data
t1wt2w_lh, t1wt2w_rh = load_marker('t1wt2w')
t1wt2w = np.concatenate([t1wt2w_lh, t1wt2w_rh])

thickness_lh, thickness_rh = load_marker('thickness')
thickness = np.concatenate([thickness_lh, thickness_rh])

# Template functional gradient
embedding = load_gradient('fc', idx=0, join=True)
```

Let's first generate some null data using spintest.

```
from brainspace.null_models import SpinPermutations
from brainspace.plotting import plot_hemispheres

# Let's create some rotations
n_rand = 1000

sp = SpinPermutations(n_rep=n_rand, random_state=0)
sp.fit(sphere_lh, points_rh=sphere_rh)

t1wt2w_rotated = np.hstack(sp.randomize(t1wt2w_lh, t1wt2w_rh))
thickness_rotated = np.hstack(sp.randomize(thickness_lh, thickness_rh))
```

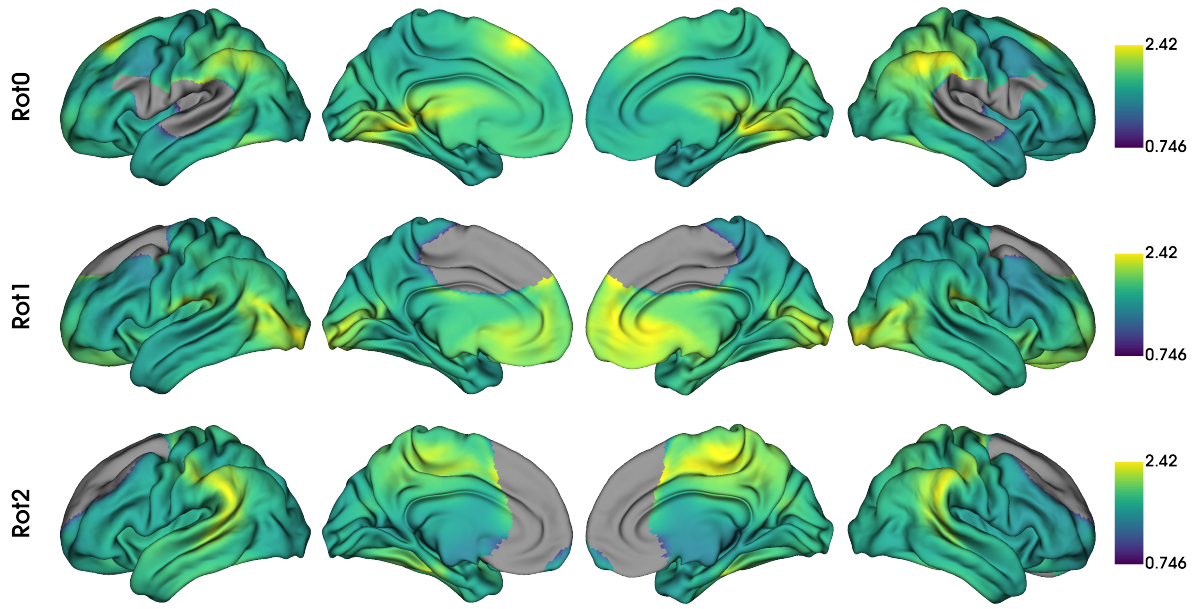
As an illustration of the rotation, let's plot the original t1w/t2w data

```
# Plot original data
plot_hemispheres(surf_lh, surf_rh, array_name=t1wt2w, size=(1200, 200), cmap='viridis
↪',
                 nan_color=(0.5, 0.5, 0.5, 1), color_bar=True, zoom=1.65)
```



as well as a few rotated versions.

```
# Plot some rotations
plot_hemispheres(surf_lh, surf_rh, array_name=t1wt2w_rotated[:3], size=(1200, 600),
                 cmap='viridis', nan_color=(0.5, 0.5, 0.5, 1), color_bar=True,
                 zoom=1.55, label_text=['Rot0', 'Rot1', 'Rot2'])
```



Warning: With spin permutations, midline vertices (i.e,, NaNs) from both the original and rotated data are discarded. Depending on the overlap of midlines in the, statistical comparisons between them may compare different numbers of features. This can bias your test statistics. Therefore, if a large portion of the sphere is not used, we recommend using Moran spectral randomization instead.

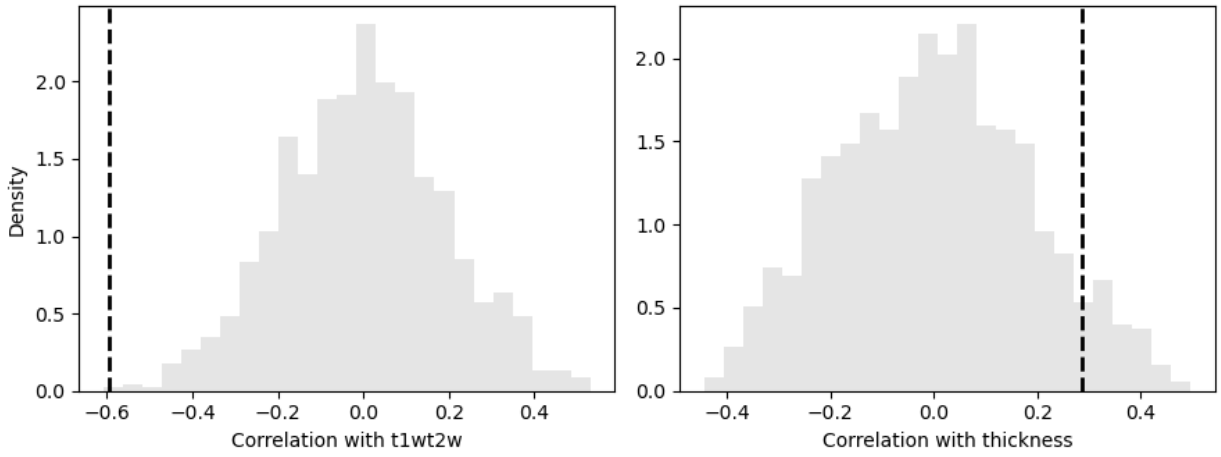
Now we simply compute the correlations between the first gradient and the original data, as well as all rotated data.

```
from matplotlib import pyplot as plt
from scipy.stats import spearmanr

fig, axs = plt.subplots(1, 2, figsize=(9, 3.5))

feats = {'t1wt2w': t1wt2w, 'thickness': thickness}
rotated = {'t1wt2w': t1wt2w_rotated, 'thickness': thickness_rotated}

r_spin = np.empty(n_rand)
mask = ~np.isnan(thickness)
for k, (fn, feat) in enumerate(feats.items()):
    r_obs, pv_obs = spearmanr(feat[mask], embedding[mask])

    # Compute perm pval
    for i, perm in enumerate(rotated[fn]):
        mask_rot = mask & ~np.isnan(perm)  # Remove midline
        r_spin[i] = spearmanr(perm[mask_rot], embedding[mask_rot])[0]
    pv_spin = np.mean(np.abs(r_spin) >= np.abs(r_obs))

    # Plot null dist
    axs[k].hist(r_spin, bins=25, density=True, alpha=0.5, color=(.8, .8, .8))
    axs[k].axvline(r_obs, lw=2, ls='--', color='k')
    axs[k].set_xlabel(f'Correlation with {fn}')
    if k == 0:
        axs[k].set_ylabel('Density')
```

(continues on next page)

(continued from previous page)

```
    print(f'{fn.capitalize()}:\n Obs : {pv_obs:.5e}\n Spin: {pv_spin:.5e}\n')

fig.tight_layout()
plt.show()
```



Out:

```
T1wt2w:
 Obs : 0.00000e+00
 Spin: 1.00000e-03

Thickness:
 Obs : 0.00000e+00
 Spin: 1.37000e-01
```

It is interesting to see that both p-values increase when taking into consideration the auto-correlation present in the surfaces. Also, we can see that the correlation with thickness is no longer statistically significant after spin permutations.

Moran Spectral Randomization

Moran Spectral Randomization (MSR) computes Moran's I, a metric for spatial auto-correlation and generates normally distributed data with similar auto-correlation. MSR relies on a weight matrix denoting the spatial proximity of features to one another. Within neuroimaging, one straightforward example of this is inverse geodesic distance i.e. distance along the cortical surface.

In this example we will show how to use MSR to assess statistical significance between cortical markers (here curvature and cortical t1wt2w intensity) and the first functional connectivity gradient. We will start by loading the left temporal lobe mask, t1w/t2w intensity as well as cortical thickness data, and a template functional gradient

```
from brainspace.datasets import load_mask

n_pts_lh = surf_lh.n_points
mask_tl, _ = load_mask(name='temporal')

# Keep only the temporal lobe.
embedding_tl = embedding[:n_pts_lh][mask_tl]
```

(continues on next page)

(continued from previous page)

```
t1wt2w_tl = t1wt2w_lh[mask_tl]
curv_tl = load_marker('curvature')[0][mask_tl]
```

We will now compute the Moran eigenvectors. This can be done either by providing a weight matrix of spatial proximity between each vertex, or by providing a cortical surface. Here we'll use a cortical surface.

```
from brainspace.null_models import MoranRandomization
from brainspace.mesh import mesh_elements as me

# compute spatial weight matrix
w = me.get_ring_distance(surf_lh, n_ring=1, mask=mask_tl)
w.data **= -1


msr = MoranRandomization(n_rep=n_rand, procedure='singleton', tol=1e-6,
                         random_state=0)
msr.fit(w)
```

Out:

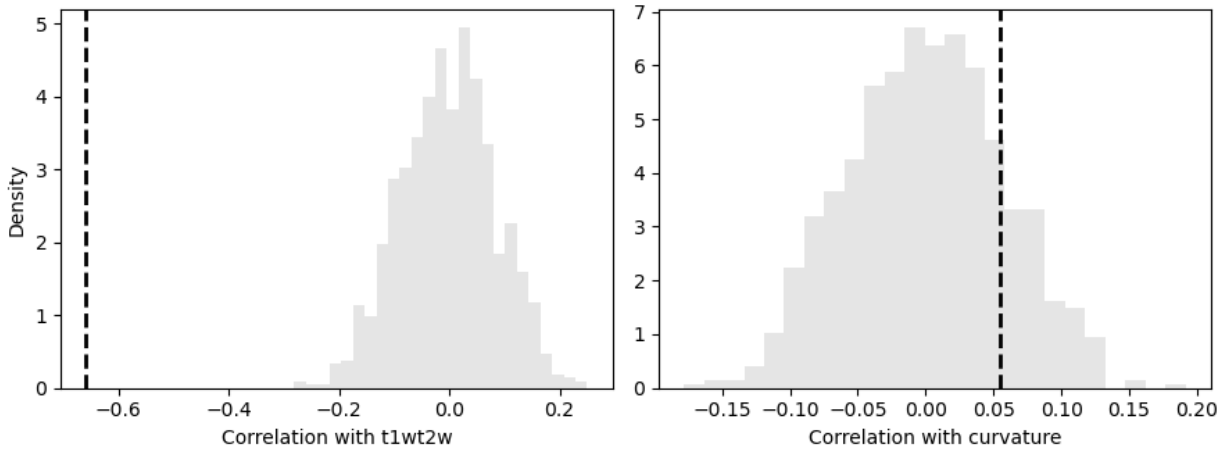
```
MoranRandomization(n_rep=1000, random_state=0, tol=1e-06)
```

Using the Moran eigenvectors we can now compute the randomized data.

```
curv_rand = msr.randomize(curv_tl)
t1wt2w_rand = msr.randomize(t1wt2w_tl)
```

Now that we have the randomized data, we can compute correlations between the gradient and the real/randomised data and generate the non-parametric p-values.

```
fig, axs = plt.subplots(1, 2, figsize=(9, 3.5))

feats = {'t1wt2w': t1wt2w_tl, 'curvature': curv_tl}
rand = {'t1wt2w': t1wt2w_rand, 'curvature': curv_rand}

for k, (fn, data) in enumerate(rand.items()):
    r_obs, pv_obs = spearmanr(feats[fn], embedding_tl, nan_policy='omit')

    # Compute perm pval
    r_rand = np.asarray([spearmanr(embedding_tl, d)[0] for d in data])
    pv_rand = np.mean(np.abs(r_rand) >= np.abs(r_obs))

    # Plot null dist
    axs[k].hist(r_rand, bins=25, density=True, alpha=0.5, color=(.8, .8, .8))
    axs[k].axvline(r_obs, lw=2, ls='--', color='k')
    axs[k].set_xlabel(f'Correlation with {fn}')
    if k == 0:
        axs[k].set_ylabel('Density')

    print(f'{fn.capitalize()}:\n Obs  : {pv_obs:.5e}\n Moran: {pv_rand:.5e}\n')

fig.tight_layout()
plt.show()
```



Out:

```
T1wt2w:
 Obs  : 0.00000e+00
 Moran: 0.00000e+00

Curvature:
 Obs  : 6.63802e-05
 Moran: 3.50000e-01
```

Variogram Matching

Here, we will repeat the same analysis using the variogram matching approach presented in (Burt et al., 2020), which generates novel brainmaps with similar spatial autocorrelation to the input data.

We will need a distance matrix that tells us what the spatial distance between our datapoints is. For this example, we will use geodesic distance.

```
from brainspace.mesh.mesh_elements import get_immediate_distance
from scipy.sparse.csgraph import dijkstra

# Compute geodesic distance
gd = get_immediate_distance(surf_lh, mask=mask_tl)
gd = dijkstra(gd, directed=False)

idx_sorted = np.argsort(gd, axis=1)
```

Now we've got everything we need to generate our surrogate datasets. By default, BrainSpace will use all available data to generate surrogate maps. However, this process is extremely computationally and memory intensive. When using this method with more than a few hundred regions, we recommend subsampling the data. This can be done using SampledSurrogateMaps instead of the SurrogateMaps.

```
from brainspace.null_models import SampledSurrogateMaps


n_surrogate_datasets = 1000

# Note: number samples must be greater than number neighbors
num_samples = 100
```

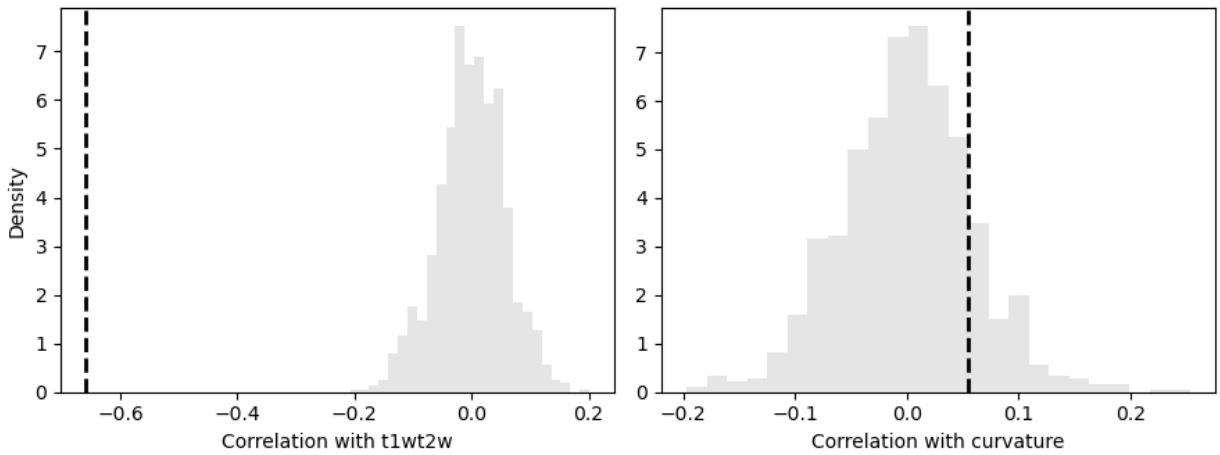
(continues on next page)

(continued from previous page)

```
num_neighbors = 50

ssm = SampledSurrogateMaps(ns=num_samples, knn=num_neighbors, random_state=0)
ssm.fit(gd, idx_sorted)

t1wt2w_surrogates = ssm.randomize(t1wt2w_tl, n_rep=n_surrogate_datasets)
curv_surrogates = ssm.randomize(curv_tl, n_rep=n_surrogate_datasets)
```

Similar to the previous case, we can now plot the results:

```
import matplotlib.pyplot as plt
from scipy.stats import spearmanr
fig, axs = plt.subplots(1, 2, figsize=(9, 3.5))

feats = {'t1wt2w': t1wt2w_tl, 'curvature': curv_tl}
rand = {'t1wt2w': t1wt2w_surrogates, 'curvature': curv_surrogates}

for k, (fn, data) in enumerate(rand.items()):
    r_obs, pv_obs = spearmanr(feats[fn], embedding_tl, nan_policy='omit')

    # Compute perm pval
    r_rand = np.asarray([spearmanr(embedding_tl, d)[0] for d in data])
    pv_rand = np.mean(np.abs(r_rand) >= np.abs(r_obs))

    # Plot null dist
    axs[k].hist(r_rand, bins=25, density=True, alpha=0.5, color=(.8, .8, .8))
    axs[k].axvline(r_obs, lw=2, ls='--', color='k')
    axs[k].set_xlabel(f'Correlation with {fn}')
    if k == 0:
        axs[k].set_ylabel('Density')

    print(f'{fn.capitalize()}:\n Obs      : {pv_obs:.5e}\n '
          f'Variogram: {pv_rand:.5e}\n')

fig.tight_layout()
plt.show()
```



Out:

