ipyvolume Documentation

Release 0.4.0

Maarten A. Breddels

Oct 27, 2017

Contents

IPyvolume is a Python library to visualize 3d volumes and glyphs (e.g. 3d scatter plots), in the Jupyter notebook, with minimal configuration and effort. It is currently pre-1.0, so use at own risk. IPyvolume's *volshow* is to 3d arrays what matplotlib's imshow is to 2d arrays.

Other (more mature but possibly more difficult to use) related packages are yt, VTK and/or Mayavi.

Feedback and contributions are welcome: Github, Email or Twitter.

CHAPTER 1

Quick intro

Volume

For quick resuls, use *ipyvolume.widgets.quickvolshow*. From a numpy array, we create two boxes, using slicing, and visualize it.

```
import numpy as np
import ipyvolume as ipv
V = np.zeros((128,128,128)) # our 3d array
# outer box
V[30:-30,30:-30,30:-30] = 0.75
V[35:-35,35:-35,35:-35] = 0.0
# inner box
V[50:-50,50:-50,50:-50] = 0.25
V[55:-55,55:-55,55:-55] = 0.0
ipv.quickvolshow(V, level=[0.25, 0.75], opacity=0.03, level_width=0.1, data_min=0,␣
↪data_max=1)
```

[widget]

Scatter plot

Simple scatter plots are also supported.

```
import ipyvolume as ipv
import numpy as np
x, y, z = np.random.random((3, 10000))
ipv.quickscatter(x, y, z, size=1, marker="sphere")
```

[widget]

Quiver plot

Quiver plots are also supported, showing a vector at each point.

```
import ipyvolume as ipv
import numpy as np
x, y, z, u, v, w = np.random.random((6, 1000))*2-1
quiver = ipv.quickquiver(x, y, z, u, v, w, size=5)
```

[widget]

Mesh plot

And surface/mesh plots, showing surfaces or wireframes.

```
import ipyvolume as ipv
x, y, z, u, v = ipv.examples.klein_bottle(draw=False)
ipv.figure()
m = ipv.plot_mesh(x, y, z, wireframe=False)
ipv.squarelim()
ipv.show()
```

[widget]

Built on Ipywidgets

For anything more sophisticed, use *ipyvolume.pylab*, ipyvolume's copy of matplotlib's 3d plotting (+ volume rendering).

Since ipyvolume is built on ipywidgets, we can link widget's properties.

```
import ipyvolume as ipv
import numpy as np
x, y, z, u, v, w = np.random.random((6, 1000))*2-1
selected = np.random.randint(0, 1000, 100)
ipv.figure()
quiver = ipv.quiver(x, y, z, u, v, w, size=5, size_selected=8, selected=selected)

from ipywidgets import FloatSlider, ColorPicker, VBox, jslink
size = FloatSlider(min=0, max=30, step=0.1)
size_selected = FloatSlider(min=0, max=30, step=0.1)
color = ColorPicker()
color_selected = ColorPicker()
jslink((quiver, 'size'), (size, 'value'))
jslink((quiver, 'size_selected'), (size_selected, 'value'))
jslink((quiver, 'color'), (color, 'value'))
jslink((quiver, 'color_selected'), (color_selected, 'value'))
VBox([ipv.gcc(), size, size_selected, color, color_selected])
```

[widget] Try changing the slider to the change the size of the vectors, or the colors.

CHAPTER 2

Quick installation

This will most likely work, otherwise read *Installation*

```
pip install ipyvolume
jupyter nbextension enable --py --sys-prefix ipyvolume
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

For conda/anaconda, use:

```
conda install -c conda-forge ipyvolume
pip install ipywidgets~=6.0.0b5 --user
```

CHAPTER 3

About

Ipyvolume is an offspring project from vaex. Ipyvolume makes use of threejs, and excellent Javascript library for OpenGL/WebGL rendering.

CHAPTER 4

Contents

Installation

Pip as root

```
pip install ipyvolume
jupyter nbextension enable --py --sys-prefix ipyvolume
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

Pip as user

```
pip install ipyvolume --user
jupyter nbextension enable --py --user ipyvolume
jupyter nbextension enable --py --user widgetsnbextension
```

Conda/Anaconda

```
conda install -c conda-forge ipyvolume
```

Examples

Mixing ipyvolume with Bokeh

This example shows how the selection from a ipyvolume quiver plot can be controlled with a bokeh scatter plot and it's selection tools.

