surfplot

Release 0.2.0

Dan Gale

Dec 20, 2022

CONTENTS

`surfplot` is a flexible and easy-to-use package that makes publication-ready brain surface plots. Users can easily set the plot views and layout, add multiple data layers, draw outlines, and further customize their figure directly using matplotlib.

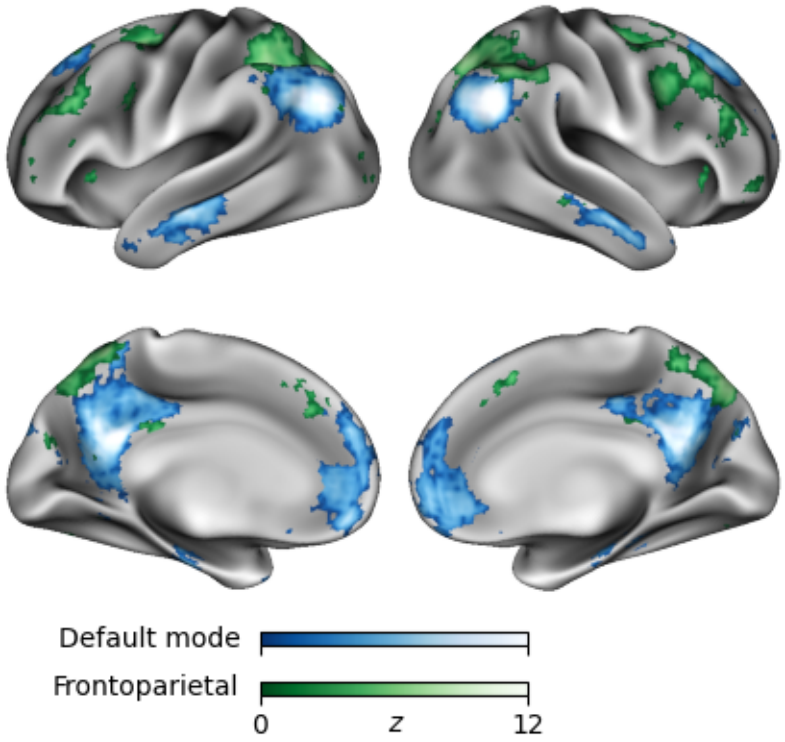


Fig. 1: Example Neurosynth association maps; see Example 1

At its core, `surfplot` is simply a high-level interface to Brainspace's excellent surface plotting and manipulation capabilities, which are built on top of Visualization Toolkit (VTK). Surfaces are rendered with Brainspace and then embedded into a matplotlib figure for easy integration with typical plotting workflows. A big thank you to the `Brainspace` developers for making this package possible.

`surfplot` is designed around common use-cases for surface plotting and popular surface plotting software (e.g., Connectome Workbench). `surfplot` also provides some additional utility functions to streamline the plotting process.

CHAPTER

ONE

GETTING STARTED

Follow the Installation Instructions to install `surfplot`, and then check out the Tutorials and Examples to learn how to get up and running! Refer to the API reference for complete documentation.

CHAPTER

TWO

CITING SURFPLOT

Please cite the following if you use `surfplot`:

Gale, Daniel J., Vos de Wael., Reinder, Benkarim, Oualid, & Bernhardt, Boris. (2021). Surfplot: Publication-ready brain surface figures (v0.1.0). Zenodo. https://doi.org/10.5281/zenodo.5567926

Vos de Wael R, Benkarim O, Paquola C, Lariviere S, Royer J, Tavakol S, Xu T, Hong S-J, Langs G, Valk S, Misic B, Milham M, Margulies D, Smallwood J, Bernhardt BC. 2020. BrainSpace: a toolbox for the analysis of macroscale gradients in neuroimaging and connectomics datasets. *Communications Biology*. 3:103. https://doi.org/10.1038/s42003-020-0794-7

CHAPTER

THREE

LICENSE INFORMATION

This codebase is licensed under the 3-clause BSD license. The full license can be found in the LICENSE file in the `surfplot` distribution.

CHAPTER

FOUR

SUPPORT

If you encounter problems or bugs with `surfplot`, or have questions or improvement suggestions, please feel free to get in touch via the Github issues.

CHAPTER

FIVE

CONTENTS

5.1 Installation

`surfplot` is installable via `pip`:

```
pip install surfplot
```

Alternatively, you can install the most up-to-date version of from GitHub:

```
git clone https://github.com/danjgale/surfplot.git
cd surfplot
pip install .
```

Note that `surfplot` requires Python 3.7+ and some key dependencies:

- brainspace (>=0.1.1)
- matplotlib (>=3.3.0)
- numpy (>=1.16.0)
- nibabel (>=3.0.0)
- vtk (>=8.1.0)

Several of the *Tutorials and Examples* make use of additional neuroimaging packages. Although these packages are not required for installation, they are recommended to make your surfplotting life a bit easier. These are:

- nilearn (for manipulating images and volume-to-surface projections)
- neuromaps (for volume-to-surface projections and surface-to-surface resampling)

5.2 API Reference

5.2.1 `surfplot.datasets` - Dataset fetchers

Convenient data fetchers for tutorial examples

surfplot.datasets.load_example_data([...])	Load example datasets for tutorials

surfplot.datasets.load_example_data

surfplot.datasets.**load_example_data**(*dataset='default_mode'*, *join=False*)

Load example datasets for tutorials

Precomputed association maps for terms 'default mode' or 'frontoparietal' using Neurosynth. In brief, maps were downloaded from Neurosynth and projected to fsLR space using `neuromaps.transforms.mni152_to_fslr`. Then, their vertex arrays were saved.

1. Default mode: https://www.neurosynth.org/analyses/terms/default%20mode/
2. Frontoparietal: https://www.neurosynth.org/analyses/terms/frontoparietal/

Yarkoni T, Poldrack RA, Nichols TE, Van Essen DC, Wager TD. 2011. Large-scale automated synthesis of human functional neuroimaging data. Nat Methods. 8:665–670.

Parameters

- **dataset** (*{'default_mode', 'frontoparietal'}, optional*) – Neurosynth association map. Default: 'default_mode'
- **join** (*bool, optional*) – Return data as a single concatenated array. Default: False, which returns left and right hemisphere arrays, respectively

Returns

numpy.ndarray – Vertex array(s)

5.2.2 surfplot.plotting - Plotting

Main module containing the Plot class

surfplot.plotting.Plot([surf_lh, surf_rh, ...])	Plot brain surfaces with data layers

surfplot.plotting.Plot

class surfplot.plotting.**Plot**(*surf_lh=None*, *surf_rh=None*, *layout='grid'*, *views=None*, *mirror_views=False*, *flip=False*, *size=(500, 400)*, *zoom=1.5*, *background=(1, 1, 1)*, *label_text=None*, *brightness=0.5*)

Plot brain surfaces with data layers

Parameters

- **surf_lh** (*str or os.PathLike or BSPolyData, optional*) – Left and right hemisphere cortical surfaces, either as a file path to a valid surface file (e.g., .gii. .surf) or a pre-loaded surface from `brainspace.mesh.mesh_io.read_surface()`. At least one hemisphere must be provided. Default: None
- **surf_rh** (*str or os.PathLike or BSPolyData, optional*) – Left and right hemisphere cortical surfaces, either as a file path to a valid surface file (e.g., .gii. .surf) or a pre-loaded surface from `brainspace.mesh.mesh_io.read_surface()`. At least one hemisphere must be provided. Default: None
- **layout** (*{'grid', 'column', 'row'}, optional*) – Layout in which to plot brain surfaces. 'row' plots brains as a single row ordered from left-to-right hemispheres (if applicable), 'column' plots brains as a single column descending from left-to-right hemispheres (if applicable). 'grid' plots surfaces as a views-by-hemisphere (left-right) array; if only one hemipshere is provided, then 'grid' is equivalent to 'row'. By default 'grid'.

- **views** (*{'lateral', 'medial', 'dorsal', 'ventral', 'anterior',}*) – 'posterior'}, str or list[str], optional Views to plot for each provided hemisphere. Views are plotted in in the order they are provided. If None, then lateral and medial views are plotted. Default: None
- **mirror_views** (*bool, optional*) – Flip the order of the right hemisphere views for 'row' or 'column' layouts, such that they mirror the left hemisphere views. Ignored if *surf_rh* is None and *layout* is 'grid'.
- **flip** (*bool, optional*) – Flip the display order of left and right hemispheres in *grid* or *row* layouts, if applicable. Useful when showing only 'anterior` or 'inferior' views. Default: False
- **size** (*tuple of int, optional*) – The size of the space to plot surfaces, defined by (width, height). Note that this differs from *figsize* in Plot.build(), which determines the overall figure size for the matplotlib figure. Default: (500, 400)
- **zoom** (*int, optional*) – Level of zoom to apply. Default: 1.5
- **background** (*tuple, optional*) – Background color, default: (1, 1, 1)
- **label_text** (*dict[str, array-like], optional*) – Brainspace label text for column/row. Possible keys are {'left', 'right', 'top', 'bottom'}, which indicate the location. See brainspace.plotting.surface_plotting.plot_surf for more details Default: None.
- **brightness** (*float, optional*) – Brightness of plain gray surface. 0 = black, 1 = white. Default: .5

Raises

ValueError – Neither *surf_lh* or *surf_rh* are provided

_add_colorbars(*location='bottom'*, *label_direction=None*, *n_ticks=3*, *decimals=2*, *fontsize=10*, *draw_border=True*, *outer_labels_only=False*, *aspect=20*, *pad=0.08*, *shrink=0.3*, *fraction=0.05*)

Draw colorbar(s) for applicable layer(s)

Parameters

- **location** (*{'left', 'right', 'top', 'bottom'}, optional*) – The location, relative to the surface plot. If location is 'top' or 'bottom', then colorbars are horizontal. If location is'left' or 'right', then colorbars are vertical.
- **label_direction** (*int or None, optional*) – Angle to draw label for colorbars, if provided. Horizontal = 0, vertical = 90. If None and *location* is 'top' or 'bottom', labels are drawn horizontally. If None and *location* is 'left' or 'right', labels are drawn vertically. Default: None
- **n_ticks** (*int, optional*) – Number of ticks to include on colorbar, default: 3 (minimum, maximum, and middle values)
- **decimals** (*int, optional*) – Number of decimals to show for colorbal tick values. Set 0 to show integers. Default: 2
- **fontsize** (*int, optional*) – Font size for colorbar labels and tick labels. Default: 10
- **draw_border** (*bool, optional*) – Draw ticks and black border around colorbar. Default: True
- **outer_labels_only** (*bool, optional*) – Show tick labels for only the outermost colorbar. This cleans up tick labels when all colorbars are the same scale. Default: False
- **pad** (*float, optional*) – Space that separates each colorbar. Default: .08