```
T1wt2w:
 Obs      : 0.00000e+00
 Variogram: 0.00000e+00

Curvature:
 Obs      : 6.63802e-05
 Variogram: 3.26000e-01
```

Total running time of the script: (3 minutes 29.601 seconds)

3.2 API Reference

3.2.1 Gradient Maps

- *Gradients*
- *Embedding*
- *Alignment*
- *Kernels*
- *Utility functions*

Gradients

`GradientMaps`([n_components, approach, ...])	Gradient maps.

brainspace.gradient.gradient.GradientMaps

class brainspace.gradient.gradient.**GradientMaps**(*n_components=10*, *approach='dm'*, *kernel=None*, *alignment=None*, *random_state=None*)

Gradient maps.

Parameters

- **n_components** (*int, optional*) – Number of gradients. Default is 10.
- **approach** (*{'dm', 'le', 'pca'} or object, optional*) – Embedding approach. Default is 'dm'. It can be a string or instance:
  - 'dm' or `DiffusionMaps`: embedding using diffusion maps.
  - 'le' or `LaplacianEigenmaps`: embedding using Laplacian eigenmaps.
  - 'pca' or `PCAMaps`: embedding using PCA.
- **kernel** (*str, callable or None, optional*) – Kernel function to build the affinity matrix. Possible options: {'pearson', 'spearman', 'cosine', 'normalized_angle', 'gaussian'}. If callable, must receive a 2D array and return a 2D square array. If None, use input matrix. Default is None.
- **alignment** (*{'procrustes', 'joint'}, object or None*) – Alignment approach. Only used when two or more datasets are provided. If None, no alignment is performed. If *object*, it accepts an instance of `ProcrustesAlignment`. Default is None.

- If 'procrustes', datasets are aligned using generalized procrustes analysis.
- If 'joint', datasets are embedded simultaneously based on a joint affinity matrix built from the individual datasets. This option is only available for 'dm' and 'le' approaches.

- **random_state** (*int or None, optional*) – Random state. Default is None.

Variables

- **lambdas** (*ndarray or list of arrays, shape = (n_components,)*) – Eigenvalues for each datatset.
- **gradients** (*ndarray or list of arrays, shape = (n_samples, n_components)*) – Gradients (i.e., eigenvectors).
- **aligned** (*None or list of arrays, shape = (n_samples, n_components)*) – Aligned gradients. None if `alignment is None` or only one dataset is used.

__init__ (*n_components=10*, *approach='dm'*, *kernel=None*, *alignment=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components, approach, kernel, ...])	Initialize self.
fit(x[, gamma, sparsity, n_iter, reference])	Compute gradients and alignment.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit (*x*, *gamma=None*, *sparsity=0.9*, *n_iter=10*, *reference=None*, ***kwargs*)
Compute gradients and alignment.

Parameters

- **x** (*ndarray or list of arrays, shape = (n_samples, n_feat)*) – Input matrix or list of matrices.
- **gamma** (*float or None, optional*) – Inverse kernel width. Only used if `kernel == 'gaussian'`. If None, `gamma=1/n_feat`. Default is None.
- **sparsity** (*float, optional*) – Proportion of the smallest elements to zero-out for each row. Default is 0.9.
- **n_iter** (*int, optional*) – Number of iterations for procrustes alignment. Default is 10.
- **reference** (*ndarray, shape = (n_samples, n_feat), optional*) – Initial reference for procrustes alignments. Only used when `alignment == 'procrustes'`. Default is None.
- **kwargs** (*kwds, optional*) – Additional keyword parameters passed to the embedding approach.

Returns **self** (*object*) – Returns self.

Embedding

Embedding([n_components])	Base class for embedding approaches.
DiffusionMaps([n_components, alpha, ...])	Diffusion maps.
LaplacianEigenmaps([n_components, ...])	Laplacian eigenmaps.
PCAMaps([n_components, random_state])	Principal component analysis.
diffusion_mapping(adj[, n_components, ...])	Compute diffusion map of affinity matrix.
laplacian_eigenmaps(adj[, n_components, ...])	Compute embedding using Laplacian eigenmaps.

brainspace.gradient.embedding.Embedding

class brainspace.gradient.embedding.**Embedding**(*n_components=2*)

Base class for embedding approaches.

Defines fit_transform method.

__init__(*n_components=2*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components])	Initialize self.
fit(x)	
fit_transform(x)	Compute embedding for *x*.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit(*x*)

fit_transform(*x*)

Compute embedding for *x*.

Parameters **x** (*ndarray, shape = (n, n)*) – Input matrix.

Returns **embedding** (*ndarray, shape(n, n_components)*) – Embedded data.

brainspace.gradient.embedding.DiffusionMaps

class brainspace.gradient.embedding.**DiffusionMaps**(*n_components=10, alpha=0.5, diffusion_time=0, random_state=None*)

Diffusion maps.

Parameters

- **n_components** (*int or None, optional*) – Number of eigenvectors. Default is 10.
- **alpha** (*float, optional*) – Anisotropic diffusion parameter, `0 <= alpha <= 1`. Default is 0.5.
- **diffusion_time** (*int, optional*) – Diffusion time or scale. If `diffusion_time == 0` use multi-scale diffusion maps. Default is 0.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Variables

- **lambdas** (*1D ndarray, shape (n_components,)*) – Eigenvalues of the affinity matrix in descending order.
- **maps** (*2D ndarray, shape (n, n_components)*) – Eigenvectors of the affinity matrix in same order. Where *n* is the number of rows of the affinity matrix.

See also:

LaplacianEigenmaps, *PCAMaps*

References

- Coifman, R.R.; S. Lafon. (2006). "Diffusion maps". Applied and Computational Harmonic Analysis 21: 5-30. doi:10.1016/j.acha.2006.04.006
- Joseph W.R., Peter E.F., Ann B.L., Chad M.S. Accurate parameter estimation for star formation history in galaxies using SDSS spectra.

__init__ (*n_components=10*, *alpha=0.5*, *diffusion_time=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components, alpha, ...])	Initialize self.
fit(affinity)	Compute the diffusion maps.
fit_transform(x)	Compute embedding for *x*.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit (*affinity*)
Compute the diffusion maps.

Parameters **affinity** (*ndarray or sparse matrix, shape = (n, n)*) – Affinity matrix.

Returns **self** (*object*) – Returns self.

fit_transform (*x*)
Compute embedding for *x*.

Parameters **x** (*ndarray, shape = (n, n)*) – Input matrix.

Returns **embedding** (*ndarray, shape(n, n_components)*) – Embedded data.

brainspace.gradient.embedding.LaplacianEigenmaps

class brainspace.gradient.embedding.**LaplacianEigenmaps** (*n_components=10*, *norm_laplacian=True*, *random_state=None*)

Laplacian eigenmaps.

Parameters

- **n_components** (*int or None, optional*) – Number of eigenvectors. Default is 10.

- **norm_laplacian** (*bool, optional*) – If True, use normalized Laplacian. Default is True.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Variables

- **lambdas** (*ndarray, shape (n_components,)*) – Eigenvalues of the affinity matrix in ascending order.
- **maps** (*ndarray, shape (n, n_components)*) – Eigenvectors of the affinity matrix in same order. Where *n* is the number of rows of the affinity matrix.

See also:

DiffusionMaps, *PCAMaps*

__init__(*n_components=10*, *norm_laplacian=True*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components, norm_laplacian, ...])	Initialize self.
fit(affinity)	Compute the Laplacian maps.
fit_transform(x)	Compute embedding for *x*.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit(*affinity*)
Compute the Laplacian maps.

Parameters **affinity** (*ndarray or sparse matrix, shape = (n, n)*) – Affinity matrix.

Returns **self** (*object*) – Returns self.

fit_transform(*x*)
Compute embedding for *x*.

Parameters **x** (*ndarray, shape = (n, n)*) – Input matrix.

Returns **embedding** (*ndarray, shape(n, n_components)*) – Embedded data.

brainspace.gradient.embedding.PCAMaps

class brainspace.gradient.embedding.**PCAMaps**(*n_components=10*, *random_state=None*)
Principal component analysis.

Parameters

- **n_components** (*int or None, optional*) – Number of principal components. Default is 10.
- **random_state** (*int, RandomState instance or None, optional*) – Random state. Default is None.

Variables

- **lambdas** (*ndarray, shape (n_components,)*) – Explained variance for first principal components in descending order.

- **maps** (*ndarray, shape (n_samples, n_components)*) – Projection of input data onto the principal components.

See also:

DiffusionMaps, *LaplacianEigenmaps*

__init__ (*n_components=10*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components, random_state])	Initialize self.
fit(x)	Compute PCA.
fit_transform(x)	Compute embedding for *x*.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit (*x*)
Compute PCA.

Parameters **x** (*ndarray, shape(n_samples, n_feat)*) – Input matrix.

Returns **self** (*object*) – Returns self.

fit_transform (*x*)
Compute embedding for *x*.

Parameters **x** (*ndarray, shape = (n, n)*) – Input matrix.

Returns **embedding** (*ndarray, shape(n, n_components)*) – Embedded data.

brainspace.gradient.embedding.diffusion_mapping

brainspace.gradient.embedding.**diffusion_mapping**(*adj*, *n_components=10*, *alpha=0.5*, *diffusion_time=0*, *random_state=None*)

Compute diffusion map of affinity matrix.

Parameters

- **adj** (*ndarray or sparse matrix, shape = (n, n)*) – Affinity matrix.
- **n_components** (*int or None, optional*) – Number of eigenvectors. If None, selection of *n_components* is based on 95% drop-off in eigenvalues. When *n_components* is None, the maximum number of eigenvectors is restricted to `n_components <= sqrt(n)`. Default is 10.
- **alpha** (*float, optional*) – Anisotropic diffusion parameter, `0 <= alpha <= 1`. Default is 0.5.
- **diffusion_time** (*int, optional*) – Diffusion time or scale. If `diffusion_time == 0` use multi-scale diffusion maps. Default is 0.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Returns

- **v** (*ndarray, shape (n, n_components)*) – Eigenvectors of the affinity matrix in same order.

- **w** (*ndarray, shape (n_components,)*) – Eigenvalues of the affinity matrix in descending order.

References

- Coifman, R.R.; S. Lafon. (2006). "Diffusion maps". Applied and Computational Harmonic Analysis 21: 5-30. doi:10.1016/j.acha.2006.04.006
- Joseph W.R., Peter E.F., Ann B.L., Chad M.S. Accurate parameter estimation for star formation history in galaxies using SDSS spectra.

brainspace.gradient.embedding.laplacian_eigenmaps

`brainspace.gradient.embedding.`**`laplacian_eigenmaps`**(*adj*, *n_components=10*, *norm_laplacian=True*, *random_state=None*)

Compute embedding using Laplacian eigenmaps.

Adapted from Scikit-learn to also provide eigenvalues.

Parameters

- **adj** (*2D ndarray or sparse matrix*) – Affinity matrix.
- **n_components** (*int, optional*) – Number of eigenvectors. Default is 10.
- **norm_laplacian** (*bool, optional*) – If True use normalized Laplacian. Default is True.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Returns

- **v** (*2D ndarray, shape (n, n_components)*) – Eigenvectors of the affinity matrix in same order. Where *n* is the number of rows of the affinity matrix.
- **w** (*1D ndarray, shape (n_components,)*) – Eigenvalues of the affinity matrix in ascending order.

References

- Belkin, M. and Niyogi, P. (2003). Laplacian Eigenmaps for dimensionality reduction and data representation. Neural Computation 15(6): 1373-96. doi:10.1162/089976603321780317

Alignment

ProcrustesAlignment([n_iter, tol, verbose])	Iterative alignment using generalized procrustes analysis.
procrustes_alignment(data[, reference, ...])	Iterative alignment using generalized procrustes analysis.
procrustes(source, target[, center, scale])	Align *source* to *target* using procrustes analysis.

brainspace.gradient.alignment.ProcrustesAlignment

class brainspace.gradient.alignment.**ProcrustesAlignment**(*n_iter=10*, *tol=1e-05*, *verbose=False*)

Iterative alignment using generalized procrustes analysis.

Parameters

- **n_iter** (*int, optional*) – Number of iterations. Default is 10.
- **tol** (*float, optional*) – Tolerance for stopping criteria. Default is 1e-5.
- **verbose** (*bool, optional*) – Verbosity. Default is False.

Variables

- **aligned** (*list of ndarray, shape = (n_samples, n_feat)*) – Aligned datsets.
- **mean** (*ndarray, shape = (n_samples, n_feat)*) – Reference dataset built in the last iteration.

__init__(*n_iter=10*, *tol=1e-05*, *verbose=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_iter, tol, verbose])	Initialize self.
fit(data[, reference])	Align data.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit(*data*, *reference=None*)

Align data.

Parameters

- **data** (*list of ndarrays, shape = (n_samples, n_feat)*) – List of datasets to align.
- **reference** (*ndarray, shape = (n_samples, n_feat), optional*) – Dataset to use as reference in the first iteration. If None, the first dataset in *data* is used as reference. Default is None.

Returns **self** (*object*) – Returns self.

brainspace.gradient.alignment.procrustes_alignment

brainspace.gradient.alignment.**procrustes_alignment**(*data*, *reference=None*, *n_iter=10*, *tol=1e-05*, *return_reference=False*, *verbose=False*)

Iterative alignment using generalized procrustes analysis.

Parameters

- **data** (*list of ndarray, shape = (n_samples, n_feat)*) – List of datasets to align.

- **reference** (*ndarray, shape = (n_samples, n_feat), optional*) – Dataset to use as reference in the first iteration. If None, the first dataset in *data* is used as reference. Default is None.
- **n_iter** (*int, optional*) – Number of iterations. Default is 10.
- **tol** (*float, optional*) – Tolerance for stopping criteria. Default is 1e-5.
- **return_reference** (*bool, optional*) – Whether to return the reference dataset built in the last iteration. Default is False.
- **verbose** (*bool, optional*) – Verbosity. Default is False.

Returns

- **aligned** (*list of ndarray, shape = (n_samples, n_feat)*) – Aligned datsets.
- **mean_dataset** (*ndarray, shape = (n_samples, n_feat)*) – Reference dataset built in the last iteration. Only if `return_reference == True`.

brainspace.gradient.alignment.procrustes

`brainspace.gradient.alignment.`**`procrustes`**(*source*, *target*, *center=False*, *scale=False*)

Align *source* to *target* using procrustes analysis.

Parameters

- **source** (*2D ndarray, shape = (n_samples, n_feat)*) – Source dataset.
- **target** (*2D ndarray, shape = (n_samples, n_feat)*) – Target dataset.
- **center** (*bool, optional*) – Center data before alignment. Default is False.
- **scale** (*bool, optional*) – Remove scale before alignment. Default is False.

Returns **aligned** (*2D ndarray, shape = (n_samples, n_feat)*) – Source dataset aligned to target dataset.

Kernels

compute_affinity(x[, kernel, sparsity, ...])	Compute affinity matrix.

brainspace.gradient.kernels.compute_affinity

`brainspace.gradient.kernels.`**`compute_affinity`**(*x*, *kernel=None*, *sparsity=0.9*, *pre_sparsify=True*, *non_negative=True*, *gamma=None*)

Compute affinity matrix.

Parameters

- **x** (*ndarray, shape = (n_samples, n_feat)*) – Input matrix.
- **kernel** (*str, None or callable, optional*) – Kernel function. If None, only sparsify. Default is None. Valid options:
  - If 'pearson', use Pearson's correlation coefficient.
  - If 'spearman', use Spearman's rank correlation coefficient.

  - If 'cosine', use cosine similarity.
  - If 'normalized_angle': use normalized angle between two vectors. This option is based on cosine similarity but provides similarities bounded between 0 and 1.
  - If 'gaussian', use Gaussian kernel or RBF.
- **sparsity** (*float or None, optional*) – Proportion of smallest elements to zero-out for each row. If None, do not sparsify. Default is 0.9.
- **pre_sparsify** (*bool, optional*) – Sparsify prior to building the affinity. If False, sparsify the final affinity matrix.
- **non_negative** (*bool, optional*) – If True, zero-out negative values. Otherwise, do nothing.
- **gamma** (*float or None, optional*) – Inverse kernel width. Only used if `kernel == 'gaussian'`. If None, `gamma = 1./n_feat`. Default is None.

Returns **affinity** (*ndarray, shape = (n_samples, n_samples)*) – Affinity matrix.

Utility functions

dominant_set(s, k[, is_thresh, norm, copy, ...])	Keep the largest elements for each row.
is_symmetric(x[, tol])	Check if input is symmetric.
make_symmetric(x[, check, tol, copy, ...])	Make array symmetric.

brainspace.gradient.utils.dominant_set

brainspace.gradient.utils.**dominant_set**(*s*, *k*, *is_thresh=False*, *norm=False*, *copy=True*, *as_sparse=True*)

Keep the largest elements for each row. Zero-out the rest.

Parameters

- **s** (*2D ndarray*) – Similarity/affinity matrix.
- **k** (*int or float*) – If int, keep top *k* elements for each row. If float, keep top *100*k* percent of elements. When float, must be in range (0, 1).
- **is_thresh** (*bool, optional*) – If True, *k* is used as threshold. Keep elements greater than *k*. Default is False.
- **norm** (*bool, optional*) – If True, normalize rows. Default is False.
- **copy** (*bool, optional*) – If True, make a copy of the input array. Otherwise, work on original array. Default is True.
- **as_sparse** (*bool, optional*) – If True, return a sparse matrix. Otherwise, return the same type of the input array. Default is True.

Returns **output** (*2D ndarray or sparse matrix*) – Dominant set.

brainspace.gradient.utils.is_symmetric

brainspace.gradient.utils.**is_symmetric**(*x*, *tol=1e-10*)

Check if input is symmetric.

Parameters

- **x** (*2D ndarray or sparse matrix*) – Input data.
- **tol** (*float, optional*) – Maximum allowed tolerance for equivalence. Default is 1e-10.

Returns **is_symm** (*bool*) – True if *x* is symmetric. False, otherwise.

Raises ValueError – If *x* is not square.

brainspace.gradient.utils.make_symmetric

brainspace.gradient.utils.**make_symmetric**(*x*, *check=True*, *tol=1e-10*, *copy=True*, *sparse_format=None*)

Make array symmetric.

Parameters

- **x** (*2D ndarray or sparse matrix*) – Input data.
- **check** (*bool, optional*) – If True, check if already symmetry first. Default is True.
- **tol** (*float, optional*) – Maximum allowed tolerance for equivalence. Default is 1e-10.
- **copy** (*bool, optional*) – If True, return a copy. Otherwise, work on *x*. If already symmetric, returns original array.
- **sparse_format** (*{'coo', 'csr', 'csc', ..}, optional*) – Format of output symmetric matrix. Only used if *x* is sparse. Default is None, uses original format.

Returns **sym** (*2D ndarray or sparse matrix.*) – Symmetrized version of *x*. Return *x* it is already symmetric.

Raises ValueError – If *x* is not square.

3.2.2 Null models

- *Spin permutations*
- *Moran spectral randomization*
- *Surrogate maps*

Spin permutations

SpinPermutations([unique, n_rep, ...])	Spin permutations.
spin_permutations(spheres, data[, unique, ...])	Generate null data using spin permutations.

brainspace.null_models.spin.SpinPermutations

class brainspace.null_models.spin.**SpinPermutations**(*unique=False*, *n_rep=100*, *random_state=None*, *surface_algorithm='FreeSurfer'*)

Spin permutations.

Parameters

- **unique** (*bool, optional*) – Whether to enforce a one-to-one correspondence between

original points and rotated ones. If True, the Hungarian algorithm is used. Default is False.

- **n_rep** (*int, optional*) – Number of randomizations. Default is 100.
- **random_state** (*int or None, optional*) – Random state. Default is None.
- **surface_algorithm** (*{'FreeSurfer', 'CIVET'}*) – For 'CIVET', no flip is required to generate the spins for the right hemisphere. Only used when `points_rh is not None`. Default is 'FreeSurfer'.

Variables

- **spin_lh** (*ndarray, shape (n_rep, n_lh)*) – Spin indices for points in left hemisphere.
- **spin_rh** (*ndarray, shape (n_rep, n_rh)*) – Spin indices for points in right hemisphere. Only if user provides right hemisphere points. None, otherwise.

See also:

spin_permutations, *MoranRandomization*

Notes

Right hemisphere permutations are generated by reflecting the rotation matrix used for the left hemisphere.

__init__ (*unique=False*, *n_rep=100*, *random_state=None*, *surface_algorithm='FreeSurfer'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([unique, n_rep, random_state, ...])	Initialize self.
fit(points_lh[, points_rh])	Compute spin indices by random rotation.
get_params([deep])	Get parameters for this estimator.
randomize(x_lh[, x_rh])	Generate random samples from *x_lh* and *x_rh*.
randomize_gen(x_lh[, x_rh])	Generate random samples from *x_lh* and *x_rh*.
set_params(**params)	Set the parameters of this estimator.

fit (*points_lh*, *points_rh=None*)
Compute spin indices by random rotation.

Parameters

- **points_lh** (*BSPolyData or ndarray, shape = (n_lh, 3)*) – Sphere for the left hemisphere. If ndarray, each row must represent a vertex in the sphere.
- **points_rh** (*BSPolyData or ndarray, shape = (n_rh, 3), optional*) – Sphere for the right hemisphere. If ndarray, row must represent a vertex in the sphere. Default is None.

Returns **self** (*object*) – Returns self.

randomize (*x_lh*, *x_rh=None*)
Generate random samples from *x_lh* and *x_rh*.

Parameters

- **x_lh** (*ndarray, shape = (n_lh,) or (n_lh, n_feat)*) – Array of variables arranged in columns, where *n_feat* is the number of variables.

- **x_rh** (*ndarray, shape = (n_rh,) or (n_rh, n_feat), optional*) – Array of variables arranged in columns for the right hemisphere. Default is None.

Returns

- **rand_lh** (*ndarray, shape = (n_rep, n_lh, n_feat)*) – Permutations of *x_lh*. If `n_feat == 1`, shape = (n_rep, n_lh).
- **rand_lh** (*ndarray, shape = (n_rep, n_rh, n_feat)*) – Permutations of *x_rh*. If `n_feat == 1`, shape = (n_rep, n_rh). None if *x_rh* is None. Only if *spin_rh_* is not None.

See also:

`SpinPermutations.randomize_gen`

randomize_gen(*x_lh*, *x_rh=None*)

Generate random samples from *x_lh* and *x_rh*.

This is the generator version of `SpinPermutations.randomize`.

Parameters

- **x_lh** (*ndarray, shape = (n_lh,) or (n_lh, n_feat)*) – Array of variables arranged in columns, where *n_feat* is the number of variables.
- **x_rh** (*ndarray, shape = (n_rh,) or (n_rh, n_feat), optional*) – Array of variables arranged in columns for the right hemisphere. Default is None.

Yields

- **rand_lh** (*ndarray, shape = (n_lh, n_feat)*) – Permutation of *x_lh*. If `n_feat == 1`, shape = (n_lh,).
- **rand_lh** (*ndarray, shape = (n_rh, n_feat)*) – Permutation of *x_rh*. If `n_feat == 1`, shape = (n_rh,). None if *x_rh* is None. Only if *spin_rh_* is not None.

See also:

`SpinPermutations.randomize`

brainspace.null_models.spin.spin_permutations

brainspace.null_models.spin.**spin_permutations**(*spheres*, *data*, *unique=False*, *n_rep=100*, *random_state=None*, *surface_algorithm='FreeSurfer'*)

Generate null data using spin permutations.

Parameters

- **spheres** (*dict[str, ndarray or BSPolyData], BSPolyData or ndarray*) – Dictionary of points in a sphere, for left ('lh' key) and right ('rh' key) hemispheres. The right hemisphere is optional. If provided, rotations are derived from the rotations computed for *points_lh* by reflecting the rotation matrix across the Y-Z plane.
- **data** (*dict[str, ndarray] or ndarray*) – Dictionary of data to randomize. Array of variables arranged in columns for each hemisphere.
- **unique** (*bool, optional*) – Whether to enforce a one-to-one correspondence between original points and rotated ones. If true, the Hungarian algorithm is used. Default is False.
- **n_rep** (*int, optional*) – Number of random rotations. Default is 100.

- **surface_algorithm** (*{'FreeSurfer', 'CIVET'}*) – For 'CIVET', no flip is required to generate the spins for the right hemisphere. Only used when `points_rh is not None`. Default is 'FreeSurfer'.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Returns

- **rand_lh** (*ndarray, shape = (n_rep, n_lh, n_feat)*) – Permutations of data in left hemisphere.
- **rand_rh** (*ndarray, shape = (n_rep, n_rh, n_feat)*) – Permutations of data in right hemisphere. Only if right data and sphere are provided.

See also:

SpinPermutations

References

- Alexander-Bloch A, Shou H, Liu S, Satterthwaite TD, Glahn DC, Shinohara RT, Vandekar SN and Raznahan A (2018). On testing for spatial correspondence between maps of human brain structure and function. NeuroImage, 178:540-51.
- Blaser R and Fryzlewicz P (2016). Random Rotation Ensembles. Journal of Machine Learning Research, 17(4): 1–26.
- https://netneurotools.readthedocs.io

Moran spectral randomization

MoranRandomization([procedure, spectrum, ...])	Moran spectral randomization.
compute_mem(w[, n_ring, spectrum, tol])	Compute Moran eigenvectors map.
moran_randomization(x, mem[, n_rep, ...])	Generate random samples from *x* based on Moran spectral randomization.

brainspace.null_models.moran.MoranRandomization

class brainspace.null_models.moran.**MoranRandomization**(*procedure='singleton', spectrum='nonzero', joint=False, n_rep=100, n_ring=1, tol=1e-10, random_state=None*)

Moran spectral randomization.

Parameters

- **procedure** (*{'singleton', 'pair'}, optional*) – Procedure to generate the random samples. Default is 'singleton'.
- **spectrum** (*{'all', 'nonzero'}, optional*) – Eigenvalues/vectors to select. If 'all', recover all eigenvectors except one. Otherwise, select all except non-zero eigenvectors. Default is 'nonzero'.
- **joint** (*boolean, optional*) – If True variables are randomized jointly. Otherwise, each variable is randomized separately. Default is False.
- **n_rep** (*int, optional*) – Number of randomizations. Default is 100.

- **n_ring** (*int, optional*) – Neighborhood size to build the weight matrix. Only used if user provides a surface mesh. Default is 1.
- **tol** (*float, optional*) – Minimum value for an eigenvalue to be considered non-zero. Default is 1e-10.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Variables

- **mev** (*1D ndarray, shape (n_components,)*) – Eigenvalues of the weight matrix in descending order.
- **mem** (*2D ndarray, shape (n_vertices, n_components)*) – Eigenvectors of the weight matrix in same order.

See also:

SpinPermutations

__init__ (*procedure='singleton'*, *spectrum='nonzero'*, *joint=False*, *n_rep=100*, *n_ring=1*, *tol=1e-10*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([procedure, spectrum, joint, ...])	Initialize self.
fit(w)	Compute Moran eigenvectors map.
get_params([deep])	Get parameters for this estimator.
randomize(x)	Generate random samples from *x*.
set_params(**params)	Set the parameters of this estimator.

fit (*w*)
Compute Moran eigenvectors map.

Parameters **w** (*BSPolyData, ndarray or sparse matrix, shape = (n_verts, n_verts)*) – Spatial weight matrix or surface. If surface, the weight matrix is built based on the inverse geodesic distance between each vertex and the vertices in its *n_ring*.

Returns **self** (*object*) – Returns self.

randomize (*x*)
Generate random samples from *x*.

Parameters **x** (*1D or 2D ndarray, shape = (n_verts,) or (n_verts, n_feat)*) – Array of variables arranged in columns, where *n_feat* is the number of variables.

Returns **output** (*ndarray, shape = (n_rep, n_verts, n_feat)*) – Random samples. If n_feat == 1, shape = (n_rep, n_verts).

brainspace.null_models.moran.compute_mem

brainspace.null_models.moran.**compute_mem** (*w*, *n_ring=1*, *spectrum='nonzero'*, *tol=1e-10*)
Compute Moran eigenvectors map.

Parameters

- **w** (*BSPolyData, ndarray or sparse matrix, shape = (n_vertices, n_vertices)*) – Spatial weight matrix or surface. If surface, the weight matrix is built based on the inverse geodesic distance between each vertex and the vertices in its *n_ring*.
- **n_ring** (*int, optional*) – Neighborhood size to build the weight matrix. Only used if user provides a surface mesh. Default is 1.
- **spectrum** (*{'all', 'nonzero'}, optional*) – Eigenvalues/vectors to select. If 'all', recover all eigenvectors except the smallest one. Otherwise, select all except non-zero eigenvectors. Default is 'nonzero'.
- **tol** (*float, optional*) – Minimum value for an eigenvalue to be considered non-zero. Default is 1e-10.

Returns

- **w** (*1D ndarray, shape (n_components,)*) – Eigenvalues in descending order. With `n_components = n_vertices - 1` if `spectrum == 'all'` and `n_components = n_vertices - n_zero` if `spectrum == 'nonzero'`, and *n_zero* is number of zero eigenvalues.
- **mem** (*2D ndarray, shape (n_vertices, n_components)*) – Eigenvectors of the weight matrix in same order.

See also:

moran_randomization, *MoranRandomization*

References

- Wagner H.H. and Dray S. (2015). Generating spatially constrained null models for irregularly spaced data using Moran spectral randomization methods. Methods in Ecology and Evolution, 6(10):1169-78.

brainspace.null_models.moran.moran_randomization

brainspace.null_models.moran.**moran_randomization**(*x*, *mem*, *n_rep=100*, *procedure='singleton'*, *joint=False*, *random_state=None*)

Generate random samples from *x* based on Moran spectral randomization.

Parameters

- **x** (*1D or 2D ndarray, shape = (n_vertices,) or (n_vertices, n_feat)*) – Array of variables arranged in columns, where *n_feat* is the number of variables.
- **mem** (*2D ndarray, shape = (n_vertices, nv)*) – Moran eigenvectors map, where *nv* is the number of eigenvectors arranged in columns.
- **n_rep** (*int, optional*) – Number of random samples. Default is 100.
- **procedure** (*{'singleton, 'pair'}, optional*) – Procedure to generate the random samples. Default is 'singleton'.
- **joint** (*boolean, optional*) – If True variables are randomized jointly. Otherwise, each variable is randomized separately. Default is False.
- **random_state** (*int or None, optional*) – Random state. Default is None.

Returns **output** (*ndarray, shape = (n_rep, n_vertices, n_feat)*) – Random samples. If `n_feat == 1`, shape = (n_rep, n_vertices).

See also:

compute_mem, *MoranRandomization*

References

- Wagner H.H. and Dray S. (2015). Generating spatially constrained null models for irregularly spaced data using Moran spectral randomization methods. Methods in Ecology and Evolution, 6(10):1169-78.

Surrogate maps

SurrogateMaps([deltas, kernel, pv, nh, ...])	Spatial autocorrelation-preserving surrogate brain maps.
SampledSurrogateMaps([ns, pv, nh, knn, b, ...])	Spatial autocorrelation-preserving surrogate brain maps wih sampling.

brainspace.null_models.variogram.SurrogateMaps

class brainspace.null_models.variogram.**SurrogateMaps**(*deltas=None*, *kernel='exp'*, *pv=25*, *nh=25*, *resample=False*, *b=None*, *n_rep=100*, *random_state=None*)

Spatial autocorrelation-preserving surrogate brain maps.

Parameters

- **deltas** (*1D ndarray or List[float], optional*) – Proportion of neighbors to include for smoothing, in (0, 1] Default is [0.1,0.2,...,0.9].
- **kernel** (*str, optional*) –

  Kernel with which to smooth permuted maps: 'gaussian' : Gaussian function. 'exp' : Exponential decay function. 'invdist' : Inverse distance. 'uniform' : Uniform weights (distance independent).

  Default is 'exp'.
- **pv** (*int, optional*) – Percentile of the pairwise distance distribution at which to truncate during variogram fitting. Default is 25.
- **nh** (*int, optional*) – Number of uniformly spaced distances at which to compute variogram. Default is 25.
- **resample** (*bool, optional*) – Resample surrogate maps' values from target brain map. Default is False.
- **b** (*float or None, optional*) – Gaussian kernel bandwidth for variogram smoothing. If None, set to three times the spacing between variogram x-coordinates. Default is None.
- **n_rep** (*int, optional*) – Number of randomizations (i.e., surrogate maps). Default is 100.
- **random_state** (*int or None, optional*) – Random state. Default is None.

See also:

SampledSurrogateMaps, *SpinPermutations*, *MoranRandomization*

Notes

Passing resample=True preserves the distribution of values in the target map, with the possibility of worsening the simulated surrogate maps' variograms fits.

__init__(*deltas=None*, *kernel='exp'*, *pv=25*, *nh=25*, *resample=False*, *b=None*, *n_rep=100*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([deltas, kernel, pv, nh, resample, ...])	Initialize self.
compute_variogram(x)	Compute variogram values (i.e., one-half squared pairwise differences).
fit(dist)	Prepare data for sorrugate map generation..
get_params([deep])	Get parameters for this estimator.
permute_map(x)	Return randomly permuted brain map.
randomize(x[, n_rep])	Generate surrogate maps from *x*.
regress(x, y)	Linearly regress *x* onto *y*.
set_params(**params)	Set the parameters of this estimator.
smooth_map(x, delta)	Smooth *x* using *delta* proportion of nearest neighbors.
smooth_variogram(v[, return_h])	Smooth a variogram.

Attributes

h	distances at which smoothed variogram is computed

compute_variogram(*x*)
Compute variogram values (i.e., one-half squared pairwise differences).

Parameters **x** (*1D ndarray*) – Brain map scalar array

Returns **v** (*ndarray, shape (N(N-1)/2,)*) – Variogram y-coordinates, i.e. 0.5 * (x_i - x_j) ^ 2

fit(*dist*)
Prepare data for sorrugate map generation..

Parameters **dist** (*filename or ndarray, shape (N,N)*) – Pairwise (geodesic) distance matrix.

Returns **self** (*object*) – Returns self.

h
distances at which smoothed variogram is computed

Type 1D ndarray

permute_map(*x*)
Return randomly permuted brain map.

Parameters **x** (*1D masked ndarray*) – Brain map scalars

Returns *1D ndarray* – Random permutation of target brain map

randomize(*x*, *n_rep=None*)
Generate surrogate maps from *x*.

Parameters

- **x** (*filename or 1D ndarray*) – Target brain map
- **n_rep** (*int or None, optional*) – Number of surrogates maps to randomly generate. If None, use the default *n_rep*.

Returns output (*ndarray, shape = (n_rep, n_verts)*) – Randomly generated map(s) with matched spatial autocorrelation.

regress(*x*, *y*)
Linearly regress *x* onto *y*.

Parameters

- **x** (*1D ndarray*) – Independent variable
- **y** (*1D ndarray*) – Dependent variable

Returns

- **alpha** (*float*) – Intercept term (offset parameter)
- **beta** (*float*) – Regression coefficient (scale parameter)
- **res** (*float*) – Sum of squared residuals

smooth_map(*x*, *delta*)
Smooth *x* using *delta* proportion of nearest neighbors.

Parameters

- **x** (*1D ndarray*) – Brain map scalars
- **delta** (*float*) – Proportion of neighbors to include for smoothing, in (0, 1)

Returns *1D ndarray* – Smoothed brain map

smooth_variogram(*v*, *return_h=False*)
Smooth a variogram.

Parameters

- **v** (*1D ndarray*) – Variogram values, i.e. 0.5 * (x_i - x_j) ^ 2
- **return_h** (*bool, default False*) – Return distances at which the smoothed variogram was computed

Returns

- *1D ndarray, shape (nh,)* – Smoothed variogram values
- *1D ndarray, shape (nh,)* – Distances at which smoothed variogram was computed (returned only if *return_h* is True)

Raises ValueError : *v* has unexpected size.

brainspace.null_models.variogram.SampledSurrogateMaps

class brainspace.null_models.variogram.**SampledSurrogateMaps**(*ns=500*, *pv=70*, *nh=25*, *knn=1000*, *b=None*, *deltas=None*, *kernel='exp'*, *resample=False*, *n_rep=100*, *random_state=None*, *verbose=False*)

Spatial autocorrelation-preserving surrogate brain maps wih sampling.

Parameters

- **ns** (*int, optional*) – Take a subsample of *ns* rows from *D* when fitting variograms. Default is 500.
- **deltas** (*1D ndarray or List[float], optional*) – Proportion of neighbors to include for smoothing, in (0, 1] Default is [0.1,0.2,. . . ,0.9].
- **kernel** (*str, optional*) –

  Kernel with which to smooth permuted maps: 'gaussian' : Gaussian function. 'exp' : Exponential decay function. 'invdist' : Inverse distance. 'uniform' : Uniform weights (distance independent).

  Default is 'exp'.
- **pv** (*int, optional*) – Percentile of the pairwise distance distribution at which to truncate during variogram fitting. Default is 25.
- **nh** (*int, optional*) – Number of uniformly spaced distances at which to compute variogram. Default is 25.
- **knn** (*int, optional*) – Number of nearest regions to keep in the neighborhood of each region. Default is 1000.
- **b** (*float or None, default None*) – Gaussian kernel bandwidth for variogram smoothing. if None, three times the distance interval spacing is used.
- **resample** (*bool, optional*) – Resample surrogate maps' values from target brain map. Default is False.
- **n_rep** (*int, optional*) – Number of randomizations (i.e., surrogate maps). Default is 100.
- **random_state** (*int or None, optional*) – Random state. Default is None.
- **verbose** (*bool, default False*) – Print surrogate count each time new surrogate map created

See also:

SurrogateMaps, *SpinPermutations*, *MoranRandomization*

Notes

Passing resample=True will preserve the distribution of values in the target map, at the expense of worsening simulated surrogate maps' variograms fits. This worsening will increase as the empirical map more strongly deviates from normality.

__init__ (*ns=500*, *pv=70*, *nh=25*, *knn=1000*, *b=None*, *deltas=None*, *kernel='exp'*, *resample=False*, *n_rep=100*, *random_state=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([ns, pv, nh, knn, b, deltas, ...])	Initialize self.
compute_variogram(x, idx)	Compute variogram of *x* using pairs of regions indexed by *idx*.
fit(dist, index)	Prepare data for surrogate map generation.
get_params([deep])	Get parameters for this estimator.
permute_map(x)	Return a random permutation of *x*.
randomize(x[, n_rep])	Generate surrogate maps from *x*.
regress(x, y)	Linearly regress *x* onto *y*.
sample(n)	Randomly sample (without replacement) brain areas for variogram computation.
set_params(**params)	Set the parameters of this estimator.
smooth_map(x, k)	Smooth *x* using *k* nearest neighboring regions.
smooth_variogram(u, v[, return_h])	Smooth a variogram.

Attributes

h	distances at which variogram is evaluated

compute_variogram (*x*, *idx*)
Compute variogram of *x* using pairs of regions indexed by *idx*.

Parameters

- **x** (*1D ndarray*) – Brain map
- **idx** (*ndarray[int], shape (ns,)*) – Indices of randomly sampled brain regions

Returns **v** (*ndarray, shape (ns,ns)*) – Variogram y-coordinates, i.e. 0.5 * (x_i - x_j) ^ 2, for i,j in idx

fit (*dist*, *index*)
Prepare data for surrogate map generation.

Parameters

- **dist** (*ndarray or memmap, shape (N,N)*) – Pairwise distance matrix. Each row of *dist* should be sorted. Indices used to sort each row are passed to through the *index* argument. See `brainspace.variogram.txt2memmap`.
- **index** (*filename or ndarray or memmap, shape(N,N)*) – Indices used to sort each row of *dist*.

Returns **self** (*object*) – Returns self.

h
distances at which variogram is evaluated

Type 1D ndarray

permute_map (*x*)
Return a random permutation of *x*.

Parameters **x** (*1D ndarray*) – Brain map

Returns *1D ndarray* – Random permutation of target brain map

randomize(*x*, *n_rep=None*)
Generate surrogate maps from *x*.

Parameters

- **x** (*filename or 1D ndarray*) – Target brain map
- **n_rep** (*int or None, optional*) – Number of surrogates maps to randomly generate. If None, use the default *n_rep*.

Returns **output** (*ndarray, shape = (n_rep, n_verts)*) – Randomly generated map(s) with matched spatial autocorrelation.

regress(*x*, *y*)
Linearly regress *x* onto *y*.

Parameters

- **x** (*1D ndarray*) – Independent variable
- **y** (*1D ndarray*) – Dependent variable

Returns

- **alpha** (*float*) – Intercept term (offset parameter)
- **beta** (*float*) – Regression coefficient (scale parameter)
- **res** (*float*) – Sum of squared residuals

sample(*n*)
Randomly sample (without replacement) brain areas for variogram computation.

Returns *ndarray, shape (ns,)* – Indices of randomly sampled areas

smooth_map(*x*, *k*)
Smooth *x* using *k* nearest neighboring regions.

Parameters

- **x** (*1D ndarray*) – Brain map
- **k** (*float*) – Number of nearest neighbors to include for smoothing

Returns **x_smooth** (*1D ndarray*) – Smoothed brain map

Notes

Assumes *dist* provided at runtime has been sorted.

smooth_variogram(*u*, *v*, *return_h=False*)
Smooth a variogram.

Parameters

- **u** (*1D ndarray*) – Pairwise distances, ie variogram x-coordinates
- **v** (*1D ndarray*) – Squared differences, ie variogram y-coordinates
- **return_h** (*bool, default False*) – Return distances at which smoothed variogram is computed

Returns

- *ndarray, shape (nh,)* – Smoothed variogram samples
- *ndarray, shape (nh,)* – Distances at which smoothed variogram was computed (returned if *return_h* is True)

Raises ValueError : *u* and *v* are not identically sized

3.2.3 Plotting functionality

- *Plotting of brain surfaces*
- *Generic plotting*

Plotting of brain surfaces

`plot_hemispheres`(surf_lh, surf_rh[, ...])	Plot left and right hemispheres in lateral and medial views.
`plot_surf`(surfs, layout[, array_name, view, ...])	Plot surfaces arranged according to the *layout*.
`build_plotter`(surfs, layout[, array_name, ...])	Build plotter arranged according to the *layout*.

brainspace.plotting.surface_plotting.plot_hemispheres

brainspace.plotting.surface_plotting.**plot_hemispheres**(*surf_lh*, *surf_rh*, *array_name=None*, *color_bar=False*, *color_range=None*, *label_text=None*, *layout_style='row'*, *cmap='viridis'*, *nan_color=(0, 0, 0, 1)*, *zoom=1*, *background=(1, 1, 1)*, *size=(400, 400)*, *interactive=True*, *embed_nb=False*, *screenshot=False*, *filename=None*, *scale=(1, 1)*, *transparent_bg=True*, ***kwargs*)

Plot left and right hemispheres in lateral and medial views.

Parameters

- **surf_lh** (*vtkPolyData or BSPolyData*) – Left hemisphere.
- **surf_rh** (*vtkPolyData or BSPolyData*) – Right hemisphere.
- **array_name** (*str, list of str, ndarray or list of ndarray, optional*) – Name of point data array to plot. If ndarray, the array is split for the left and right hemispheres. If list, plot one row per array. If None, plot surfaces without any array data. Default is None.
- **color_bar** (*bool, optional*) – Plot color bar for each array (row). Default is False.
- **color_range** (*{'sym'}, tuple or sequence.*) – Range for each array name. If 'sym', uses a symmetric range. Only used if array has positive and negative values. Default is None.

- **label_text** (*dict[str, array-like], optional*) – Label text for column/row. Possible keys are {'left', 'right', 'top', 'bottom'}, which indicate the location. Default is None.
- **layout_style** (*str*) – Layout style for hemispheres. If 'row', layout is a single row alternating lateral and medial views, from left to right. If 'grid', layout is a 2x2 grid, with lateral views in the top row, medial views in the bottom row, and left and right columns. Default is 'row'.
- **nan_color** (*tuple*) – Color for nan values. Default is (0, 0, 0, 1).
- **zoom** (*float or sequence of float, optional*) – Zoom applied to the surfaces in each layout entry.
- **background** (*tuple*) – Background color. Default is (1, 1, 1).
- **cmap** (*str, optional*) – Color map name (from matplotlib). Default is 'viridis'.
- **size** (*tuple, optional*) – Window size. Default is (800, 200).
- **interactive** (*bool, optional*) – Whether to enable interaction. Default is True.
- **embed_nb** (*bool, optional*) – Whether to embed figure in notebook. Only used if running in a notebook. Default is False.
- **screenshot** (*bool, optional*) – Take a screenshot instead of rendering. Default is False.
- **filename** (*str, optional*) – Filename to save the screenshot. Default is None.
- **transparent_bg** (*bool, optional*) – Whether to us a transparent background. Only used if `screenshot==True`. Default is False.
- **scale** (*tuple, optional*) – Scale (magnification). Only used if `screenshot==True`. Default is None.
- **kwargs** (*keyword-valued args*) – Additional arguments passed to the plotter.

Returns **figure** (*Ipython Image or None*) – Figure to plot. None if using vtk for rendering (i.e., `embed_nb == False`).

See also:

build_plotter, *plot_surf*

brainspace.plotting.surface_plotting.plot_surf

brainspace.plotting.surface_plotting.**plot_surf**(*surfs*, *layout*, *array_name=None*, *view=None*, *color_bar=None*, *color_range=None*, *share=False*, *label_text=None*, *cmap='viridis'*, *nan_color=(0, 0, 0, 1)*, *zoom=1*, *background=(1, 1, 1)*, *size=(400, 400)*, *embed_nb=False*, *interactive=True*, *scale=(1, 1)*, *transparent_bg=True*, *screenshot=False*, *filename=None*, *return_plotter=False*, *suppress_warnings=False*, ***kwargs*)

Plot surfaces arranged according to the *layout*.

Parameters

- **surfs** (*dict[str, BSPolyData]*) – Dictionary of surfaces.
- **layout** (*array-like, shape = (n_rows, n_cols)*) – Array of surface keys in *surfs*. Specifies how window is arranged.
- **array_name** (*array-like, optional*) – Names of point data array to plot for each layout entry. Use a tuple with multiple array names to plot multiple arrays (overlays) per layout entry. If None, plot surfaces without any array data. Default is None.
- **view** (*array-like, optional*) – View for each layout entry. Possible views are {'lateral', 'medial', 'ventral', 'dorsal'}. If None, use default view. Default is None.
- **color_bar** (*{'left', 'right', 'top', 'bottom'} or None, optional*) – Location where color bars are rendered. If None, color bars are not included. Default is None.
- **color_range** (*{'sym'}, tuple or sequence.*) – Range for each array name. If 'sym', uses a symmetric range. Only used if array has positive and negative values. Default is None.