Ipyvolume quiver plot

The 3d quiver plot is done using ipyvolume

```
In [1]: import ipyvolume
        import ipyvolume as ipv
        import vaex
```

We load some data from vaex, but only use the first 10 000 samples for performance reasons of Bokeh.

```
In [2]: ds = vaex.example()
        N = 10000
```

We make a quiver plot using ipyvolume's matplotlib's style api.

```
In [3]: ipv.figure()
        quiver = ipv.quiver(ds.data.x[:N],  ds.data.y[:N],  ds.data.z[:N],
                            ds.data.vx[:N], ds.data.vy[:N], ds.data.vz[:N],
                            size=1, size_selected=5, color_selected="grey")
        ipv.xyzlim(-30, 30)
        ipv.show()
```

```
A Jupyter Widget
```

Bokeh scatter part

The 2d scatter plot is done using Bokeh.

```
In [4]: from bokeh.io import output_notebook, show
        from bokeh.plotting import figure
        import ipyvolume.bokeh
        output_notebook()
```

```
In [5]: tools = "wheel_zoom,box_zoom,box_select,lasso_select,help,reset,"
        p = figure(title="E Lz space", tools=tools, webgl=True, width=500, height=500)
        r = p.circle(ds.data.Lz[:N], ds.data.E[:N],color="navy", alpha=0.2)
        # A 'trick' from ipyvolume to link the selection (one way traffic atm)
        ipyvolume.bokeh.link_data_source_selection_to_widget(r.data_source, quiver, 'selected')
        show(p)
```

Now try doing a selection and see how the above 3d quiver plot reflects this selection.

```
In [ ]: # this code is currently broken
        # import ipywidgets
        #out = ipywidgets.Output()
        #with out:
        #    show(p)
        #ipywidgets.HBox([out, ipv.gcc()])
```

Embedding in html

A bit of a hack, but it is possible to embed the widget and the bokeh part into a single html file (use at own risk).

```
In [8]: from bokeh.resources import CDN
        from bokeh.embed import components

        script, div = components((p))
        template_options = dict(extra_script_head=script + CDN.render_js() + CDN.render_css(),
                                body_pre="<h2>Do selections in 2d (bokeh)<h2>" + div + "<h2>And see t
        ipyvolume.embed.embed_html("tmp/bokeh.html",
```

```
                                    [ipv.gcc(), ipyvolume.bokeh.wmh], all_states=True,
                                    template_options=template_options)
```

```
In [7]: !open tmp/bokeh.html
```

Mixing ipyvolume with bqplot

This example shows how the selection from a ipyvolume quiver plot can be controlled with a bqplot scatter plot and it's selection tools. We first get a small dataset from vaex

```
In [1]: import numpy as np
        import vaex
```

```
In [2]: ds = vaex.example()
        N = 2000 # for performance reasons we only do a subset
        x, y, z, vx, vy, vz, Lz, E = [ds.columns[k][:N] for k in "x y z vx vy vz Lz E".split()]
```

bqplot scatter plot

And create a scatter plot with bqplot

```
In [3]: import bqplot.pyplot as plt
```

```
In [4]: plt.figure(1, title="E Lz space")
        scatter = plt.scatter(Lz, E,
                              selected_style={'opacity': 0.2, 'size':1, 'stroke': 'red'},
                              unselected_style={'opacity': 0.2, 'size':1, 'stroke': 'blue'},
                              default_size=1,
                             )
        plt.brush_selector()
        plt.show()
```

```
A Jupyter Widget
```

Ipyvolume quiver plot

And use ipyvolume to create a quiver plot

```
In [5]: import ipyvolume.pylab as p3
```

```
In [6]: p3.clear()
        quiver = p3.quiver(x, y, z, vx, vy, vz, size=2, size_selected=5, color_selected="blue")
        p3.show()
```

```
A Jupyter Widget
```

Linking ipyvolume and bqplot

Using jslink, we link the `selected` properties of both widgets, and we display them next to eachother using a VBox.

```
In [7]: from ipywidgets import jslink, VBox
```

```
In [8]: jslink((scatter, 'selected'), (quiver, 'selected'))
```

```
In [9]: hbox = VBox([p3.current.container, plt.figure(1)])
        hbox
```

```
A Jupyter Widget
```