- **aspect** (*float, optional*) – Ratio of long to short dimensions. Default: 20
- **shrink** (*float, optional*) – Fraction by which to multiply the size of the colorbar. Default: .3
- **fraction** (*float, optional*) – Fraction of original axes to use for colorbar. Default: .05

add_layer(*data*, *cmap='viridis'*, *alpha=1*, *color_range=None*, *as_outline=False*, *zero_transparent=True*, *cbar=True*, *cbar_label=None*)

Add plotting layer to surface(s)

Parameters

- **data** (*str or os.PathLike, numpy.ndarray, dict, nibabel.gifti.gifti.GiftiImage, or nibabel.cifti2.cifti2.Cifti2Image*) – Vertex data to plot on surfaces. Must be a valid file path of a GIFTI or CIFTI image, a loaded GIFTI or CIFTI image, a numpy array with length equal to the total number of vertices in the provided surfaces (e.g., 32k in left surface + 32k in right surface = 64k total), or a dictionary with 'left' and/or 'right keys. If a numpy array, vertices are assumed to be in order of left-to-right, if applicable. If a dictionary, then values can be any of the possible types mentioned above, assuming that the vertices match the vertices of their respective surface.
- **cmap** (*matplotlib colormap name or object, optional*) – Colormap to use for data, default: 'viridis'
- **alpha** (*float, optional*) – Colormap opacity (0 to 1). Default: 1
- **color_range** (*tuple[float, float], optional*) – Minimum and maximum value for color map. If None, then the minimum and maximum values in *data* are used. Default: None
- **as_outline** (*bool, optional*) – Plot only an outline of contiguous vertices with the same value. Useful if plotting regions of interests, atlases, or discretized data. Not recommended for continous data. Default: False
- **zero_transparent** (*bool, optional*) – Set vertices with value of 0 to NaN, which will turn them transparent on the surface. Useful when value of 0 has no importance (e.g., thresholded data, an atlas). Default: True
- **cbar** (*bool, optional*) – Show colorbar for layer, default: True
- **cbar_label** (*str, optional*) – Label to include with colorbar if shown. Note that this is not required for the colorbar to be drawn. Default: None

Raises

- **ValueError** – *data* keys must be 'left' and/or 'right'
- **TypeError** – *data* is neither an instance of str or os.PathLike, numpy.ndarray, dict, nibabel.gifti.gifti.GiftiImage, or nibabel.cifti2.cifti2.Cifti2Image

build(*figsize=None*, *colorbar=True*, *cbar_kws=None*, *scale=(2, 2)*)

Build matplotlib figure of surface plot

Parameters

- **figsize** (*tuple, optional*) – Overall figure size, specified by (width, height). Default: None, which will determine the figure size based on the *size* parameter.
- **colorbar** (*bool, optional*) – Draw colorbars for each applicable layer, default: True

- **cbar_kws** (*dict, optional*) – Keyword arguments for `_add_colorbar()`. Default: None, which will plot the default colorbar parameters.
- **scale** (*tuple, optional*) – Amount to scale the surface plot. Default: (2, 2), which is a good baseline for higher resolution plotting.

Returns
matplotlib.pyplot.figure – Surface plot figure

render(*offscreen=True*)

Generate surface plot with all provided layers

Parameters
offscreen (*bool, optional*) – Render offscreen. Default: True

Returns
brainspace.plotting.base.Plotter – Surface plot

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

View Brainspace vtk surface rendering

Notes

This only shows the plot created by brainspace.plotting.surface_plotting.plot_surf, and will not include colorbars created by `plot()` or any other matplotlib components.

Parameters

- **embed_nb** (*bool, optional*) – Whether to embed figure in notebook. Only used if running in a notebook. Default: False
- **interactive** (*bool, optional*) – Whether to enable interaction, default: True
- **scale** (*tuple, optional*) – Amount to scale the surface plot, default: (1, 1)

Returns
Ipython Image or vtk panel – Brainspace surface plot rendering

5.2.3 `surfplot.utils` - Utilities

Utility functions that streamline preparing vertex arrays for plotting

surfplot.utils.add_fslr_medial_wall(data[, ...])	Add medial wall to data in fsLR space
surfplot.utils.threshold(data, thresh[, ...])	Threshold vertex array

surfplot.utils.add_fslr_medial_wall

surfplot.utils.**add_fslr_medial_wall**(*data*, *split=False*)

Add medial wall to data in fsLR space

Data in 32k fs_LR space (e.g., Human Connectome Project data) often exclude the medial wall in their data arrays, which results in a total of 59412 vertices across hemispheres. This function adds back in the missing medial wall vertices to produce a data array with the full 64984 vertices, which is required for plotting with 32k density fsLR surfaces.

Parameters

- **data** (*numpy.ndarray*) – Surface vertices. Must have exactly 59412 or 64984 vertices. Note that if 64984 vertices are present, then the medial wall is already included. If so, then only hemisphere splitting will be performed, if applied.
- **split** (*bool*) – Return left and right hemipsheres as separate arrays. Default: False

Returns
numpy.ndarray – Vertices with medial wall included (64984 vertices total)

Raises
ValueError – *data* has the incorrect number of vertices (59412 or 64984 only accepted)

surfplot.utils.threshold

surfplot.utils.**threshold**(*data*, *thresh*, *binarize=False*, *two_sided=True*)

Threshold vertex array

Parameters

- **data** (*numpy.ndarray*) – Vertex array
- **thresh** (*float*) – Threshold value. All values below or equal to threshold are set 0.
- **binarize** (*bool, optional*) – Set all values above threshold to 1. Default: False
- **two_sided** (*bool, optional*) – Apply thresholding to both positive and negative values. Default: True

Returns
numpy.ndarray – Thresholded data

5.3 Tutorials and Examples

Many of these tutorials and examples rely on neuromaps to automatically download surfaces, rather than relying on specifying local file paths. Should you wish to run these tutorials/examples yourself, be sure to install `neuromaps` via `pip`:

```
pip install neuromaps
```

5.3.1 Tutorials

These tutorials provide a step-by-step walkthrough of `surfplot`, covering each major feature in detail.

Tutorial 1: Quick Start

This tutorial gives a quick overview of `surfplot` before diving into more detail in subsequent tutorials. The aim here is to get a flavour of how `surfplot` works and what can be plotted.

Getting surfaces

First, we need to get some brain surfaces. Here, we'll use the Conte69 Human Connectome Project surface (A.K.A *fsLR* surfaces). We can import and call the `neuromaps.datasets.fetch_fslr()` function, and then select the 'inflated' surface, which will give the file paths of the left and right hemisphere GIFTI files:

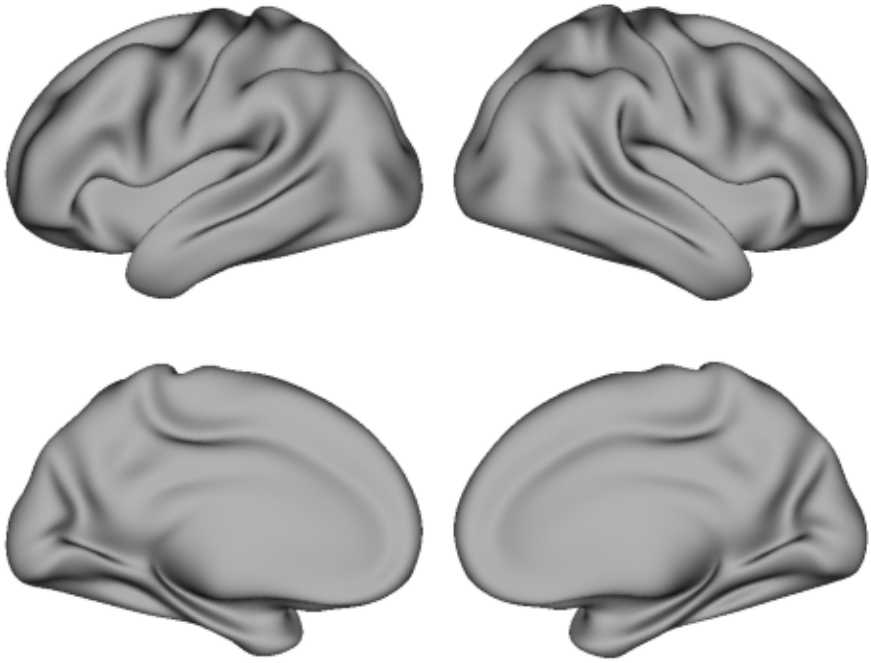
```
from neuromaps.datasets import fetch_fslr

surfaces = fetch_fslr()
lh, rh = surfaces['inflated']
```

Making a plot

Brain plots are created using the `brainspace.plotting.Plot` class. We can pass both of our surfaces to the *surf_lh* and *surf_rh* parameters. These parameters accept file paths/names, or preloaded surfaces from `brainspace.mesh.mesh_io.read_surface()`.

Then, we can call `build()` method to make the figure, which returns a `matplotlib` figure, *fig*.

```
from surfplot import Plot

p = Plot(surf_lh=lh, surf_rh=rh)
fig = p.build()
# show figure, as you typically would with matplotlib
fig.show()
```



Adding layers

Once the plot has been set up by instantiating the `Plot` class, adding data is as simple as adding plotting layers using the `add_layer()` method.

Let's first add some shading. We already have the Freesurfer sulc maps in our *surface* variable, which are accessed here with the 'sulc' key.