- **share** (*{'row', 'col', 'both'} or bool, optional*) – If `share == 'row'`, point data for surfaces in the same row share same data range. If `share == 'col'`, the same but for columns. If `share == 'both'`, all data shares same range. If True, similar to `share == 'both'`. Default is False.
- **label_text** (*dict[str, array-like], optional*) – Label text for column/row. Possible keys are {'left', 'right', 'top', 'bottom'}, which indicate the location. Default is None.
- **cmap** (*str or sequence of str, optional*) – Color map name (from matplotlib) for each array name. Default is 'viridis'.
- **nan_color** (*tuple*) – Color for nan values. Default is (0, 0, 0, 1).
- **zoom** (*float or sequence of float, optional*) – Zoom applied to the surfaces in each layout entry.
- **background** (*tuple*) – Background color. Default is (1, 1, 1).
- **size** (*tuple, optional*) – Window size. Default is (400, 400).
- **interactive** (*bool, optional*) – Whether to enable interaction. Default is True.
- **embed_nb** (*bool, optional*) – Whether to embed figure in notebook. Only used if running in a notebook. Default is False.
- **screenshot** (*bool, optional*) – Take a screenshot instead of rendering. Default is False.
- **filename** (*str, optional*) – Filename to save the screenshot. Default is None.
- **transparent_bg** (*bool, optional*) – Whether to us a transparent background. Only used if `screenshot==True`. Default is False.
- **scale** (*tuple, optional*) – Scale (magnification). Only used if `screenshot==True`. Default is None.
- **return_plotter** (*bool, optional*) – If True, return plotter instead of returning image or rendering.
- **suppress_warnings** (*bool, optional*) – Whether to suppress warnings. Default is False.

- **kwargs** (*keyword-valued args*) – Additional arguments passed to the renderers, actors, mapper or plotter. Keywords starting with:
  - '**renderer__**' are passed to the renderers.
  - '**actor__**' are passed to the actors.
  - '**mapper__**' are passed to the mappers.

  The rest of keywords are passed to the plotter.

Returns **figure** (*Ipython Image or panel or None*) – Figure to plot. None if using vtk for rendering (i.e., `embed_nb == False`).

See also:

build_plotter, *plot_hemispheres*

Notes

If sequences, shapes of *array_name*, *view* and *zoom* must be equal or broadcastable to the shape of *layout*. Renderer keywords must also be broadcastable to the shape of *layout*.

If sequences, shapes of *cmap* and *cbar_range* must be equal or broadcastable to the shape of *array_name*, including the number of array names per entry. Actor and mapper keywords must also be broadcastable to the shape of *array_name*.

brainspace.plotting.surface_plotting.build_plotter

brainspace.plotting.surface_plotting.**build_plotter**(*surfs*, *layout*, *array_name=None*, *view=None*, *color_bar=None*, *color_range=None*, *share=False*, *label_text=None*, *cmap='viridis'*, *nan_color=(0, 0, 0, 1)*, *zoom=1*, *background=(1, 1, 1)*, *size=(400, 400)*, ***kwargs*)

Build plotter arranged according to the *layout*.

Parameters

- **surfs** (*dict[str, BSPolyData]*) – Dictionary of surfaces.
- **layout** (*array-like, shape = (n_rows, n_cols)*) – Array of surface keys in *surfs*. Specifies how window is arranged.
- **array_name** (*array-like, optional*) – Names of point data array to plot for each layout entry. Use a tuple with multiple array names to plot multiple arrays (overlays) per layout entry. If None, plot surfaces without any array data. Default is None.
- **view** (*array-like, optional*) – View for each layout entry. Possible views are {'lateral', 'medial', 'ventral', 'dorsal'}. If None, use default view. Default is None.
- **color_bar** (*{'left', 'right', 'top', 'bottom'} or None, optional*) – Location where color bars are rendered. If None, color bars are not included. Default is None.
- **color_range** (*{'sym'}, tuple or sequence.*) – Range for each array name. If 'sym', uses a symmetric range. Only used if array has positive and negative values. Default is None.

- **share** (*{'row', 'col', 'both'} or bool, optional*) – If `share == 'row'`, point data for surfaces in the same row share same data range. If `share == 'col'`, the same but for columns. If `share == 'both'`, all data shares same range. If True, similar to `share == 'both'`. Default is False.
- **label_text** (*dict[str, array-like], optional*) – Label text for column/row. Possible keys are {'left', 'right', 'top', 'bottom'}, which indicate the location. Default is None.
- **cmap** (*str or sequence of str, optional*) – Color map name (from matplotlib) for each array name. Default is 'viridis'.
- **nan_color** (*tuple*) – Color for nan values. Default is (0, 0, 0, 1).
- **zoom** (*float or sequence of float, optional*) – Zoom applied to the surfaces in each layout entry.
- **background** (*tuple*) – Background color. Default is (1, 1, 1).
- **size** (*tuple, optional*) – Window size. Default is (400, 400).
- **kwargs** (*keyword-valued args*) – Additional arguments passed to the renderers, actors, mapper, color_bar or plotter. Keywords starting with:
  - '**renderer__**' are passed to the renderers.
  - '**actor__**' are passed to the actors.
  - '**mapper__**' are passed to the mappers.
  - '**cb__**' are passed to color bar actors.
  - '**text__**' are passed to color text actors.

  The rest of keywords are passed to the plotter.

Returns **plotter** (*Plotter*) – An instance of Plotter.

See also:

plot_surf, *plot_hemispheres*

Notes

If sequences, shapes of *array_name*, *view* and *zoom* must be equal or broadcastable to the shape of *layout*. Renderer keywords must also be broadcastable to the shape of *layout*.

If sequences, shapes of *cmap* and *cbar_range* must be equal or broadcastable to the shape of *array_name*, including the number of array names per entry. Actor and mapper keywords must also be broadcastable to the shape of *array_name*.

Generic plotting

Plotter([nrow, ncol, offscreen, ...])	
GridPlotter([nrow, ncol, try_qt, offscreen])	

brainspace.plotting.base.Plotter

class brainspace.plotting.base.**Plotter**(*nrow=1*, *ncol=1*, *offscreen=None*, *force_close=False*, *try_qt=False*, ***kwargs*)

__init__(*nrow=1*, *ncol=1*, *offscreen=None*, *force_close=False*, *try_qt=False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

AddRenderer([row, col, renderer])	
__init__([nrow, ncol, offscreen, ...])	Initialize self.
close()	
close_all()	
key_quit([obj, event])	
quit(*args)	
screenshot(filename[, transparent_bg, scale])	
show([embed_nb, interactive, ...])	
to_notebook([transparent_bg, scale])	
to_numpy([transparent_bg, scale])	
to_panel([scale])	

Attributes

DICT_PLOTTERS

AddRenderer(*row=None*, *col=None*, *renderer=None*, ***kwargs*)

DICT_PLOTTERS = {}

close()

classmethod close_all()

key_quit(*obj=None*, *event=None*)

quit(**args*)

screenshot(*filename*, *transparent_bg=True*, *scale=(1, 1)*)

show(*embed_nb=False*, *interactive=True*, *transparent_bg=True*, *scale=(1, 1)*)

to_notebook(*transparent_bg=True*, *scale=(1, 1)*)

to_numpy(*transparent_bg=True*, *scale=(1, 1)*)

to_panel(*scale=(1, 1)*)

brainspace.plotting.base.GridPlotter

class brainspace.plotting.base.**GridPlotter**(*nrow=1*, *ncol=1*, *try_qt=True*, *offscreen=None*, ***kwargs*)

__init__(*nrow=1*, *ncol=1*, *try_qt=True*, *offscreen=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

AddRenderer(row, col[, renderer])	
AddRenderers(**kwargs)	
__init__([nrow, ncol, try_qt, offscreen])	Initialize self.
close()	
close_all()	
key_quit([obj, event])	
quit(*args)	
screenshot(filename[, transparent_bg, scale])	
show([embed_nb, interactive, ...])	
to_notebook([transparent_bg, scale])	
to_numpy([transparent_bg, scale])	
to_panel([scale])	

Attributes

DICT_PLOTTERS

AddRenderer(*row*, *col*, *renderer=None*, ***kwargs*)

AddRenderers(***kwargs*)

DICT_PLOTTERS = {}

close()

classmethod close_all()

key_quit(*obj=None*, *event=None*)

quit(**args*)

screenshot(*filename*, *transparent_bg=True*, *scale=(1, 1)*)

show(*embed_nb=False*, *interactive=True*, *transparent_bg=True*, *scale=(1, 1)*)

to_notebook(*transparent_bg=True*, *scale=(1, 1)*)

to_numpy(*transparent_bg=True*, *scale=(1, 1)*)

to_panel(*scale=(1, 1)*)

3.2.4 Datasets

- *Surfaces*
- *Cortical features*
- *Connectivity matrices*

Surfaces

load_conte69([as_sphere, with_normals, join])	Load conte69 surfaces.

brainspace.datasets.load_conte69

brainspace.datasets.**load_conte69**(*as_sphere=False*, *with_normals=True*, *join=False*)
Load conte69 surfaces.

Parameters

- **as_sphere** (*bool, optional*) – Return spheres instead of cortical surfaces. Default is False.
- **with_normals** (*bool, optional*) – Whether to compute surface normals. Default is True.
- **join** (*bool, optional*) – If False, return one surface for left and right hemispheres. Otherwise, return a single surface as a combination of both left and right surfaces. Default is False.

Returns **surf** (*tuple of BSPolyData or BSPolyData*) – Surfaces for left and right hemispheres. If `join == True`, one surface with both hemispheres.

Cortical features

load_mask([name, join])	Load mask for conte69.
load_parcellation(name[, scale, join])	Load parcellation for conte69.
load_marker(name[, join])	Load cortical data for conte69.
load_gradient(name[, idx, join])	Load gradient for conte69.

brainspace.datasets.load_mask

brainspace.datasets.**load_mask**(*name='midline'*, *join=False*)
Load mask for conte69.

Parameters

- **name** (*{'midline', 'temporal'} or None, optional*) – Region name. If 'midline', load mask for all cortex. Default is 'midline'.
- **join** (*bool, optional*) – If False, return one array for each hemisphere. Otherwise, return a single array for both left and right hemispheres. Default is False.

Returns **mask** (*tuple of ndarrays or ndarray*) – Boolean masks for left and right hemispheres. If `join == True`, one mask with both hemispheres.

brainspace.datasets.load_parcellation

brainspace.datasets.**load_parcellation**(*name*, *scale=400*, *join=False*)
Load parcellation for conte69.

Parameters

- **name** (*{'schaefer', 'vosdewael'}*) – Parcellation name, either 'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of aparc.
- **scale** (*{100, 200, 300, 400}, optional*) – Number of parcels. Default is 400.
- **join** (*bool, optional*) – If False, return one array for each hemisphere. Otherwise, return a single array for both left and right hemisphere. Default is False.

Returns **parcellation** (*tuple of ndarrays or ndarray*) – Parcellations for left and right hemispheres. If `join == True`, one parcellation with both hemispheres.

brainspace.datasets.load_marker

`brainspace.datasets.`**`load_marker`**(*name*, *join=False*)

Load cortical data for conte69.

Markers are derived from the discovery (main) datasets of (Vos de Wael et al., 2018, PNAS).

Parameters

- **name** (*{'curvature', 'thickness', 't1wt2w'}*) – Marker name.
- **join** (*bool, optional*) – If False, return one array for each hemisphere. Otherwise, return a single array for both left and right hemispheres. Default is False.

Returns **marker** (*tuple of ndarrays or ndarray*) – Marker data for left and right hemispheres. If `join == True`, one array with both hemispheres.

brainspace.datasets.load_gradient

`brainspace.datasets.`**`load_gradient`**(*name*, *idx=0*, *join=False*)

Load gradient for conte69.

Parameters

- **name** (*{'fc', 'mpc'}*) – The type of gradient, either 'fc' for functional connectivity or 'mpc' for microstructural profile covariance.
- **idx** (*int, optional*) – Gradient index. Default is 0 (first gradient).
- **join** (*bool, optional*) – If False, return one array for each hemisphere. Otherwise, return a single array for both left and right hemispheres. Default is False.

Returns **marker** (*tuple of ndarrays or ndarray*) – Gradients for left and right hemispheres. If `join == True`, one gradient array with both hemispheres.

Connectivity matrices

load_group_fc(parcellation[, scale, group])	Load group level connectivity matrix for a given parcellation.
load_group_mpc(parcellation[, scale])	Load group level connectivity matrix for a given parcellation.

brainspace.datasets.load_group_fc

`brainspace.datasets.`**`load_group_fc`**(*parcellation*, *scale=400*, *group='main'*)

Load group level connectivity matrix for a given parcellation.

Connectivity is derived from a the discovery (main) and validation (holdout) datasets of (Vos de Wael et al., 2018, PNAS).

Parameters

- **parcellation** (*{'schaefer', 'vosdewael'}*) – Parcellation name, either

'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of aparc.

- **scale** (*{100, 200, 300, 400}, optional*) – Number of parcels. Default is 400.
- **group** (*{'main', 'holdout'}*) – Group of subjects used to derive the connectivity matrix. Default is 'main'.

Returns **conn** (*2D ndarray, shape = (scale, scale)*) – Connectivity matrix.

brainspace.datasets.load_group_mpc

brainspace.datasets.**load_group_mpc**(*parcellation*, *scale=400*)

Load group level connectivity matrix for a given parcellation.

MPC is derived from the dataset used by (Paquola et al., 2019, PLoS biology).

Parameters

- **parcellation** (*{'schaefer', 'vosdewael'}*) – Parcellation name, either 'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of aparc.
- **scale** (*{100, 200, 300, 400}, optional*) – Number of parcels. Default is 400.

Returns **conn** (*2D ndarray, shape = (scale, scale)*) – Connectivity matrix.

3.2.5 Mesh

BrainSpace provides basic functionality for working with surface meshes. This functionality is built on top of the Visualization Toolkit (VTK).

- *Read/Write functionality*
- *Surface creation*
- *Elements*
- *Connectivity*
- *Operations on meshes*
- *Operations on mesh data*
- *Mesh clustering*

Read/Write functionality

read_surface(ipth[, itype, return_data, update])	Read surface data.
write_surface(ifilter, opth[, oformat, otype])	Write surface data.

brainspace.mesh.mesh_io.read_surface

brainspace.mesh.mesh_io.**read_surface**(*ipth*, *itype=None*, *return_data=True*, *update=True*)

Read surface data.

See *itype* for supported file types.

Parameters

- **ipth** (*str*) – Input filename.
- **itype** (*{'ply', 'vtp', 'vtk', 'fs', 'asc', 'gii'}, optional*) – Input file type. If None, it is deduced from *ipth*. Files compressed with gzip (.gz) are also supported. Default is None.
- **return_data** (*bool, optional*) – Whether to return data instead of filter. Default is False
- **update** (*bool, optional*) – Whether to update filter When return_data=True, filter is automatically updated. Default is True.

Returns **output** (*BSAlgorithm or BSPolyData*) – Surface as a filter or BSPolyData.

Notes

Function can read FreeSurfer geometry data in binary ('fs') and ascii ('asc') format. Gifti surfaces can also be loaded if nibabel is installed.

See also:

write_surface

brainspace.mesh.mesh_io.write_surface

brainspace.mesh.mesh_io.**write_surface**(*ifilter*, *opth*, *oformat=None*, *otype=None*)

Write surface data.

See *otype* for supported file types.

Parameters

- **ifilter** (*BSAlgorithm or BSDataObject*) – Input filter or data.
- **opth** (*str*) – Output filename.
- **oformat** (*{'ascii', 'binary'}, optional*) – File format. Defaults to writer's default format. Only used when writer accepts format. Default is None.
- **otype** (*{'ply', 'vtp', 'vtk', 'fs', 'asc', 'gii'}, optional*) – File type. If None, type is deduced from *opth*. Default is None.

Notes

Function can save data in FreeSurfer binary ('fs') and ascii ('asc') format. Gifti surfaces can also be saved if nibabel is installed.

See also:

read_surface

Surface creation

build_polydata(points[, cells])	Build surface (PolyData) from points and cells.
to_lines(surf)	Convert all cells in PolyData to lines.
to_vertex(surf)	Convert all cells in PolyData to vertex cells.

brainspace.mesh.mesh_creation.build_polydata

brainspace.mesh.mesh_creation.**build_polydata**(*points*, *cells=None*)
Build surface (PolyData) from points and cells.

Parameters

- **points** (*ndarray, shape = (n_points, 3)*) – Array of points.
- **cells** (*ndarray, shape = (n_cells, nd), optional*) – Array of cells. Cells can be vertex (nd=1), line (nd=2) or triangle (nd=3). Default is None (no topology information).

Returns **output** (*BSPolyData*) – Returns surface (PolyData).

See also:

to_vertex, *to_lines*

Notes

Point ids within cells must start from 0 (first point) and contain all points.

brainspace.mesh.mesh_creation.to_lines

brainspace.mesh.mesh_creation.**to_lines**(*surf*)
Convert all cells in PolyData to lines.

Parameters **surf** (*vtkPolyData or BSPolyData*) – Input surface.

Returns **output** (*BSPolyData*) – PolyData with lines.

See also:

to_vertex, *build_polydata*

brainspace.mesh.mesh_creation.to_vertex

brainspace.mesh.mesh_creation.**to_vertex**(*surf*)
Convert all cells in PolyData to vertex cells.

Parameters **surf** (*vtkPolyData or BSPolyData*) – Input surface.

Returns **output** (*BSPolyData*) – PolyData with vertex points.

See also:

to_lines, *build_polydata*

Elements

get_cells(surf)	Get surface cells.
get_points(surf[, mask])	Get surface points.
get_edges(surf[, mask])	Get surface edges.

brainspace.mesh.mesh_elements.get_cells

brainspace.mesh.mesh_elements.**get_cells**(*surf*)

Get surface cells.

Parameters **surf** (*vtkDataSet or BSDataSet*) – Input surface.

Returns **cells** (*ndarray, shape (n_cells, nd)*) – Array of cells. The value of nd depends on the topology. If vertex (nd=1), line (nd=2) or poly (nd=3). Each element is a point id.

Raises ValueError – If *surf* contains different cell types.

See also:

get_points, *get_edges*

brainspace.mesh.mesh_elements.get_points

brainspace.mesh.mesh_elements.**get_points**(*surf*, *mask=None*)

Get surface points.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only get points within the mask. Default is None.

Returns **points** (*ndarray, shape (n_points, 3)*) – Array of points.

See also:

get_cells, *get_edges*

brainspace.mesh.mesh_elements.get_edges

brainspace.mesh.mesh_elements.**get_edges**(*surf*, *mask=None*)

Get surface edges.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only use points within the mask. Default is None.

Returns **edges** (*ndarray, shape (n_edges, 2)*) – Array of edges. Each element is a point id.

See also:

get_edge_length, *get_points*, *get_cells*

Connectivity

get_cell2point_connectivity(surf[, dtype])	mask,	Get cell to point connectivity.
get_point2cell_connectivity(surf[, dtype])	mask,	Get point to cell connectivity.

Continued on next page

Table 33 – continued from previous page

get_cell_neighbors(surf[, include_self, ...])	Get cell connectivity based on shared edges.
get_immediate_adjacency(surf[, ...])	Get immediate adjacency matrix.
get_ring_adjacency(surf[, n_ring, ...])	Get adjacency in the neighborhood of each point.
get_immediate_distance(surf[, metric, mask, ...])	Get immediate distance matrix.
get_ring_distance(surf[, n_ring, metric, ...])	Get distance matrix in the neighborhood of each point.

brainspace.mesh.mesh_elements.get_cell2point_connectivity

brainspace.mesh.mesh_elements.**get_cell2point_connectivity**(*surf*, *mask=None*, *dtype=<class 'numpy.uint8'>*)

Get cell to point connectivity.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only get points within the mask. Default is None.
- **dtype** (*dtype, optional*) – Data type. Default is uint8.

Returns **output** (*sparse matrix, shape (n_cells, n_points)*) – The connectivity matrix. The (i,j) entry is 1 if the i-th cell uses the j-th point.

See also:

get_point2cell_connectivity, *get_cell_neighbors*

Notes

This function returns the transpose of *get_point2cell_connectivity*.

brainspace.mesh.mesh_elements.get_point2cell_connectivity

brainspace.mesh.mesh_elements.**get_point2cell_connectivity**(*surf*, *mask=None*, *dtype=<class 'numpy.uint8'>*)

Get point to cell connectivity.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only get points within the mask. Default is None.
- **dtype** (*dtype, optional*) – Data type. Default is uint8.

Returns **output** (*sparse matrix, shape (n_points, n_cells)*) – The connectivity matrix. The (i,j) entry is 1 if the j-th cell uses the i-th point.

Notes

This function returns the transpose of *get_cell2point_connectivity*.

See also:

get_cell2point_connectivity, *get_cell_neighbors*

brainspace.mesh.mesh_elements.get_cell_neighbors

brainspace.mesh.mesh_elements.**get_cell_neighbors**(*surf*, *include_self=True*, *with_edge=True*, *dtype=<class 'numpy.uint8'>*)

Get cell connectivity based on shared edges.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **include_self** (*bool, optional*) – If True, set diagonal elements to 1. Default is True.
- **with_edge** (*bool, optional*) – If True, neighboring cells are based on shared edges. Otherwise, cells must share, at least, one point. Default is True.
- **dtype** (*dtype, optional*) – Data type. Default is uint8.

Returns **output** (*sparse matrix, shape (n_cells, n_cells)*) – The connectivity matrix. The (i,j) entry is 1 if cells i and j share an edge.

See also:

get_point2cell_connectivity, *get_cell2point_connectivity*

brainspace.mesh.mesh_elements.get_immediate_adjacency

brainspace.mesh.mesh_elements.**get_immediate_adjacency**(*surf*, *include_self=True*, *mask=None*, *dtype=<class 'numpy.uint8'>*)

Get immediate adjacency matrix.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **include_self** (*bool, optional*) – If True, set diagonal elements to 1. Default is True.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only use points within the mask. Default is None.
- **dtype** (*dtype, optional*) – Data type. Default is uint8.

Returns **adj** (*sparse matrix, shape (n_points, n_points)*) – Immediate adjacency matrix.

See also:

get_ring_adjacency, *get_immediate_distance*, *get_ring_distance*

Notes

Immediate adjacency: set to one all entries of points that share and edge with current point.

brainspace.mesh.mesh_elements.get_ring_adjacency

brainspace.mesh.mesh_elements.**get_ring_adjacency**(*surf*, *n_ring=1*, *include_self=True*, *mask=None*, *dtype=<class 'numpy.uint8'>*)

Get adjacency in the neighborhood of each point.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **n_ring** (*int, optional*) – Size of neighborhood. Default is 1.
- **include_self** (*bool, optional*) – If True, set diagonal elements to 1. Otherwise, the diagonal is set to 0. Default is True.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only use points within the mask. Default is None.
- **dtype** (*dtype, optional*) – Data type. Default is uint8.

Returns **adj** (*sparse matrix, shape (n_points, n_points)*) – Adjacency matrix in *n_ring* ring.

See also:

get_immediate_adjacency, *get_immediate_distance*, *get_ring_distance*

brainspace.mesh.mesh_elements.get_immediate_distance

brainspace.mesh.mesh_elements.**get_immediate_distance**(*surf*, *metric='euclidean'*, *mask=None*, *dtype=<class 'float'>*)

Get immediate distance matrix.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only use points within the mask. Default is None.
- **metric** (*{'euclidean', 'sqeuclidean'}, optional*) – Distance metric. Default is 'euclidean'.
- **dtype** (*dtype, optional*) – Data type. Default is float.

Returns **dist** (*sparse matrix, shape (n_points, n_points)*) – Immediate distance matrix.

See also:

get_immediate_adjacency, *get_ring_adjacency*, *get_ring_distance*

Notes

Immediate distance: Euclidean distance with all points that share and edge with current point.

brainspace.mesh.mesh_elements.get_ring_distance

brainspace.mesh.mesh_elements.**get_ring_distance**(*surf*, *n_ring=1*, *metric='geodesic'*, *mask=None*, *dtype=<class 'float'>*)

Get distance matrix in the neighborhood of each point.

Parameters

- **surf** (*vtkDataSet or BSDataSet*) – Input surface.
- **n_ring** (*int, optional*) – Size of neighborhood. Default is 1.
- **metric** (*{'euclidean', 'sqeuclidean', 'geodesic'}, optional*) – Distance metric. Default is 'geodesic'.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only use points within the mask. Default is None.
- **dtype** (*dtype, optional*) – Data type. Default is np.float.

Returns **dist** (*sparse matrix, shape (n_points, n_points)*) – Distance matrix in *n_ring* ring..

See also:

get_immediate_adjacency, *get_ring_adjacency*, *get_immediate_distance*

Notes

Distance is only computed for points in the ring of current point. When using geodesic, shortest paths are restricted to points within the ring.

Operations on meshes

drop_cells(surf, array[, low, upp])	Remove surface cells whose values fall within the threshold.
mask_cells(surf, mask)	Mask surface cells.
select_cells(surf, array[, low, upp])	Select surface cells whose values fall within the threshold.
drop_points(surf, array[, low, upp])	Remove surface points whose values fall within the threshold.
mask_points(surf, mask)	Mask surface points.
select_points(surf, array[, low, upp])	Select surface points whose values fall within the threshold.
get_connected_components(surf[, labeling, ...])	Get connected components.
downsample_with_parcellation(surf, labeling)	Downsample surface according to labeling.

brainspace.mesh.mesh_operations.drop_cells

brainspace.mesh.mesh_operations.**drop_cells**(*surf*, *array*, *low=-inf*, *upp=inf*)

Remove surface cells whose values fall within the threshold.

Points corresponding to these cells are also removed.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **array** (*str or 1D ndarray*) – Array used to perform selection. If str, it must be an array in the CellData attributes of the PolyData.
- **low** (*float or -np.inf*) – Lower threshold. Default is -np.inf.
- **upp** (*float or np.inf*) – Upper threshold. Default is np.inf.

Returns **surf_selected** (*vtkPolyData or BSPolyData*) – PolyData after thresholding.

See also:

drop_points, *select_cells*, *mask_cells*

brainspace.mesh.mesh_operations.mask_cells

brainspace.mesh.mesh_operations.**mask_cells**(*surf*, *mask*)

Mask surface cells.

Points corresponding to these cells are also kept.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **mask** (*1D ndarray*) – Binary boolean array. Zero elements are discarded.

Returns **surf_masked** (*vtkPolyData or BSPolyData*) – PolyData after masking.

See also:

mask_points, *drop_cells*, *select_cells*

brainspace.mesh.mesh_operations.select_cells

brainspace.mesh.mesh_operations.**select_cells**(*surf*, *array*, *low=-inf*, *upp=inf*)

Select surface cells whose values fall within the threshold.

Points corresponding to these cells are also kept.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **array** (*str or 1D ndarray*) – Array used to perform selection. If str, it must be an array in the CellData attributes of the PolyData.
- **low** (*float or -np.inf*) – Lower threshold. Default is -np.inf.
- **upp** (*float or np.inf*) – Upper threshold. Default is np.inf.

Returns **surf_selected** (*vtkPolyData or BSPolyData*) – PolyData after selection.

See also:

select_points, *drop_cells*, *mask_cells*

brainspace.mesh.mesh_operations.drop_points

brainspace.mesh.mesh_operations.**drop_points**(*surf*, *array*, *low=-inf*, *upp=inf*)

Remove surface points whose values fall within the threshold.

Cells corresponding to these points are also removed.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **array** (*str or 1D ndarray*) – Array used to perform selection. If str, it must be an array in the PointData attributes of the PolyData.
- **low** (*float or -np.inf*) – Lower threshold. Default is -np.inf.
- **upp** (*float or np.inf*) – Upper threshold. Default is np.inf.

Returns **surf_selected** (*vtkPolyData or BSPolyData*) – PolyData after thresholding.

See also:

drop_cells, *select_points*, *mask_points*

brainspace.mesh.mesh_operations.mask_points

brainspace.mesh.mesh_operations.**mask_points**(*surf*, *mask*)

Mask surface points.

Cells corresponding to these points are also kept.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **mask** (*1D ndarray*) – Binary boolean array. Zero elements are discarded.

Returns **surf_masked** (*vtkPolyData or BSPolyData*) – PolyData after masking.

See also:

mask_cells, *drop_points*, *select_points*

brainspace.mesh.mesh_operations.select_points

brainspace.mesh.mesh_operations.**select_points**(*surf*, *array*, *low=-inf*, *upp=inf*)

Select surface points whose values fall within the threshold.

Cells corresponding to these points are also kept.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **array** (*str or 1D ndarray*) – Array used to perform selection. If str, it must be an array in the PointData attributes of the PolyData.
- **low** (*float or -np.inf*) – Lower threshold. Default is -np.inf.
- **upp** (*float or np.inf*) – Upper threshold. Default is np.inf.

Returns **surf_selected** (*vtkPolyData or BSPolyData*) – PolyData after selection.

See also:

select_cells, *drop_points*, *mask_points*

brainspace.mesh.mesh_operations.get_connected_components

brainspace.mesh.mesh_operations.**get_connected_components**(*surf*, *labeling=None*, *mask=None*, *fill=0*, *append=False*, *key='components'*)

Get connected components.

Connected components are based on connectivity (and same label if *labeling* is provided).

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **labeling** (*str or 1D ndarray, optional*) – Array with labels. If str, it must be in the point data attributes of *surf*. Default is None. If provided, connectivity is based on neighboring points with the same label.
- **mask** (*str or 1D ndarray, optional*) – Boolean mask. If str, it must be in the point data attributes of *surf*. Default is None. If specified, only consider points within the mask.
- **fill** (*int or float, optional*) – Value used for entries out of the mask. Only used if the *target_mask* is provided. Default is 0.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'components'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – 1D array with different labels for each connected component. Return ndarray if `append == False`. Otherwise, return input surface with the new array.

Notes

VTK point data does not accept boolean arrays. If the mask is provided as a string, the mask is built from the corresponding array such that any value larger than 0 is True.

brainspace.mesh.mesh_operations.downsample_with_parcellation

brainspace.mesh.mesh_operations.**downsample_with_parcellation**(*surf*, *labeling*, *name='parcel'*)

Downsample surface according to labeling.

Such that, each parcel centroid is used as a point in the new downsampled surface. Connectivity is based on neighboring parcels.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **labeling** (*str or 1D ndarray*) – Array of labels used to perform the downsampling. If str, it must be an array in the PointData attributes of *surf*.

- **name** (*str, optional*) – Name of the downsampled parcellation appended to the PointData of the new surface. Default is 'parcel'.

Returns **res** (*BSPolyData*) – Downsampled surface.

Operations on mesh data

compute_cell_area(surf[, append, key])	Compute cell area.
compute_cell_center(surf[, append, key])	Compute center of cells (parametric center).
get_n_adjacent_cells(surf[, append, key])	Compute number of adjacent cells for each point.
map_celldata_to_pointdata(surf, cell_data[, ...])	Map cell data to point data.
map_pointdata_to_celldata(surf, point_data)	Map point data to cell data.
compute_point_area(surf[, cell_area, ...])	Compute point area from its adjacent cells.
get_labeling_border(surf, labeling[, ...])	Get labeling borders.
get_parcellation_centroids(surf, labeling[, ...])	Compute parcels centroids.
propagate_labeling(surf, labeling[, ...])	Propagate labeling on surface points.

brainspace.mesh.array_operations.compute_cell_area

brainspace.mesh.array_operations.**compute_cell_area**(*surf*, *append=False*, *key='cell_area'*)

Compute cell area.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **append** (*bool, optional*) – If True, append array to cell data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's cell data attributes. Only used if `append == True`. Default is 'cell_area'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.compute_cell_center

brainspace.mesh.array_operations.**compute_cell_center**(*surf*, *append=False*, *key='cell_center'*)

Compute center of cells (parametric center).

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **append** (*bool, optional*) – If True, append array to cell data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's cell data attributes. Only used if `append == True`. Default is 'cell_center'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.get_n_adjacent_cells

brainspace.mesh.array_operations.**get_n_adjacent_cells**(*surf*, *append=False*, *key='point_ncells'*)

Compute number of adjacent cells for each point.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **append** (*bool, optional*) – If True, append array to cell data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'point_ncells'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.map_celldata_to_pointdata

brainspace.mesh.array_operations.**map_celldata_to_pointdata**(*surf*, *cell_data*, *red_func='mean'*, *dtype=None*, *append=False*, *key=None*)

Map cell data to point data.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **cell_data** (*str, 1D ndarray*) – Array with cell data. If str, it must be in cell data attributes of *surf*.
- **red_func** (*str or callable, optional.*) – Function used to compute point data from data of neighboring cells. If str, options are {'sum', 'mean', 'mode', 'one_third', 'min', 'max'}. Default is 'mean'.
- **dtype** (*dtype, optional*) – Data type of new array. If None, use the same data type of cell data array. Default is None.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is None.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.map_pointdata_to_celldata

brainspace.mesh.array_operations.**map_pointdata_to_celldata**(*surf*, *point_data*, *red_func='mean'*, *dtype=None*, *append=False*, *key=None*)

Map point data to cell data.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **point_data** (*str, 1D ndarray*) – Array with point data. If str, it is in the point data attributes of *surf*. If ndarray, use this array as point data.
- **red_func** (*{'sum', 'mean', 'mode', 'min', 'max'} or callable, optional*) – Function used to compute data of each cell from data of its points. Default is 'mean'.
- **dtype** (*dtype, optional*) – Data type of new array. If None, use the same data type of point data array. Default is None.
- **append** (*bool, optional*) – If True, append array to cell data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's cell data attributes. Only used if `append == True`. Default is None.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.compute_point_area

brainspace.mesh.array_operations.**compute_point_area**(*surf*, *cell_area=None*, *area_as='one_third'*, *append=False*, *key='point_area'*)

Compute point area from its adjacent cells.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **cell_area** (*str, 1D ndarray or None, optional*) – Array with cell areas. If str, it must be in the cell data attributes of *surf*. If None, cell areas are computed first. Default is None.
- **area_as** (*{'one_third', 'sum', 'mean'}, optional*) – Compute point area as 'one_third', 'sum' or 'mean' of adjacent cells. Default is 'one_third'.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'point_area'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – 1D array with point area. Return ndarray if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.get_labeling_border

brainspace.mesh.array_operations.**get_labeling_border**(*surf*, *labeling*, *append=False*, *key='border'*)

Get labeling borders.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.

- **labeling** (*str, 1D ndarray*) – Array with labels. If str, it must be in the point data attributes of *surf*.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'border'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – A 1D array with ones in the borders. Return array if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.get_parcellation_centroids

brainspace.mesh.array_operations.**get_parcellation_centroids**(*surf*, *labeling*, *non_centroid=0*, *mask=None*, *append=False*, *key='centroids'*)

Compute parcels centroids.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **labeling** (*str, 1D ndarray*) – Array with labels. If str, it must be in the point data attributes of *surf*. If ndarray, use this array as the labeling.
- **non_centroid** (*int, optional*) – Label assigned to non-centroid points. Default is 0.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, only consider points within the mask. Default is None.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'centroids'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – A 1D array with the centroids assigned to their corresponding labels and the rest of points assigned *non_centroid*. Return array if `append == False`. Otherwise, return input surface with the new array.

brainspace.mesh.array_operations.propagate_labeling

brainspace.mesh.array_operations.**propagate_labeling**(*surf*, *labeling*, *no_label=nan*, *mask=None*, *alpha=0.99*, *n_iter=30*, *tol=0.001*, *n_ring=1*, *mode='connectivity'*, *append=False*, *key='propagated'*)

Propagate labeling on surface points.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **labeling** (*str, 1D ndarray*) – Array with initial labels. If str, it must be in the point data attributes of *surf*. If ndarray, use this array as the initial labeling.

- **no_label** (*int or np.nan, optional*) – Value for unlabeled points. Default is np.nan.
- **mask** (*1D ndarray, optional*) – Binary mask. If specified, propagation is only performed on points within the mask. Default is None.
- **alpha** (*float, optional*) – Clamping factor such that `0 < aplha < 1`. Deault is 0.99.
- **n_iter** (*int, optional*) – Maximum number of propagation iterations. Default is 30.
- **tol** (*float, optional*) – Convergence tolerance. Default is 0.001.
- **n_ring** (*positive int, optional*) – Consider points in the n-th ring to label the unlabeled points. Default is 1.
- **mode** (*{'connectivity', 'distance'}, optional*) – Propagation based on connectivity or geodesic distance. Default is 'connectivity'.
- **append** (*bool, optional*) – If True, append array to point data attributes of input surface and return surface. Otherwise, only return array. Default is False.
- **key** (*str, optional*) – Array name to append to surface's point data attributes. Only used if `append == True`. Default is 'propagated'.

Returns **output** (*vtkPolyData, BSPolyData or ndarray*) – A 1D array with the propagated labeling. Return array if `append == False`. Otherwise, return input surface with the new array.

References

- Zhou, D., Bousquet, O., Lal, T. N., Weston, J., & Schölkopf, B. (2004). Learning with local and global consistency. Advances in neural information processing systems, 16(16), 321-328.

Mesh clustering

Clustering and sampling of surface vertices.

cluster_points(surf[, n_clusters, is_size, ...])	Clustering of surface points.
sample_points_clustering(surf[, keep, mask, ...])	Sample equidistant points from surface based on clustering.

brainspace.mesh.mesh_cluster.cluster_points

brainspace.mesh.mesh_cluster.**cluster_points**(*surf*, *n_clusters=100*, *is_size=False*, *mask=None*, *with_centers=True*, *random_state=None*, *approach='kmeans'*, *n_init=3*, *n_jobs=1*)

Clustering of surface points.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **n_clusters** (*int, optional*) – Number of clusters. Default is 100.
- **is_size** (*bool, optional*) – If True, interpret *n_clusters* as cluster size. Default is False.

- **mask** (*1D ndarray, optional*) – Mask for surface points. Points outside the mask (i.e., False) are discarded from clustering. Default is None.
- **with_centers** (*bool, optional*) – If True, an array of labels with the closest points to the centroid of each cluster is returned. Default is True.
- **random_state** (*int, RandomState instance or None, optional*) – Random state. Default is None.
- **approach** (*{'kmeans', 'ward'}, optional*) – Clustering method: k-means or hierarchical with ward linkage. Hierarchical clustering is faster but k-means provides better results. Default is 'kmeans'.
- **n_init** (*int, optional*) – Number of k-means repetitions. Only used when `approach == 'kmeans'`. Default is 3.
- **n_jobs** (*int or None, optional*) – The number of parallel jobs. Only used when `approach == 'kmeans'`. Default is 1.

Returns

- **cluster_labels** (*1D ndarray, shape (n_points,)*) – Array of cluster labels. If *mask* is provided, points out of the mask are assigned label 0.
- **center_labels** (*1D ndarray, shape (n_points,)*) – Array with centers labeled with their corresponding cluster label. The rest of points is assigned label 0. Returned only if `with_centers=True`.

Notes

Valid cluster labels start from 1. If the mask is provided, zeros are assigned to the points outside the mask.

brainspace.mesh.mesh_cluster.sample_points_clustering

brainspace.mesh.mesh_cluster.**sample_points_clustering**(*surf*, *keep=0.1*, *mask=None*, *random_state=None*, *approach='kmeans'*, *n_init=3*, *n_jobs=1*)

Sample equidistant points from surface based on clustering.

Parameters

- **surf** (*vtkPolyData or BSPolyData*) – Input surface.
- **keep** (*float or int, optional*) – If float, percentage of points to sample. Must be `0 < keep < 1`. If int, number of points to sample. Default is 0.1.
- **mask** (*1D ndarray, optional*) – Mask for surface points. Points outside the mask (i.e., False) are discarded from sampling. Default is None.
- **random_state** (*int, RandomState instance or None, optional*) – Random state. Default is None.
- **approach** (*{'kmeans', 'ward'}, optional*) – Clustering approach: k-means or hierarchical with ward linkage. Hierarchical is faster but k-means provides better results. Default is 'kmeans'.
- **n_init** (*int, optional*) – Number of k-means repetitions. Only used when `approach == 'kmeans'`. Default is 3.

- **n_jobs** (*int or None, optional*) – The number of parallel jobs. Only used when `approach == 'kmeans'`. Default is 1.

Returns **sampled** (*1D ndarray, shape (n_points,)*) – Array with sampled points marked with 1 in their corresponding positions. The rest is 0.

See also:

cluster_points, sample_points_decimation

Notes

This method first clusters the surface points and then selects the points closest to the centroids as the sampled points.

3.2.6 VTK interface

Surface mesh functionality provided in BrainSpace is built on top of the Visualization Toolkit (VTK). BrainSpace provides several wrappers for most data objects and some filters in VTK. Here we present a subset of this functionality. Please also refer to *VTK wrapping* document for an introduction to the wrapping interface.

- *Basic wrapping*
- *Pipeline functionality*
- *VTK wrappers*
  - *Data objects*
  - *Algorithms*
  - *Mappers*
  - *Actors*
  - *Lookup tables*
  - *Rendering*
  - *Properties*
  - *Miscellanea*
- *Decorators*

Basic wrapping

wrap_vtk(obj, **kwargs)	Wrap input object to BSVTKObjectWrapper or one of its subclasses.
is_vtk(obj)	Check if *obj* is a vtk object.
is_wrapper(obj)	Check if *obj* is a wrapper.
BSVTKObjectWrapper(vtkobject, **kwargs)	Base class for all classes that wrap VTK objects.

brainspace.vtk_interface.wrappers.base.wrap_vtk

brainspace.vtk_interface.wrappers.base.**wrap_vtk**(*obj*, ***kwargs*)

Wrap input object to BSVTKObjectWrapper or one of its subclasses.

Parameters

- **obj** (*vtkObject or BSVTKObjectWrapper*) – Input object.
- **kwargs** (*kwds, optional*) – Additional keyword parameters are passed to vtk object.

Returns **wrapper** (*BSVTKObjectWrapper*) – The wrapped object.

brainspace.vtk_interface.wrappers.base.is_vtk

brainspace.vtk_interface.wrappers.base.**is_vtk**(*obj*)

Check if *obj* is a vtk object.

Parameters **obj** (*object*) – Any object.

Returns **res** (*bool*) – True if *obj* is a VTK object. False, otherwise.

brainspace.vtk_interface.wrappers.base.is_wrapper

brainspace.vtk_interface.wrappers.base.**is_wrapper**(*obj*)

Check if *obj* is a wrapper.

Parameters **obj** (*object*) – Any object.

Returns **res** (*bool*) – True if *obj* is a VTK wrapper. False, otherwise.

brainspace.vtk_interface.wrappers.base.BSVTKObjectWrapper

class brainspace.vtk_interface.wrappers.base.**BSVTKObjectWrapper**(*vtkobject*, ***kwargs*)

Base class for all classes that wrap VTK objects.

Adapted from dataset_adapter, with additional setVTK and getVTK methods. Create an instance if class is passed instead of object.

This class holds a reference to the wrapped VTK object. It also forwards unresolved methods to the underlying object by overloading __getattr__. This class also supports all VTK setters and getters to be used like properties/attributes dropping the get/set prefix. This is case insensitive.

Parameters

- **vtkobject** (*type or object*) – VTK class or object.
- **kwargs** (*optional keyword parameters*) – Parameters used to invoke set methods on the vtk object.

Variables **VTKObject** (*vtkObject*) – A VTK object.

Examples