Embedding

We embed the two widgets in an html file, creating a standlone plot.

```
In [10]: import ipyvolume.embed
         # if we don't do this, the bqplot will be really tiny in the standalone html
         bqplot_layout = hbox.children[1].layout
         bqplot_layout.min_width = "400px"
```

```
In [14]: ipyvolume.embed.embed_html("bqplot.html", hbox, offline=True, devmode=True)
```

```
In [ ]: !open bqplot.html
```

MCMC & why 3d matters

This example (although quite artificial) shows that viewing a posterior (ok, I have flat priors) in 3d can be quite useful. While the 2d projection may look quite 'bad', the 3d volume rendering shows that much of the volume is empty, and the posterior is much better defined than it seems in 2d.

```
In [3]: import pylab
        import scipy.optimize as op
        import emcee
        import numpy as np
        %matplotlib inline
```

```
In [4]: # our 'blackbox' 3 parameter model which is highly degenerate
        def f_model(x, a, b, c):
            return x * np.sqrt(a**2 +b**2 + c**2) + a*x**2 + b*x**3
```

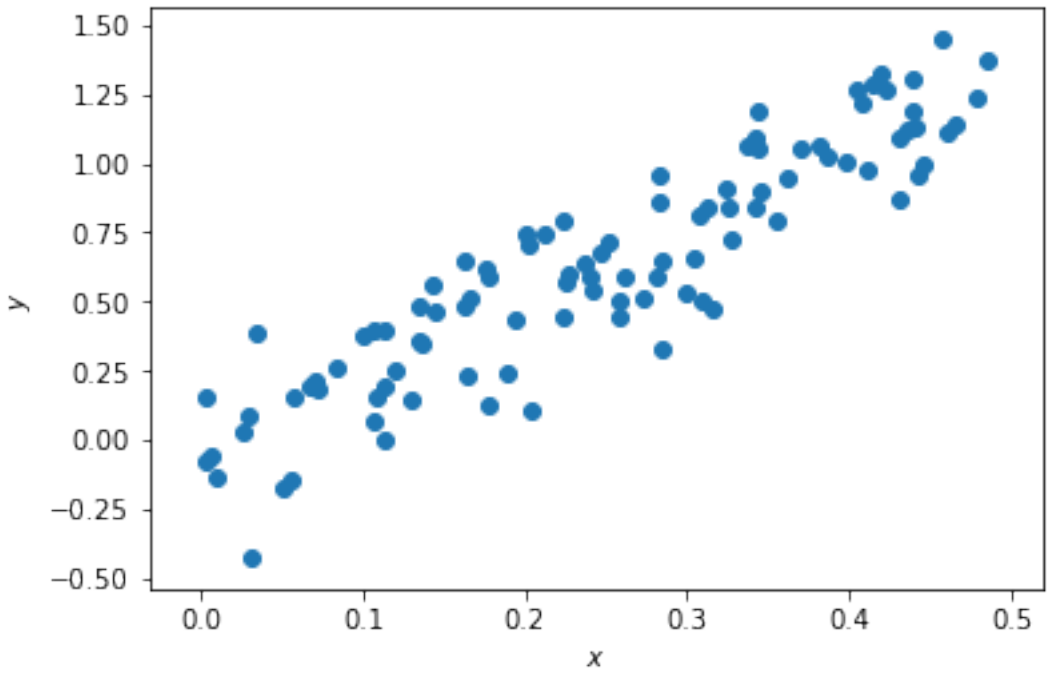
```
In [5]: N = 100
        a_true, b_true, c_true = -1., 2., 1.5

        # our input and output
        x = np.random.rand(N)*0.5#+0.5
        y = f_model(x, a_true, b_true, c_true)

        # + some (known) gaussian noise
        error = 0.2
        y += np.random.normal(0, error, N)

        # and plot our data
        pylab.scatter(x, y);
        pylab.xlabel("$x$")
        pylab.ylabel("$y$")
```

```
Out[5]: <matplotlib.text.Text at 0x10d7b35c0>
```




```
In [6]: # our likelihood
        def lnlike(theta, x, y, error):
            a, b, c = theta
            model =  f_model(x, a, b, c)
            chisq = 0.5*(np.sum((y-model)**2/error**2))
            return -chisq
        result = op.minimize(lambda *args: -lnlike(*args), [a_true, b_true, c_true], args=(x, y, erro
        # find the max likelihood
        a_ml, b_ml, c_ml = result["x"]
        print("estimates", a_ml, b_ml, c_ml)
        print("true values", a_true, b_true, c_true)
        result["message"]
```