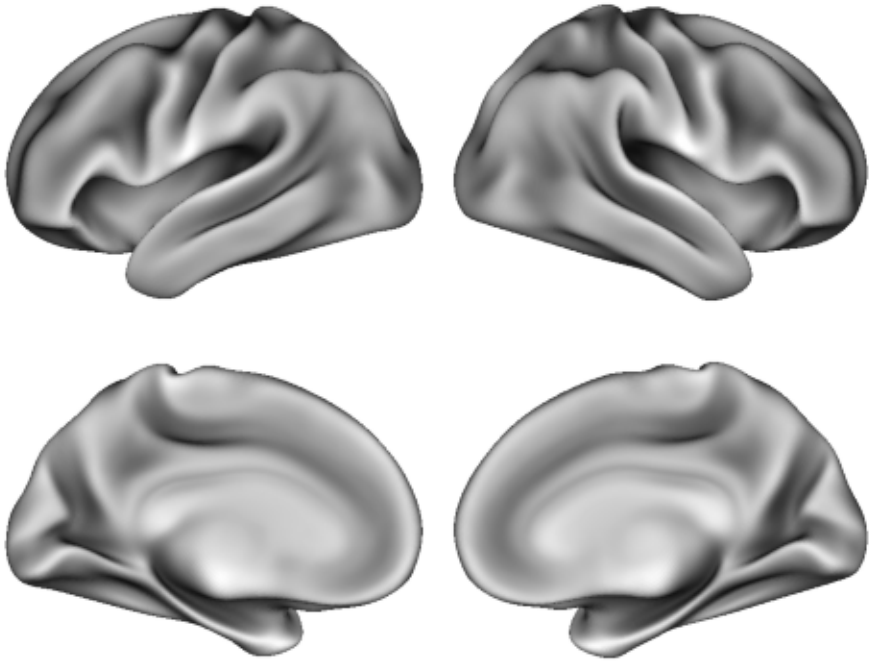
We can pass our sulc maps to the `add_layer()` method with the first positional parameter, *data*, which accepts either a dictionary with 'left' and 'right' keys, or a `numpy` array. *Tutorial 3: Types of Input Data* covers what types of data can be passed to the *data* parameter.

```
sulc_lh, sulc_rh = surfaces['sulc']
p.add_layer({'left': sulc_lh, 'right': sulc_rh}, cmap='binary_r', cbar=False)
```

Above, we've also used a grayscale colormap (*cmap*) and turned off the colorbar (*cbar*) for this particular layer.

Now, let's plot our updated figure:

```
fig = p.build()
fig.show()
```



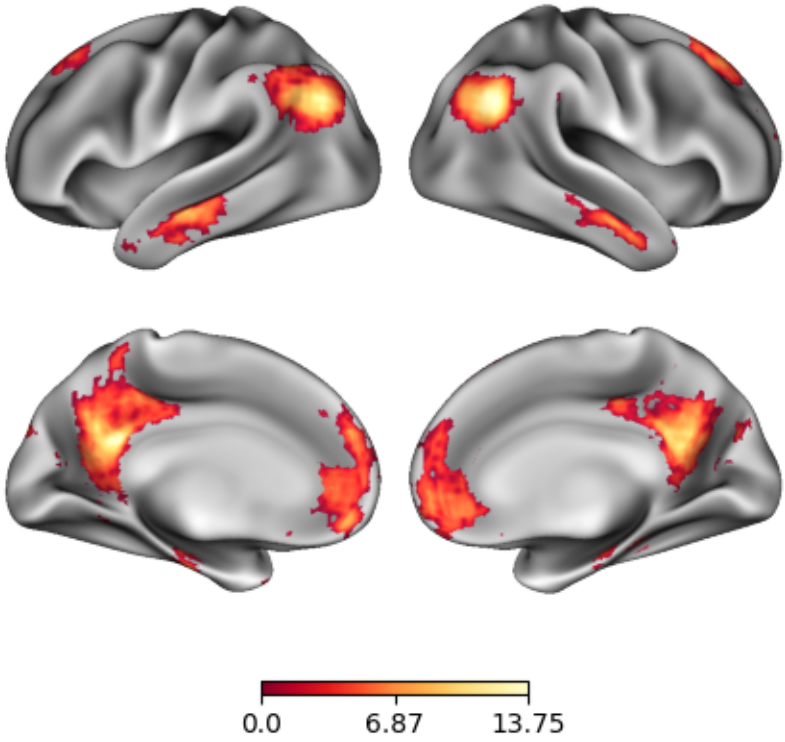
Finally, let's add some statistical data. We can load some example data packaged with `surfplot` using *load_example_data()*. By default, it loads an association map of the term 'default mode' computed from Neurosynth. For convenience, this map has already been projected from a volume in MNI152 coordinates to a fsLR surface using `neuromaps`, and the *lh_data* and *rh_data* variables are just numpy arrays of the vertices:

```
from surfplot.datasets import load_example_data
lh_data, rh_data = load_example_data()
print(lh_data)
```

```
[6.6808 0.     0.     ... 0.     0.     0.    ]
```

We can add each array as a layer using a dictionary like before. By default a colorbar will be added for this layer, and its range is determined by the minimum and maximum values (this can be adjusted with the *color_range* parameter).

```
p.add_layer({'left': lh_data, 'right': rh_data}, cmap='YlOrRd_r')
fig = p.build()
fig.show()
# sphinx_gallery_thumbnail_number = 3
```



Total running time of the script: (0 minutes 0.987 seconds)

Tutorial 2: Types of Surfaces

This tutorial covers what types of surfaces that can be plotted with `surfplot`.

Types of surfaces

Put briefly, `surfplot` can take file paths to any valid surface file(s) with geometry data. Under the hood, *`Plot`* runs `brainspace.mesh.mesh_io.read_surface()` to load files. Typically, these will be Freesurfer or GIFTI files.

`plottingPlot` can also read instances of `brainspace.vtk_interface.wrappers.data_object.BSPolyData`, which are returned by `read_surface()`. So, pre-loaded surfaces with `read_surface()` can be plotted as well.

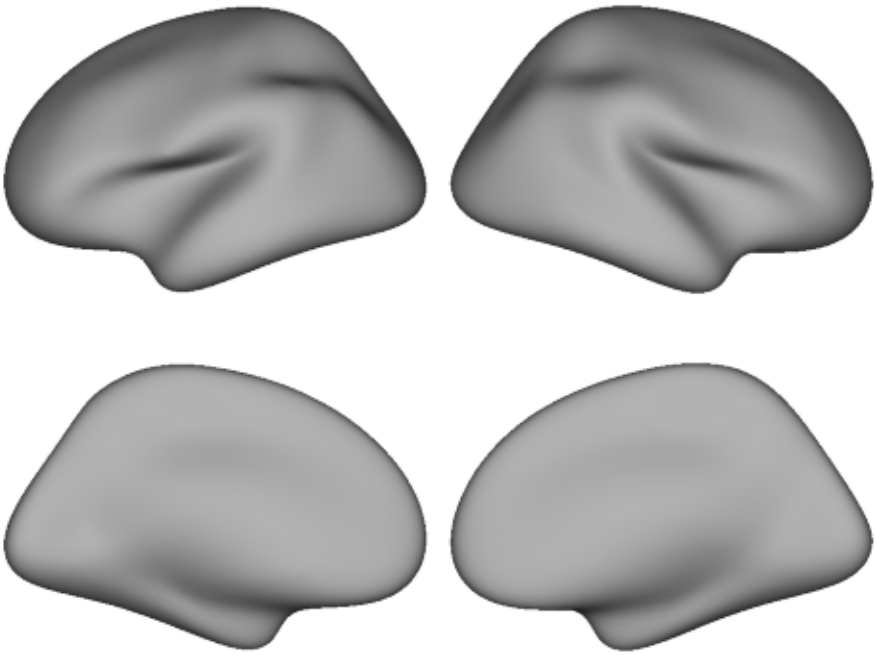
Beyond that, `surfplot` is invariant to the actual brain surfaces you wish to use. Common human surfaces include *fsaverage* surfaces packaged with Freesurfer, and Human Connectome Project *fsLR* surfaces (downloadable here). Several different human surfaces can also all be found on OSF here. Non-human surfaces can also be plotted, such as the NMTv2 Macaque surfaces.

Note that throughout these tutorials, surfaces are automatically fetched using `neuromaps` or `Brainspace` to avoid having to specify local files. It is also possible to fetch fsaverage surfaces using `nilearn`. These are all great options to automically get surfaces in your workflow, and make reproducibility and portability of your code a bit more feasible.

Note: Make sure to *always* use the correct surface for your data. Double check the number of vertices in both your data and the surfaces you are using for plotting.

In *Tutorial 1: Quick Start* we plotted *fsLR* surfaces. For the sake of demonstration, let's plot Freesurfer's *fsaverage* here, again using `neuromaps` fetching functions.

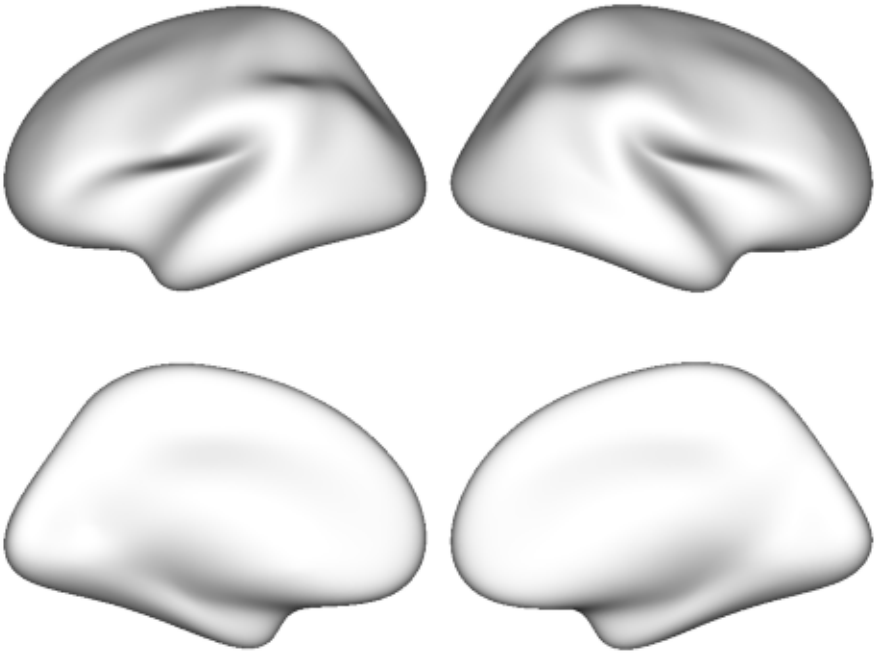
```
from neuromaps.datasets import fetch_fsaverage
from surfplot import Plot

surfaces = fetch_fsaverage(density='164k')
lh, rh = surfaces['inflated']

# make figure
p = Plot(lh, rh)
fig = p.build()
fig.show()
```



Brightness

By default, `Plot` will plot a medium-gray surface, typical of most surface plotting packages like Connectome Workbench. The brightness of the blank surface can be adjusted using the *brightness* parameter, if desired. Values range from 0 (black) to 1 (white). For example:

```
p = Plot(lh, rh, brightness=.8)
fig = p.build()
fig.show()
```



Total running time of the script: (0 minutes 0.584 seconds)

Tutorial 3: Types of Input Data

This tutorial covers what types of data can be passed to the *data* parameter of the `add_layer()` method of `Plot`.

data accepts four different types of data:

1. A numpy array of vertex data
2. A file path of a valid GIFTI or CIFTI file
3. Instances of `nibabel.gifti.gifti.GiftiImage` or `nibabel.cifti2.cifti2.Cifti2Image`
4. A dictionary with 'left' and/or 'right' keys to explicity assign any of the above data types to either hemisphere.