```
>>> from vtkmodules.vtkRenderingCorePython import vtkPolyDataMapper
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtkPolyDataMapper())
>>> m1
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4b70198>
>>> m1.VTKObject
(vtkRenderingOpenGL2Python.vtkOpenGLPolyDataMapper)0x7f38a4bee888
```

Passing class and additional keyword arguments:

```
>>> m2 = BSVTKObjectWrapper(vtkPolyDataMapper, arrayId=3,
...                         colorMode='mapScalars')
>>> # Get color name, these are all the same
>>> m2.VTKObject.GetColorModeAsString()
'MapScalars'
>>> m2.GetColorModeAsString()
'MapScalars'
>>> m2.colorModeAsString
'MapScalars'
>>> # Get array id
>>> m2.VTKObject.GetArrayId()
3
>>> m2.GetArrayId()
3
>>> m2.arrayId
3
```

We can change array id and color mode as follows:

```
>>> m2.arrayId = 0
>>> m2.VTKObject.GetArrayId()
0
>>> m2.colorMode = 'default'
>>> m2.VTKObject.GetColorModeAsString()
'Default'
```

__init__(*vtkobject*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(vtkobject, **kwargs)	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

Pipeline functionality

serial_connect(*filters[, as_data, update, port])	Connect filters serially.
get_output(ftr[, as_data, update, port])	Get output from filter.
to_data(ftr[, port])	Extract data from filter.
connect(ftr0, ftr1[, port0, port1, add_conn])	Connection of two filters.

brainspace.vtk_interface.pipeline.serial_connect

brainspace.vtk_interface.pipeline.**serial_connect**(**filters*, *as_data=True*, *update=True*, *port=0*)

Connect filters serially.

Parameters

- ***filters** (*sequence of tuple or list*) – Input filters to serially connect. Each input takes one of the following formats:

  1. First filter in sequence: `(f0, op=0)`
     - *f0* (vtkAlgorithm, *BSAlgorithm*, vtkDataObject or *BSDataObject*) - This is the first filter.
     - *op* (int, optional) - This is the output port of *f0*. Default is 0.
  2. Last filter in sequence: `(ic=None, ip=0, fn)`
     - *ic* (int, optional) - This is the input connection of the input port *ip* of filter *fn*. Default is None.
     - *ip* (int, optional) - This is the input port of *fn*. Must be specified when *ic* is not None. Default is 0.
     - *fn* (vtkAlgorithm or *BSAlgorithm*) - This is the last filter.
  3. Intermediate filters: `(ic=None, ip=0, fi, op=0)`
     - *ic* (int, optional) - This is the input connection of the input port *ip* of filter *fi*. Default is None.
     - *ip* (int, optional) - This is the input port of *fi*. Must be specified when *ic* is not None. Default is 0.
     - *fi* (vtkAlgorithm or *BSAlgorithm*) - This is a filter.
     - *op* (int, optional) - This is the output port of *fi*. Default is 0.

- **as_data** (*bool, optional*) – Return data instead of filter. If True, last filter is automatically updated. Default is True.
- **update** (*bool, optional*) – Update last filter. Only used when `as_data == False`. Default is True.
- **port** (*int, optional*) – Port to update or get data from. When port is -1, refers to all ports. Default is 0.

Returns **output** (*BSAlgorithm or BSDataObject*) – Last filter or its output.

Examples

In VTK:

```
>>> # point source
>>> ps = vtk.vtkPointSource()
>>> ps.SetNumberOfPoints(100)
>>> # delauny
>>> dn = vtk.vtkDelaunay2D()
>>> dn.SetTolerance(0.01)
>>> dn.SetInputConnection(0, ps.GetOutputPort(0))
>>> # smooth
>>> sf = vtk.vtkWindowedSincPolyDataFilter()
>>> sf.SetInputConnection(0, dn.GetOutputPort(0))
>>> sf.SetNumberOfIterations(20)
>>> # update and get output
>>> sf.Update()
```

(continues on next page)

(continued from previous page)

```
>>> sf.GetOutput(0)
(vtkCommonDataModelPython.vtkPolyData)0x7f0134fffb28
```

With *serial_connect* function:

```
>>> from brainspace.vtk_interface.pipeline import serial_connect
>>> # point source
>>> ps = vtk.vtkPointSource()
>>> ps.SetNumberOfPoints(100)
>>> # delauny
>>> dn = vtk.vtkDelaunay2D()
>>> dn.SetTolerance(0.01)
>>> # smooth
>>> sf = vtk.vtkWindowedSincPolyDataFilter()
>>> sf.SetNumberOfIterations(20)
>>> # Connection
>>> serial_connect((ps, 0), (None, 0, dn, 0), (None, 0, sf), as_data=True,
...                port=0)
<brainspace.vtk_interface.wrappers.BSPolyData at 0x7f0134efb048>
>>> # This can be shortened, since no input connection is needed
>>> serial_connect((ps, 0), (0, dn, 0), (0, sf), as_data=True, port=0)
<brainspace.vtk_interface.wrappers.BSPolyData at 0x7f0134ee9128>
>>> # And shortened even further since the default input and output
>>> # ports are 0
>>> serial_connect((ps,), (dn,), (sf,), as_data=True, port=0)
<brainspace.vtk_interface.wrappers.BSPolyData at 0x7f0134ee92b0>
>>> # This is the same
>>> serial_connect(ps, dn, sf)
<brainspace.vtk_interface.wrappers.BSPolyData at 0x7f0134eee898>
```

brainspace.vtk_interface.pipeline.get_output

brainspace.vtk_interface.pipeline.**get_output**(*ftr*, *as_data=True*, *update=True*, *port=0*)

Get output from filter.

Parameters

- **ftr** (vtkAlgorithm or *BSAlgorithm*) – Input filter.
- **as_data** (*bool, optional*) – Return data as BSDataObject instead of *BSAlgorithm*. If True, the filter is automatically updated. Default is True.
- **update** (*bool, optional*) – Update filter. Only used when *as_data* is False. Default is True.
- **port** (*int or None, optional*) – Output port to update or get data from. Only used when input is vtkAlgorithm. When port is -1, refers to all ports. When None, call Update() with no arguments. Not used, when *ftr* is a sink (i.e., 0 output ports), call Update(). Default is 0.

Returns **poly** (*BSAlgorithm or BSDataObject*) – Returns filter or its output. If port is -1, returns all outputs in a list if `as_data == True`.

brainspace.vtk_interface.pipeline.to_data

brainspace.vtk_interface.pipeline.**to_data**(*ftr*, *port=0*)

Extract data from filter.

Parameters

- **ftr** (vtkAlgorithm or *BSAlgorithm*) – Input filter.
- **port** (*int, optional*) – Port to get data from. When port is -1, refers to all ports. Default is 0.

Returns **data** (*BSDataObject or list of BSDataObject*) – Returns the output of the filter. If port is -1 and number of output ports > 1, then return list of outputs.

Notes

Filters are automatically updated to get the output.

brainspace.vtk_interface.pipeline.connect

brainspace.vtk_interface.pipeline.**connect**(*ftr0*, *ftr1*, *port0=0*, *port1=0*, *add_conn=False*)

Connection of two filters.

Connects the output port *port0* of filter *ftr0* with the input port *port1* of filter *ftr1*.

Parameters

- **ftr0** (*vtkAlgorithm, vtkDataSet, BSAlgorithm or BSDataSet*) – The input filter. May be a filter or dataset.
- **ftr1** (*vtkAlgorithm or BSAlgorithm*) – The output filter.
- **port0** (*int, optional*) – Output port of *ftr0*. Not used if *ftr0* is a dataset. Default is 0.
- **port1** (*int, optional*) – Input port of *ftr1*. Default is 0.
- **add_conn** (*bool or int, optional*) – Connect to specific connection of *port1*. If False, use *SetInputConnection* or *SetInputData* (all other added connections to *port1* are removed). Otherwise, use *AddInputConnection* or *AddInputData*. If int, add to given connection (e.g., *SetInputConnectionByNumber* or *SetInputDataByNumber*). Only used if *port1* accepts more than one connection (i.e., repeatable). Default is False.

Returns **ftr1** (*BSAlgorithm*) – Returns (wrapped) *frt1* after connecting it with the input filter.

VTK wrappers

Data objects

BSDataObject([vtkobject])	Wrapper for vtkDataObject.
BSTable([vtkobject])	Wrapper for vtkTable.
BSCompositeDataSet([vtkobject])	Wrapper for vtkCompositeDataSet.
BSDataSet([vtkobject])	Wrapper for vtkDataSet.
BSPointSet([vtkobject])	Wrapper for vtkPointSet.

Continued on next page

Table 41 – continued from previous page

BSPolyData([vtkobject])	Wrapper for vtkPolyData.
BSUnstructuredGrid([vtkobject])	Wrapper for vtkUnstructuredGrid.

brainspace.vtk_interface.wrappers.data_object.BSDataObject

class brainspace.vtk_interface.wrappers.data_object.**BSDataObject**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkDataObject.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

FieldData	This property returns the field data of a data object.
field_keys	Returns keys of field data.
n_field_data	Returns number of entries in field data.
vtk_map	Dictionary of vtk setter and getter methods.

FieldData
This property returns the field data of a data object.

GetAttributes(*type*)
Returns the attributes specified by the type as a DataSetAttributes instance.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

field_keys
Returns keys of field data.

Type list of str

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_field_data
Returns number of entries in field data.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSTable

class brainspace.vtk_interface.wrappers.data_object.**BSTable**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkTable.

__init__ (*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetRowData()	Returns the row data as a DataSetAttributes instance.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

FieldData	This property returns the field data of a data object.
RowData	This property returns the row data of the table.
field_keys	Returns keys of field data.
n_field_data	Returns number of entries in field data.
vtk_map	Dictionary of vtk setter and getter methods.

FieldData
This property returns the field data of a data object.

GetAttributes (*type*)
Returns the attributes specified by the type as a DataSetAttributes instance.

GetFieldData ()
Returns the field data as a DataSetAttributes instance.

GetRowData ()
Returns the row data as a DataSetAttributes instance.

RowData
This property returns the row data of the table.

field_keys
Returns keys of field data.

Type list of str

getVTK (**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_field_data
Returns number of entries in field data.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSCompositeDataSet

class brainspace.vtk_interface.wrappers.data_object.**BSCompositeDataSet**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkCompositeDataSet.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a CompositeDataSetAttributes instance.
GetCellData()	Returns the cell data as a DataSetAttributes instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetNumberOfCells()	Returns the total number of cells of all datasets in the composite dataset.
GetNumberOfElements(assoc)	Returns the total number of cells or points depending on the value of assoc which can be ArrayAssociation.POINT or ArrayAssociation.CELL.
GetNumberOfPoints()	Returns the total number of points of all datasets in the composite dataset.
GetPointData()	Returns the point data as a DataSetAttributes instance.
GetPoints()	Returns the points as a VTKCompositeDataArray instance.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

CellData	This property returns the cell data of a dataset.
FieldData	This property returns the field data of a dataset.
PointData	This property returns the point data of the dataset.
Points	This property returns the points of the dataset.
field_keys	Returns keys of field data.
n_field_data	Returns number of entries in field data.
vtk_map	Dictionary of vtk setter and getter methods.

CellData
This property returns the cell data of a dataset.

FieldData
This property returns the field data of a dataset.

GetAttributes(*type*)
Returns the attributes specified by the type as a CompositeDataSetAttributes instance.

GetCellData()
Returns the cell data as a DataSetAttributes instance.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

GetNumberOfCells()
Returns the total number of cells of all datasets in the composite dataset. Note that this traverses the whole composite dataset every time and should not be called repeatedly for large composite datasets.

GetNumberOfElements(*assoc*)
Returns the total number of cells or points depending on the value of assoc which can be ArrayAssociation.POINT or ArrayAssociation.CELL.

GetNumberOfPoints()
Returns the total number of points of all datasets in the composite dataset. Note that this traverses the whole composite dataset every time and should not be called repeatedly for large composite datasets.

GetPointData()
Returns the point data as a DataSetAttributes instance.

GetPoints()
Returns the points as a VTKCompositeDataArray instance.

PointData
This property returns the point data of the dataset.

Points
This property returns the points of the dataset.

field_keys
Returns keys of field data.

Type list of str

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_field_data
Returns number of entries in field data.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSDataSet

class brainspace.vtk_interface.wrappers.data_object.**BSDataSet**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkDataSet.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetCellData()	Returns the cell data as a DataSetAttributes instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetPointData()	Returns the point data as a DataSetAttributes instance.
__init__([vtkobject])	Initialize self.
append_array(array[, name, at, ...])	Append array to attributes.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
get_array([name, at, return_name])	Return array in attributes.
remove_array([name, at])	Remove array from vtk dataset.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

CellData	This property returns the cell data of a dataset.
FieldData	This property returns the field data of a data object.
PointData	This property returns the point data of the dataset.
cell_keys	Returns keys of cell data.

Continued on next page

Table 49 – continued from previous page

cell_types	Returns cell types of the object.
field_keys	Returns keys of field data.
has_only_line	Returns True if object has only lines.
has_only_quad	Returns True if object has only quad cells.
has_only_triangle	Returns True if object has only triangles.
has_only_vertex	Returns True if object has only vertex cells.
has_unique_cell_type	Returns True if object has a unique cell type.
n_cell_data	Returns number of entries in cell data.
n_cells	Returns number of cells.
n_field_data	Returns number of entries in field data.
n_point_data	Returns number of entries in point data.
n_points	Returns number of points.
number_of_cell_types	Returns number of cell types.
point_keys	Returns keys of point data.
vtk_map	Dictionary of vtk setter and getter methods.

CellData
This property returns the cell data of a dataset.

FieldData
This property returns the field data of a data object.

GetAttributes(*type*)
Returns the attributes specified by the type as a DataSetAttributes instance.

GetCellData()
Returns the cell data as a DataSetAttributes instance.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

GetPointData()
Returns the point data as a DataSetAttributes instance.

PointData
This property returns the point data of the dataset.

append_array(*array*, *name=None*, *at=None*, *convert_bool='warn'*, *overwrite='warn'*)
Append array to attributes.

Parameters

- **array** (*1D or 2D ndarray*) – Array to append to the dataset.
- **name** (*str or None, optional*) – Array name. If None, a random string is generated and returned. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attribute to append data to. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f') data. If None, it will attempt to append data to the attributes with the same number of elements. Only considers points and cells. If both have the same number of elements or the size of the array does not coincide with any of them, it raises an exception. Default is None.
- **convert_bool** (*bool or {'warn', 'raise'}, optional*) – If True append array after conversion to uint8. If False, array is not appended. If 'warn', issue a warning but append the array. If raise, raise an exception.

- **overwrite** (*bool or {'warn', 'raise'}, optional*) – If True append array even if its name already exists. If False, array is not appended, issue a warning. If 'warn', issue a warning but append the array. If raise, raise an exception.

Returns **name** (*str*) – Array name used to append the array to the dataset.

cell_keys
Returns keys of cell data.

Type list of str

cell_types
Returns cell types of the object.

Type list of int

field_keys
Returns keys of field data.

Type list of str

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

get_array(*name=None*, *at=None*, *return_name=False*)
Return array in attributes.

Parameters

- **name** (*str, list of str or None, optional*) – Array names. If None, return all arrays. Cannot be None if `at == None`. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to get the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, get array name from all attributes that have an array with the same array name. Cannot be None if `name == None`. Default is None.

- **return_name** (*bool, optional*) – Whether to return array names too. Default is False.

Returns

- **arrays** (*VTKArray or list of VTKArray*) – Data arrays. None is returned if *name* does not exist.
- **names** (*str or list of str*) – Names of returned arrays. Only if `return_name == True`.

has_only_line
Returns True if object has only lines. False, otherwise.

Type bool

has_only_quad
Returns True if object has only quad cells. False, otherwise.

Type bool

has_only_triangle
Returns True if object has only triangles. False, otherwise.

Type bool

has_only_vertex
Returns True if object has only vertex cells. False, otherwise.

Type bool

has_unique_cell_type
Returns True if object has a unique cell type. False, otherwise.

Type bool

n_cell_data
Returns number of entries in cell data.

Type int

n_cells
Returns number of cells.

Type int

n_field_data
Returns number of entries in field data.

Type int

n_point_data
Returns number of entries in point data.

Type int

n_points
Returns number of points.

Type int

number_of_cell_types
Returns number of cell types.

Type int

point_keys
Returns keys of point data.

Type list of str

remove_array(*name=None*, *at=None*)
Remove array from vtk dataset.

Parameters

- **name** (*str, list of str or None, optional*) – Array name to remove. If None, remove all arrays. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to remove the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, remove array name from all attributes. Default is None.

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSPointSet

class brainspace.vtk_interface.wrappers.data_object.**BSPointSet**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkPointSet.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetCellData()	Returns the cell data as a DataSetAttributes instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetPointData()	Returns the point data as a DataSetAttributes instance.
GetPoints()	Returns the points as a VTKArray instance.
SetPoints(pts)	Given a VTKArray instance, sets the points of the dataset.
__init__([vtkobject])	Initialize self.
append_array(array[, name, at, ...])	Append array to attributes.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
get_array([name, at, return_name])	Return array in attributes.
remove_array([name, at])	Remove array from vtk dataset.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

CellData	This property returns the cell data of a dataset.
FieldData	This property returns the field data of a data object.
PointData	This property returns the point data of the dataset.
Points	This property returns the point coordinates of dataset.
cell_keys	Returns keys of cell data.
cell_types	Returns cell types of the object.
field_keys	Returns keys of field data.
has_only_line	Returns True if object has only lines.
has_only_quad	Returns True if object has only quad cells.
has_only_triangle	Returns True if object has only triangles.
has_only_vertex	Returns True if object has only vertex cells.
has_unique_cell_type	Returns True if object has a unique cell type.
n_cell_data	Returns number of entries in cell data.
n_cells	Returns number of cells.
n_field_data	Returns number of entries in field data.
n_point_data	Returns number of entries in point data.
n_points	Returns number of points.
number_of_cell_types	Returns number of cell types.
point_keys	Returns keys of point data.
vtk_map	Dictionary of vtk setter and getter methods.

CellData
: This property returns the cell data of a dataset.

FieldData
: This property returns the field data of a data object.

GetAttributes(*type*)
: Returns the attributes specified by the type as a DataSetAttributes instance.

GetCellData()
: Returns the cell data as a DataSetAttributes instance.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

GetPointData()
Returns the point data as a DataSetAttributes instance.

GetPoints()
Returns the points as a VTKArray instance. Returns None if the dataset has implicit points.

PointData
This property returns the point data of the dataset.

Points
This property returns the point coordinates of dataset.

SetPoints(*pts*)
Given a VTKArray instance, sets the points of the dataset.

append_array(*array*, *name=None*, *at=None*, *convert_bool='warn'*, *overwrite='warn'*)
Append array to attributes.

Parameters

- **array** (*1D or 2D ndarray*) – Array to append to the dataset.
- **name** (*str or None, optional*) – Array name. If None, a random string is generated and returned. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attribute to append data to. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f') data. If None, it will attempt to append data to the attributes with the same number of elements. Only considers points and cells. If both have the same number of elements or the size of the array does not coincide with any of them, it raises an exception. Default is None.
- **convert_bool** (*bool or {'warn', 'raise'}, optional*) – If True append array after conversion to uint8. If False, array is not appended. If 'warn', issue a warning but append the array. If raise, raise an exception.
- **overwrite** (*bool or {'warn', 'raise'}, optional*) – If True append array even if its name already exists. If False, array is not appended, issue a warning. If 'warn', issue a warning but append the array. If raise, raise an exception.

Returns **name** (*str*) – Array name used to append the array to the dataset.

cell_keys
Returns keys of cell data.

Type list of str

cell_types
Returns cell types of the object.

Type list of int

field_keys
Returns keys of field data.

Type list of str

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

get_array(*name=None*, *at=None*, *return_name=False*)
Return array in attributes.

Parameters

- **name** (*str, list of str or None, optional*) – Array names. If None, return all arrays. Cannot be None if `at == None`. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to get the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, get array name from all attributes that have an array with the same array name. Cannot be None if `name == None`. Default is None.
- **return_name** (*bool, optional*) – Whether to return array names too. Default is False.

Returns

- **arrays** (*VTKArray or list of VTKArray*) – Data arrays. None is returned if *name* does not exist.
- **names** (*str or list of str*) – Names of returned arrays. Only if `return_name == True`.

has_only_line
Returns True if object has only lines. False, otherwise.

Type bool

has_only_quad
Returns True if object has only quad cells. False, otherwise.

Type bool

has_only_triangle
Returns True if object has only triangles. False, otherwise.

Type bool

has_only_vertex
Returns True if object has only vertex cells. False, otherwise.

Type bool

has_unique_cell_type
Returns True if object has a unique cell type. False, otherwise.

Type bool

n_cell_data
Returns number of entries in cell data.

Type int

n_cells
Returns number of cells.

Type int

n_field_data
Returns number of entries in field data.

Type int

n_point_data
Returns number of entries in point data.

Type int

n_points
Returns number of points.

Type int

number_of_cell_types
Returns number of cell types.

Type int

point_keys
Returns keys of point data.

Type list of str

remove_array(*name=None*, *at=None*)
Remove array from vtk dataset.

Parameters

- **name** (*str, list of str or None, optional*) – Array name to remove. If None, remove all arrays. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to remove the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, remove array name from all attributes. Default is None.

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSPolyData

class brainspace.vtk_interface.wrappers.data_object.**BSPolyData**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyData.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetCellData()	Returns the cell data as a DataSetAttributes instance.
GetCells2D()	Return cells as a 2D ndarray.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetLines()	Returns the lines as a 1D VTKArray instance.
GetLines2D()	Returns the lines as a 2D VTKArray instance.
GetPointData()	Returns the point data as a DataSetAttributes instance.
GetPoints()	Returns the points as a VTKArray instance.
GetPolygons()	Returns the polys as a VTKArray instance.
GetPolys()	Returns the polys as a 1D VTKArray instance.
GetPolys2D()	Returns the polys as a 2D VTKArray instance.
GetVerts()	Returns the verts as a 1D VTKArray instance.
GetVerts2D()	Returns the verts as a 2D VTKArray instance.
SetLines(lines)	Set lines.
SetPoints(pts)	Given a VTKArray instance, sets the points of the dataset.
SetPolys(polys)	Set polys.
SetVerts(verts)	Set verts.

Continued on next page

Table 52 – continued from previous page

__init__([vtkobject])	Initialize self.
append_array(array[, name, at, ...])	Append array to attributes.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
get_array([name, at, return_name])	Return array in attributes.
remove_array([name, at])	Remove array from vtk dataset.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

CellData	This property returns the cell data of a dataset.
FieldData	This property returns the field data of a data object.
PointData	This property returns the point data of the dataset.
Points	This property returns the point coordinates of dataset.
Polygons	This property returns the connectivity of polygons.
cell_keys	Returns keys of cell data.
cell_types	Returns cell types of the object.
field_keys	Returns keys of field data.
has_only_line	Returns True if object has only lines.
has_only_quad	Returns True if object has only quad cells.
has_only_triangle	Returns True if object has only triangles.
has_only_vertex	Returns True if object has only vertex cells.
has_unique_cell_type	Returns True if object has a unique cell type.
lines	Return lines as a 1D VTKArray.
lines2D	Return lines as a 2D VTKArray if possible.
n_cell_data	Returns number of entries in cell data.
n_cells	Returns number of cells.
n_field_data	Returns number of entries in field data.
n_point_data	Returns number of entries in point data.
n_points	Returns number of points.
number_of_cell_types	Returns number of cell types.
point_keys	Returns keys of point data.
polys	Return polys as a 1D VTKArray.
polys2D	Return polys as a 2D VTKArray if possible.
verts	Return verts as a 1D VTKArray.
verts2D	Return verts as a 2D VTKArray if possible.
vtk_map	Dictionary of vtk setter and getter methods.

CellData
This property returns the cell data of a dataset.

FieldData
This property returns the field data of a data object.

GetAttributes(*type*)
Returns the attributes specified by the type as a DataSetAttributes instance.

GetCellData()
Returns the cell data as a DataSetAttributes instance.

GetCells2D()
Return cells as a 2D ndarray.

Returns **cells** (*2D ndarray, shape = (n_points, n)*) – PolyData cells.

Raises `ValueError` – If PolyData has different cell types.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

GetLines()
Returns the lines as a 1D VTKArray instance.

GetLines2D()
Returns the lines as a 2D VTKArray instance.

Returns **lines** (*2D ndarray, shape = (n_points, n)*) – PolyData lines.

Raises `ValueError` – If PolyData has different line types.

GetPointData()
Returns the point data as a DataSetAttributes instance.

GetPoints()
Returns the points as a VTKArray instance. Returns None if the dataset has implicit points.

GetPolygons()
Returns the polys as a VTKArray instance.

GetPolys()
Returns the polys as a 1D VTKArray instance.

GetPolys2D()
Returns the polys as a 2D VTKArray instance.

Returns **polys** (*2D ndarray, shape = (n_points, n)*) – PolyData polys.

Raises `ValueError` – If PolyData has different poly types.

GetVerts()
Returns the verts as a 1D VTKArray instance.

GetVerts2D()
Returns the verts as a 2D VTKArray instance.

Returns **verts** (*2D ndarray, shape = (n_points, n)*) – PolyData verts.

Raises `ValueError` – If PolyData has different vertex types.

PointData
This property returns the point data of the dataset.

Points
This property returns the point coordinates of dataset.

Polygons
This property returns the connectivity of polygons.

SetLines(*lines*)
Set lines.

Parameters **lines** (*1D or 2D ndarray*) – If 2D, shape = (n_points, n), and n is the number of points per line. All lines must use the same number of points.

SetPoints(*pts*)
Given a VTKArray instance, sets the points of the dataset.

SetPolys(*polys*)
Set polys.

Parameters **polys** (*1D or 2D ndarray*) – If 2D, shape = (n_points, n), and n is the number of points per poly. All polys must use the same number of points.

SetVerts (*verts*)
Set verts.

Parameters **verts** (*1D or 2D ndarray*) – If 2D, shape = (n_points, n), and n is the number of points per vertex. All verts must use the same number of points.

append_array (*array*, *name=None*, *at=None*, *convert_bool='warn'*, *overwrite='warn'*)
Append array to attributes.

Parameters

- **array** (*1D or 2D ndarray*) – Array to append to the dataset.
- **name** (*str or None, optional*) – Array name. If None, a random string is generated and returned. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attribute to append data to. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f') data. If None, it will attempt to append data to the attributes with the same number of elements. Only considers points and cells. If both have the same number of elements or the size of the array does not coincide with any of them, it raises an exception. Default is None.
- **convert_bool** (*bool or {'warn', 'raise'}, optional*) – If True append array after conversion to uint8. If False, array is not appended. If 'warn', issue a warning but append the array. If raise, raise an exception.
- **overwrite** (*bool or {'warn', 'raise'}, optional*) – If True append array even if its name already exists. If False, array is not appended, issue a warning. If 'warn', issue a warning but append the array. If raise, raise an exception.

Returns **name** (*str*) – Array name used to append the array to the dataset.

cell_keys
Returns keys of cell data.

Type list of str

cell_types
Returns cell types of the object.

Type list of int

field_keys
Returns keys of field data.

Type list of str

getVTK (**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

get_array (*name=None*, *at=None*, *return_name=False*)
Return array in attributes.

Parameters

- **name** (*str, list of str or None, optional*) – Array names. If None, return all arrays. Cannot be None if `at == None`. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to get the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, get array name from all attributes that have an array with the same array name. Cannot be None if `name == None`. Default is None.
- **return_name** (*bool, optional*) – Whether to return array names too. Default is False.

Returns

- **arrays** (*VTKArray or list of VTKArray*) – Data arrays. None is returned if *name* does not exist.
- **names** (*str or list of str*) – Names of returned arrays. Only if `return_name == True`.

has_only_line
Returns True if object has only lines. False, otherwise.

Type bool

has_only_quad
Returns True if object has only quad cells. False, otherwise.

Type bool

has_only_triangle
Returns True if object has only triangles. False, otherwise.

Type bool

has_only_vertex
Returns True if object has only vertex cells. False, otherwise.

Type bool

has_unique_cell_type
Returns True if object has a unique cell type. False, otherwise.

Type bool

lines
Return lines as a 1D VTKArray.

lines2D
Return lines as a 2D VTKArray if possible.

n_cell_data
Returns number of entries in cell data.

Type int

n_cells
Returns number of cells.

Type int

n_field_data
Returns number of entries in field data.

Type int

n_point_data
Returns number of entries in point data.

Type int

n_points
Returns number of points.

Type int

number_of_cell_types
Returns number of cell types.

Type int

point_keys
Returns keys of point data.

Type list of str

polys
Return polys as a 1D VTKArray.

polys2D
Return polys as a 2D VTKArray if possible.

remove_array(*name=None*, *at=None*)
Remove array from vtk dataset.

Parameters

- **name** (*str, list of str or None, optional*) – Array name to remove. If None, remove all arrays. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to remove the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, remove array name from all attributes. Default is None.

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.

- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

verts
Return verts as a 1D VTKArray.

verts2D
Return verts as a 2D VTKArray if possible.

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.data_object.BSUnstructuredGrid

class brainspace.vtk_interface.wrappers.data_object.**BSUnstructuredGrid**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkUnstructuredGrid.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetAttributes(type)	Returns the attributes specified by the type as a DataSetAttributes instance.
GetCellData()	Returns the cell data as a DataSetAttributes instance.
GetCellLocations()	Returns the cell locations as a VTKArray instance.
GetCellTypes()	Returns the cell types as a VTKArray instance.
GetCells()	Returns the cells as a VTKArray instance.
GetFieldData()	Returns the field data as a DataSetAttributes instance.
GetPointData()	Returns the point data as a DataSetAttributes instance.
GetPoints()	Returns the points as a VTKArray instance.
SetCells(cellTypes, cellLocations, cells)	Given cellTypes, cellLocations, cells as VTKArrays, populates the unstructured grid data structures.

Continued on next page

Table 54 – continued from previous page

SetPoints(pts)	Given a VTKArray instance, sets the points of the dataset.
__init__([vtkobject])	Initialize self.
append_array(array[, name, at, ...])	Append array to attributes.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
get_array([name, at, return_name])	Return array in attributes.
remove_array([name, at])	Remove array from vtk dataset.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

CellData	This property returns the cell data of a dataset.
CellLocations	This property returns the locations of cells.
CellTypes	This property returns the types of cells.
Cells	This property returns the connectivity of cells.
FieldData	This property returns the field data of a data object.
PointData	This property returns the point data of the dataset.
Points	This property returns the point coordinates of dataset.
cell_keys	Returns keys of cell data.
cell_types	Returns cell types of the object.
field_keys	Returns keys of field data.
has_only_line	Returns True if object has only lines.
has_only_quad	Returns True if object has only quad cells.
has_only_triangle	Returns True if object has only triangles.
has_only_vertex	Returns True if object has only vertex cells.
has_unique_cell_type	Returns True if object has a unique cell type.
n_cell_data	Returns number of entries in cell data.
n_cells	Returns number of cells.
n_field_data	Returns number of entries in field data.
n_point_data	Returns number of entries in point data.
n_points	Returns number of points.
number_of_cell_types	Returns number of cell types.
point_keys	Returns keys of point data.
vtk_map	Dictionary of vtk setter and getter methods.

CellData
This property returns the cell data of a dataset.

CellLocations
This property returns the locations of cells.

CellTypes
This property returns the types of cells.

Cells
This property returns the connectivity of cells.

FieldData
This property returns the field data of a data object.

GetAttributes(*type*)
Returns the attributes specified by the type as a DataSetAttributes instance.

GetCellData()
Returns the cell data as a DataSetAttributes instance.

GetCellLocations()
Returns the cell locations as a VTKArray instance.

GetCellTypes()
Returns the cell types as a VTKArray instance.

GetCells()
Returns the cells as a VTKArray instance.

GetFieldData()
Returns the field data as a DataSetAttributes instance.

GetPointData()
Returns the point data as a DataSetAttributes instance.

GetPoints()
Returns the points as a VTKArray instance. Returns None if the dataset has implicit points.

PointData
This property returns the point data of the dataset.

Points
This property returns the point coordinates of dataset.

SetCells(*cellTypes*, *cellLocations*, *cells*)
Given cellTypes, cellLocations, cells as VTKArrays, populates the unstructured grid data structures.

SetPoints(*pts*)
Given a VTKArray instance, sets the points of the dataset.

append_array(*array*, *name=None*, *at=None*, *convert_bool='warn'*, *overwrite='warn'*)
Append array to attributes.

Parameters

- **array** (*1D or 2D ndarray*) – Array to append to the dataset.
- **name** (*str or None, optional*) – Array name. If None, a random string is generated and returned. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attribute to append data to. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f') data. If None, it will attempt to append data to the attributes with the same number of elements. Only considers points and cells. If both have the same number of elements or the size of the array does not coincide with any of them, it raises an exception. Default is None.
- **convert_bool** (*bool or {'warn', 'raise'}, optional*) – If True append array after conversion to uint8. If False, array is not appended. If 'warn', issue a warning but append the array. If raise, raise an exception.
- **overwrite** (*bool or {'warn', 'raise'}, optional*) – If True append array even if its name already exists. If False, array is not appended, issue a warning. If 'warn', issue a warning but append the array. If raise, raise an exception.

Returns **name** (*str*) – Array name used to append the array to the dataset.

cell_keys
Returns keys of cell data.

Type list of str

cell_types
Returns cell types of the object.

Type list of int

field_keys
Returns keys of field data.

Type list of str

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

get_array(*name=None*, *at=None*, *return_name=False*)
Return array in attributes.

Parameters

- **name** (*str, list of str or None, optional*) – Array names. If None, return all arrays. Cannot be None if `at == None`. Default is None.
- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to get the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, get array name from all attributes that have an array with the same array name. Cannot be None if `name == None`. Default is None.
- **return_name** (*bool, optional*) – Whether to return array names too. Default is False.

Returns

- **arrays** (*VTKArray or list of VTKArray*) – Data arrays. None is returned if *name* does not exist.
- **names** (*str or list of str*) – Names of returned arrays. Only if `return_name == True`.

has_only_line
Returns True if object has only lines. False, otherwise.

Type bool

has_only_quad
Returns True if object has only quad cells. False, otherwise.

Type bool

has_only_triangle
Returns True if object has only triangles. False, otherwise.

Type bool

has_only_vertex
Returns True if object has only vertex cells. False, otherwise.

Type bool

has_unique_cell_type
Returns True if object has a unique cell type. False, otherwise.

Type bool

n_cell_data
Returns number of entries in cell data.

Type int

n_cells
Returns number of cells.

Type int

n_field_data
Returns number of entries in field data.

Type int

n_point_data
Returns number of entries in point data.

Type int

n_points
Returns number of points.

Type int

number_of_cell_types
Returns number of cell types.

Type int

point_keys
Returns keys of point data.

Type list of str

remove_array(*name=None*, *at=None*)
Remove array from vtk dataset.

Parameters

- **name** (*str, list of str or None, optional*) – Array name to remove. If None, remove all arrays. Default is None.

- **at** (*{'point', 'cell', 'field', 'p', 'c', 'f'} or None, optional.*) – Attributes to remove the array from. Points (i.e., 'point' or 'p'), cells (i.e., 'cell' or 'c') or field (i.e., 'field' or 'f'). If None, remove array name from all attributes. Default is None.

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

Algorithms

BSAlgorithm([vtkobject])	Wrapper for vtkAlgorithm.
BSPolyDataAlgorithm([vtkobject])	Wrapper for vtkPolyDataAlgorithm.
BSWindowToImageFilter([vtkobject])	Wrapper for vtkWindowToImageFilter.
BSImageWriter([vtkobject])	Wrapper for vtkImageWriter.
BSBMPWriter([vtkobject])	Wrapper for vtkBMPWriter.
BSJPEGWriter([vtkobject])	Wrapper for vtkJPEGWriter.
BSPNGWriter([vtkobject])	Wrapper for vtkPNGWriter.
BSPostScriptWriter([vtkobject])	Wrapper for vtkPostScriptWriter.
BSTIFFWriter([vtkobject])	Wrapper for vtkTIFFWriter.

brainspace.vtk_interface.wrappers.algorithm.BSAlgorithm

class brainspace.vtk_interface.wrappers.algorithm.**BSAlgorithm**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkAlgorithm.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSPolyDataAlgorithm

class brainspace.vtk_interface.wrappers.algorithm.**BSPolyDataAlgorithm**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyDataAlgorithm.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSWindowToImageFilter

class brainspace.vtk_interface.wrappers.algorithm.**BSWindowToImageFilter**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkWindowToImageFilter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSImageWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSImageWriter**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkImageWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSBMPWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSBMPWriter**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkBMPWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSJPEGWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSJPEGWriter**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkJPEGWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSPNGWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSPNGWriter**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPNGWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSPostScriptWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSPostScriptWriter**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPostScriptWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSTIFFWriter

class brainspace.vtk_interface.wrappers.algorithm.**BSTIFFWriter**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkTIFFWriter.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter

Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink

Returns True if self is a sink. False, otherwise.

Type bool

is_source

Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

Mappers

BSDataSetMapper([vtkobject])	Wrapper for vtkDataSetMapper.
BSPolyDataMapper([vtkobject])	Wrapper for vtkPolyDataMapper.
BSLabeledContourMapper([vtkobject])	Wrapper for vtkLabeledContourMapper.
BSLabeledDataMapper([vtkobject])	Wrapper for vtkLabeledDataMapper.
BSLabelPlacementMapper([vtkobject])	Wrapper for vtkLabelPlacementMapper.
BSPolyDataMapper2D([vtkobject])	Wrapper for vtkPolyDataMapper2D.
BSTextMapper2D([vtkobject])	Wrapper for vtkPolyDataMapper2D.

brainspace.vtk_interface.wrappers.algorithm.BSDataSetMapper

class brainspace.vtk_interface.wrappers.algorithm.**BSDataSetMapper**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkDataSetMapper.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetArrayId(idx)	Set array id.
SetArrayName(name)	Set array id.
SetColorTransferFunction([obj])	Set lookup table using a ColorTransferFunction.
SetDiscretizableColorTransferFunction([obj])	Set lookup table using a DiscretizableColorTransferFunction.
SetLookupTable([obj])	Set lookup table.
SetLookupTableWithEnabling([obj])	Set lookup table using a LookupTableWithEnabling.
SetWindowLevelLookupTable([obj])	Set lookup table using a WindowLevelLookupTable.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

SetArrayId(*idx*)
Set array id.

Wraps the *SetArrayId* method of *vtkMapper* such that the access mode is changed to accept setting the array id.

Parameters **idx** (*int*) – Array id.

SetArrayName(*name*)
Set array id.

Wraps the *SetArrayName* method of *vtkMapper* such that the access mode is changed to accept setting the array name.

Parameters **name** (*str*) – Array name.

SetColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a ColorTransferFunction.

Parameters

- **obj** (*vtkColorTransferFunction or BSColorTransferFunction, optional*) – Lookup table. If None, the color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetDiscretizableColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a DiscretizableColorTransferFunction.

Parameters

- **obj** (*vtkDiscretizableColorTransferFunction or*) – BSDiscretizableColorTransferFunction, optional Lookup table. If None, the discretizable color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTable(*obj=None*, ***kwargs*)
Set lookup table.

Wraps the *SetLookupTable* method of *vtkMapper* to accept a *vtkLookupTable* or BSLookupTable.

Parameters

- **obj** (*vtkLookupTable or BSLookupTable, optional*) – Lookup table. If None, a LookupTable is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTableWithEnabling(*obj=None*, ***kwargs*)
Set lookup table using a LookupTableWithEnabling.

Parameters

- **obj** (*vtkLookupTableWithEnabling or BSLookupTableWithEnabling, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetWindowLevelLookupTable(*obj=None*, ***kwargs*)
Set lookup table using a WindowLevelLookupTable.

Parameters

- **obj** (*vtkWindowLevelLookupTable or BSWindowLevelLookupTable, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSPolyDataMapper

class brainspace.vtk_interface.wrappers.algorithm.**BSPolyDataMapper**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyDataMapper.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetArrayId(idx)	Set array id.
SetArrayName(name)	Set array id.
SetColorTransferFunction([obj])	Set lookup table using a ColorTransferFunction.
SetDiscretizableColorTransferFunction([obj])	Set lookup table using a DiscretizableColorTransferFunction.
SetLookupTable([obj])	Set lookup table.
SetLookupTableWithEnabling([obj])	Set lookup table using a LookupTableWithEnabling.
SetWindowLevelLookupTable([obj])	Set lookup table using a WindowLevelLookupTable.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

SetArrayId(*idx*)
Set array id.

Wraps the *SetArrayId* method of *vtkMapper* such that the access mode is changed to accept setting the array id.

Parameters **idx** (*int*) – Array id.

SetArrayName(*name*)
Set array id.

Wraps the *SetArrayName* method of *vtkMapper* such that the access mode is changed to accept setting the array name.

Parameters **name** (*str*) – Array name.

SetColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a ColorTransferFunction.

Parameters

- **obj** (*vtkColorTransferFunction or BSColorTransferFunction, optional*) – Lookup table. If None, the color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetDiscretizableColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a DiscretizableColorTransferFunction.

Parameters

- **obj** (*vtkDiscretizableColorTransferFunction or*) – BSDiscretizableColorTransferFunction, optional Lookup table. If None, the discretizable color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTable(*obj=None*, ***kwargs*)
Set lookup table.

Wraps the *SetLookupTable* method of *vtkMapper* to accept a *vtkLookupTable* or BSLookupTable.

Parameters

- **obj** (*vtkLookupTable or BSLookupTable, optional*) – Lookup table. If None, a LookupTable is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTableWithEnabling(*obj=None*, ***kwargs*)
Set lookup table using a LookupTableWithEnabling.

Parameters

- **obj**(*vtkLookupTableWithEnabling or BSLookupTableWithEnabling, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetWindowLevelLookupTable(*obj=None*, ***kwargs*)
Set lookup table using a WindowLevelLookupTable.

Parameters

- **obj** (*vtkWindowLevelLookupTable or BSWindowLevelLookupTable, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSLabeledContourMapper

class brainspace.vtk_interface.wrappers.algorithm.**BSLabeledContourMapper**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkLabeledContourMapper.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetArrayId(idx)	Set array id.
SetArrayName(name)	Set array id.
SetColorTransferFunction([obj])	Set lookup table using a ColorTransferFunction.
SetDiscretizableColorTransferFunction([obj])	Set lookup table using a DiscretizableColorTransferFunction.
SetLookupTable([obj])	Set lookup table.
SetLookupTableWithEnabling([obj])	Set lookup table using a LookupTableWithEnabling.
SetTextProperty([obj])	Set text property.

Continued on next page

Table 80 – continued from previous page

SetTextPropertyMapping(mapping)	
SetWindowLevelLookupTable([obj])	Set lookup table using a WindowLevelLookupTable.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

SetArrayId(*idx*)
Set array id.

Wraps the *SetArrayId* method of *vtkMapper* such that the access mode is changed to accept setting the array id.

Parameters **idx** (*int*) – Array id.

SetArrayName(*name*)
Set array id.

Wraps the *SetArrayName* method of *vtkMapper* such that the access mode is changed to accept setting the array name.

Parameters **name** (*str*) – Array name.

SetColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a ColorTransferFunction.

Parameters

- **obj** (*vtkColorTransferFunction or BSColorTransferFunction, optional*) – Lookup table. If None, the color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetDiscretizableColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a DiscretizableColorTransferFunction.

Parameters

- **obj** (*vtkDiscretizableColorTransferFunction or*) – BSDiscretizableColorTransferFunction, optional Lookup table. If None, the discretizable color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTable(*obj=None*, ***kwargs*)
Set lookup table.

Wraps the *SetLookupTable* method of *vtkMapper* to accept a *vtkLookupTable* or BSLookupTable.

Parameters

- **obj** (*vtkLookupTable or BSLookupTable, optional*) – Lookup table. If None, a LookupTable is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTableWithEnabling(*obj=None*, ***kwargs*)
Set lookup table using a LookupTableWithEnabling.

Parameters

- **obj** (*vtkLookupTableWithEnabling or BSLookupTableWithEnabling, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetTextProperty(*obj=None*, ***kwargs*)
Set text property.

Wraps the *SetTextProperty* method of *vtkLabeledContourMapper* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Label text property. If None, the property is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the property.

SetTextPropertyMapping(*mapping*)

SetWindowLevelLookupTable(*obj=None*, ***kwargs*)
Set lookup table using a WindowLevelLookupTable.

Parameters

- **obj** (*vtkWindowLevelLookupTable or BSWindowLevelLookupTable, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSLabeledDataMapper

class brainspace.vtk_interface.wrappers.algorithm.**BSLabeledDataMapper**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkLabeledDataMapper.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSLabelPlacementMapper

class brainspace.vtk_interface.wrappers.algorithm.**BSLabelPlacementMapper**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkLabelPlacementMapper.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSPolyDataMapper2D

class brainspace.vtk_interface.wrappers.algorithm.**BSPolyDataMapper2D**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyDataMapper2D.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetColorTransferFunction([obj])	Set lookup table using a ColorTransferFunction.
SetDiscretizableColorTransferFunction([obj])	Set lookup table using a DiscretizableColorTransferFunction.
SetLookupTable([obj])	Set lookup table.
SetLookupTableWithEnabling([obj])	Set lookup table using a LookupTableWithEnabling.
SetWindowLevelLookupTable([obj])	Set lookup table using a WindowLevelLookupTable.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

SetColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a ColorTransferFunction.

Parameters

- **obj** (*vtkColorTransferFunction or BSColorTransferFunction*,

optional) – Lookup table. If None, the color transfer function is created. Default is None.

- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetDiscretizableColorTransferFunction(*obj=None*, ***kwargs*)
Set lookup table using a DiscretizableColorTransferFunction.

Parameters

- **obj** (*vtkDiscretizableColorTransferFunction or*) – BSDiscretizableColorTransferFunction, optional Lookup table. If None, the discretizable color transfer function is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTable(*obj=None*, ***kwargs*)
Set lookup table.

Wraps the *SetLookupTable* method of *vtkMapper* to accept a *vtkLookupTable* or BSLookupTable.

Parameters

- **obj** (*vtkLookupTable or BSLookupTable, optional*) – Lookup table. If None, a LookupTable is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetLookupTableWithEnabling(*obj=None*, ***kwargs*)
Set lookup table using a LookupTableWithEnabling.

Parameters

- **obj**(*vtkLookupTableWithEnabling or BSLookupTableWithEnabling, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

SetWindowLevelLookupTable(*obj=None*, ***kwargs*)
Set lookup table using a WindowLevelLookupTable.

Parameters

- **obj** (*vtkWindowLevelLookupTable or BSWindowLevelLookupTable, optional*) – Lookup table. If None, the lut is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the lookup table.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK (**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.algorithm.BSTextMapper2D

class brainspace.vtk_interface.wrappers.algorithm.**BSTextMapper2D**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyDataMapper2D.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetTextProperty([obj])	Set text property.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

is_filter	Returns True if self is a filter.
is_sink	Returns True if self is a sink.
is_source	Returns True if self is a source.
nic	Returns number of total input connections
nip	Returns number of input ports
nop	Returns number of output ports
vtk_map	Dictionary of vtk setter and getter methods.

SetTextProperty(*obj=None*, ***kwargs*)
Set text property.

Wraps the *SetTextProperty* method of *vtkTextActor* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Label text property. If None, the property is created. Default is None.

- **kwargs** (*optional keyword arguments*) – Arguments are used to set the property.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

is_filter
Returns True if self is a filter. False, otherwise.

A filter that is not a source nor a sink.

Type bool

is_sink
Returns True if self is a sink. False, otherwise.

Type bool

is_source
Returns True if self is a source. False, otherwise.

Type bool

nic
Returns number of total input connections

Type int

nip
Returns number of input ports

Type int

nop
Returns number of output ports

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

Actors

BSActor2D([vtkobject])	Wrapper for vtkActor2D.
BSScalarBarActor([vtkobject])	Wrapper for vtkScalarBarActor.
BSTexturedActor2D([vtkobject])	Wrapper for vtkTexturedActor2D.
BSTextActor([vtkobject])	Wrapper for vtkTextActor.
BSActor([vtkobject])	Wrapper for vtkActor.

brainspace.vtk_interface.wrappers.actor.BSActor2D

class brainspace.vtk_interface.wrappers.actor.**BSActor2D**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkActor2D.

Unresolved requests are forwarded to its 2D property.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetProperty()	Get property.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

GetProperty()
Get property.

Wraps the *GetProperty* method of *vtkActor2D* to return a wrapped property.

Returns **prop** (*BSProperty2D*) – Actor's property.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
```