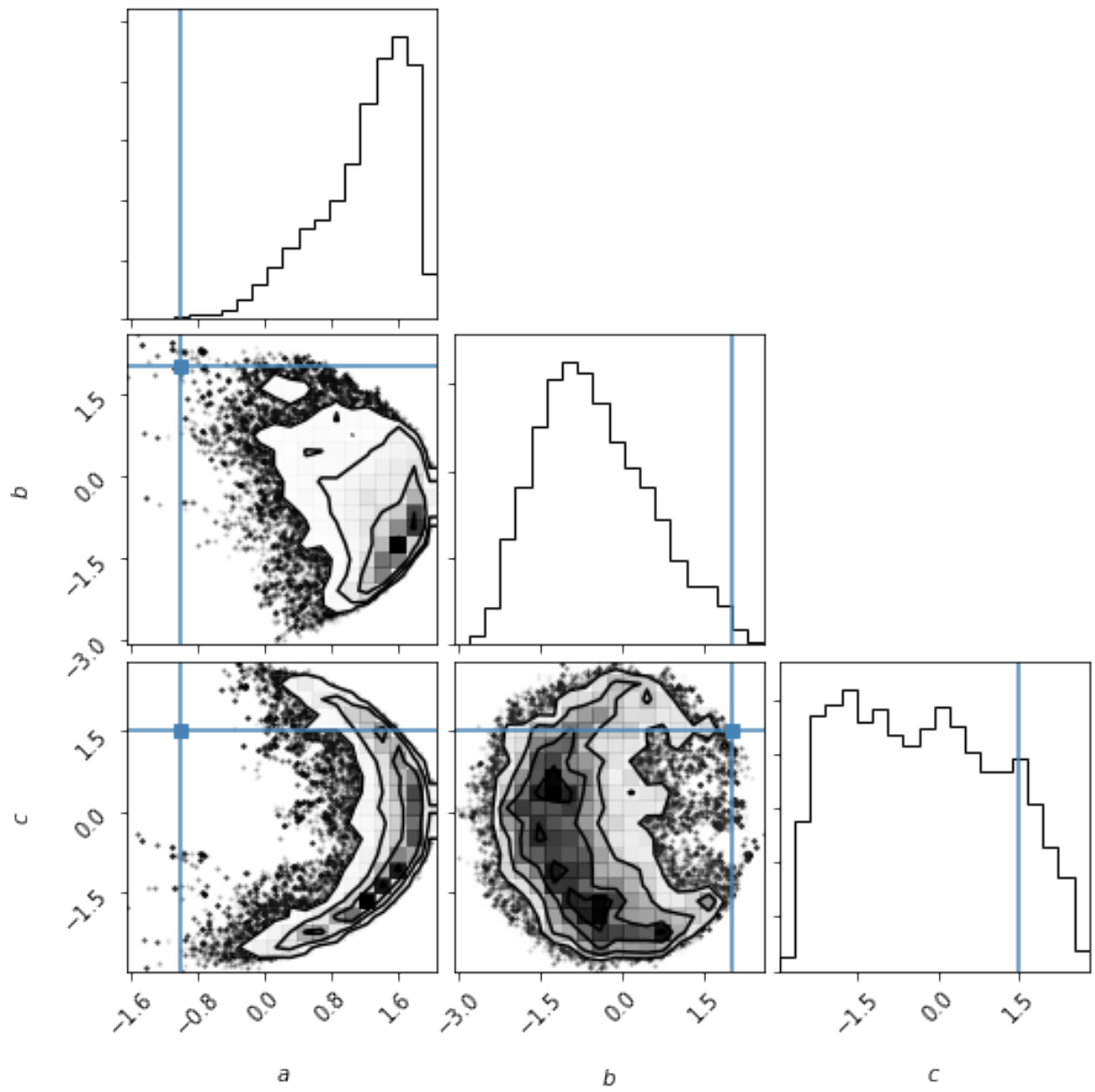
```
estimates 1.74022905195 -1.23351935318 -1.68793098984e-05
true values -1.0 2.0 1.5
```

```
Out[6]: 'Optimization terminated successfully.'
```

```
In [7]: # do the mcmc walk
        ndim, nwalkers = 3, 100
        pos = [result["x"] + np.random.randn(ndim)*0.1 for i in range(nwalkers)]
        sampler = emcee.EnsembleSampler(nwalkers, ndim, lnlike, args=(x, y, error))
        sampler.run_mcmc(pos, 1500);
        samples = sampler.chain[:, 50:, :].reshape((-1, ndim))
```