This flexibility makes it easy to plot any surface data by accommodating both GIFTI and CIFTI data. Let's dig into this further.

Getting data

Here we'll reuse the Conte69 fsLR surfaces and Freesurfer sulc maps we used in *Tutorial 1: Quick Start*, both of which are downloaded via `neuromaps`. We'll also reuse the example data.

```
from neuromaps.datasets import fetch_fslr
from surfplot.datasets import load_example_data

surfaces = fetch_fslr()
lh, rh = surfaces['inflated']
```

Arrays

A numpy array can be passed to *data* in the `add_layer()` method. Importantly, the length of this array **must equal the total number of vertices of the hemispheres that are plotted.** With our surfaces, we can check their vertices using `nibabel`:

```
import nibabel as nib
print('left', nib.load(lh).darrays[0].dims)
print('right', nib.load(rh).darrays[0].dims)
```

```
left [32492, 3]
right [32492, 3]
```

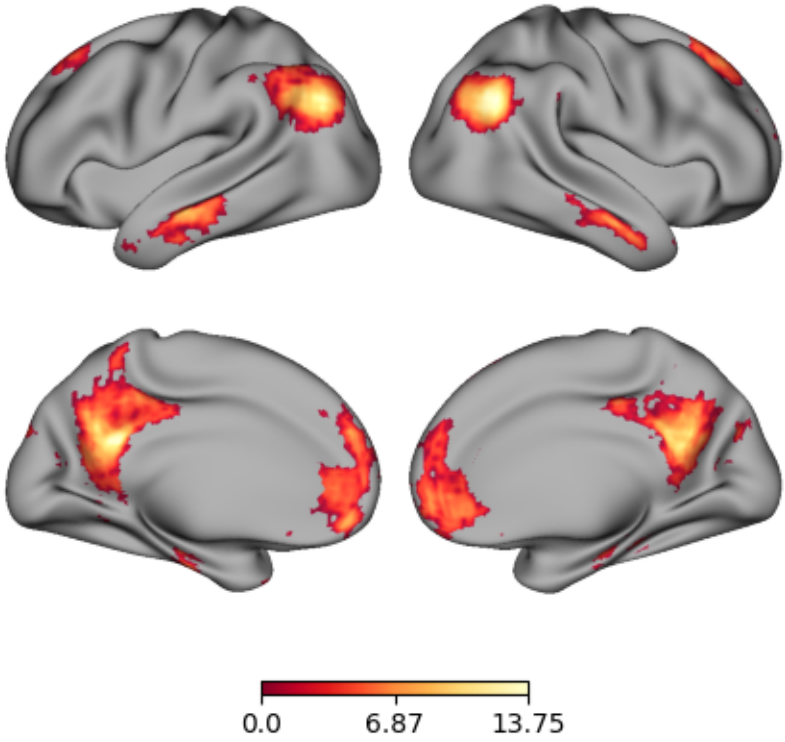
Therefore, our data must have a length of 32492 + 32492 = 64984 if we want to plot both hemispheres. Let's check this first:

```
# return a single concatenated array from both hemispheres
data = load_example_data(join=True)
print(len(data) == 64984)
```

```
True
```

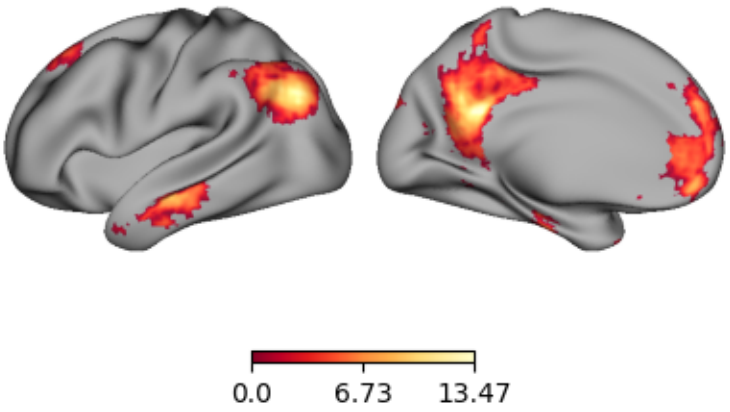
Perfect, now let's plot:

```
from surfplot import Plot

p = Plot(surf_lh=lh, surf_rh=rh)
p.add_layer(data, cmap='YlOrRd_r')
fig = p.build()
fig.show()
```



Note that passing a single array **assumes it goes from the left hemisphere to the right**. If we want to plot just one hemisphere, then we have to update our data accordingly. Be sure to plot the correct data!

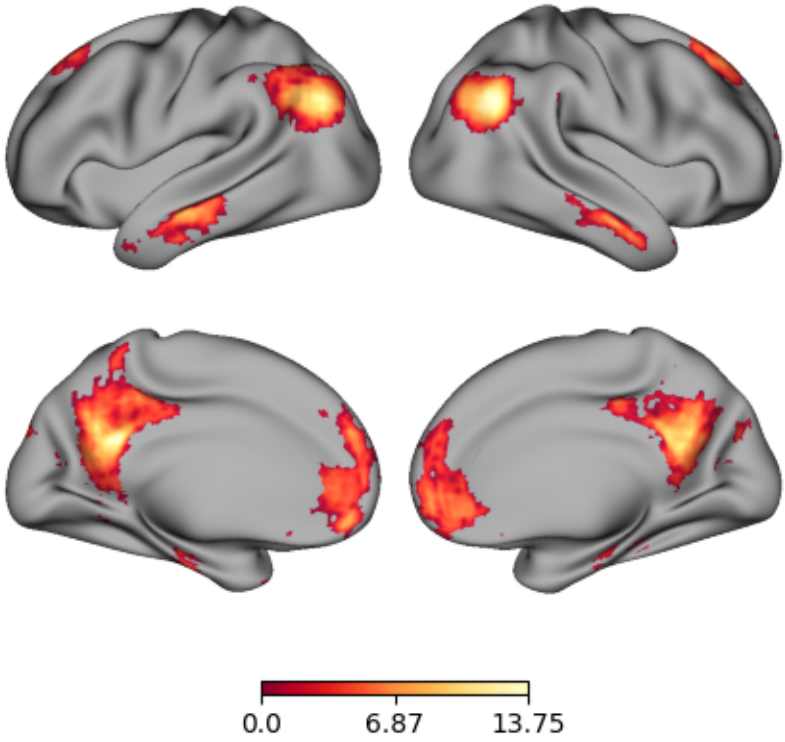
```
p = Plot(surf_lh=lh, zoom=1.2, size=(400, 200))
# left hemisphere is the first 32492 vertices
p.add_layer(data[:32492], cmap='YlOrRd_r')
fig = p.build()
fig.show()
# sphinx_gallery_thumbnail_number = 2
```



Using a dictionary

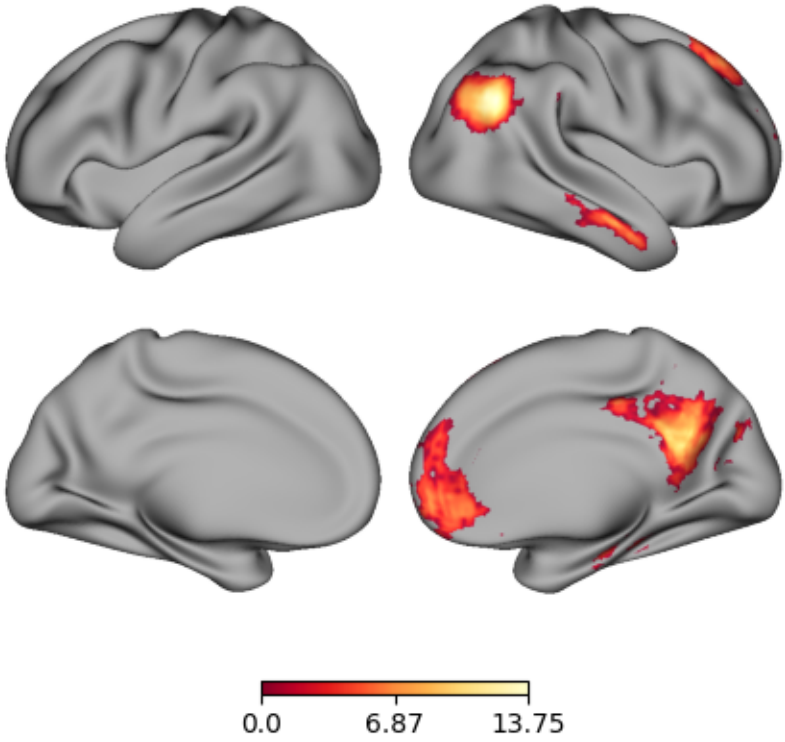
To be explicit about which data is passed to which hemisphere, it is also possible to use a dictionary to assign data to a hemisphere. The dictionary **must** have 'left' and/or 'right' keys only. This is exactly how data was passed to the final figure in *Tutorial 1: Quick Start*. Note that the length of each array must equal the number of vertices in their respective hemispheres.

```
# return as separate arrays for each hemisphere
lh_data, rh_data = load_example_data()

p = Plot(surf_lh=lh, surf_rh=rh)
p.add_layer({'left': lh_data, 'right': rh_data}, cmap='YlOrRd_r')
fig = p.build()
fig.show()
```



Using a dictionary, we can also only plot data for a specific hemisphere, e.g., the right:

```
p = Plot(surf_lh=lh, surf_rh=rh)
p.add_layer({'right': rh_data}, cmap='YlOrRd_r')
fig = p.build()
fig.show()
```