(continues on next page)

(continued from previous page)

```
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.actor.BSScalarBarActor

class brainspace.vtk_interface.wrappers.actor.**BSScalarBarActor**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkScalarBarActor.

Unresolved requests are forwarded to its 2D property.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetProperty()	Get property.
SetAnnotationTextProperty([obj])	Set annotation text property.
SetBackgroundProperty([obj])	Set background property.
SetFrameProperty([obj])	Set frame property.
SetLabelTextProperty([obj])	Set label text property.
SetTitleTextProperty([obj])	Set title text property.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

GetProperty()
Get property.

Wraps the *GetProperty* method of *vtkActor2D* to return a wrapped property.

Returns **prop** (*BSProperty2D*) – Actor's property.

SetAnnotationTextProperty(*obj=None*, ***kwargs*)
Set annotation text property.

Wraps the *SetAnnotationTextProperty* method of *vtkScalarBarActor* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Annotation text

property. If None, the property is created. Default is None.

- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

SetBackgroundProperty(*obj=None*, ***kwargs*)
Set background property.

Wraps the *SetBackgroundProperty* method of *vtkScalarBarActor* to accept a *vtkProperty2D* or BSProperty2D.

Parameters

- **obj** (*vtkProperty2D or BSProperty2D, optional*) – Background property. If None, the property is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

SetFrameProperty(*obj=None*, ***kwargs*)
Set frame property.

Wraps the *SetFrameProperty* method of *vtkScalarBarActor* to accept a *vtkProperty2D* or BSProperty2D.

Parameters

- **obj** (*vtkProperty2D or BSProperty2D, optional*) – Frame property. If None, the property is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

SetLabelTextProperty(*obj=None*, ***kwargs*)
Set label text property.

Wraps the *SetLabelTextProperty* method of *vtkScalarBarActor* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Label text property. If None, the property is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

SetTitleTextProperty(*obj=None*, ***kwargs*)
Set title text property.

Wraps the *SetTitleTextProperty* method of *vtkScalarBarActor* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Title text property. If None, the property is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.actor.BSTexturedActor2D

class brainspace.vtk_interface.wrappers.actor.**BSTexturedActor2D**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkTexturedActor2D.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetProperty()	Get property.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

GetProperty()
Get property.

Wraps the *GetProperty* method of *vtkActor2D* to return a wrapped property.

Returns **prop** (*BSProperty2D*) – Actor's property.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.actor.BSTextActor

class brainspace.vtk_interface.wrappers.actor.**BSTextActor**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkTextActor.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetProperty()	Get property.
SetTextProperty([obj])	Set text property.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

GetProperty()
Get property.

Wraps the *GetProperty* method of *vtkActor2D* to return a wrapped property.

Returns **prop** (*BSProperty2D*) – Actor's property.

SetTextProperty(*obj=None*, ***kwargs*)
Set text property.

Wraps the *SetTextProperty* method of *vtkTextActor* to accept a *vtkTextProperty* or BSTextProperty.

Parameters

- **obj** (*vtkTextProperty or BSTextProperty, optional*) – Label text property. If None, the property is created. Default is None.

- **kwargs** (*optional keyword arguments*) – Arguments are use to set the property.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.actor.BSActor

class brainspace.vtk_interface.wrappers.actor.**BSActor**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkActor.

Unresolved requests are forwarded to its property.

Examples

```
>>> from brainspace.vtk_interface.wrappers import BSActor
>>> a = BSActor()
>>> a.GetProperty().GetOpacity()
1.0
>>> a.GetOpacity() # It is forwarded to the property
1.0
>>> a.opacity = .5
>>> a.VTKObject.GetProperty().GetOpacity()
0.5
```

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

GetProperty()	Get property.
SetDataSetMapper([obj])	Set DataSetMapper.
SetLabeledContourMapper([obj])	Set LabeledContourMapper.
SetMapper([obj])	Set mapper.
SetPolyDataMapper([obj])	Set a PolyDataMapper.
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

GetProperty()

Get property.

Wraps the *GetProperty* method of *vtkActor* to return a wrapped property.

Returns **prop** (*BSProperty*) – Actor's property.

SetDataSetMapper(*obj=None*, ***kwargs*)

Set DataSetMapper.

Parameters

- **obj** (*vtkMapper or BSMapper, optional*) – Mapper. If None, the mapper is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the mapper.

SetLabeledContourMapper(*obj=None*, ***kwargs*)
Set LabeledContourMapper.

Parameters

- **obj** (*vtkMapper or BSMapper, optional*) – Mapper. If None, the mapper is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the mapper.

SetMapper(*obj=None*, ***kwargs*)
Set mapper.

Wraps the *SetMapper* method of *vtkActor* to accept a *vtkMapper* or BSMapper.

Parameters

- **obj** (*vtkMapper or BSMapper, optional*) – Mapper. If None, a PolyDataMapper is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the mapper.

SetPolyDataMapper(*obj=None*, ***kwargs*)
Set a PolyDataMapper.

Parameters

- **obj** (*vtkMapper or BSMapper, optional*) – Mapper. If None, the mapper is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the mapper.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

Lookup tables

BSScalarsToColors([vtkobject])	Wrapper for vtkScalarsToColors.
BSLookupTable([vtkobject])	Wrapper for vtkLookupTable.
BSLookupTableWithEnabling([vtkobject])	Wrapper for vtkLookupTableWithEnabling.
BSWindowLevelLookupTable([vtkobject])	Wrapper for vtkWindowLevelLookupTable.
BSColorTransferFunction([vtkobject])	Wrapper for vtkColorTransferFunction.
BSDiscretizableColorTransferFunction([vtkobject])	Wrapper for vtkDiscretizableColorTransferFunction.

brainspace.vtk_interface.wrappers.lookup_table.BSScalarsToColors

class brainspace.vtk_interface.wrappers.lookup_table.**BSScalarsToColors**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkScalarsToColors.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetAnnotations(values, annotations)	
__init__([vtkobject])	Initialize self.

Continued on next page

Table 102 – continued from previous page

getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetAnnotations (*values*, *annotations*)

getVTK (**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK (**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
```

(continues on next page)

(continued from previous page)

```
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.lookup_table.BSLookupTable

class brainspace.vtk_interface.wrappers.lookup_table.**BSLookupTable**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkLookupTable.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetNumberOfColors()	
SetAnnotations(values, annotations)	
SetNumberOfColors(n)	
SetTable(table)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_values	Returns number of table values.
vtk_map	Dictionary of vtk setter and getter methods.

GetNumberOfColors()

SetAnnotations(*values*, *annotations*)

SetNumberOfColors(*n*)

SetTable(*table*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_values
Returns number of table values.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.lookup_table.BSLookupTableWithEnabling

class brainspace.vtk_interface.wrappers.lookup_table.**BSLookupTableWithEnabling**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkLookupTableWithEnabling.

__init__ (*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetNumberOfColors()	
SetAnnotations(values, annotations)	
SetEnabledArray(array)	
SetNumberOfColors(n)	
SetTable(table)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_values	Returns number of table values.
vtk_map	Dictionary of vtk setter and getter methods.

GetNumberOfColors ()

SetAnnotations (*values*, *annotations*)

SetEnabledArray (*array*)

SetNumberOfColors (*n*)

SetTable (*table*)

getVTK (**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_values
Returns number of table values.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.lookup_table.BSWindowLevelLookupTable

class brainspace.vtk_interface.wrappers.lookup_table.**BSWindowLevelLookupTable**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkWindowLevelLookupTable.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

GetNumberOfColors()	
SetAnnotations(values, annotations)	
SetNumberOfColors(n)	
SetTable(table)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_values	Returns number of table values.
vtk_map	Dictionary of vtk setter and getter methods.

GetNumberOfColors()

SetAnnotations(*values*, *annotations*)

SetNumberOfColors(*n*)

SetTable(*table*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_values
Returns number of table values.

Type int

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.lookup_table.BSColorTransferFunction

class brainspace.vtk_interface.wrappers.lookup_table.**BSColorTransferFunction**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkColorTransferFunction.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetAnnotations(values, annotations)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetAnnotations(*values*, *annotations*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.lookup_table.BSDiscretizableColorTransferFunction

class brainspace.vtk_interface.wrappers.lookup_table.**BSDiscretizableColorTransferFunction**(

Wrapper for vtkDiscretizableColorTransferFunction.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

SetAnnotations(values, annotations)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetAnnotations(*values*, *annotations*)

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

Rendering

BSRenderer([vtkobject])	Wrapper for vtkRenderer.
BSRenderWindow([vtkobject])	Wrapper for vtkRenderWindow.
BSRenderWindowInteractor([vtkobject])	Wrapper for vtkRenderWindowInteractor.
BSGenericRenderWindowInteractor([vtkobject])	Wrapper for vtkGenericRenderWindowInteractor.
BSInteractorStyle([vtkobject])	Wrapper for vtkInteractorStyle.
BSInteractorStyleJoystickCamera([vtkobject])	Wrapper for vtkInteractorStyleJoystickCamera.
BSInteractorStyleJoystickActor([vtkobject])	Wrapper for vtkInteractorStyleJoystickActor.
BSInteractorStyleTerrain([vtkobject])	Wrapper for vtkInteractorStyleTerrain.
BSInteractorStyleRubberBandZoom([vtkobject])	Wrapper for vtkInteractorStyleRubberBandZoom.
BSInteractorStyleTrackballActor([vtkobject])	Wrapper for vtkInteractorStyleTrackballActor.
BSInteractorStyleTrackballCamera([vtkobject])	Wrapper for vtkInteractorStyleTrackballCamera.
BSInteractorStyleImage([vtkobject])	Wrapper for vtkInteractorStyleImage.
BSInteractorStyleRubberBandPick([vtkobject])	Wrapper for vtkInteractorStyleRubberBandPick.
BSInteractorStyleSwitchBase([vtkobject])	Wrapper for vtkInteractorStyleSwitchBase.
BSInteractorStyleSwitch([vtkobject])	Wrapper for vtkInteractorStyleSwitch.
BSCamera([vtkobject])	Wrapper for vtkCamera.

brainspace.vtk_interface.wrappers.renderer.BSRenderer

class brainspace.vtk_interface.wrappers.renderer.**BSRenderer**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkRenderer.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

AddActor([obj])	Set mapper.
AddActor2D([obj])	Set mapper.
AddScalarBarActor([obj])	
AddTextActor([obj])	
__init__([vtkobject])	Initialize self.

Continued on next page

Table 115 – continued from previous page

getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

AddActor (*obj=None*, ***kwargs*)
Set mapper.

Wraps the *AddActor* method of *vtkRenderer* to accept a *vtkActor* or BSActor.

Parameters

- **obj** (*vtkActor or BSActor, optional*) – Actor. If None, the actor is created. Default is None.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the actor.

AddActor2D (*obj=None*, ***kwargs*)
Set mapper.

Wraps the *AddActor2D* method of *vtkViewport* to accept a *vtkActor2D* or BSActor2D.

Parameters

- **obj** (*vtkActor or BSActor*) – 2D Actor.
- **kwargs** (*optional keyword arguments*) – Arguments are used to set the actor.

AddScalarBarActor (*obj=None*, ***kwargs*)

AddTextActor (*obj=None*, ***kwargs*)

getVTK (**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns results (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSRenderWindow

class brainspace.vtk_interface.wrappers.renderer.**BSRenderWindow**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkRenderWindow.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

AddRenderer([obj])	
SetInteractor([obj])	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

AddRenderer(*obj=None*, ***kwargs*)

SetInteractor(*obj=None*, ***kwargs*)

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSRenderWindowInteractor

class brainspace.vtk_interface.wrappers.renderer.**BSRenderWindowInteractor**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkRenderWindowInteractor.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetInteractorStyle([obj])	
SetInteractorStyleImage([obj])	
SetInteractorStyleJoystickActor([obj])	
SetInteractorStyleJoystickCamera([obj])	
SetInteractorStyleNone()	
SetInteractorStyleRubberBandPick([obj])	
SetInteractorStyleRubberBandZoom([obj])	
SetInteractorStyleSwitch([obj])	
SetInteractorStyleTerrain([obj])	
SetInteractorStyleTrackBallCamera([obj])	
SetInteractorStyleTrackballActor([obj])	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetInteractorStyle(*obj=None*, ***kwargs*)

SetInteractorStyleImage(*obj=None*, ***kwargs*)

SetInteractorStyleJoystickActor(*obj=None*, ***kwargs*)

SetInteractorStyleJoystickCamera(*obj=None*, ***kwargs*)

SetInteractorStyleNone()

SetInteractorStyleRubberBandPick(*obj=None*, ***kwargs*)

SetInteractorStyleRubberBandZoom(*obj=None*, ***kwargs*)

SetInteractorStyleSwitch(*obj=None*, ***kwargs*)

SetInteractorStyleTerrain(*obj=None*, ***kwargs*)

SetInteractorStyleTrackBallCamera(*obj=None*, ***kwargs*)

SetInteractorStyleTrackballActor(*obj=None*, ***kwargs*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSGenericRenderWindowInteractor

class brainspace.vtk_interface.wrappers.renderer.**BSGenericRenderWindowInteractor**(*vtkobject=No*
***kwargs*)

Wrapper for vtkGenericRenderWindowInteractor.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetInteractorStyle([obj])	
SetInteractorStyleImage([obj])	
SetInteractorStyleJoystickActor([obj])	
SetInteractorStyleJoystickCamera([obj])	
SetInteractorStyleNone()	
SetInteractorStyleRubberBandPick([obj])	
SetInteractorStyleRubberBandZoom([obj])	
SetInteractorStyleSwitch([obj])	
SetInteractorStyleTerrain([obj])	
SetInteractorStyleTrackBallCamera([obj])	
SetInteractorStyleTrackballActor([obj])	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetInteractorStyle(*obj=None*, ***kwargs*)

SetInteractorStyleImage(*obj=None*, ***kwargs*)

SetInteractorStyleJoystickActor(*obj=None*, ***kwargs*)

SetInteractorStyleJoystickCamera(*obj=None*, ***kwargs*)

SetInteractorStyleNone()

SetInteractorStyleRubberBandPick(*obj=None*, ***kwargs*)

SetInteractorStyleRubberBandZoom(*obj=None*, ***kwargs*)

SetInteractorStyleSwitch(*obj=None*, ***kwargs*)

SetInteractorStyleTerrain(*obj=None*, ***kwargs*)

SetInteractorStyleTrackBallCamera(*obj=None*, ***kwargs*)

SetInteractorStyleTrackballActor(*obj=None*, ***kwargs*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyle

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyle**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkInteractorStyle.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleJoystickCamera

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleJoystickCamera**(*vtkobject=Non*
***kwargs*)

Wrapper for vtkInteractorStyleJoystickCamera.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
```

(continues on next page)

(continued from previous page)

```
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleJoystickActor

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleJoystickActor**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleJoystickActor.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleTerrain

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleTerrain**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleTerrain.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleRubberBandZoom

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleRubberBandZoom**(*vtkobject=Nor*
***kwargs*)

Wrapper for vtkInteractorStyleRubberBandZoom.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple.

Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleTrackballActor

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleTrackballActor**(*vtkobject=No*
***kwargs*)

Wrapper for vtkInteractorStyleTrackballActor.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleTrackballCamera

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleTrackballCamera**(*vtkobject=N*
***kwargs*)

Wrapper for vtkInteractorStyleTrackballCamera.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
```

(continues on next page)

(continued from previous page)

```
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleImage

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleImage**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleImage.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleRubberBandPick

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleRubberBandPick**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleRubberBandPick.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleSwitchBase

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleSwitchBase**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleSwitchBase.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple.

Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSInteractorStyleSwitch

class brainspace.vtk_interface.wrappers.renderer.**BSInteractorStyleSwitch**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkInteractorStyleSwitch.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.renderer.BSCamera

class brainspace.vtk_interface.wrappers.renderer.**BSCamera**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkCamera.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
```

(continues on next page)

(continued from previous page)

```
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

Properties

BSProperty([vtkobject])	Wrapper for vtkProperty.
BSProperty2D([vtkobject])	Wrapper for vtkProperty2D.
BSTextProperty([vtkobject])	Wrapper for vtkTextProperty.

brainspace.vtk_interface.wrappers.property.BSProperty

class brainspace.vtk_interface.wrappers.property.**BSProperty**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkProperty.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK (**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK (**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.property.BSProperty2D

class brainspace.vtk_interface.wrappers.property.**BSProperty2D**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkProperty2D.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
```

(continues on next page)

(continued from previous page)

```
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.property.BSTextProperty

class brainspace.vtk_interface.wrappers.property.**BSTextProperty**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkTextProperty.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

Miscellanea

BSCellArray([vtkobject])	Wrapper for vtkCellArray.
BSGL2PSExporter([vtkobject])	Wrapper for vtkGL2PSExporter.
BSCollection([vtkobject])	Wrapper for vtkCollection.
BSPropCollection([vtkobject])	Wrapper for vtkPropCollection.
BSActor2DCollection([vtkobject])	Wrapper for vtkActor2DCollection.
BSActorCollection([vtkobject])	Wrapper for vtkActorCollection.
BSProp3DCollection([vtkobject])	Wrapper for vtkProp3DCollection.
BSMapperCollection([vtkobject])	Wrapper for vtkMapperCollection.
BSRendererCollection([vtkobject])	Wrapper for vtkRendererCollection.
BSPolyDataCollection([vtkobject])	Wrapper for vtkPolyDataCollection.
BSTextPropertyCollection([vtkobject])	Wrapper for vtkTextPropertyCollection.
BSCoordinate([vtkobject])	Wrapper for vtkCoordinate.

brainspace.vtk_interface.wrappers.misc.BSCellArray

class brainspace.vtk_interface.wrappers.misc.**BSCellArray**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkCellArray.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

SetCells(n_cells, cells)	
__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

SetCells(*n_cells*, *cells*)

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple.

Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSGL2PSExporter

class brainspace.vtk_interface.wrappers.misc.**BSGL2PSExporter**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkGL2PSExporter.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSCollection

class brainspace.vtk_interface.wrappers.misc.**BSCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
```

(continues on next page)

(continued from previous page)

```
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSPropCollection

class brainspace.vtk_interface.wrappers.misc.**BSPropCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPropCollection.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSActor2DCollection

class brainspace.vtk_interface.wrappers.misc.**BSActor2DCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkActor2DCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSActorCollection

class brainspace.vtk_interface.wrappers.misc.**BSActorCollection**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkActorCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSProp3DCollection

class brainspace.vtk_interface.wrappers.misc.**BSProp3DCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkProp3DCollection.

__init__(*vtkobject=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.

- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSMapperCollection

class brainspace.vtk_interface.wrappers.misc.**BSMapperCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkMapperCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSRendererCollection

class brainspace.vtk_interface.wrappers.misc.**BSRendererCollection**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkRendererCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSPolyDataCollection

class brainspace.vtk_interface.wrappers.misc.**BSPolyDataCollection**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkPolyDataCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)
Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSTextPropertyCollection

class brainspace.vtk_interface.wrappers.misc.**BSTextPropertyCollection**(*vtkobject=None*, ***kwargs*)
Wrapper for vtkTextPropertyCollection.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

n_items	
vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)

Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

n_items

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map
Dictionary of vtk setter and getter methods.

Type dict

brainspace.vtk_interface.wrappers.misc.BSCoordinate

class brainspace.vtk_interface.wrappers.misc.**BSCoordinate**(*vtkobject=None*, ***kwargs*)

Wrapper for vtkCoordinate.

__init__(*vtkobject=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([vtkobject])	Initialize self.
getVTK(*args, **kwargs)	Invoke get methods on the vtk object.
setVTK(*args, **kwargs)	Invoke set methods on the vtk object.

Attributes

vtk_map	Dictionary of vtk setter and getter methods.

getVTK(**args*, ***kwargs*)
Invoke get methods on the vtk object.

Parameters

- **args** (*list of str*) – Method that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **results** (*dict*) – Dictionary of results where the keys are the method names and the values the results.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
```

(continues on next page)

(continued from previous page)

```
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.getVTK('arrayId', colorModeAsString=None)
{'arrayId': -1, 'colorModeAsString': 'Default'}
>>> m1.getVTK('colorModeAsString', arrayId=None)
{'colorModeAsString': 'Default', 'arrayId': -1}
>>> m1.getVTK(numberOfInputConnections=0)
{'numberOfInputConnections': 0}
```

setVTK(**args*, ***kwargs*)

Invoke set methods on the vtk object.