Posterior in 2d

```
In [8]: # plot the 2d pdfs
        import corner
        fig = corner.corner(samples, labels=["$a$", "$b$", "$c$"],
                            truths=[a_true, b_true, c_true])
```



Posterior in 3d

```
In [9]: import vaex
        import scipy.ndimage
        import ipyvolume
```

```
In [10]: ds = vaex.from_arrays(a=samples[...,0].copy(), b=samples[...,1].copy(), c=samples[...,2].cop
         # get 2d histogram
         v = ds.count(binby=["a", "b", "c"], shape=64)
         # smooth it for visual pleasure
         v = scipy.ndimage.gaussian_filter(v, 2)
```

```
In [11]: ipyvolume.quickvolshow(v, lighting=True)
```

```
A Jupyter Widget
```

Note that actually a large part of the volume is empty.

Meshes / Surfaces

Meshes (or surfaces) in ipyvolume consist of triangles, and are defined by their coordinate (vertices) and faces/triangles, which refer to three vertices.

```
In [1]: import ipyvolume.pylab as p3
        import numpy as np
```

Triangle meshes

Lets first construct a 'solid', a tetrahedron, consisting out of 4 vertices, and 4 faces (triangles) using plot_trisurf

```
In [2]: s = 1/2**0.5
        # 4 vertices for the tetrahedron
        x = np.array([1.,  -1, 0,  0])
        y = np.array([0,   0, 1., -1])
        z = np.array([-s, -s, s,  s])
        # and 4 surfaces (triangles), where the number refer to the vertex index
        triangles = [(0, 1, 2), (0, 1, 3), (0, 2, 3), (1,3,2)]
```

```
In [3]: p3.figure()
        # we draw the tetrahedron
        p3.plot_trisurf(x, y, z, triangles=triangles, color='orange')
        # and also mark the vertices
        p3.scatter(x, y, z, marker='sphere', color='blue')
        p3.xyzlim(-2, 2)
        p3.show()
```

```
A Jupyter Widget
```

Surfaces

To draw parametric surfaces, which go from $\mathbb{R}^2 \rightarrow \mathbb{R}^3$, it's convenient to use plot_surface, which takes 2d numpy arrays as arguments, assuming they form a regular grid (meaning you do not need to provide the triangles, since they can be inferred from the shape of the arrays). Note that plot_wireframe has a similar api, as does plot_mesh which can do both the surface and wireframe at the same time.

```
In [4]: # f(u, v) -> (u, v, u*v**2)
        a = np.arange(-5, 5)
        U, V = np.meshgrid(a, a)
        X = U
        Y = V
        Z = X*Y**2
        p3.figure()
        p3.plot_surface(X, Z, Y, color="orange")
        p3.plot_wireframe(X, Z, Y, color="black")
        p3.show()
```

```
A Jupyter Widget
```

Colors

Vertices can take colors as well, as the example below (adapted from matplotlib) shows.

```
In [5]: X = np.arange(-5, 5, 0.25*1)
        Y = np.arange(-5, 5, 0.25*1)
        X, Y = np.meshgrid(X, Y)
        R = np.sqrt(X**2 + Y**2)
        Z = np.sin(R)

In [6]: from matplotlib import cm
        colormap = cm.coolwarm
        znorm = Z - Z.min()
        znorm /= znorm.ptp()
        znorm.min(), znorm.max()
        color = colormap(znorm)

In [7]: p3.figure()
        mesh = p3.plot_surface(X, Z, Y, color=color[...,0:3])
        p3.show()
```