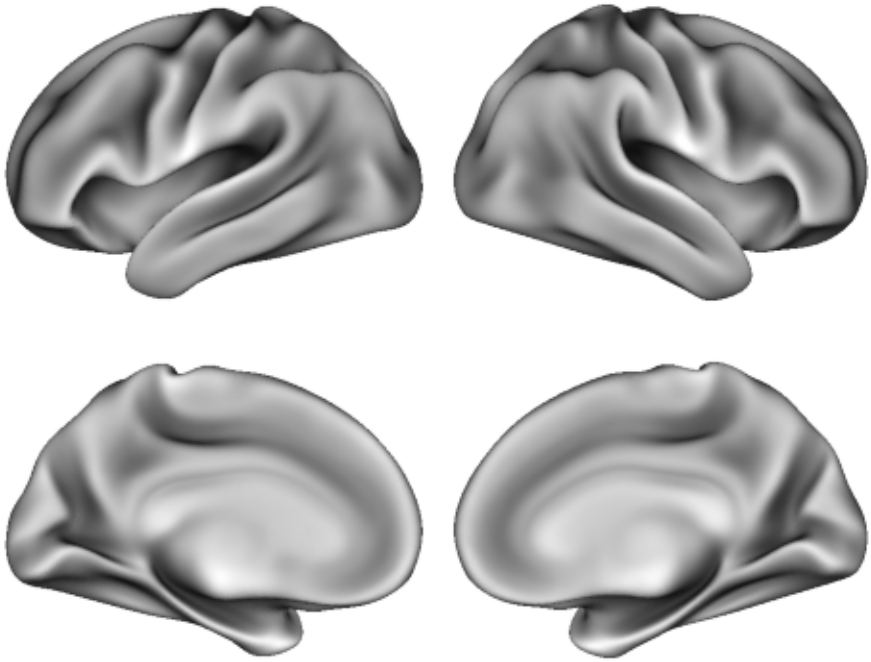


Using dictionaries is necessary when plotting data from left and/or right GIFTI files, which we'll cover in the next section.

File names

It is possible to directly pass in file names, assuming that they're valid and readable with `nibabel`. These files must be either GIFTI or CIFTI images. When plotting both hemispheres, you will need a dictionary to assign each each GIFTI to a hemisphere. To test this out, let's get the downloaded sulc maps and add them:

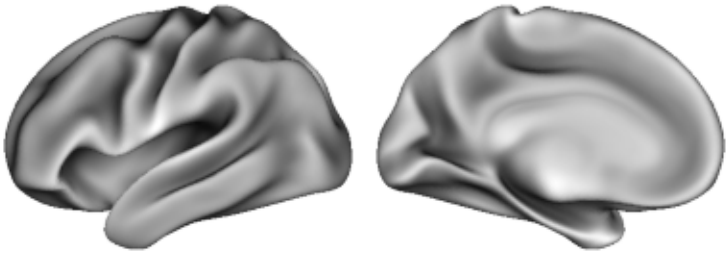
```
lh_sulc, rh_sulc = surfaces['sulc']
p = Plot(surf_lh=lh, surf_rh=rh)
p.add_layer({'left': lh_sulc, 'right': rh_sulc}, cmap='binary_r', cbar=False)
fig = p.build()
fig.show()
```



Loaded files

Finally, if a file was already loaded into Python using `nibabel`, then it can also be plotted. For example, with single hemisphere:

```
img = nib.load(lh_sulc)

p = Plot(surf_lh=lh, zoom=1.2, size=(400, 200))
p.add_layer(img, cmap='binary_r', cbar=False)
fig = p.build()
fig.show()
```



Altogether, this flexibility makes it easy to plot data in a variety of different workflows and usecases. As always, be sure to check that the data is passed to the correct hemisphere, and that the number of vertices in the data match the number of vertices of the surface(s)!

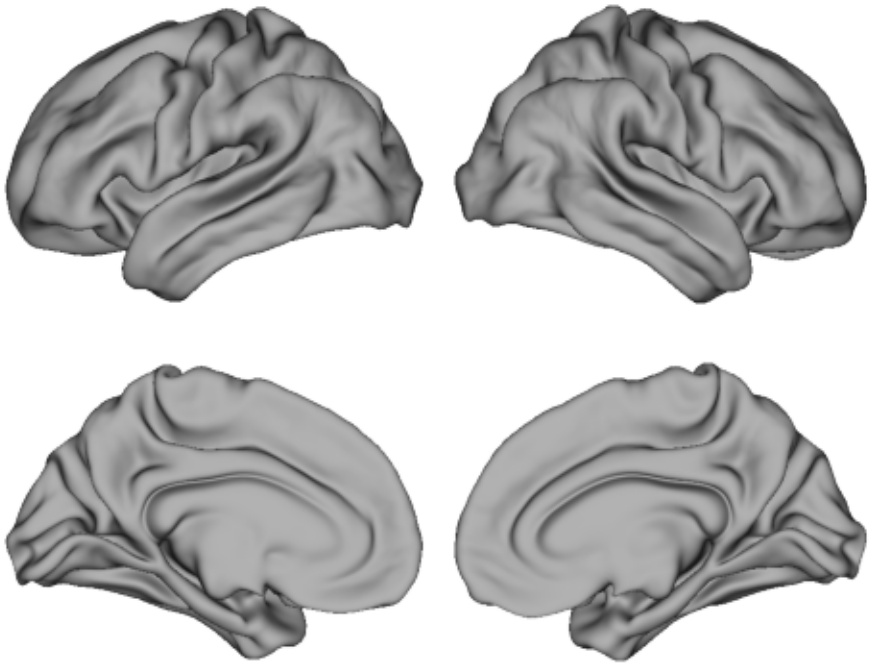
Total running time of the script: (0 minutes 0.941 seconds)

Tutorial 4: Layouts and Views

This tutorial covers the layout and view options provided by `surfplot`.

For variety, let's import the left and right Conte69 midthickness surface directly using `load_conte69()`. Then, we'll make a blank surface plot using the default layout and view, which is a 2x2 'grid' of lateral and medial views that is identical to the default setup in Connectome Workbench.

```
from brainspace.datasets import load_conte69
from surfplot import Plot

lh, rh = load_conte69()
p = Plot(lh, rh)
fig = p.build()
fig.show()
```

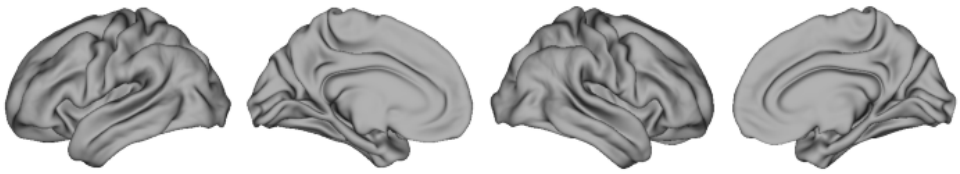


Layout

The layout can be adjusted with the *layout* parameter, and has 3 options: 'grid' (default, shown above), 'row', or 'column'. The *size* and *zoom* parameters will have to be adjusted based on the layout and number of views.

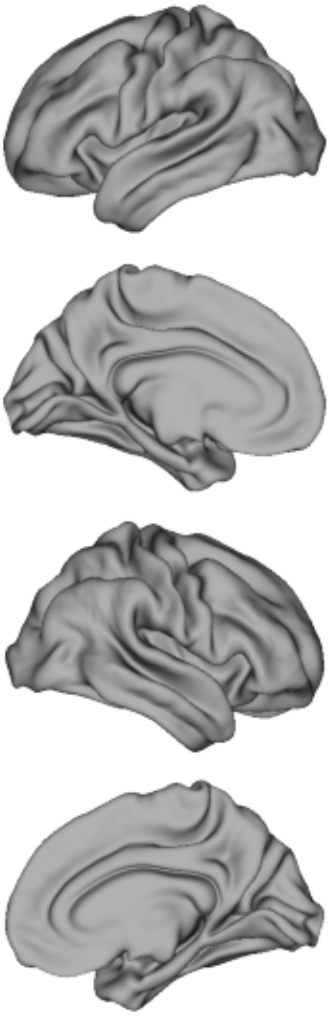
Above we see that 'grid' gives us a views-by-hemisphere grid, where the left hemisphere is the left column and the right hemisphere is the right column. Meanwhile, the 'row' layout gives a single horizontal row of brains:

```
p = Plot(lh, rh, size=(800, 200), zoom=1.2, layout='row')
fig = p.build()
fig.show()
```



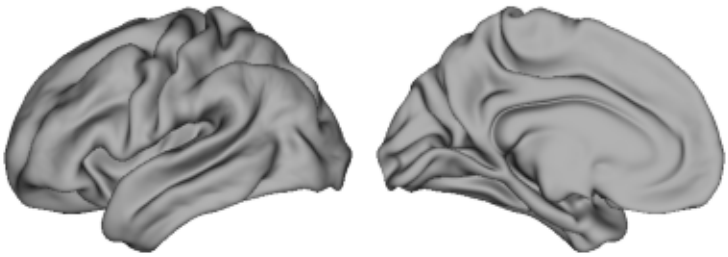
The 'column' layout gives a single vertical column of brains.

```
p = Plot(lh, rh, size=(200, 600), zoom=1.6, layout='column')
fig = p.build()
fig.show()
# sphinx_gallery_thumbnail_number = 3
```



As well, it's also possible to plot just one hemisphere. If the layout is set as default ('grid'), then a single hemisphere is plotted as row:

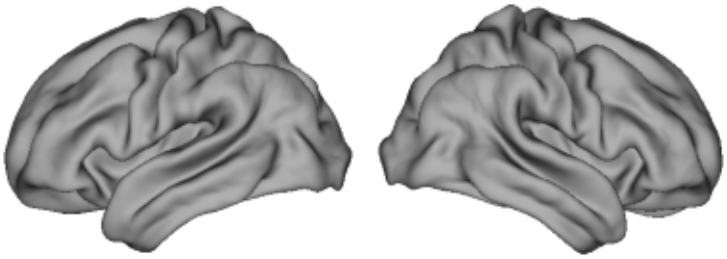
```
p = Plot(lh, size=(400, 200), zoom=1.2)
fig = p.build()
fig.show()
```



Views

`surfplot` makes it easy to configure the view(s) you wish to use. One or more views can be specified through the *views* parameter. As we've seen before, the default is to include lateral and medial views. It is also possible to show just one view:

```
p = Plot(lh, rh, size=(400, 200), zoom=1.2, views='lateral')
fig = p.build()
fig.show()
```