Parameters

- **args** (*list of str*) – Setter methods that require no arguments.
- **kwargs** (*list of keyword-value arguments*) – key-word arguments can be use for methods that require arguments. When several arguments are required, use a tuple. Methods that require no arguments can also be used here using None as the argument.

Returns **self** (*BSVTKObjectWrapper object*) – Return self.

Examples

```
>>> import vtk
>>> from brainspace.vtk_interface.wrappers import BSVTKObjectWrapper
>>> m1 = BSVTKObjectWrapper(vtk.vtkPolyDataMapper())
>>> m1.setVTK(arrayId=3, colorMode='mapScalars')
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f38a4ace320>
>>> m1.arrayId
3
>>> m1.colorModeAsString
'MapScalars'
```

vtk_map

Dictionary of vtk setter and getter methods.

Type dict

Decorators

wrap_input(*xargs[, skip])	Decorator to wrap the arguments of a function.
wrap_output(func)	Decorator to wrap output of function.
unwrap_input(*xargs[, vtype, skip])	Decorator to unwrap input arguments of function.
unwrap_output([vtype])	Decorator to wrap both arguments and output of a function.
append_vtk([to])	Decorator to append data to surface.

brainspace.vtk_interface.decorators.wrap_input

brainspace.vtk_interface.decorators.**wrap_input**(**xargs*, *skip=False*)

Decorator to wrap the arguments of a function.

An object is wrapped only if it is an instance of `vtkObject` or one of its subclasses.

Parameters

- **xargs** (*sequence of int and str*) – Positional indices (integers) and keys as strings (for keyword args) to wrap. If none specified, try to wrap all args.
- **skip** (*bool, optional*) – Wrap all arguments except those in *xargs*. Default is False.

See also:

wrap_output, wrap_func, *unwrap_input*

brainspace.vtk_interface.decorators.wrap_output

brainspace.vtk_interface.decorators.**wrap_output**(*func*)
Decorator to wrap output of function.

The output is wrapped only if is an instance of vtkObject or one of its subclasses.

Parameters **func** (*callable*) – Function whose output is wrapped.

See also:

wrap_input, wrap_func, *unwrap_output*

brainspace.vtk_interface.decorators.unwrap_input

brainspace.vtk_interface.decorators.**unwrap_input**(**xargs*, *vtype=False*, *skip=False*)
Decorator to unwrap input arguments of function.

An object is unwrapped only if it is an instance of *BSVTKObjectWrapper* or one of its subclasses.

Parameters

- **xargs** (*sequence of int and str*) – Positional indices (integers) and keys as strings (for keyword args) to unwrap. If no specified, try to unwrap all args.
- **skip** (*bool, optional*) – Unwrap all arguments except those in *xargs*. Default is False.

See also:

unwrap_output, unwrap_func, *wrap_input*

brainspace.vtk_interface.decorators.unwrap_output

brainspace.vtk_interface.decorators.**unwrap_output**(*vtype=False*)
Decorator to wrap both arguments and output of a function.

An object is wrapped only if it is an instance of vtkObject or one of its subclasses.

Parameters

- **xargs** (*sequence of int and str*) – Positional indices (integers) and keys as strings (for keyword args) to wrap. If no specified, try to wrap all args.
- **inp** (*bool, optional*) – If True, wrap input arguments. Default is True.
- **out** (*bool, optional*) – If True, wrap output. Default is True.
- **skip** (*bool, optional*) – Wrap all arguments except those in *xargs*. Default is False.

See also:

wrap_input, *wrap_output*, unwrap_func

brainspace.vtk_interface.decorators.append_vtk

brainspace.vtk_interface.decorators.**append_vtk**(*to='point'*)
Decorator to append data to surface.

Parameters **to** (*{'point', 'cell', 'field'}, optional*) – Append data to PointData, CellData or FieldData. Default is 'point'.

Returns **wrapped_func** (*callable*) – Wrapped function.

Notes

All functions using this decorator must:

- Return a ndarray. The size of the array must be consistent with the data it will be appended to (e.g., number of points if `to == 'point'`), except for FieldData.
- Have the following 2 key-value arguments:
  1. **append (bool, optional)** If True, append data to surface. Otherwise, return data.
  2. **key (str, optional)** Array names of data.

See also:

compute_cell_area, *get_n_adjacent_cells*

3.2.7 Utils

- *Parcellation*

Parcellation

relabel(lab[, new_labels])	Relabel array.
relabel_consecutive(lab[, start_from])	Relabel array with consecutive values.
relabel_by_overlap(lab, ref_lab)	Relabel according to overlap with reference.
map_to_mask(values, mask[, fill, axis])	Assign data to mask.
map_to_labels(source_val, target_lab[, ...])	Map data in source to target according to their labels.
reduce_by_labels(values, labels[, weights, ...])	Summarize data in *values* according to *labels*.

brainspace.utils.parcellation.relabel

brainspace.utils.parcellation.**relabel**(*lab*, *new_labels=None*)
Relabel array.

Parameters

- **lab** (*array_like*) – Array to relabel.
- **new_labels** (*array_like or dict, optional*) – New labels. If dict, provide new label for each label in input array. If array_like, mapping is performed in ascending

order. If None, relabel consecutively, starting from 0. Default is None.

Returns **new_lab** (*ndarray*) – Array with new labels.

brainspace.utils.parcellation.relabel_consecutive

brainspace.utils.parcellation.**relabel_consecutive**(*lab*, *start_from=0*)

Relabel array with consecutive values.

Parameters

- **lab** (*ndarray*) – Array to relabel.
- **start_from** (*int, optional*) – Initial label. The default is 0.

Returns **new_lab** (*ndarray*) – Array with consecutive labels.

brainspace.utils.parcellation.relabel_by_overlap

brainspace.utils.parcellation.**relabel_by_overlap**(*lab*, *ref_lab*)

Relabel according to overlap with reference.

Parameters

- **lab** (*ndarray, shape = (n_labels,)*) – Array of labels.
- **ref_lab** (*ndarray, shape = (n_labels,)*) – Reference array of labels.

Returns **new_lab** (*ndarray, shape = (n_labels,)*) – Array relabeled using the reference array.

Notes

Correspondences between labels are based on largest overlap using the Hungarian algorithm.

brainspace.utils.parcellation.map_to_mask

brainspace.utils.parcellation.**map_to_mask**(*values*, *mask*, *fill=0*, *axis=0*)

Assign data to mask.

Parameters

- **values** (*ndarray, shape = (n_values,) or (n_samples, n_values)*) – Source array of values.
- **mask** (*ndarray, shape = (n_mask,)*) – Mask of boolean values. Data is mapped to mask. If *values* is 2D, the mask is applied according to *axis*.
- **fill** (*float, optional*) – Value used to fill elements outside the mask. Default is 0.
- **axis** (*{0, 1}, optional*) – If `axis == 0` map rows. Otherwise, map columns. Default is 0.

Returns **output** (*ndarray*) – Values mapped to mask. If *values* is 1D, shape (n_mask,). When *values* is 2D, shape (n_samples, n_mask) if `axis == 0` and (n_mask, n_samples) otherwise.

brainspace.utils.parcellation.map_to_labels

brainspace.utils.parcellation.**map_to_labels**(*source_val*, *target_lab*, *mask=None*, *fill=0*, *axis=0*, *source_lab=None*)

Map data in source to target according to their labels.

Target labels are sorted in ascending order, such that the smallest label indexes the value at position 0 in *source_val*. If *source_lab* is specified, any label in *target_lab* must be in *source_lab*.

Parameters

- **source_val** (*ndarray, shape = (n_values,) or (n_samples, n_values)*) – Source array of values.
- **target_lab** (*ndarray, shape = (n_labels,)*) – Target labels.
- **mask** (*int or ndarray, shape = (n_labels,), optional*) – If mask is not None, only consider target labels in mask. If int, it indicates the label denoting label to discard. Default is None.
- **fill** (*float, optional*) – Value used to fill elements outside the mask. Only used if mask is not None. Default is 0.
- **axis** (*{0, 1}, optional*) – If `axis == 0`, map rows. Otherwise, map columns. Default is 0.
- **source_lab** (*ndarray, shape = (n_values,), optional*) – Source labels for source values. If None, use unique labels in *target_lab* in ascending order. Default is None.

Returns **target_val** (*ndarray, shape = (n_labels,) or (n_samples, n_labels)*) – Target array with corresponding source values.

brainspace.utils.parcellation.reduce_by_labels

brainspace.utils.parcellation.**reduce_by_labels**(*values*, *labels*, *weights=None*, *target_labels=None*, *red_op='mean'*, *axis=0*, *dtype=None*)

Summarize data in *values* according to *labels*.

Parameters

- **values** (*1D or 2D ndarray*) – Array of values.
- **labels** (*1D ndarray, shape = (n_lab,)*) – Labels used summarize values.
- **weights** (*1D ndarray, shape = (n_lab,), optional*) – Weights associated with labels. Only used when *red_op* is 'average', 'mean', 'sum' or 'mode'. Weights are not normalized. Default is None.
- **target_labels** (*1D ndarray, optional*) – Target labels. Arrange new array following the ordering of labels in the *target_labels*. When None, new array is arranged in ascending order of *labels*. Default is None.
- **red_op** (*str or callable, optional*) – How to summarize data. If str, options are: {'min', 'max', 'sum', 'mean', 'median', 'mode', 'average'}. If callable, it should receive a 1D array of values, array of weights (or None) and return a scalar value. Default is 'mean'.

- **dtype** (*dtype, optional*) – Data type of output array. When None, if *red_op* in {'min', 'max', 'sum', 'mode'}, output is same type as *values*, otherwise output is float. Default is None.
- **axis** (*{0, 1}, optional*) – If `axis == 0`, apply to each row (reduce number of columns per row). Otherwise, apply to each column (reduce number of rows per column). Default is 0.

Returns **target_values** (*ndarray*) – Summarized target values.

3.3 VTK wrapping

Surface mesh functionality provided in BrainSpace is built on top of the Visualization Toolkit (VTK). This document introduces the wrapping interface used by BrainSpace to work with VTK objects.

- *Wrapping interface*
- *Pipeline liaisons*
- *Data object wrappers*
- *Plotting*

3.3.1 Wrapping interface

BSVTKObjectWrapper is the base class for all wrappers implemented in BrainSpace. Wrapping a VTK object is done with the *wrap_vtk* function. When wrapping a VTK object, if the corresponding wrapper does not exist, it falls back to *BSVTKObjectWrapper*. Check the API for the complete list of wrappers implemented in the current version of BrainSpace.

The *BSVTKObjectWrapper* is a wrapper that extends the Python object wrapper in VTK `VTKObjectWrapper` to provide easier access to VTK setter and getter class methods. The wrapper, since it is a subclass of `VTKObjectWrapper`, holds a reference to the VTK object in the `BSVTKObjectWrapper.VTKObject` attribute. And further includes the following functionality:

1. *setVTK* and *getVTK* to invoke several setter/getter methods on the VTK object:

```
>>> import vtk

>>> # Let's create a sphere with VTK
>>> s = vtk.vtkSphere()
>>> s
(vtkCommonDataModelPython.vtkSphere)0x7f610d222f48

>>> # And check the default values
>>> s.GetRadius()
0.5
>>> s.GetCenter()
(0.0, 0.0, 0.0)

>>> # We are going to wrap the sphere
>>> from brainspace.vtk_interface import wrap_vtk
>>> ws = wrap_vtk(s)

>>> # ws is an instance of BSVTKObjectWrapper
>>> ws
```

(continues on next page)

(continued from previous page)

```
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f60cd7d6f60>

>>> # and holds a reference to the VTK sphere
>>> ws.VTKObject
(vtkCommonDataModelPython.vtkSphere)0x7f610d222f48

>>> # Now we can invoke getter methods as follows:
>>> ws.getVTK('radius', center=None)
{'radius': 0.5, 'center': (0.0, 0.0, 0.0)}

>>> # and set different values
>>> ws.setVTK(radius=2, center=(1.5, 1.5, 1.5))
<brainspace.vtk_interface.base.BSVTKObjectWrapper at 0x7f60cd7d6f60>

>>> # To check that everything works as expected
>>> s.GetRadius()
2.0
>>> s.GetCenter()
(1.5, 1.5, 1.5)

>>> # these methods can be invoked on the wrapper, which forwards
>>> # them to the VTK object
>>> ws.GetRadius()
2.0
```

2. Calling VTK setters and getters can also be treated as attributes, by overloading `__setattr__` and `__getattr__`:

```
>>> # we can access to the radius and center
>>> ws.radius
2.0
>>> ws.center
(1.5, 1.5, 1.5)

>>> # and even set new values
>>> ws.radius = 8
>>> ws.center = (10, 10, 10)

>>> # check that everything's ok
>>> s.GetRadius()
8.0
>>> s.GetCenter()
(10.0, 10.0, 10.0)
```

This functionality is available for all methods that start with Get/Set. To see all the methods available, *BSVTKObjectWrapper* holds a dictionary *vtk_map* for each wrapped class:

```
>>> ws.vtk_map.keys()
dict_keys(['set', 'get'])

>>> # for getter methods
>>> ws.vtk_map['get']
{'addressasstring': 'GetAddressAsString',
 'center': 'GetCenter',
 'classname': 'GetClassName',
 'command': 'GetCommand',
```

(continues on next page)

(continued from previous page)

```
 'debug': 'GetDebug',
 'globalwarningdisplay': 'GetGlobalWarningDisplay',
 'mtime': 'GetMTime',
 'radius': 'GetRadius',
 'referencecount': 'GetReferenceCount',
 'transform': 'GetTransform'}
```

Note that this approach is case-insensitive. But we recommend using camel case, at least, for methods with more that one word:

```
>>> # To access the reference count, for example, these are the same
>>> ws.referencecount
1
>>> ws.ReferenceCount
1
```

wrap_vtk provides a nice way to wrap and simultaneously set the values for a VTK object:

```
>>> ws2 = wrap_vtk(vtk.vtkSphere(), radius=10, center=(5, 5, 0))
>>> ws2.getVTK('radius', 'center')
{'radius': 10.0, 'center': (5.0, 5.0, 0.0)}

>>> ws2.VTKObject.GetRadius()
10.0
>>> ws2.VTKObject.GetCenter()
(5.0, 5.0, 0.0)
```

In VTK, among setter methods, we have state methods with the form **Set**Something**To**Value. Using the previous functionality, these methods can be called as follows:

```
>>> # Let's create a mapper
>>> m = vtk.vtkPolyDataMapper()

>>> # This class has several state methods to set the color mode
>>> [m for m in dir(m) if m.startswith('SetColorModeTo')]
['SetColorModeToDefault',
 'SetColorModeToDirectScalars',
 'SetColorModeToMapScalars']

>>> # The default value is
>>> m.GetColorModeAsString()
'Default'

>>> # Now we are going to wrap the VTK object
>>> wm = wrap_vtk(m)
>>> wm
<brainspace.vtk_interface.wrappers.BSPolyDataMapper at 0x7f60ada07828>

>>> # and change the default value as we know so far
>>> # Note that we use None because the method accepts no arguments
>>> wm.colorModeToMapScalars = None  # same as wm.SetColorMoteToMapScalars()
>>> wm.GetColorModeAsString()
'MapScalars'

>>> # state methods can also be used as follows
>>> wm.colorMode = 'DirectScalars'  # which is the default
```

(continues on next page)

(continued from previous page)

```
>>> wm.GetColorModeAsString()
'Default'

>>> # This can be used when wrapping
>>> m2 = wrap_vtk(vtkPolyDataMapper(), colorMode='mapScalars', scalarVisibility=False)
>>> m2.getVTK('colorModeAsString', 'scalarVisibility')
{'colorModeAsString': 'MapScalars', 'scalarVisibility': 0}
```

In the example above, the wrapper class of our mapper is no longer *BSVTKObjectWrapper* but *BSPolyDataMapper*. This is because *wrap_vtk* looks for a convenient wrapper by searching the hierarchy of wrappers in a bottom-up fashion, and *BSPolyDataMapper* is a wrapper that is already implemented in the current version of BrainSpace. We can also see this with VTK actor:

```
>>> wa = wrap_vtk(vtk.vtkActor())
>>> wa
<brainspace.vtk_interface.wrappers.BSActor at 0x7f60cd749e80>

>>> # When a wrapper exists, the VTK object can be created directly
>>> from brainspace.vtk_interface.wrappers import BSActor
>>> wa2 = BSActor()
<brainspace.vtk_interface.wrappers.BSActor at 0x7f60cce8fac8>

>>> # and can be created with arguments
>>> # for example, setting the previous mapper
>>> wa3 = BSActor(mapper=wm)
>>> wa3.mapper.VTKObject is wm.VTKObject
True
>>> wa3.mapper.VTKObject is m
True
```

BSActor is a special wrapper, because calls to setter and getter methods can be forwarded also to its property (i.e., GetProperty()). Methods are first forwarded to the VTK object of the actor, but if they do not exist, they are forwarded then to the property. As of the current version, this is only implemented for *BSActor*:

```
>>> # To see the opacity using the VTK object
>>> wa3.VTKObject.GetProperty().GetOpacity()
1.0

>>> # this wa3.VTKObject.GetOpacity() raises an exception

>>> # Now, using the wrapper
>>> wa3.GetOpacity()
1.0
>>> # or
>>> wa3.opacity
1.0

>>> # and we can set the opacity
>>> wa3.opacity = 0.25
>>> wa3.VTKObject.GetProperty().GetOpacity()
0.25
```

The advantage of this approach over existing packages that build over VTK is that we do not need to learn about all the new API. If the user is familiar with VTK, then using this approach is straightforward, we can invoke the setter and getter methods by simply stripping the Get/Set prefixes.

3.3.2 Pipeline liaisons

VTK workflow is based on connecting (a source to) several filters (and to a sink). This often makes the code very cumbersome. Let's see a dummy example:

```
>>> # Generate point cloud
>>> point_source = vtk.vtkPointSource()
>>> point_source.SetNumberOfPoints(25)

>>> # Build convex hull from point cloud
>>> delauny = vtk.vtkDelaunay2D()
>>> delauny.SetInputConnection(point_source.GetOutputPort())
>>> delauny.SetTolerance(0.01)

>>> # Smooth convex hull
>>> smooth_filter = vtk.vtkWindowedSincPolyDataFilter()
>>> smooth_filter.SetInputConnection(delauny.GetOutputPort())
>>> smooth_filter.SetNumberOfIterations(20)
>>> smooth_filter.FeatureEdgeSmoothingOn()
>>> smooth_filter.NonManifoldSmoothingOn()

>>> # Compute normals
>>> normals_filter = vtk.vtkPolyDataNormals()
>>> normals_filter.SetInputConnection(smooth_filter.GetOutputPort())
>>> normals_filter.SplittingOff()
>>> normals_filter.ConsistencyOn()
>>> normals_filter.AutoOrientNormalsOn()
>>> normals_filter.ComputePointNormalsOn()

>>> # Execute pipeline
>>> normals_filter.Update()

>>> # Get the output
>>> output1 = normals_filter.GetOutput()
>>> output1
(vtkCommonDataModelPython.vtkPolyData)0x7f60cceabb28

>>> output1.GetNumberOfPoints()
25
```

For these scenarios, Brainspace provides the *serial_connect* function to serially connect several filters and skip the boilerplate of connecting filters. The previous example can be rewritten as follows:

```
>>> from brainspace.vtk_interface.pipeline import serial_connect

>>> # Generate point cloud
>>> point_source = wrap_vtk(vtk.vtkPointSource, numberOfPoints=25)

>>> # Build convex hull from point cloud
>>> delauny = wrap_vtk(vtk.vtkDelaunay2D, tolerance=0.01)

>>> # Smooth convex hull
>>> smooth_filter = wrap_vtk(vtk.vtkWindowedSincPolyDataFilter,
...                          numberOfIterations=20, featureEdgeSmoothing=True,
...                          nonManifoldSmoothing=True)

>>> # Compute normals
>>> normals_filter = wrap_vtk(vtk.vtkPolyDataNormals, splitting=False,
```

(continues on next page)

(continued from previous page)

```
...                             consistency=True, autoOrientNormals=True,
...                             computePointNormals=True)

>>> # Execute and get the output
>>> output2 = serial_connect(point_source, delauny, smooth_filter,
...                          normals_filter, as_data=True)
>>> output2
<brainspace.vtk_interface.wrappers.BSPolyData at 0x7f60a3f5fa20>

>>> output2.GetNumberOfPoints()
25
```

First, note that we can simply provide the VTK class instead of the object to *wrap_vtk*. Furthermore, the output object of the previous pipeline is a polydata. This brings us to one of the most important wrappers in BrainSpace, *BSPolyData*, a wrapper for VTK polydata objects.

3.3.3 Data object wrappers

BrainSpace is intended primarily to work with triangular surface meshes of the brain. Hence, the importance of *BSPolyData*. VTK already provides very good wrappers for VTK data objects in the `dataset_adapter` module. In BrainSpace, these wrappers are simply extended to incorporate the aforementioned functionality of the *BSVTKObjectWrapper* and some additional features:

```
>>> # Using the output from previous example
>>> output2.numberOfPoints
25
>>> output2.n_points
25

>>> # Check if polydata has only triangles
>>> output2.has_only_triangle
True

>>> # Since a polydata can hold vertices, lines, triangles, and their
>>> # poly versions, polygons are returned as a 1D array
>>> output2.Polygons.shape
(144,)

>>> # we further provide an additional method to recover the polygons:
>>> output2.GetCells2D().shape
(36, 3)

>>> # this method raises an exception if the polydata holds different
>>> # cell or polygon types, this can be checked with
>>> output2.has_unique_cell_type
True

>>> # to get the cell types
>>> output2.cell_types
array([5])

>>> vtk.VTK_TRIANGLE == 5
True

>>> # To get the point data
```

(continues on next page)

(continued from previous page)

```
>>> output2.PointData.keys()
['Normals']

>>> output2.point_keys
['Normals']

>>> output2.PointData['Normals'].shape
(25, 3)

>>> # or
>>> output2.get_array(name='Normals', at='point').shape
(25, 3)

>>> # we do not have to specify the attributes
>>> output2.get_array(name='Normals')
(25, 3)

>>> # raises exception if name is in more than one attribute (e.g., point
>>> # and cell data)
>>> dummy_cell_normals = np.zeros((output2.n_cells, 3))
>>> output2.append_array(dummy_cell_normals, name='Normals', at='cell')

>>> output2.get_array(name='Normals') # Raise exception!
```

Most properties and methods of *BSPolyData* are inherited from *BSDataSet*. Check out their documentations for more information.

3.3.4 Plotting

BrainSpace offers two high-level plotting functions: *plot_surf* and *plot_hemispheres*. These functions are based on the wrappers of the corresponding VTK objects. We have already seen above the *BSPolyDataMapper* and *BSActor* class wrappers. Here we will show how rendering is performed using these wrappers. The base class for all plotters is `BasePlotter`, with subclasses *Plotter* and *GridPlotter*. These classes forward unknown set/get requests to *BSRenderWindow*, which is a wrapper of `vtkRenderWindow`. Let's see a simple example:

```
>>> ipth = 'path_to_surface'

>>> from brainspace.mesh import mesh_io as mio
>>> from brainspace.plotting.base import Plotter
>>> # from brainspace.vtk_interface.wrappers import BSLookupTable

>>> surf = mio.read_surface(ipth, return_data=True)

>>> yeo7_colors = np.array([[0, 0, 0, 255],
...                         [0, 118, 14, 255],
...                         [230, 148, 34, 255],
...                         [205, 62, 78, 255],
...                         [120, 18, 134, 255],
...                         [220, 248, 164, 255],
...                         [70, 130, 180, 255],
...                         [196, 58, 250, 255]], dtype=np.uint8)

>>> # create a plotter with 2 rows and 2 columns
>>> # other arguments are forwarded to BSRenderWindow (i.e., vtkRenderWindow)
>>> p = Plotter(n_rows=2, n_cols=2, try_qt=False, size=(400, 400))
```

(continues on next page)

(continued from previous page)

```
>>> # Add first renderer, span all columns of first row
>>> ren1 = p.AddRenderer(row=0, col=None, background=(1,1,1))

>>> # Add actor (actor created if not provided)
>>> ac1 = ren1.AddActor()

>>> # Set mapper (mapper created if not provided)
>>> m1 = ac1.SetMapper(inputDataObject=surf, colorMode='mapScalars',
...                    scalarMode='usePointFieldData',
...                    interpolateScalarsBeforeMapping=False,
...                    arrayName='yeo7', useLookupTableScalarRange=True)

>>> # Set mapper lookup table (created if not provided)
>>> lut1 = m1.SetLookupTable(numberOfTableValues=8, Range=(0, 7),
...                          table=yeo7_colors)

>>> # Add second renderer in first column of second row
>>> ren2 = p.AddRenderer(row=1, col=0, background=(1,1,1))
>>> ac2 = ren2.AddActor(opacity=1, edgeVisibility=0)
>>> m2 = ac2.SetMapper(inputData=surf)

>>> # plot in notebook
>>> p.show(interactive=False, embed_nb=True)
```

See that we can use *AddRenderer* to create the renderer if it is not provided as an argument. The same happens with methods *AddActor*, *SetMapper* and *SetLookupTable*, from *BSRenderer*, *BSActor* and *BSPolyDataMapper* wrapper classes, respectively.

The only difference between *Plotter* and *GridPlotter* is that in the latter a renderer is restricted to a single entry, cannot span more than one entry.

CHAPTER 4

MATLAB Package

This page contains links to descriptions of all MATLAB code available in this toolbox as well as the tutorials. The code is divided into five groups: the "main object" which performs the computations, "surface handling" functions for read/writing and parcellating surfaces, "visualization" which plots data, "data loaders" which loads sample data for our tutorials, and "support functions" which are requisites for the other functions, but are not intended for usage by the user.

4.1 Tutorials

4.1.1 Tutorial 1: Building your first gradient

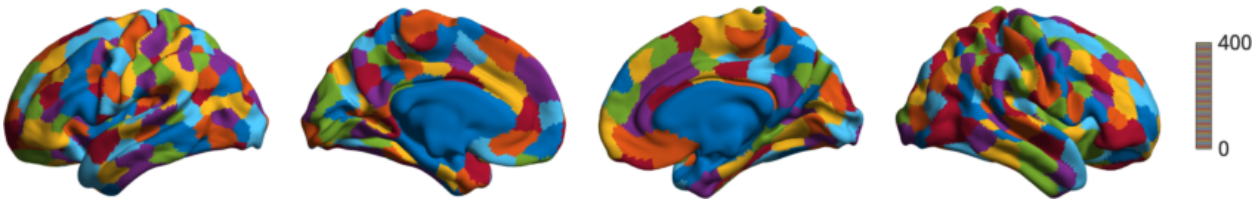
In this example, we will derive a gradient and do some basic inspections to determine which gradients may be of interest and what the multidimensional organization of the gradients looks like.

We'll first start by loading some sample data. Note that we're using parcellated data for computational efficiency.

```
% First load mean connectivity matrix and Schaefer parcellation
conn_matrix = load_group_fc('schaefer',400);
labeling = load_parcellation('schaefer',400);

% The loader functions output data in a struct array for when you
% load multiple parcellations. Let's just bring them to numeric arrays.
conn_matrix = conn_matrix.schaefer_400;
labeling = labeling.schaefer_400;

% and load the conte69 hemisphere surfaces
[surf_lh, surf_rh] = load_conte69();
```

Let's first look at the parcellation scheme we're using.

```
h = plot_hemispheres(labeling, {surf_lh,surf_rh});
colormap(h.handles.figure,lines(401))
```

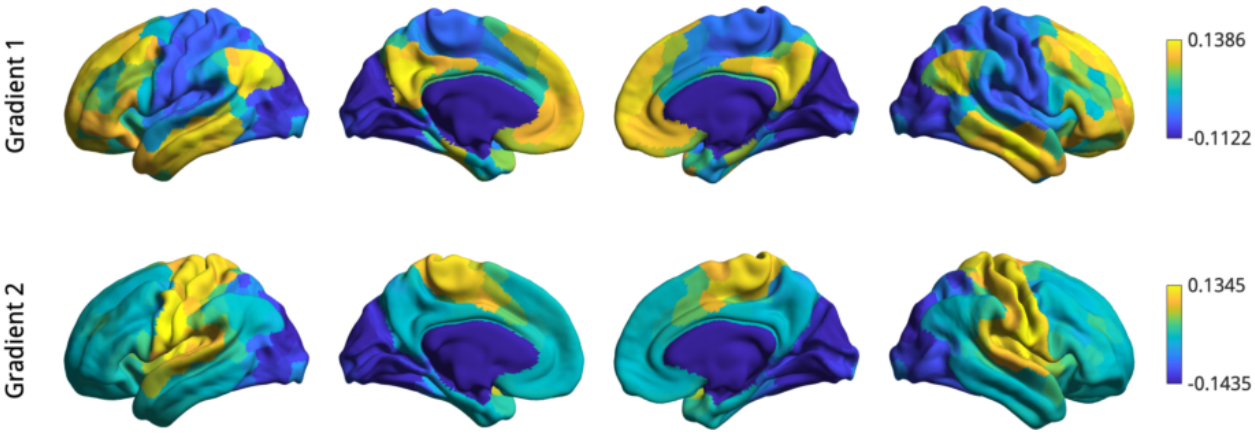


and let's construct our gradients.

```
% Construct the gradients
gm = GradientMaps();
gm = gm.fit(conn_matrix);
```

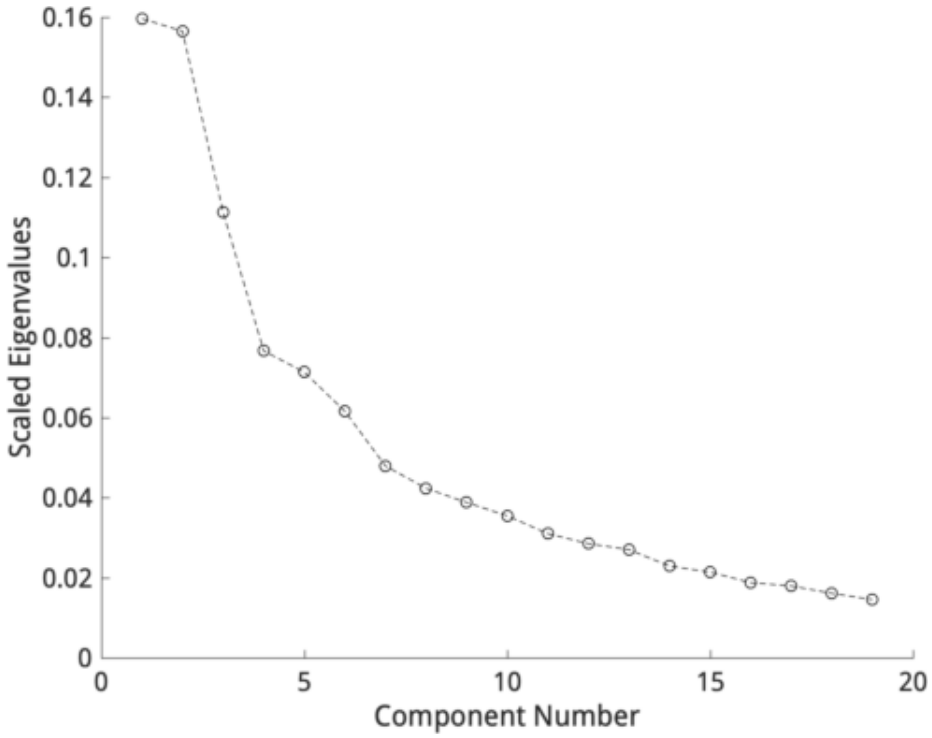
Note that the default parameters (normalized angle kernel, diffusion embedding approach, 10 components) will be reported in the console. Once you have your gradients a good first step is to simply inspect what they look like. Let's have a look at the first two gradients.

```
plot_hemispheres(gm.gradients{1}(:,1:2),{surf_lh,surf_rh}, ...
                 'parcellation', labeling, ...
                 'labeltext',{'Gradient 1','Gradient 2'});
```



But which gradients should you keep for your analysis? In some cases you may have an a-priori interest in some previously defined set of gradients. When you don not have a pre-defined set, you can instead look at the lambdas (eigenvalues) of each component in a scree plot. Higher eigenvalues (or lower in laplacian eigenmapping) are more important, so one can choose a cut-off based on a scree plot.

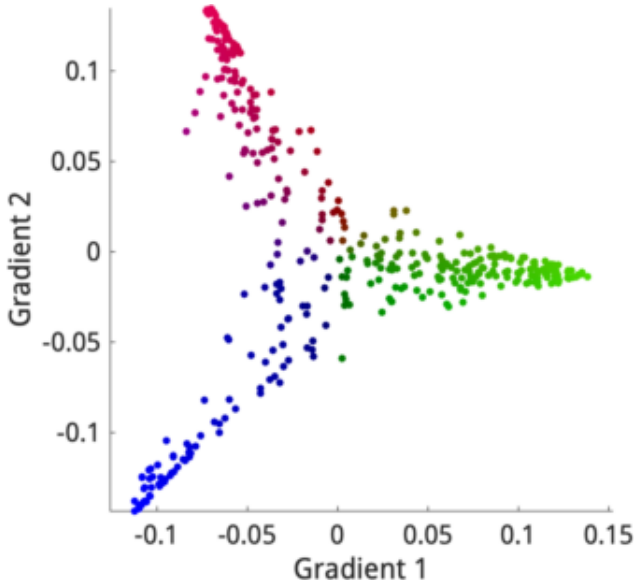
```
scree_plot(gm.lambda{1});
```



A reasonable choice based on this scree plot would be to focus on the first two gradients as they provide a good trade-off between keeping few components whilst retaining a large amount of variance.

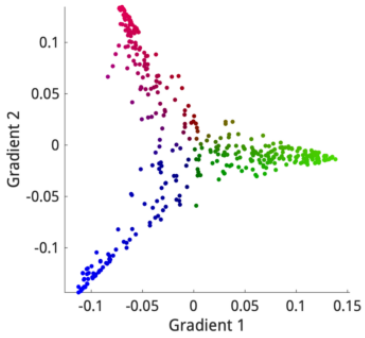
Inspecting gradients together can be quite informative. BrainSpace provides tools for plotting a set of gradients in 2D or 3D space, and assigning them colors based on their position. This color can then be propagated to the surface to get an idea of the multidimensional interaction between the gradients. You do this as follows:

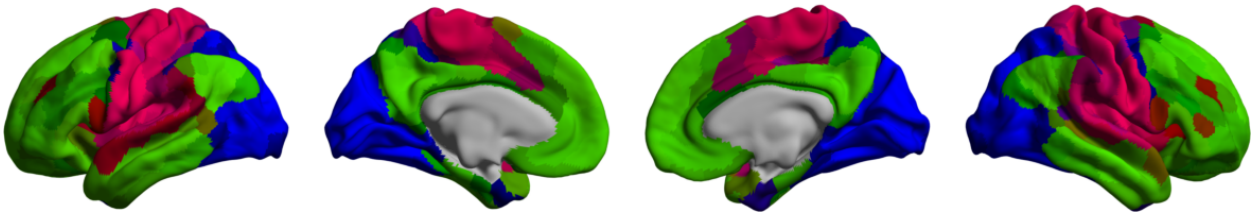
```
gradient_in_euclidean(gm.gradients{1}(:,1:2));
```



We can see that the values of each region are relatively clustered along three lines, colored here in red, green, and blue. If we want to put these colors on the cortical surface, we simply provide the same function with the surface (and parcellation if using parcellated data).

```
gradient_in_euclidean(gm.gradients{1}(:,1:2),{surf_lh,surf_rh},labeling);
```





It now becomes quite evident that the three lines we see in the scatter plot correspond to the somatomotor (red), default mode (green) and visual (blue) networks.