A Jupyter Widget

Texture mapping

Texture mapping can be done by providing a PIL image, and UV coordiante (texture coordinates, between 0 and 1). Note that like almost anything in ipyvolume, these u & v coordinates can be animated, as well as the textures.

```
In [8]: import PIL.Image
        image = PIL.Image.open('data/jupyter.png')

In [9]: fig = p3.figure()
        p3.style.use('dark')
        # we create a sequence of 8 u v coordinates so that the texture moves across the surface.
        u = np.array([X/5 +np.sin(k/8*np.pi)*4. for k in range(8)])
        v = np.array([-Y/5*(1-k/7.) + Z*(k/7.) for k in range(8)])
        mesh = p3.plot_mesh(X, Z, Y, u=u, v=v, texture=image, wireframe=False)
        p3.animation_control(mesh, interval=800, sequence_length=8)
        p3.show()
```

A Jupyter Widget

We now make a small movie / animated gif of 30 frames.

```
In [10]: frames = 30
         p3.movie('movie.gif', frames=frames)
```

A Jupyter Widget

And play that movie on a square

```
In [11]: p3.figure()
         x = np.array([-1.,  1,  1,  -1])
         y = np.array([-1,  -1, 1., 1])
         z = np.array([0., 0, 0., 0])
         u = x / 2 + 0.5
         v = y / 2 + 0.5
         # square
         triangles = [(0, 1, 2), (0, 2, 3)]
         m = p3.plot_trisurf(x, y, z, triangles=triangles, u=u, v=v, texture=PIL.Image.open('movie.gi
         p3.animation_control(m, sequence_length=frames)
         p3.show()
```

A Jupyter Widget

Animation

All (or most of) the changes in scatter and quiver plots are (linearly) interpolated. On top top that, scatter plots and quiver plots can take a sequence of arrays (the first dimension), where only one array is visualized. Together this can make smooth animations with coarse timesteps. Lets see an example.

```
In [1]: import ipyvolume.pylab as p3
        import numpy as np
```

Basic animation

```
In [2]: # only x is a sequence of arrays
        x = np.array([[-1, -0.8], [1, -0.1], [0., 0.5]])
        y = np.array([0.0, 0.0])
        z = np.array([0.0, 0.0])
        p3.figure()
        s = p3.scatter(x, y, z)
        p3.xyzlim(-1, 1)
        p3.animation_control(s) # shows controls for animation controls
        p3.show()
```

```
A Jupyter Widget
```

You can control which array to visualize, using the `scatter.sequence_index` property. Actually, the `pylab.animate_glyphs` is connecting the `Slider` and `Play` button to that property, but you can also set it from Python.

```
In [3]: s.sequence_index = 1
```

Animating color and size

We now demonstrate that you can also animate color and size

```
In [4]: # create 2d grids: x, y, and r
        u = np.linspace(-10, 10, 25)
        x, y = np.meshgrid(u, u)
        r = np.sqrt(x**2+y**2)
        print("x,y and z are of shape", x.shape)
        # and turn them into 1d
        x = x.flatten()
        y = y.flatten()
        r = r.flatten()
        print("and flattened of shape", x.shape)
```

```
x,y and z are of shape (25, 25)
and flattened of shape (625,)
```

Now we only animate the z component

```
In [5]: # create a sequence of 15 time elements
        time = np.linspace(0, np.pi*2, 15)
        z = np.array([(np.cos(r + t) * np.exp(-r/5)) for t in time])
        print("z is of shape", z.shape)
```

```
z is of shape (15, 625)
```

```
In [6]: # draw the scatter plot, and add controls with animate_glyphs
        p3.figure()
        s = p3.scatter(x, z, y, marker="sphere")
        p3.animation_control(s, interval=200)
```

```
        p3.ylim(-3,3)
        p3.show()
```

```
A Jupyter Widget
```

```
In [7]: # Now also include, color, which containts rgb values
        color = np.array([[np.cos(r + t), 1-np.abs(z[i]), 0.1+z[i]*0] for i, t in enumerate(time)])
        size = (z+1)
        print("color is of shape", color.shape)
```

```
color is of shape (15, 3, 625)
```

color is of the wrong shape, the last dimension should contain the rgb value, i.e. the shape of should be (15, 2500, 3)

```
In [8]: color = np.transpose(color, (0, 2, 1)) # flip the last axes
```

```
In [9]: p3.figure()
        s = p3.scatter(x, z, y, color=color, size=size, marker="sphere")
        p3.animation_control(s, interval=200)
        p3.ylim(-3,3)
        p3.show()
```

```
A Jupyter Widget
```