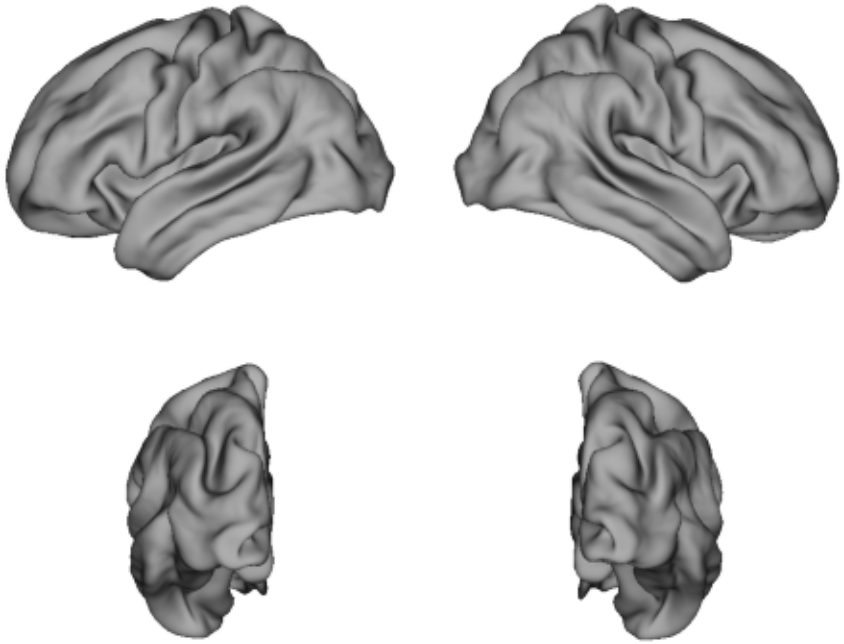
It is also possible to show more than just lateral and medial views, such as 'posterior'. Note that views are plotted in order in which they appear in the list:

```
p = Plot(lh, rh, size=(500, 400), zoom=1.4,  views=['lateral', 'posterior'])
fig = p.build()
```

(continues on next page)

(continued from previous page)

```
fig.show()
```



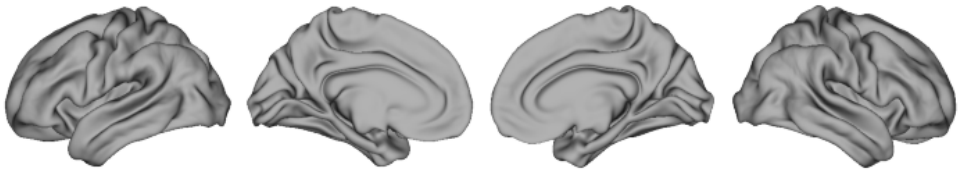
All possible views are shown here (the right hemisphere for brevity):

```
all_views = ['lateral', 'medial', 'dorsal', 'ventral', 'anterior', 'posterior']
p = Plot(surf_rh=rh, size=(900, 200), zoom=.8, layout='row', views=all_views)
fig = p.build()
fig.show()
```

Views can also be mirrored when both hemipsheres are plotted and *layout* is either 'row' or 'column'. Specifically,

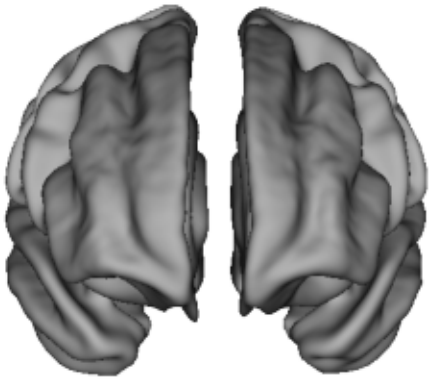
the right hemisphere view order is reversed. For example, plotting default lateral and medial views and setting *mirror_views=True* will situate the medial views in the middle for a symmetrical figure:

```
p = Plot(lh, rh, size=(800, 200), zoom=1.2, layout='row', mirror_views=True)
fig = p.build()
fig.show()
```



Finally, it is possible to flip the left and right hemisphere. This is useful when plotting just the 'anterior' or 'ventral' for both hemispheres. For example:

```
p = Plot(lh, rh, size=(200, 200), zoom=3, views='anterior', flip=True)
fig = p.build()
fig.show()
```



Total running time of the script: (0 minutes 0.961 seconds)

Tutorial 5: Colors and colorbars

This tutorial demonstrates how to configure the colorbar(s) with `surfplot`.

Layer color maps and colorbars

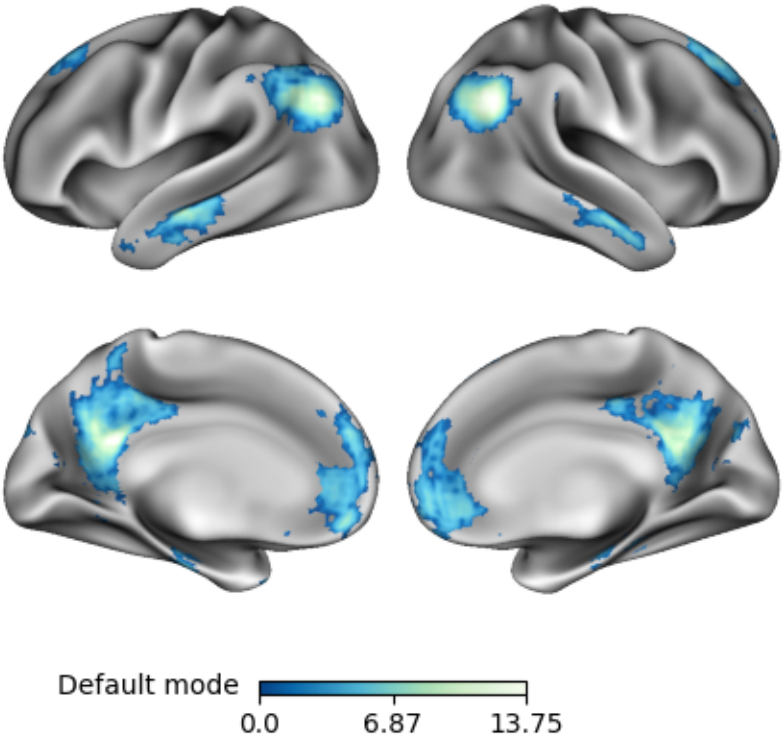
The color map can be specified for each added plotting layer using the *cmap* parameter of *add_layer()*, along with the associated `matplotlib` colorbar drawn if specified. The colobar can be turned off by *cbar=False*. The range of the colormap is specified with the *color_range* parameter, which takes a tuple of (*minimum*, *maximum*) values. If no color range is specified (the default, i.e. *None*), then the color range is computed automically based on the minimum and maximum of the data.

Let's get started by setting up a plot with surface shading added as well. Following the first initial steps of *Tutorial 1: Quick Start* :

```
from neuromaps.datasets import fetch_fslr
from surfplot import Plot

surfaces = fetch_fslr()
lh, rh = surfaces['inflated']
p = Plot(lh, rh)

sulc_lh, sulc_rh = surfaces['sulc']
p.add_layer({'left': sulc_lh, 'right': sulc_rh}, cmap='binary_r', cbar=False)
```

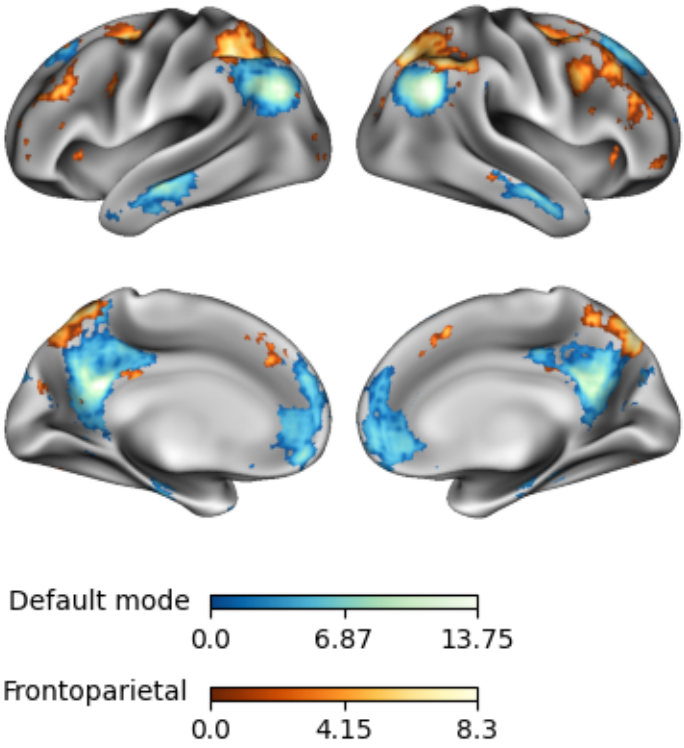
Now let's add a plotting layer with a colorbar using the example data. The *cmap* parameter accepts any named matplotlib colormap, or a colormap object. This means that `surfplot` can work with pretty much any colormap, including those from seaborn and cmasher, for example.

```
from surfplot.datasets import load_example_data

# default mode network associations
default = load_example_data(join=True)
p.add_layer(default, cmap='GnBu_r', cbar_label='Default mode')
fig = p.build()
fig.show()
```



cbar_label added a text label to the colorbar. Although not necessary in cases where a single layer/colorbar is shown, it can be useful when adding multiple layers. To demonstrate that, let's add another layer using the *frontoparietal* network associations from *load_example_data()*:

```
fronto = load_example_data('frontoparietal', join=True)
p.add_layer(fronto, cmap='YlOrBr_r', cbar_label='Frontoparietal')
fig = p.build()
fig.show()
```



The order of the colorbars is always based on the order of the layers, where the outermost colorbar is the last (i.e. uppermost) plotting layer. Of course, more layers and colorbars can lead to busy-looking figure, so be sure not to overdo it.