This concludes the first tutorial. In the next tutorial we will have a look at how to customize the methods of gradient estimation, as well as gradient alignments.

4.1.2 Tutorial 2: Customizing and aligning gradients

In this tutorial you'll learn about the methods available within the `GradientMaps` class. The flexible usage of this class allows for the customization of gradient computation with different kernels and dimensionality reductions, as well as aligning gradients from different datasets. This tutorial will only show you how to apply these techniques, for in-depth descriptions we recommend you read the main manuscript.

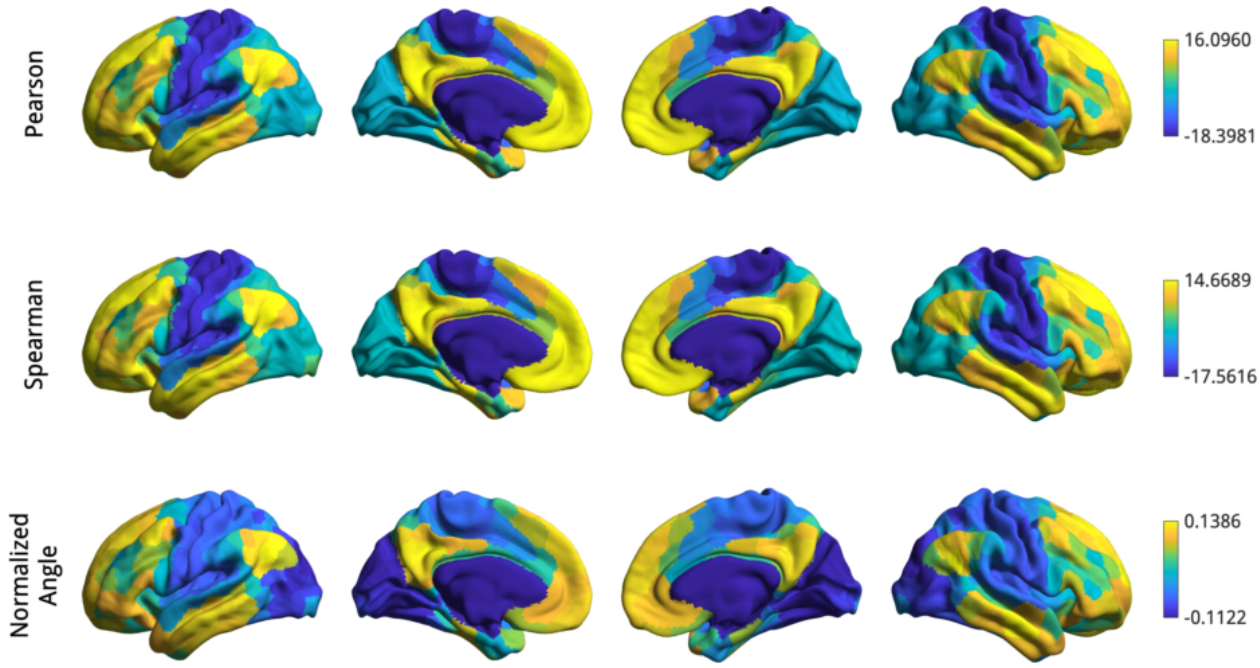
Customizing Gradient Computation

As before, we'll start by loading the sample data.

```
% First load mean connectivity matrix and Schaefer parcellation
conn_matrix = load_group_fc('schaefer',400);
labeling = load_parcellation('schaefer',400);

% The loader functions output data in a struct array for when you
% load multiple parcellations. Let's just bring them to numeric arrays.
conn_matrix = conn_matrix.schaefer_400;
labeling = labeling.schaefer_400;

% and load the conte69 hemisphere surfaces
[surf_lh, surf_rh] = load_conte69();
```

The `GradientMaps` object allows for many different kernels and dimensionality reduction techniques (for a full list see *GradientMaps*). Let's have a look at three different kernels.

```
kernels = {'pearson', ...
           'spearman', ...
           'normalizedAngle'}; % Case-insensitive
for ii = 1:numel(kernels)
    gm_k{ii} = GradientMaps('kernel',kernels{ii},'approach','dm');
    gm_k{ii} = gm_k{ii}.fit(conn_matrix);
end

label_text = {'Pearson','Spearman',{'Normalized','Angle'}};
plot_hemispheres([gm_k{1}.gradients{1}(:,1), ...
    gm_k{2}.gradients{1}(:,1), ...
    gm_k{3}.gradients{1}(:,1)], ...
    {surf_lh,surf_rh}, ...
    'parcellation', labeling, ...
    'labeltext',label_text);
```



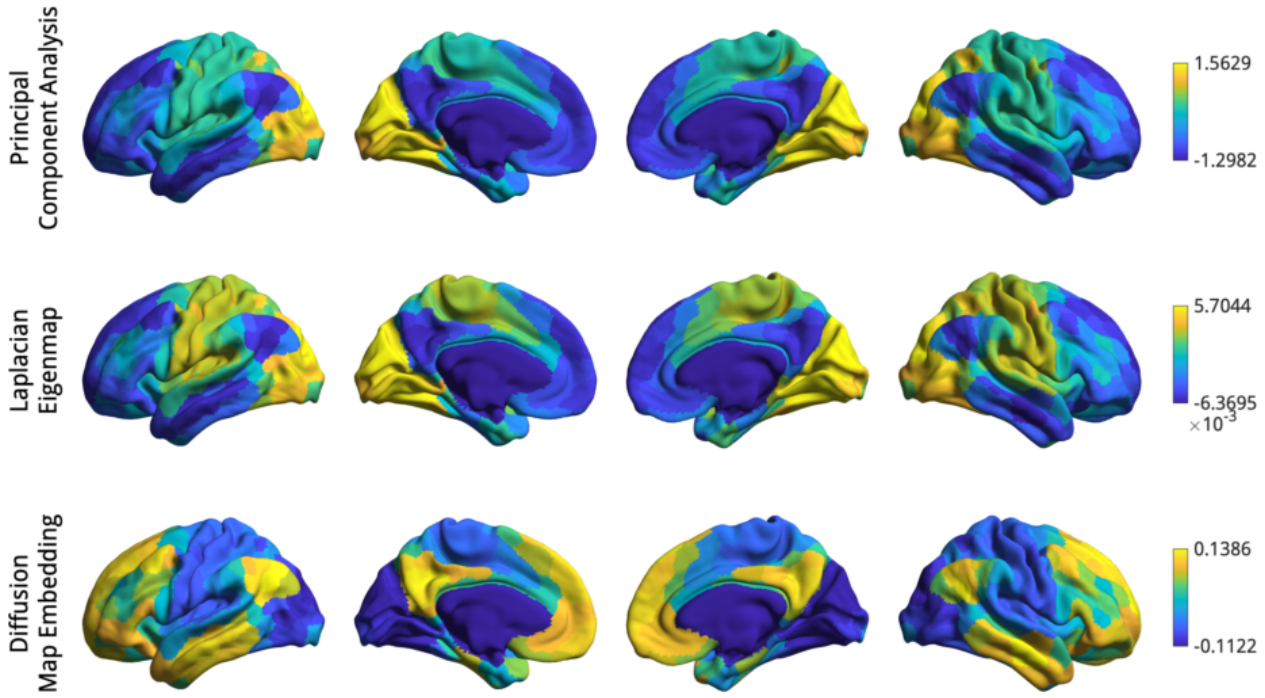
It seems the gradients provided by these kernels are quite similar although their scaling is quite different. Do note that the gradients are in arbitrary units, so the smaller/larger axes across kernels do not imply anything. Similar to using different kernels, we can also use different dimensionality reduction techniques.

```
% Compute gradients for every approach.
embeddings = {'principalComponentAnalysis', ...
              'laplacianEigenmap', ...
              'diffusionEmbedding'}; % case-insensitve
for ii = 1:numel(embeddings)
    gm_m{ii} = GradientMaps('kernel','na','approach',embeddings{ii});
    gm_m{ii} = gm_m{ii}.fit(conn_matrix);
end

% Plot all to one figure.
label_text = {{'Principal','Component Analysis'}, ...
    {'Laplacian','Eigenmap'}, {'Diffusion','Map Embedding'}};
plot_hemispheres([gm_m{1}.gradients{1}(:,1), ...
    gm_m{2}.gradients{1}(:,1), ...
```

(continues on next page)

(continued from previous page)

```
    gm_m{3}.gradients{1}(:,1)], ...
    {surf_lh,surf_rh}, ...
    'parcellation', labeling, ...
    'labeltext', label_text);
```



Here we do see some substantial differences: PCA appears to find a slightly different axis, with the somatomotor in the middle between default mode and visual, whereas LE and DM both find the canonical first gradient but their signs are flipped! Fortunately, the sign of gradients is arbitrary, so we could simply multiply either the LM and DM gradient by -1 to make them more comparable.

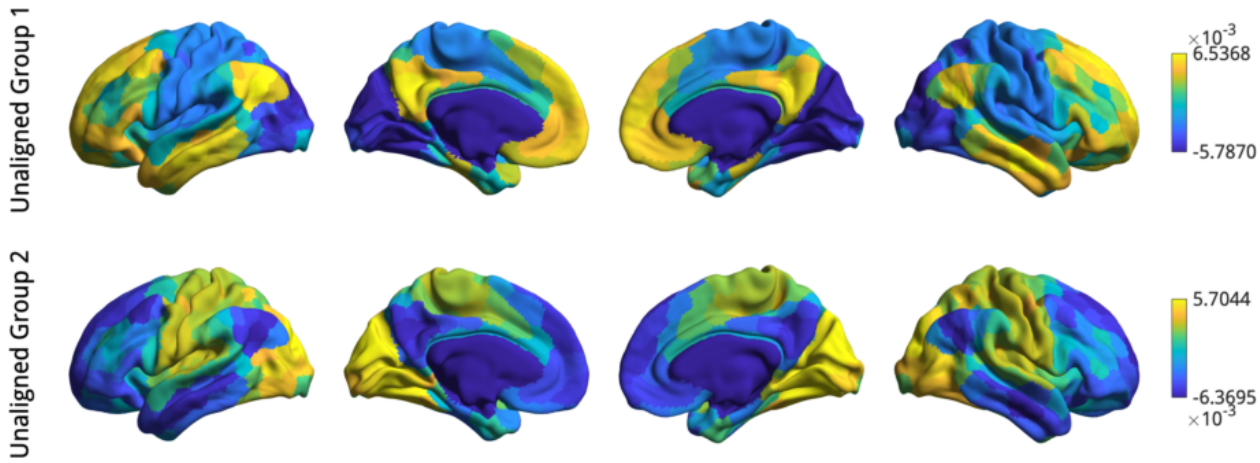
Gradient Alignment

A more principled way of increasing comparability across gradients are alignment techniques. BrainSpace provides two alignment techniques: Procrustes analysis, and joint alignment. For this example we will load functional connectivity data of a second subject group and align it with the first group using a normalized angle kernel and laplacian eigenmap approach.

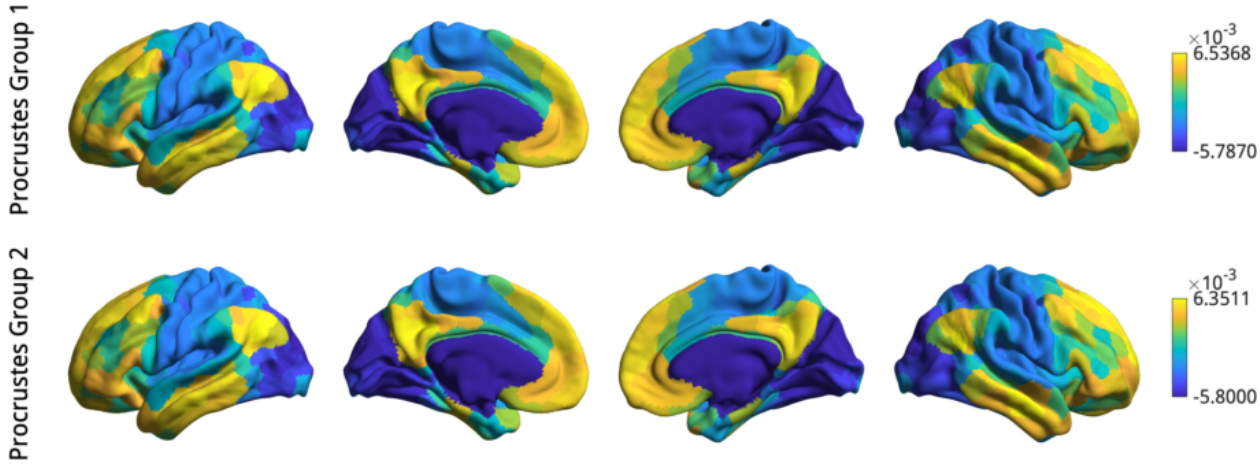
```
conn_matrix2 = load_group_fc('schaefer',400,'holdout');
conn_matrix2 = conn_matrix2.schaefer_400;
Gp = GradientMaps('kernel','na','approach','le','alignment','pa');
Gj = GradientMaps('kernel','na','approach','le','alignment','ja');

Gp = Gp.fit({conn_matrix2,conn_matrix});
Gj = Gj.fit({conn_matrix2,conn_matrix});
```

Here, `Gp` contains the Procrustes aligned data and `Gj` contains the joint aligned data. Let's plot them, but in separate figures to keep things organized. Note that the gradients in the first element of the gradients/aligned property correspond to the first data matrix provided in fit() and the gradients in the second element correspond to the second data matrix.

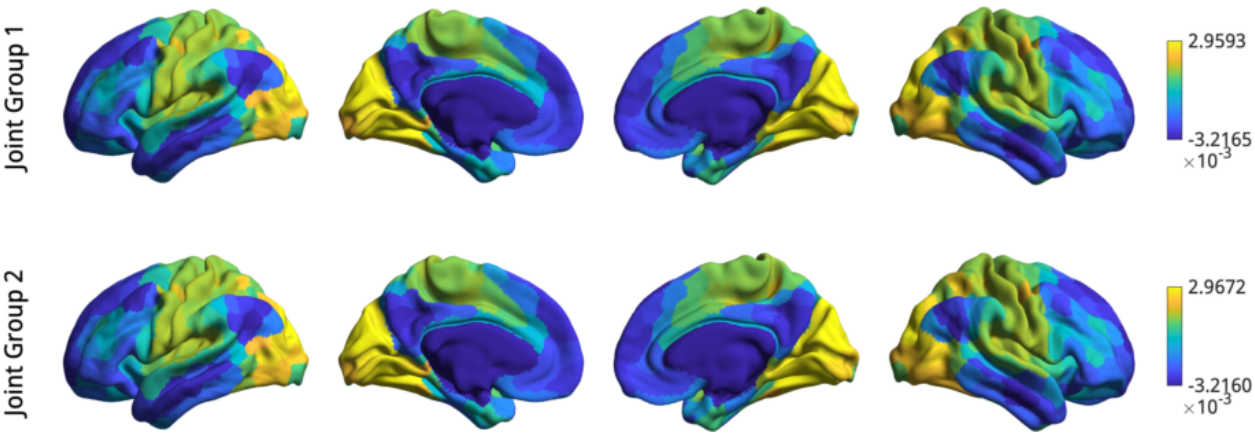
```
plot_hemispheres([Gp.gradients{1}(:,1),Gp.gradients{2}(:,1)], ...
    {surf_lh,surf_rh}, 'parcellation', labeling, ...
    'labeltext',{'Unaligned Group 1','Unaligned Group 2'});
```




```
plot_hemispheres([Gp.aligned{1}(:,1),Gp.aligned{2}(:,1)], ...
    {surf_lh,surf_rh},'parcellation',labeling, ...
    'labeltext',{'Procrustes Group 1','Procrustes Group 2'});
```




```
plot_hemispheres([Gj.aligned{1}(:,1),Gj.aligned{2}(:,1)], ...
    {surf_lh,surf_rh},'parcellation',labeling, ...
    'labeltext',{'Joint Group 1','Joint Group 2'});
```



Before gradient alignment, the first gradient is reversed, but both alignments resolve this issue. If the input data was less similar, alignments may also resolve changes in the order of the gradients. However, you should always inspect the output of an alignment; if the input data are sufficiently dissimilar then the alignment may produce odd results.

In some instances, you may want to align gradients to an out-of-sample gradient, for example when aligning individuals to a hold-out group gradient. When performing a Procrustes alignemnt, a 'reference' can be specified. The first alignment iteration will then be to the reference. For purposes of this example, we will use the gradient of the hold-out group as the reference.

```
Gref = GradientMaps('kernel','na','approach','le');
Gref = Gref.fit(conn_matrix2);

Galign = GradientMaps('kernel','na','approach','le','alignment','pa');
Galign = Galign.fit(conn_matrix,'reference',Gref.gradients{1});
```

The gradients in `Galign.aligned` are now aligned to the reference gradients.

Gradient Fusion

We can also fuse data across multiple modalities and build mutli-modal gradients as was done by (Paquola et al., 2019). In this case we only look at one set of output gradients, rather than one per modality.

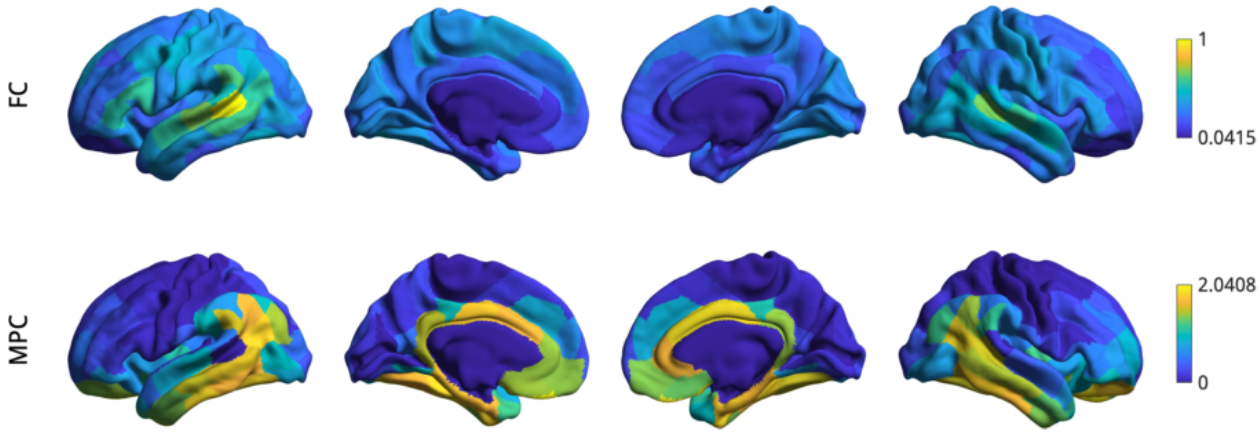
First, let's load the example data of microstructural profile covariance and functional connectivity.

```
% First load mean connectivity matrix and parcellation
mpc = load_group_mpc('vosdewael',200);
fc = load_group_fc('vosdewael',200);
labeling = load_parcellation('vosdewael',200);

% The loader functions output data in a struct array for when you
% load multiple parcellations. Let's just bring them to numeric arrays.
mpc = mpc.vosdewael_200;
fc = fc.vosdewael_200;
labeling = labeling.vosdewael_200;

% and load the conte69 hemisphere surfaces
[surf_lh, surf_rh] = load_conte69();

% visualise the features from a seed region
h = plot_hemispheres([fc(:,1),mpc(:,1)], ...
            {surf_lh,surf_rh}, ...
```

(continues on next page)

(continued from previous page)

```
            'parcellation', labeling, ...
            'labeltext',{'FC','MPC'});
```



In order to fuse the matrices, we simply pass the matrices to the fusion command which will rescale and horizontally concatenate the matrices. To do this, you will need the fusion function, which is not included with the BrainSpace package.

```
function fused_matrix = fusion(M)
% FUSION   Fuses matrices from multiple modalities for deep embedding.
%
%   fused_matrix = fusion(M) rank orders the input
%   matrices and scales them to the maximum of the sparsest matrix.
%   M must be a cell array of matrices with the same dimensionality.

% Check the input for nans, infs, and negatives.
if any(cellfun(@(x) any(x(:)<0) || any(isnan(x(:))) || any(isinf(x(:))), M))
error('Negative numbers, nans, and infs are not allowed in the input matrices.');
end

% Check matrix sizes
sz = cellfun(@size,M,'Uniform',false);
if any(cellfun(@numel,sz) > 2)
error('Input matrices may not have more than two dimensions.')
end
if ~all(cellfun(@(x)all(x==sz{1}),sz))
error('Input matrices must have equal dimensions.')
end

% Reshape data to be a vector per input matrix
vectorized = cellfun(@(x)x(:),M,'Uniform',false);
data_matrix = cat(2,vectorized{:});
data_matrix(data_matrix==0) = nan;

% Get rank order and scale to 1 through the maximum of the smallest rank
ranking = tiedrank(data_matrix);
max_val = min(max(ranking));
rank_scaled = rescale(ranking,1,max_val,'InputMin',min(ranking),'InputMax',
↪max(ranking));

% Reshape to output format and remove nans.
fused_matrix = reshape(rank_scaled, size(M{1},1), []);
```

(continues on next page)

(continued from previous page)

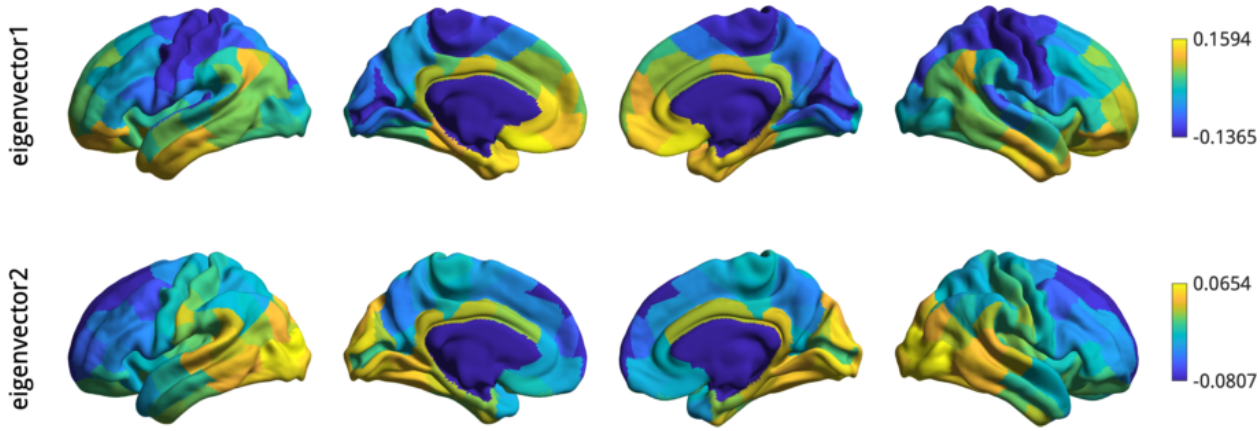
```
fused_matrix(isnan(fused_matrix)) = 0;
end
```

With this function in your path, you can now run the following.

```
% Negative numbers are not allowed in fusion.
fc(fc<0) = 0;

% fuse the matrices
fused_matrix = fusion({fc,mpc});
```

We then use this output in the fit function. This will convert the long horizontal array into a square affinity matrix, and then perform embedding.

```
% resolve the gradients
gm = GradientMaps('kernel','na');
gm = gm.fit(fused_matrix, 'Sparsity', 0);
h = plot_hemispheres(gm.gradients{1}(:,1:2), ...
        {surf_lh,surf_rh}, ...
         'parcellation', labeling, ...
         'labeltext',{'eigenvector1','eigenvector2'});
```



Note: The mpc matrix presented here match the subject cohort of (Paquola et al., 2019). Other matrices in this package match the subject groups used by (Vos de Wael et al., 2018). We make direct comparisons in our tutorial for didactic purposes only.

That concludes the second tutorial. In the third tutorial we will consider null hypothesis testing of comparisons between gradients and other markers.

4.1.3 Tutorial 3: Null models for gradient significance

In this tutorial we assess the significance of correlations between the first canonical gradient and data from other modalities (curvature, cortical thickness and T1w/T2w image intensity). A normal test of the significance of the correlation cannot be used, because the spatial auto-correlation in MRI data may bias the test statistic. In this tutorial we will show two approaches for null hypothesis testing: spin permutations and Moran spectral randomization.

Note: When using these approaches to compare gradients to non-gradient markers, we recommend randomizing the

non-gradient markers as these randomizations need not maintain the statistical independence between gradients.

Spin Permutations

Here, we use the spin permutations approach previously proposed in (Alexander-Bloch et al., 2018), which preserves the auto-correlation of the permuted feature(s) by rotating the feature data on the spherical domain. We will start by loading the conte69 surfaces for left and right hemispheres, their corresponding spheres, midline mask, and t1w/t2w intensity as well as cortical thickness data, and a template functional gradient.

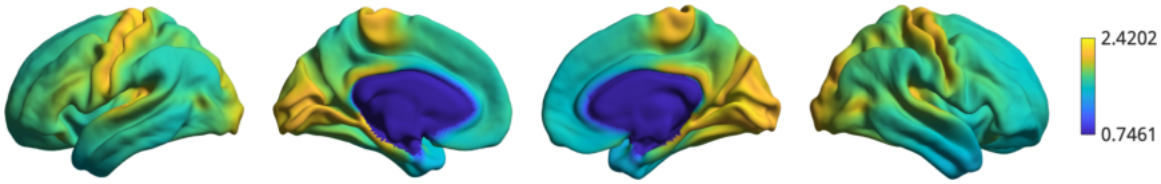
```
% load the conte69 hemisphere surfaces and spheres
[surf_lh, surf_rh] = load_conte69;
[sphere_lh, sphere_rh] = load_conte69('spheres');

% Load the data
[t1wt2w_lh,t1wt2w_rh] = load_marker('t1wt2w');
[thickness_lh,thickness_rh] = load_marker('thickness');

% Template functional gradient
embedding = load_gradient('fc',1);
```

Let's first generate some null data using spintest.

```
% Let's create some rotations
n_permutations = 1000;
y_rand = spin_permutations({[t1wt2w_lh,thickness_lh],[t1wt2w_rh,thickness_rh]}, ...
                  {sphere_lh,sphere_rh}, ...
                  n_permutations,'random_state',0);

% Merge the rotated data into single vectors
t1wt2w_rotated = squeeze([y_rand{1}(:,1,:); y_rand{2}(:,1,:)]);
thickness_rotated = squeeze([y_rand{1}(:,2,:); y_rand{2}(:,2,:)]);
```

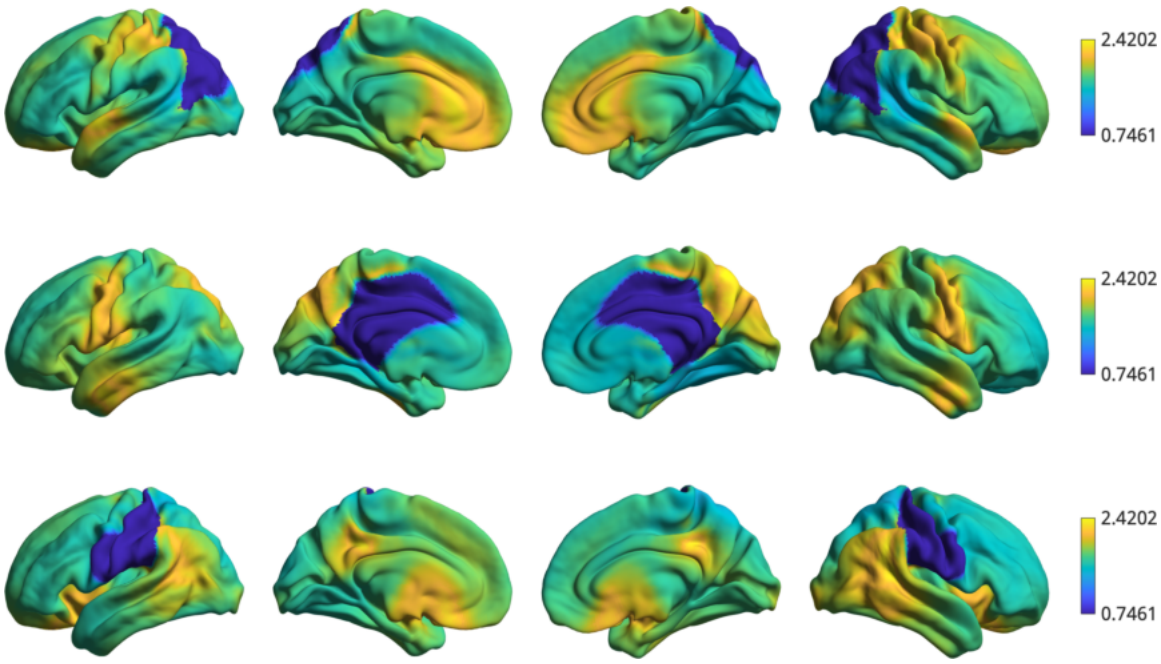
As an illustration of the rotation, let's plot the original t1w/t2w data

```
% Plot original data
h1 = plot_hemispheres([t1wt2w_lh;t1wt2w_rh],{surf_lh,surf_rh});
```



as well as a few rotated version.

```
% Plot a few of the rotations
h2 = plot_hemispheres(t1wt2w_rotated(:,1:3),{surf_lh,surf_rh});
```



Warning: Depending on the overlap of midlines (i.e. NaNs) in the original data and in the rotation, statistical comparisons between them may compare different numbers of features. This can bias your test statistics. Therefore, if a large portion of the sphere is not used, we recommend using Moran spectral randomization instead.

Now we simply compute the correlations between the first gradient and the original data, as well as all rotated data.

```
% Find correlation between FC-G1 with thickness and T1w/T2w
[r_original_thick, pval_thick_spin] = corr(embedding,[thickness_lh;thickness_rh], ...
                'rows','pairwise','type','spearman');
% pval_thick_spin = 0

[r_original_t1wt2w, pval_t1wt2w_spin] = corr(embedding,[t1wt2w_lh;t1wt2w_rh], ...
                'rows','pairwise','type','spearman');
% pval_t1wt2w_spin = 0

r_rand_thick = corr(embedding,thickness_rotated, ...
            'rows','pairwise','type','spearman');
r_rand_t1wt2w = corr(embedding,t1wt2w_rotated, ...
            'rows','pairwise','type','spearman');
```

To find a p-value, we simply compute the percentile rank of the true correlation in the distribution or random correlations. Assuming a threshold of p<0.05 for statistical significance and disregarding multiple comparison corrections, we consider the correlation to be significant if it is lower or higher than the 2.5th/97.5th percentile, respectively.

```
% Compute percentile rank.
 prctile_rank_thick = mean(r_original_thick > r_rand_thick);
 % prctile_rank_thick = 0.9410

 prctile_rank_t1wt2w = mean(r_original_t1wt2w > r_rand_t1wt2w);
 % prctile_rank_t1wt2w = 0

 significant_thick = prctile_rank_thick < 0.025 || prctile_rank_thick >= 0.975;
 significant_t1wt2w = prctile_rank_t1wt2w < 0.025 || prctile_rank_t1wt2w >= 0.975;
```

If significant is true, then we have found a statistically significant correlation. Alternatively, one could also test the one-tailed hypothesis whether the percentile rank is lower or higher than the 5th/95th percentile, respectively.

Moran Spectral Randomization

Moran Spectral Randomization (MSR) computes Moran's I, a metric for spatial auto-correlation and generates normally distributed data with similar auto-correlation. MSR relies on a weight matrix denoting the spatial proximity of features to one another. Within neuroimaging, one straightforward example of this is inverse geodesic distance i.e. distance along the cortical surface.

In this example we will show how to use MSR to assess statistical significance between cortical markers (here curvature and cortical t1wt2w intensity) and the first functional connectivity gradient. We will start by loading the conte69 surfaces for left and right hemispheres, a left temporal lobe mask, t1w/t2w intensity as well as cortical thickness data, and a template functional gradient.

```
% load the conte69 hemisphere surfaces and spheres
[surf_lh, surf_rh] = load_conte69();

% Load the data
t1wt2w_lh = load_marker('t1wt2w');
curv_lh = load_marker('curvature');

% Load mask
temporal_mask_tmp = load_mask('temporal');

% There's a one vertex overlap between the HCP midline mask (i.e. nans) and
% our temporal mask.
temporal_mask_lh = temporal_mask_tmp & ~isnan(t1wt2w_lh);

% Load the embedding
embedding = load_gradient('fc',1);
embedding_lh = embedding(1:end/2);

% Keep only the temporal lobe.
embedding_tl = embedding(temporal_mask_lh);
t1wt2w_tl = t1wt2w_lh(temporal_mask_lh);
curv_tl = curv_lh(temporal_mask_lh);
```

We will now compute the Moran eigenvectors. This can be done either by providing a weight matrix of spatial proximity between each vertex, or by providing a cortical surface (see also: *compute_mem*). Here we'll use a cortical surface.

```
n_ring = 1;
MEM = compute_mem(surf_lh,'n_ring',n_ring,'mask',~temporal_mask_lh);
```

Using the Moran eigenvectors we can now compute the randomized data. As the computationally intensive portion of MSR is mostly in *compute_mem*, we can push the number of permutations a bit further.

```
n_rand = 10000;
y_rand = moran_randomization([curv_tl,t1wt2w_tl],MEM,n_rand, ...
    'procedure','singleton','joint',true,'random_state',0);

curv_rand = squeeze(y_rand(:,1,:));
t1wt2w_rand = squeeze(y_rand(:,2,:));
```

Now that we have the randomized data, we can compute correlations between the gradient and the real/randomized data.

```
[r_original_curv,pval_curv_moran] = corr(embedding_tl,curv_tl,'type','spearman');
% pval_curv_moran = 6.6380e-05

[r_original_t1wt2w,pval_t1wt2w_moran] = corr(embedding_tl,t1wt2w_tl,'type','spearman
↪');
% pval_t1wt2w_moran = 0

r_rand_curv = corr(embedding_tl,curv_rand,'type','spearman');
r_rand_t1wt2w = corr(embedding_tl,t1wt2w_rand,'type','spearman');
```

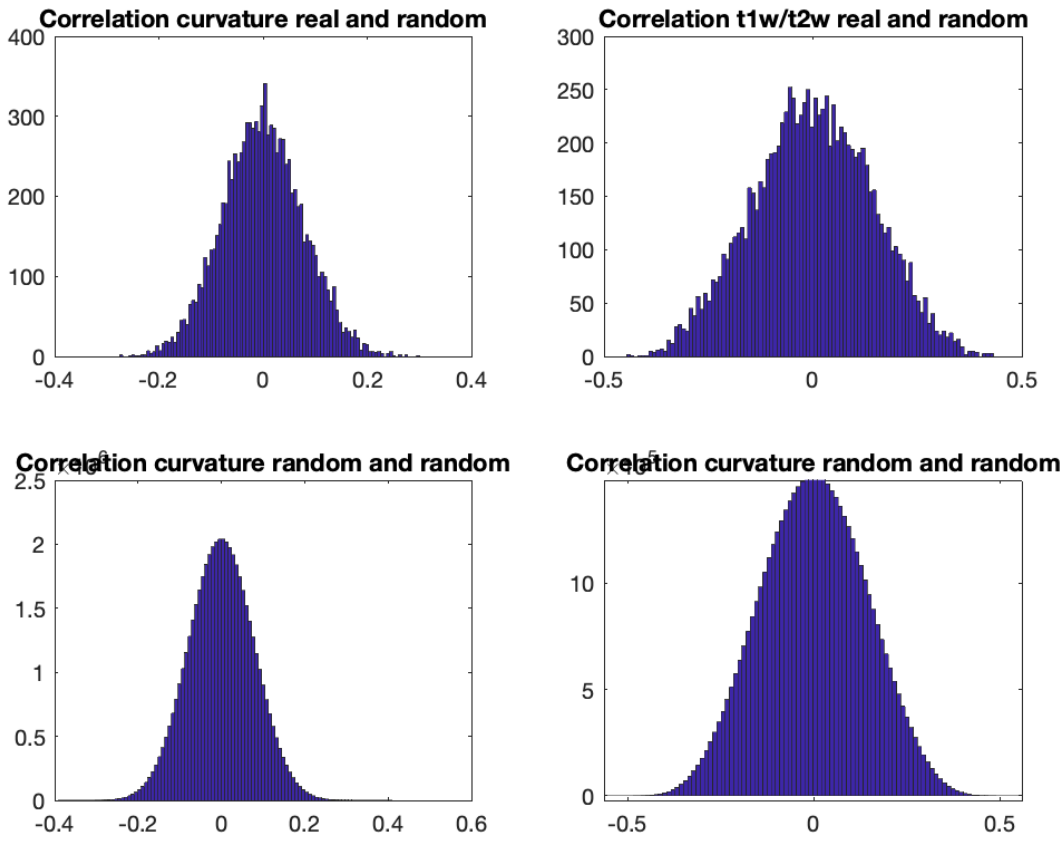
To find a p-value, we simply compute the percentile rank of the true correlation in the distribution or random correlations. Assuming a threshold of p<0.05 for statistical significance and disregarding multiple comparison corrections, we consider the correlation to be significant if it is lower or higher than the 2.5th/97.5th percentile, respectively.

```
prctile_rank_curv = mean(r_original_curv > r_rand_curv);
% prctile_rank_curv = 0.8249

prctile_rank_t1wt2w = mean(r_original_t1wt2w > r_rand_t1wt2w);
% prctile_rank_t1wt2w = 0

significant_curv = prctile_rank_curv < 0.025 || prctile_rank_curv >= 0.975;
significant_t1wt2w = prctile_rank_t1wt2w < 0.025 || prctile_rank_t1wt2w >= 0.975;
```

If significant is true, then we have found a statistically significant correlation. Alternatively, one could also test the one-tailed hypothesis whether the percentile rank is lower or higher than the 5th/95th percentile, respectively.

There are some scenarios where MSR results do not follow a normal distribution. It is relatively simple to check whether this occurs in our data by visualizing the null distributions. Check this interesting paper for more information (Burt et al.,2020).

```
% Compute the correlations between real and random data.
upper_triangle = triu(ones(size(curv_rand,2),'logical'),1);
r_real_rand_curv = corr(curv_tl,curv_rand);
r_real_rand_t1wt2w = corr(t1wt2w_tl,t1wt2w_rand);
r_rand_rand_curv = corr(curv_rand);
r_rand_rand_t1wt2w = corr(t1wt2w_rand);
r_rand_rand_curv = r_rand_rand_curv(upper_triangle);
r_rand_rand_t1wt2w = r_rand_rand_t1wt2w(upper_triangle);

% Plot histograms
figure('Color','w');
subplot(2,2,1);
hist(r_real_rand_curv,100);
title('Correlation curvature real and random');
subplot(2,2,2);
hist(r_real_rand_t1wt2w,100);
title('Correlation t1w/t2w real and random');
subplot(2,2,3);
hist(r_rand_rand_curv,100);
title('Correlation curvature random and random');
subplot(2,2,4);
hist(r_rand_rand_t1wt2w,100);
title('Correlation curvature random and random');
```



Indeed, our histograms appear to be normally distributed. This concludes the third and last tutorial. You should now be familliar with all the functionality of the BrainSpace toolbox. For more details on any specific function, please see *MATLAB Package*.

Variogram Matching

Here, we use the variogram matching presented in (Burt et al., 2020), which generates novel brainmaps with similar spatial autocorrelation to the input data. We will start by loading the conte69 surfaces for left and right hemispheres, a left temporal lobe mask, t1w/t2w intensity as well as cortical thickness data.

```
% load the conte69 hemisphere surfaces and spheres
[surf_lh, surf_rh] = load_conte69();

% Load the data
t1wt2w_lh = load_marker('t1wt2w');
curv_lh = load_marker('curvature');

% Load mask
temporal_mask_tmp = load_mask('temporal');

% There's a one vertex overlap between the HCP midline mask (i.e. nans) and
% our temporal mask.
temporal_mask_lh = temporal_mask_tmp & ~isnan(t1wt2w_lh);
```

(continues on next page)

(continued from previous page)

```
% Load the embedding
embedding = load_gradient('fc',1);
embedding_lh = embedding(1:end/2);

% Keep only the temporal lobe.
t1wt2w_tl = t1wt2w_lh(temporal_mask_lh);
curv_tl = curv_lh(temporal_mask_lh);
```

Next, we will need a distance matrix that tells us what the spatial distance between our datapoints is. For this example, we will use geodesic distance.

```
G = surface_to_graph(surf_lh, 'geodesic', ~temporal_mask_lh, true);
geodesic_distance = distances(G);
```

Now we've got everything we need to generate our surrogate datasets. By default, BrainSpace will use all available data to generate surrogate maps. However, this process is extremely computationally and memory intensive. When using this method with more than a few hundred regions, we recommend subsampling the data. This can be done using the 'ns' and 'knn' name-value pairs. 'ns' determines how many data points to sample for the generation of the variogram; 'knn' determines how many neighbors to use in the smoothing step.

```
random_initialization = 0;
n_surrogate_datasets = 10;

% Note: num_samples must be greater than num_neighbors
num_samples = 200;
num_neighbors = 100;

obj_subsample = variogram(geodesic_distance, 'ns', num_samples, ...
    'knn', num_neighbors, 'random_state', random_initialization);
surrogates_subsample = obj_subsample.fit(t1wt2w_tl, n_surrogate_datasets);
```

Note: The variogram class also supports parallel processing with the 'num_workers' Name-Value pair. This functionality requires the Parallel Computing Toolbox.

The correlation between the real data, surrogate data, and the marker of interest can then be compared as follows. Note that, for this example, we only generated 10 surogate datasets. For any practical usage we recommend generating at least 1000.

```
r_real = corr(t1wt2w_tl, curv_tl);
r_surrogate = corr(surrogates_subsample, curv_tl);
prctile_rank = mean(r_real > r_surrogate);
```

4.2 Main Object

4.2.1 GradientMaps

Synopsis

The core object of the MATLAB BrainSpace package (source code).

Usage

```
gm = GradientMaps(varargin_1);
gm = gm.fit({data_matrix_1,...,data_matrix_n},varargin_2);
```

- *gm*: the GradientMaps object.
- *data_matrix_n*: the input feature matrix
- *varargin_1*: a set of name-value pairs (see below).
- *varargin_2*: a set of name-value pairs (see below).

Description

Properties

The **method** property is a structure array which itself consists of four fields. Each of these is set at the initalization of the object (see below) and cannot be modifed afterwards. The fields are: "kernel", "approach", "alignment", "n_components", and "verbose".

The **gradients** property is a cell array containing the (unaligned) gradients of each input matrix. Each cell is an n-by-m matrix where n is the number of datapoints and m the number of components. In joint embedding the gradients of all data sets are computed simultaneously, and thus no unaligned gradients are stored.

The **aligned** property is a cell array of identical dimensions to the gradients property. If an alignment was requested, then the aligned data are stored here.

The **lambda** property stores the variance explained (for PCA) or the eigenvalues (for LE and DM). Note that all computed lambdas are provided even if this is more than the number of requested components.

Initialization

A basic GradientMaps object can initialized by simply running it without arguments i.e. `gm = GradientMaps();` . However, several name-value pairs can be provided to alter its behavior.

'kernel'

- 'none', ''
- 'pearson', 'p'
- 'spearman', 'sm'
- 'gaussian', 'g'
- 'cosine similarity', 'cs', 'cossim', 'cosine_similarity'
- 'normalized angle', 'na', 'normangle', 'normalized_angle' (default)
- a function handle (the function will be applied to the data matrix)

'approach'

- 'principal component analysis', 'pca'
- 'laplacian eigenmap', 'le'
- 'diffusion embedding', 'dm' (default)
- a function handle (the function will be applied to the post-kernel data matrix)

'alignment'

- 'none', '' (default)
- 'procrustes analysis', 'pa', 'procrustes'
- 'joint alignment', 'ja', 'joint'
- a function handle (the function will be applied to the post-manifold data matrix)

'random_state'

- Any input accepted by MATLAB's `rng` function, or nan to use the current random state. (default: nan)

'n_components'

- Any natural number in double format. (default: 10)

'verbose'

- Determines whether non-warning/error messages are diplayed (default: false).

Putting it all together, an example initialization could be: `gm = GradientMaps('kernel','g', 'approach','pca','alignment','','random_state',10,'verbose',true);`

Public Methods

Public methods are accesible to the end-user. Disregarding the constructor (see initialization section), the GradientMaps class contains one public method.

fit

Uses the settings set in the methods to compute the gradients of all provided data matrices. `varargin` can be used to pro

- sparsity (default: 90)

  Sets the sparsity at which the data matrix is thresholded.

- tolerance (default: 1e-6)

  Floating point errors may cause the kernel to output asymmetric matrices. This number denotes the amount of asymmetry that is allowed before an error is thrown.

- gamma (default: 1 / number_of_data_points)

  The gamma parameter used in the Gaussian kernel.

- alpha (default: 0.5)

  The alpha paramter used in diffusion embedding.

- diffusion_time (default: 0)

  The diffusion time used in diffusion embedding. Leave at 0 for automatic estimation.

- niterations (default: 10)

  The number of iterations in Procrustes analysis.

- reference (default: gradients of the first data matrix)

  The target for alignment for the first iteration of Procrustes analysis.

Example usage: `fit({data_matrix_1,data_matrix_2,...,data_matrix_n},'sparsity', 75)`

Private Methods

Private methods are not accesible to the user, but are called by other methods i.e. GradientMaps initialization and GradientMaps.fit. The GradientMaps class contains three private methods. As these methods are not intended for user interaction, we only provide a basic explanation here.

- *set(obj,varargin)*: used for setting properties of the GradientMaps class.
- *kernels(obj,data,varargin)*: performs kernel computations.
- *approaches(obj,data,varargin)*: performs dimensionality reduction.

4.3 Surface Handling

4.3.1 read_surface

Synopsis

Reads surfaces (source code).

Usage

```
surf_out = read_surface(file);
```

- *file*: File to read from.
- *surf_out*: Output surface.

Description

Reads surfaces from the disk. Accepted formats are gifti files (provided the gifti library is installed), freesurfer files, .mat files and .obj files.

When provided with a .mat file, the file must contain variables corresponding to a MATLAB surface i.e. 'vertices' and 'faces' or a SurfStat surface i.e. 'coord and 'tri'. 'vertices' is an n-by-3 list of vertex coordinates, 'faces' is an m-by-3 matrix of triangles, 'coord', is a 3-by-n list of vertex coordinates, and 'tri' is identical to faces in the MATLAB format.

4.3.2 write_surface

Synopsis

Writes surfaces (source code).

Usage

```
write_surface(surface,path);
```

- *surface*: surface loaded into MATLABB
- *path*: path to write to

Description

Writes surfaces to the designated output path. Accepted formats are .gii files (provided the gifti library is installed), freesurfer files, .mat files and .obj files. File type is determined by the extension; if the filetype is not recognized the default is freesurfer format.

When saving a .mat file, the first will save the fields inside the surface structure.

4.3.3 combine_surfaces

Synopsis

Combines two surfaces into a single surface (source code).

Usage

```
SC = combine_surfaces(S1,S2);
SC = combine_surfaces(S1,S2,format);
```

- *S1*, *S2*: two surfaces
- *format*: the output format; either 'SurfStat' (default) or 'MATLAB'.
- *S*: Combined surface.

Description

This can be used to merge left and right hemisphere surfaces into a single surface. The input surfaces can be any file readable by *read_surface* or a surface loaded into memory.

A MATLAB structure consists of two fields: vertices, which is a n-by-3 list of vertex coordinates, and faces, which is a m-by-3 matrix of triangles. A SurfStat format consists of two fields: coord, which is a 3-by-n list of vertex coordinates, and tri, which is identical to faces.

4.3.4 split_surfaces

Synopsis

Splits surfaces in a single variable into separate variables. (source code).

Usage

```
varargout = split_surfaces(surface);
```

- *surface*: surface loaded into MATLABB
- *varargout*: Variables to load the surfaces into.

Description

If you have a variable containing multiple disconnected surfaces, this function will split them up and return each in a separate output variable. Accepts all surfaces readable by *convert_surface*.

4.3.5 full2parcel

Synopsis

Converts data from a full data matrix to parcellated data (source code).

Usage

```
parcellated_data = full2parcel(data,parcellation);
```

- *data*: an n-by-m data matrix.
- *parcellation*: a 1-by-m parcel vector containing natural numbers.
- *parcellated_data*: n-by-max(parcellation) matrix containing the mean column of each parcel.

Description

A useful tool for quickly moving data between vertex and parcel level, especially in combination with *parcel2full*. For more flexible usage, see also *labelmean*.

4.3.6 parcel2full

Synopsis

Converts data from a parcellated data matrix to full data matrix by assigning each vertex the value of its parcel (source code).

Usage

```
full_data = parcel2full(parcellated_data,parcellation);
```

- *parcellated_data*: an n-by-max(parcellation) data matrix.
- *parcellation*: a 1-by-m parcel vector containing natural numbers.
- *full_data*: n-by-m matrix containing the data at the vertex level.

Description

A useful tool for quickly moving data between parcel and vertex level, especially in combination with *full2parcel*. This is mostly used for plotting parcellated data on the surface.

4.4 Null Hypothesis Testing

4.4.1 spin_permutations

Synopsis

Performs a spin test to generate null data for hypothesis testing (source code).

Usage

```
null_data = spin_permutations(data,spheres,n_rep,varargin);
```

- *data*: An n-by-m matrix of data to be randomised (single sphere) or cell array of n-by-m matrices (two spheres).
- *spheres*: Cell array containing spheres.
- *n_rep*: The number of permutation to perform.
- *null_data*: The randomised data.
- *varargin*: See name-value pairs below.

Description

Spin test as described by (Alexander-Bloch, 2018). Data is either an n-by-m matrix where n is the number of vertices and/or parcels or, when spinning on two spheres, a two-element cell array each containing an n-by-m matrix. Spheres is either a file in a format readable by *read_surface*, a sphere loaded into memory, or a two-element cell array containing files/spheres corresponding to the respective elements in the data cell array.

Name-Value pairs

- *'parcellation'*: a n-by-1 vector containing the parcellation scheme. If you are performing vertexwise analysis, do not provide this pair.
- *'surface_algorithm'*: program used to generate the spheres. Either 'FreeSurfer' (default), or 'CIVET'. If Freesurfer, rotations are flipped along the x-axis for the second sphere.

4.4.2 compute_mem

Synopsis

Computes the moran eigenvectors required for Moran spectral randomization (source code).

Usage

```
MEM = compute_mem(W,varargin);
```

- *W*: A matrix denoting distance between features or a cortical surface.
- *varargin*: Name-value pairs (see below).

Description

The Moran eigenvectors hold information on the spatial autocorrelation of the data. Their computation is kept separate from the randomization as these eigenvectors can be used for any feature on the same surface. These eigenvectors are used for Moran spectral randomization by *moran_randomization*. See also (Wagner and Dray, 2015).

If W is provided as a cortical surface, then this can either be a surface in memory or a file readable by *read_surface*.

Name-Value pairs

- *n_ring*: Only used if W is a surface. Vertices that are within *n_ring* steps of each other have their distance computed (Default: 1).
- *mask*: Only used if W is a surface. A n-by-1 logical denoting a mask. `n` denotes the number of vertices. Vertices corresponding to True values are discarded when computing the eigenvectors. You can also provide an empty logical array to discard nothing (Default: []).
- *spectrum*: Determines the behavior for discarding eigenvectors with eigenvalue=0. Set to 'all' for discarding only one and reorthogonalizing the remainder or 'NonZero' for discarding all zero eigenvalues (Default: 'all').

4.4.3 moran_randomization

Synopsis

Computes the moran eigenvectors required for Moran spectral randomization (source code).

Usage

```
Y_rand = moran_randomization(Y,MEM,n_rep,varargin);
```

- *Y*: An n-by-m data matrix to randomize where n is number of datapoints and m are different modalities.
- *MEM*: Moran eigenvectors as returned by *compute_mem*.
- *n_rep*: Number of pertubations
- *varargin*: See name-value pairs below.
- *Y_rand*: Randomized data.

Description

Implementation of Moran spectral randomization as presented by (Wagner and Dray, 2015). This function uses the eigenvectors computed by *compute_mem* to generate null model data with similar spatial autocorrelation. The implemented procedures are 'singleton' and 'pair'. Singleton matches the input data's autocorrelation more closely at the cost of fewer possible randomizations (max: 2^n). In most use-cases this allows for ample randomizations. In cases where the maximum number of randomizations becomes restrictive, we recommend using the pair procedure instead.

Name-Value Pairs

- *procedure*: Randomization procedure; either 'singleton' or 'pair'.
- *joint*: If true, randomizes different modalities identically.
- *random_state*: Initilaization of the random state. Accepts any argument accepted by rng() or nan for no initialization.

4.5 Visualization

4.5.1 gradient_in_euclidean

Synopsis

Plots gradient data in 3D Euclidean space (source code).

Usage

```
handles = gradient_in_euclidean(gradients)
handles = gradient_in_euclidean(gradients,surface)
handles = gradient_in_euclidean(gradients,surface,parcellation)
```

- *gradients*: an n-by-3 matrix of gradients (usually the first three).
- *surface*: a surface readable by *convert_surface* or a two-element cell array containing left and right hemispheric surfaces in that order.
- *parcellation*: an m-by-1 vector containing the parcellation scheme, where m is the number of vertices.
- *handles*: a structure containing the handles of the graphics objects.

Description

Produces a 3D scatter plot of gradient data in Euclidean space with each datapoint color coded by their location. If provided a surface (and parcellation), also produces surface plots with the colors projected back to the surface.

BrainSpace only provides basic figure building functionality. For more information on how to use MATLAB to create publication-ready figures we recommend delving into graphics object properties (e.g. figure, axes, surface).

4.5.2 plot_hemispheres

Synopsis

Plots data on the cortical surface (source code).

Usage

```
obj = plot_hemispheres(data,surface,varargin);
```

- *data*: an n-by-m vector of data to plot where n is the number of vertices or parcels, and m the number of markers to plot.
- *surface*: a surface readable by *convert_surface* or a two-element cell array containing left and right hemispheric surfaces in that order.
- *varargin*: Name-Value Pairs (see below).
- *obj*: an object allowing for further modificatio of the figure (see below).

Description

Plots any data vector onto cortical surfaces. Data is always provided as a single vector; if two surfaces are provided then the *n* vertices of the first surface will be assigned datapoints 1:*n* and the second surface is assigned the remainder. If a parcellation scheme is provided, data should have as many datapoints as there are parcels.

BrainSpace only provides basic figure building functionality. For more information on how to use MATLAB to create publication-ready figures we recommend delving into graphics object properties (e.g. figure, axes, surface). Also see the source code for basic graphic object property modifications.

Name-Value Pairs

- *parcellation*: an k-by-1 vector containing the parcellation scheme, where k is the number of vertices. Default [].
- *labeltext*: A cell array of m elements containing labels for each column of data. These will be printed next to the hemispheres. Default: [].
- *views*: A character vector containing the requested view angles. Options are: l(ateral), m(edial), i(nferior), s(uperior), a(nterior), and p(osterior). Default: 'lm'.

Public methods

Public methods can be used with obj.(method) e.g. obj.colorlimits to use the colorlimits method.

- *colorlimits(limits)*: Sets the color limits of each row. Limits must be a 2-element numeric vector or n-by-2 matrix where n is the number of columns in obj.data. The first column of limits denotes the minimum color limit and the second the maximum. When limits is a 2-element vector, then the limits are applied to all axes, with limits(1) as minimum and limits(2) as maximum.
- *colorMaps(maps)*: Maps must either be an n-by-3 color map, or a cell array with the same number of elements as columns in obj.data, each containing n-by-3 colormaps.
- *labels(varargin)*: Modifies the properties of the labeltext. Varargin are valid name-value pairs for MATLAB's text function. e.g. *obj.labels('FontSize',25)*

4.5.3 scree_plot

Synopsis

Produces a scree plot of the lambdas (source code).

Usage

```
handles = scree_plot(lambdas)
```

- *lambdas*: a vector of lambdas; can be taken from the *GradientMaps* object.
- *handles*: a structure containing the handles of the graphics objects.

Description

scree_plot plots the lambdas scaled to a sum of 1. It is a useful tool for identifying the difference in the importance of each eigenvector (i.e. gradient) with higher lambdas being more important in principal component analysis and diffusion embedding, and lower ones more important in Laplacian eigenmaps.

BrainSpace only provides basic figure building functionality. For more information on how to use MATLAB to create publication-ready figures we recommend delving into graphics object properties (e.g. figure, axes, surface).

4.6 Data Loaders

4.6.1 load_conte69

Synopsis

Loads conte69 surfaces (source code).

Usage

```
[surf_lh,surf_rh] = load_conte69(name)
```

- *name* The type of surface. Either 'surfaces' for neocortex, 'spheres' for cortical spheres or '5k_surfaces' for downsampled neocortex.
- *surf_lh* Left hemispheric surface.
- *surf_rh* Right hemispheric surface.

4.6.2 load_group_fc

Synopsis

Loads group level functional connectivity matrices (source code).

Usage

```
conn_matrices = load_group_fc(parcellation,scale,group)
```

- *parcellation*: Name of the parcellation, either 'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of the Desikan-Killiany atlas. Both may be provided as a cell or string array.
- *scale*: Scale of the parcellation. Either 100, 200, 300, or 400. Multiple may be provided as a vector.
- *group*: Loads data from the main group if set to 'main' (default) or the holdout group if set to 'holdout'.
- *conn_matrices*: Structure of all requested data.

Note: Data matrices were drived from the discovery (main) and validation (holdout) groups of (Vos de Wael et al., 2018).

4.6.3 load_group_mpc

Synopsis

Loads group level microstructural profile covariance matrices (source code).

Usage

```
conn_matrices = load_group_mpc(parcellation,scale,group)
```

- *parcellation*: Name of the parcellation, either 'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of the Desikan-Killiany atlas. Both may be provided as a cell or string array.
- *scale*: Scale of the parcellation. Either 100, 200, 300, or 400. Multiple may be provided as a vector.
- *group*: Loads data from the main group if set to 'main' (default) or the holdout group if set to 'holdout'.
- *conn_matrices*: Structure of all requested data.

Note: The mpc matrix presented here match the subject cohort of (Paquola et al., 2019). Other matrices in this package match the subject groups used by (Vos de Wael et al., 2018). We make direct comparisons in our tutorial for didactic purposes only.

4.6.4 load_mask

Synopsis

Loads cortical masks (source code).

Usage

```
[mask_lh,mask_rh] = load_mask(name)
```

- *name*: Type of mask: either 'midline' for the midline or 'temporal' for the temporal lobe.
- *mask_lh*: Left hemispheric mask.
- *mask_rh*: Right hemispheric mask.

4.6.5 load_marker

Synopsis

Loads metric data (source code).

Usage

```
[metric_lh,metric_rh] = load_marker(name)
```

- *name*: The type of surface. Either 'thickness' for cortical thickness, 't1wt2w' for t1w/t2w intensity or 'curvature' for curvature.
- *metric_lh*: Data on left hemisphere.
- *metric_rh*: Data on right hemisphere.

Note: Data matrices were drived from the discovery (main) group of (Vos de Wael et al., 2018).

4.6.6 load_parcellation

Synopsis

Loads parcellation schemes (source code).

Usage

```
labels = load_parcellation(parcellation,scale)
```

- *parcellation*: Name of the parcellation, either 'schaefer' for Schaefer (functional) parcellations or 'vosdewael' for a subparcellation of the Desikan-Killiany atlas. Both may be provided as a cell or string array.
- *scale*: Scale of the parcellation. Either 100, 200, 300, or 400. Multiple may be provided as a vector.
- *labels*: Structure containing the parcellation schemes.

4.6.7 load_gradient

Synopsis

Loads template gradients (source code).

Usage

```
data = load_gradient(name,number)
```

- *name*: The type of gradient, either 'fc' for functional connectivity or 'mpc' for microstructural profile covariance.
- *number*: The rank of the gradient, either 1 or 2.
- *data*: Output data.

Description

Loads normative functional connectivity and microstructural profile covariance gradients, both computed from the HCP dataset. Gradients were computed with a cosine similarity kernel and diffusion mapping on a downsampled 5K cortical mesh. Resulting gradients were upsampled to the 32K mesh.

4.7 Support Functions

4.7.1 SurfStatReadSurf1

Synopsis

SurfStat's surface loader, used here for loading Freesurfer and .obj files. For surface loading in BrainSpace, please use *read_surface* instead (source code).

4.7.2 SurfStatWriteSurf1

Synopsis

SurfStat's surface writer, used here for writing Freesurfer and .obj files. For surface writing in BrainSpace, please use *write_surface* instead (source code).

4.7.3 convert_surface

Synopsis

Loads, converts, and writes surfaces (source code).

Usage

```
S = convert_surface(file);
S = convert_surface(surface_struct,varargin);
```

- *file*: path to a gifti, freesurfer, or .obj file.
- *surface_struct*: a surface stored either in MATLAB format or SurfStat format.
- *varargin*: see Name-Value pairs below.
- *S*: Output surface.

Description

This function is BrainSpace's surface loader/writer. When provided with the path to a surface file it will load gifti files (provided the gifti library is installed), freesurfer files, .mat files and .obj files. It can also be provided with a structure variable containing a surface in either MATLAB format or SurfStat format. When provided with a 'path' name-value pair, the input surface will also be written to the disk.

A MATLAB surface is a structure consisting of two fields: 'vertices', which is an n-by-3 list of vertex coordinates, and 'faces', which is an m-by-3 matrix of triangles. A SurfStat format consists of two fields: 'coord', which is a 3-by-n list of vertex coordinates, and 'tri', which is identical to faces in the MATLAB format.

Name-Value pairs

- *format*: the format to convert to; either SurfStat (default) or MATLAB
- *path*: a path to write the surface to (Default: ''). If used, only one surface can be provided in S.

4.7.4 diffusion_mapping

Synopsis

Performs the diffusion mapping computations (source code).

Usage

```
[gradients,lambdas] = diffusion_mapping(data, n_components, alpha, diffusion_time,␣
↪random_state);
```

- *data*: the data matrix to perform the diffusion mapping on.
- *n_components*: the number of components to return.
- *alpha*: the alpha parameter.
- *diffusion_time*: the diffusion_time parameter; set to 0 for automatic estimation.
- *random_state*: Input passed to the rng() function for randomization initialization (default: no initialization).
- *gradients*: the output gradients.
- *lambdas*: the output eigenvalues.

Description

Performs the diffusion procedure. This implementation is based on the mapalign package by Satrajid Ghosh.

4.7.5 labelmean

Synopsis

Takes the mean of columns with the same label (source code).

Usage

```
label_data = labelmean(data,label,varargin);
```

- *data*: an n-by-m data matrix.
- *label*: a 1-by-m label vector containing natural numbers.
- *label_data*: n-by-max(label) matrix containing the mean column of each label.
- *varargin*: See below.

Description

A fast way of computing the mean for each label. Accepts the following additional arguments in varargin:

- *sortByLabelVector*: Sorts the columns of `label_data` by the order of appearance in the `label` vector, rather than ascending order.
- *ignoreWarning*: Does not display order of column sorting or warnings due to 0s in the label vector.

4.7.6 laplacian_eigenmap

Synopsis

Performs the laplacian eigenmap computations (source code).

Usage

```
[gradients, lambdas] = laplacian_eigenmap(data, n_components)
```

- *data*: the data matrix to perform the laplacian eigenmapping on.
- *n_components*: the number of components to return.
- *gradients*: the output gradients.
- *mapping*: structure containing the output gradients and lambdas.

Description

Performs the laplacian eigenmap procedure. Original implemented in the MATLAB Toolbox for Dimensionality Reduction by Laurens van der Maaten. Modified for use in the BrainSpace toolbox.

4.7.7 procrustes_alignment

Synopsis

Performs gradient alignment through iterative singular value decomposition (source code).

Usage

```
aligned = procrustes_alignment(gradients,varargin)
```

- *gradients*: cell array of gradients
- *varargin*: Name-value pairs (see below).
- *aligned*: Aligned gradients.

Description

Gradient alignment through Procrustes analysis [see (Langs et al., 2015)]. On the first iteration all gradients are aligned to either the first provided gradient set or to a template (see Name-Value pairs) using singular value decomposition. On every subsequent iteration alignment is to the mean.

Name-Value pairs

- *nIterations*: Number of iterations to run (default: 100).
- *reference*: Template to align to on the first iteration (default: first provided gradient set)

4.7.8 surface_to_graph

Synopsis

Converts a surface to a graph (source code).

Usage

```
G = surface_to_graph(S,distance)
G = surface_to_graph(S,distance,mask)
G = surface_to_graph(S,distance,mask,removeDegreeZero)
```

- *G*: Output graph
- *S*: a surface.
- *distance*: either 'geodesic' for returning a weighted graph, or 'mesh' for unweighted.
- *mask*: a logical mask where True denotes vertices to remove (Default: []).
- *removeDegreeZero*: a logical, if true then vertices with degree 0 are removed from the output graph.

Description

Converts surfaces readable by *convert_surface* into a graph. Masks can be used to remove portions of the surfaces (e.g. midline)

CHAPTER 5

References

When using BrainSpace, please cite the following manuscript:

- Vos de Wael R, Benkarim O, Paquola C, Lariviere S, Royer J, Tavakol S, Xu T, Hong S, Langs G, Valk S, Misic B, Milham M, Margulies D, Smallwood J, Bernhardt B (2020). BrainSpace: a toolbox for the analysis of macroscale gradients in neuroimaging and connectomics datasets. Commun Biol 3, 103.

The following references used methodology similar, or identical, to those described here:

Functional gradients of the adult connectome.

- Margulies D, Ghosh SS, Goulas A, Falkiewicz M, Huntenburg JM, Langs G, Bezgin G, Eickhoff S, Castellanos FX, Petrides M, Jefferies E, Smallwood J (2016) Situating the default-mode network along a principal gradient of macroscale cortical organization. PNAS

Functional gradients of the neonatal connectome.

- Larivière S, Vos de Wael R, Hong S, Lowe A, Tavakol S, Paquola C, Schrader DS, Bernhardt BC (2019) Multiscale structure-function gradients in the neonatal connectome. Cerebral Cortex. 2019 April 10. doi: 10.1093/cercor/bhz069

Functional gradients in autism.

- Hong S, Vos de Wael R, Bethlehem R, Lariviere S, Paquola C, DiMartino A, Milham M, Margulies D, Smallwood J, Bernhardt BC (2019) Atypical functional connectome hierarchy in autism. Nature Communications, in press

Functional gradients in aging.

- Lowe A, Paquola C, Vos de Wael R, Girn M, Caldairou B, Larivière S, Tavakol S, Royer J, Schrader DV, Bernasconi N, Bernasconi A, Spreng RN, Bernhardt BC (2019) Targeting age-related differences in brain and cognition with multimodal imaging and connectome topography profiling. Human Brain Mapping, in press.

Functional gradients of the nonhuman primate connectome.

- Manea, A. M. G., Zilverstand, A., Ugurbil, K., Heilbronner, S. R., & Zimmermann, J. (2021). Intrinsic timescales as an organizational principle of neural processing across the whole rhesus macaque brain. bioRxiv, 2021.2010.2005.463277. doi:10.1101/2021.10.05.463277

Meta-analytic co-activation and functional gradients in the hippocampus.

- Vos de Wael R, Larivière S, Caldairou B, Hong S, Jefferies E, Margulies DS, Smallwood J, Bernasconi N, Bernhardt BC (2018) Anatomical and microstructural determinants of hippocampal subfield functional connectome embedding. PNAS 115(40):10154-10159.

Gradients of microstructural markers.

- Paquola C, Vos de Wael R, Wagstyl K, Bethlehem R, Seidlitz J, Hong S, Bullmore ET, Evans AC, Misic B, Margulies DS, Smallwood J, Bernhardt BC (2019) Dissociations between microstructural and functional hierarchies within regions of transmodal cortex. PLoS Biology, in press.
- Paquola C, Bethlehem RAI, Seidlitz J, Wagstyl K, Romero-Garcia, Whitaker KJ, Vos de Wael R, Williams GB, NSPN Consortium, Vertes PE, Bernhardt BC, Bullmore ET (2019). A moment of change: shifts in myeloarchitecture profiles characterize adolescent development of cortical gradients. Preprint: https://www.biorxiv.org/content/10.1101/706341v1

Please send us your gradient papers, so that we can list them here as well!

CHAPTER 6

Funding

Our research is kindly supported by:

- Canadian Institutes of Health Research (CIHR)
- National Science and Engineering Research Council of Canada (NSERC)
- Azrieli Center for Autism Research
- The Montreal Neurological Institute
- SickKids Foundation

We also like to thank these funders for training/salary support

- Savoy Foundation for Epilepsy (to RvdW)
- Healthy Brain and Healthy Lives (to OB)
- Fonds de la Recherche du Quebec - Sante (to BB)

CHAPTER 7

Authors

- Reinder Vos de Wael, MICA Lab - Montreal Neurological Institute
- Oualid Benkarim, MICA Lab - Montreal Neurological Institute
- Boris Bernhardt, MICA Lab - Montreal Neurological Institute

CHAPTER 8

License

The BrainSpace source code is available under the BSD (3-Clause) license.

CHAPTER 9

Support

If you have problems installing the software or questions about usage and documentation, or something else related to BrainSpace, you can post to the Issues section of our repository.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W