cbar_kws

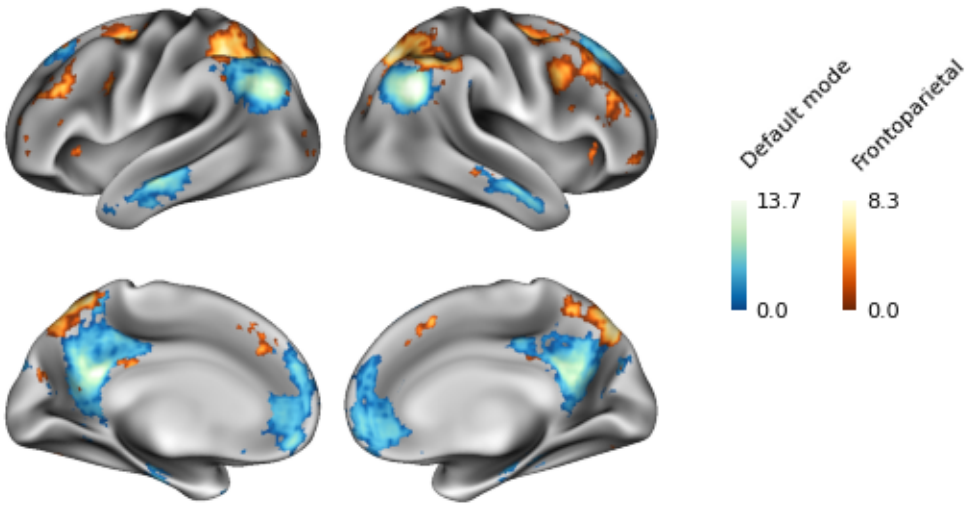
Once all layers have been added, the positioning and style can be adjusted using the *cbar_kws* parameter in `build()`, which are keyword arguments for `surfplot.plotting.Plot._add_colorbars()`. Each one is briefly described below (see `_add_colorbars()` for more detail):

1. *location*: The location, relative to the surface plot
2. *label_direction*: Angle to draw label for colorbars
3. *n_ticks*: Number of ticks to include on colorbar
4. *decimals*: Number of decimals to show for colorbal tick values
5. *fontsize*: Font size for colorbar labels and tick labels
6. *draw_border*: Draw ticks and black border around colorbar
7. *outer_labels_only*: Show tick labels for only the outermost colorbar
8. *aspect*: Ratio of long to short dimensions
9. *pad*: Space that separates each colorbar

10. *shrink*: Fraction by which to multiply the size of the colorbar
11. *fraction*: Fraction of original axes to use for colorbar

Let's plot colorbars on the right, which will generate vertical colorbars instead of horizontal colorbars. We'll also add some style changes for a cleaner look:

```
kws = {'location': 'right', 'label_direction': 45, 'decimals': 1,
       'fontsize': 8, 'n_ticks': 2, 'shrink': .15, 'aspect': 8,
       'draw_border': False}
fig = p.build(cbar_kws=kws)
fig.show()
# sphinx_gallery_thumbnail_number = 3
```

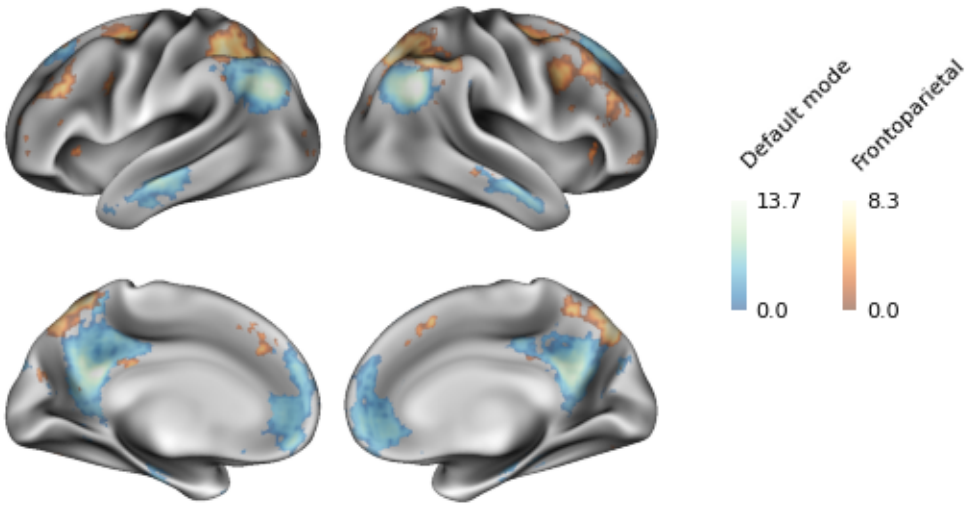


Be sure to check out *Example 1: Multiple Stat Maps* for another example of colorbar styling.

Transparency

The transparency of the plotting layers can be adjusted by the `alpha` parameter. This may be preferred in cases with overlapping plotting layers. We can recreate the example above with transparent maps like so:

```
p = Plot(lh, rh)
p.add_layer({'left': sulc_lh, 'right': sulc_rh}, cmap='binary_r', cbar=False)
p.add_layer(default, cmap='GnBu_r', cbar_label='Default mode', alpha=.5)
p.add_layer(fronto, cmap='YlOrBr_r', cbar_label='Frontoparietal', alpha=.5)
fig = p.build(cbar_kws=kws)
fig.show()
```



Although these particular maps are largely non-overlapping, you can see some small overlap at the edges of the default mode and frontoparietal clusters thanks to the transparency.

Total running time of the script: (0 minutes 0.783 seconds)

Tutorial 6: Regions and Parcellations

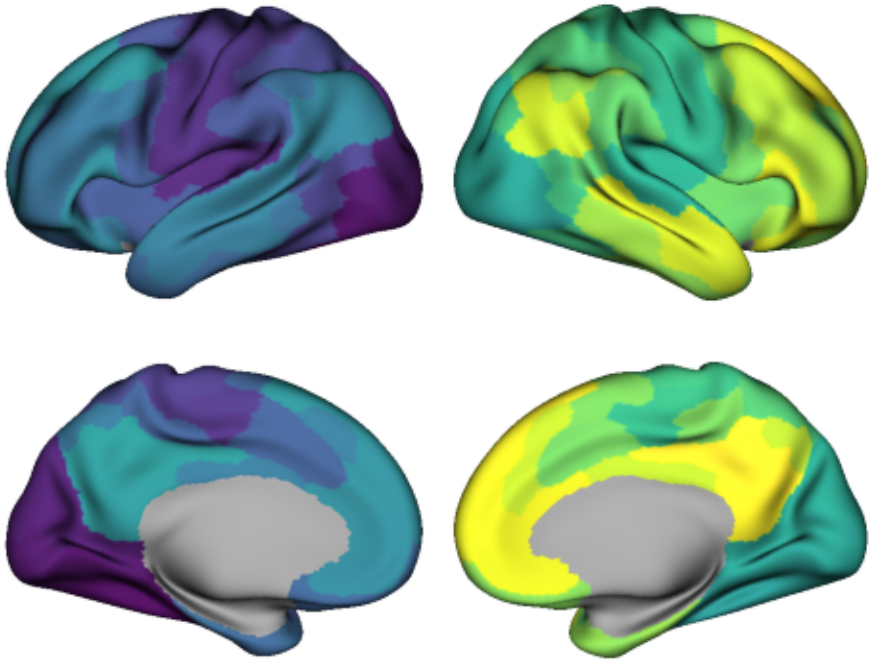
This tutorial demonstrates how to plot brain regions.

Regions and parcellations can be plotted with `brainplot` as one or more layers, and it's possible to add region outlines by simply adding a layer with the *as_outline* parameter.

Parcellations

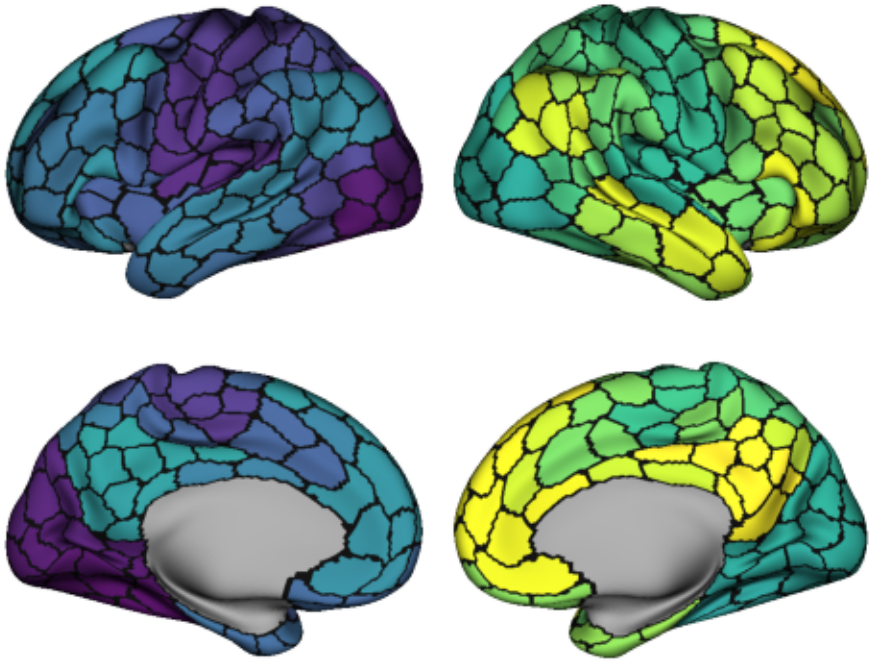
Multiple brain regions can be plotted as a single layer as long as the vertices in different regions have different numerical labels/values, which is typical for any parcelation. To demonstrate, we can use the `load_parcellation()` from `Brainspace` to load the Schaefer 400 parcellation.

```
from neuromaps.datasets import fetch_fslr
from surfplot import Plot
from brainspace.datasets import load_parcellation

surfaces = fetch_fslr()
lh, rh = surfaces['inflated']
p = Plot(lh, rh)

# add schaefer parcellation (no color bar needed)
lh_parc, rh_parc = load_parcellation('schaefer')
p.add_layer({'left': lh_parc, 'right': rh_parc}, cbar=False)

fig = p.build()
fig.show()
```



Now can add a second layer of just the region outlines. This is done by setting *as_outline=True*. The color of the outlines are set by the *cmap* parameter, as with any data. To show black outlines, we can just use the *gray* colormap.

```
p.add_layer({'left': lh_parc, 'right': rh_parc}, cmap='gray',
            as_outline=True, cbar=False)
fig = p.build()
fig.show()
```



Regions of Interest

Often times we want to show a selection of regions, instead of all regions. These could be regions from a parcellation, regions defined from a functional localizer, etc.

Let's select two regions from the Schaefer parcellation and zero-out the remaining regions. We'll just stick with the left hemisphere here.

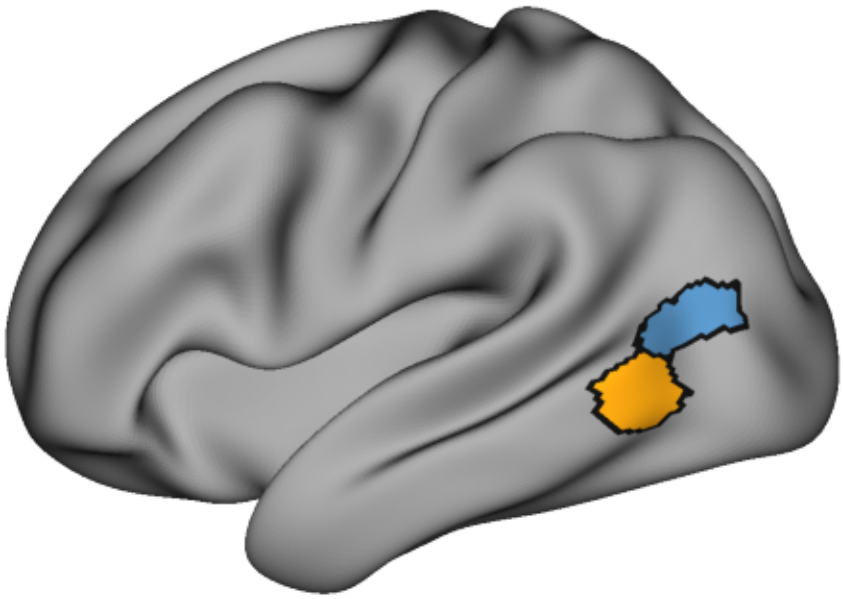
```
import numpy as np
region_numbers = [71, 72]
# zero-out all regions except 71 and 72
regions = np.where(np.isin(lh_parc, region_numbers), lh_parc, 0)
```

Although we can use a pre-defined color map, we might want to define a custom colormap where we can define the exact color for each region. This is possible using `matplotlib`:

```
from matplotlib.colors import ListedColormap

colors = ['orange', 'steelblue']
cmap = ListedColormap(colors, 'regions', N=2)
```

Now we can plot both regions with their outlines: only need to show the left lateral view

```
p = Plot(lh, views='lateral')
p.add_layer(regions, cmap=cmap, cbar=False)
p.add_layer(regions, cmap='gray', as_outline=True, cbar=False)

fig = p.build()
fig.show()
# sphinx_gallery_thumbnail_number = 3
```



Note: Multiple regions can also be plotted as individual layers, rather than combined as a single layer, as shown here. In this case, the vertex array(s) for each layer would be binary.

Total running time of the script: (0 minutes 0.506 seconds)

5.3.2 Examples

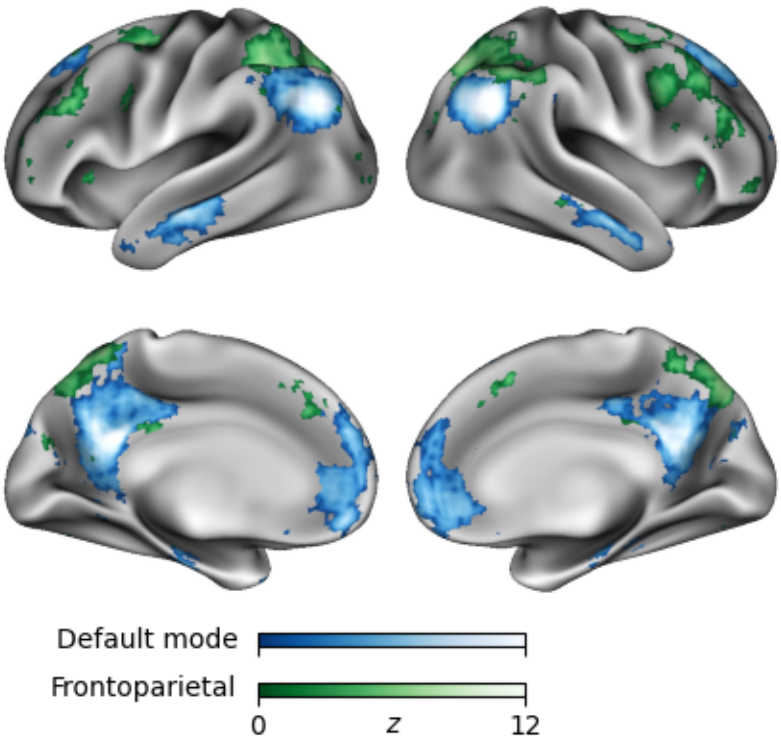
These examples demonstrate 'publication-ready' figures created with `surfplot` based on common use-cases. These are intended to show more typical 'real-world' `surfplot` workflows.

Examples

These examples demonstrate 'publication-ready' figures created with `surfplot` based on common use-cases. These are intended to show more typical 'real-world' `surfplot` workflows.

Example 1: Multiple Stat Maps

This example shows multiple statistical maps on a surface with some extra stylizing for a clean-looking figure.




```
# Code source: Dan Gale
# License: BSD 3 clause

from surfplot import Plot
from surfplot.datasets import load_example_data
from neuromaps.datasets import fetch_fslr

```

(continues on next page)

(continued from previous page)

```
surfaces = fetch_fslr()
lh, rh = surfaces['inflated']

p = Plot(lh, rh)

# shading
lh_sulc, rh_sulc = surfaces['sulc']
p.add_layer({'left': lh_sulc, 'right': rh_sulc}, cmap='binary_r', cbar=False)

color_range = (0, 12)

# add default mode association stats
default = load_example_data(join=True)
p.add_layer(default, cmap='Blues_r', color_range=color_range,
            cbar_label='Default mode')

# add frontoparietal assocation stats
fronto = load_example_data('frontoparietal', join=True)
p.add_layer(fronto, cmap='Greens_r', color_range=color_range,
            cbar_label='Frontoparietal')

# create a clean looking set of colorbars. Only show labels for outer colorbar,
# given that both colorbars have the same range.
cbar_kws = dict(outer_labels_only=True, pad=.02, n_ticks=2, decimals=0)
fig = p.build(cbar_kws=cbar_kws)
# add units to colorbar
fig.axes[1].set_xlabel('z', labelpad=-11, fontstyle='italic')
fig.show()
```

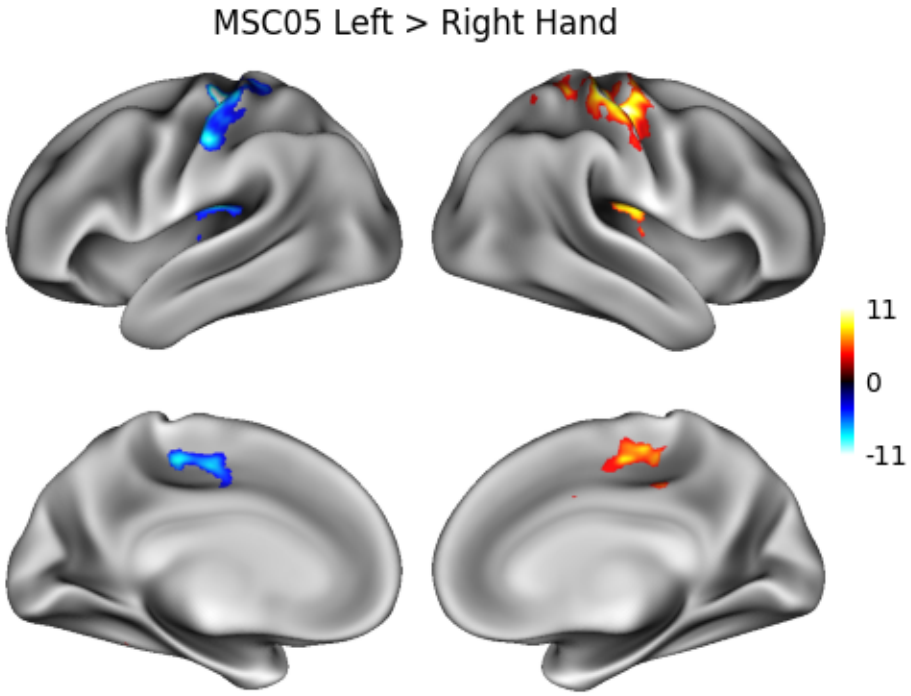
Total running time of the script: (0 minutes 0.225 seconds)

Example 2: Plotting volumetric data

This example shows how to project data from a NIFTI volume onto a surface, and then display the result.

Data is a Left vs Right hand localizer contrast (t-map) for a single subject of the Midnight Scan Club dataset[1]. Data is obtained from Neurovault via `nilearn` and then projected from MNI152 coordinates to fsLR surfaces using `neuromaps`.

[1] Gordon EM, et al. 2017. Precision Functional Mapping of Individual Human Brains. Neuron. 95:791–807.e7.




```
# Code source: Dan Gale
# License: BSD 3 clause

from nilearn.datasets import fetch_neurovault_ids
from nilearn.plotting.cm import _cmap_d as nilearn_cmaps
from neuromaps.transforms import mni152_to_fslr
from neuromaps.datasets import fetch_fslr
from surfplot import Plot
from surfplot.utils import threshold

data = fetch_neurovault_ids(image_ids=[47307], verbose=0)
img = data['images'][0]

# project from MNI to fslr; GIFTI surfaces are returned
gii_lh, gii_rh = mni152_to_fslr(img)

# threshold after projection to avoid interpolation artefacts
data_lh = threshold(gii_lh.agg_data(), 3)
data_rh = threshold(gii_rh.agg_data(), 3)

# get surfaces + sulc maps
surfaces = fetch_fslr()
```

(continues on next page)

(continued from previous page)

```
lh, rh = surfaces['inflated']
sulc_lh, sulc_rh = surfaces['sulc']

p = Plot(lh, rh)
p.add_layer({'left': sulc_lh, 'right': sulc_rh}, cmap='binary_r', cbar=False)

# cold_hot is a common diverging colormap for neuroimaging
cmap = nilearn_cmaps['cold_hot']
p.add_layer({'left': data_lh, 'right': data_rh}, cmap=cmap,
            color_range=(-11, 11))

# make a nice vertical colorbar on the right side of the figure
kws = dict(location='right', draw_border=False, aspect=10, shrink=.2,
           decimals=0, pad=0)
fig = p.build(cbar_kws=kws)
fig.axes[0].set_title('MSC05 Left > Right Hand', pad=-3)
fig.show()
```

Total running time of the script: (0 minutes 0.318 seconds)

INDEX

Symbols

A

B

L

M

P

R

S

T