Creating movie files

We now make a movie, with a 2 second duration, where we rotate the camera, and change the size of the scatter points.

```
In [10]: def set_view(figure, framenr, fraction):
             p3.view(fraction*360, (fraction - 0.5) * 180)
             s.size = size * (1+0.5*np.sin(fraction * 6 * np.pi))
         p3.movie('wave.gif', set_view, fps=20, frames=40)
```

```
A Jupyter Widget
```

```
In [11]: # include the gif with base64 encoding
         import IPython.display
         import base64
         with open('wave.gif', 'rb') as gif:
             url = b"" +base64.b64encode(gif.read())
         IPython.display.Image(url=url.decode('ascii'))
```

```
Out[11]: <IPython.core.display.Image object>
```

Animated quiver

Not only scatter plots can be animated, quiver as well, so the direction vector (vx, vy, vz) can also be animated, as shown in the example below, which is a (subsample of) a simulation of a small galaxy orbiting a host galaxy (not visible).

```
In [12]: import ipyvolume.datasets
         stream = ipyvolume.datasets.animated_stream.fetch()
         print("shape of steam data", stream.data.shape) # first dimension contains x, y, z, vx, vy,
```

```
shape of steam data (6, 200, 1250)
```

```
In [13]: fig = p3.figure()
         # instead of doing x=stream.data[0], y=stream.data[1], ... vz=stream.data[5], use *stream.da
         # limit to 50 timesteps to avoid having a huge notebook
         q = p3.quiver(*stream.data[:,0:50,:200], color="red", size=7)
         p3.style.use("dark") # looks better
```

```
        p3.animation_control(q, interval=200)
        p3.show()
```

```
A Jupyter Widget
```

```
In [ ]: fig.animation = 0 # set to 0 for no interpolation
```

API docs

Note that ipyvolume.pylab and ipyvolume.widgets are imported in the ipyvolume namepsace, to you can access ipyvolume.scatter instead of ipyvolume.pylab.scatter.

ipyvolume.pylab

ipyvolume.pylab.**plot**(*x*, *y*, *z*, *color='red'*, ***kwargs*)

Plot a line in 3d

Parameters

- **x** – numpy array of shape (N,) or (S, N) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z
- **color** – color for each point/vertex/symbol, can be string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (N, 3) or (S, N, 3)
- **kwargs** – extra arguments passed to the Scatter constructor

Returns *Scatter*

ipyvolume.pylab.**scatter**(*x*, *y*, *z*, *color='red'*, *size=2*, *size_selected=2.6*, *color_selected='white'*, *marker='diamond'*, *selection=None*, ***kwargs*)

Plots many markers/symbols in 3d

Parameters

- **x** – numpy array of shape (N,) or (S, N) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z
- **color** – color for each point/vertex/symbol, can be string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (N, 3) or (S, N, 3)
- **size** – float representing the size of the glyph in percentage of the viewport, where 100 is the full size of the viewport
- **size_selected** – like size, but for selected glyphs
- **color_selected** – like color, but for selected glyphs
- **marker** – name of the marker, options are: 'arrow', 'box', 'diamond', 'sphere'
- **selection** – numpy array of shape (N,) or (S, N) with indices of x,y,z arrays of the selected markers, which can have a different size and color
- **kwargs** –

Returns *Scatter*

ipyvolume.pylab.**quiver**(*x, y, z, u, v, w, size=20, size_selected=26.0, color='red', color_selected='white', marker='arrow', **kwargs*)

Create a quiver plot, which is like a scatter plot but with arrows pointing in the direction given by u, v and w

Parameters

- **x** – numpy array of shape (N,) or (S, N) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z
- **u** – numpy array of shape (N,) or (S, N) indicating the x component of a vector (can be a sequence)
- **v** – idem for y
- **w** – idem for z
- **size** – float representing the size of the glyph in percentage of the viewport, where 100 is the full size of the viewport
- **size_selected** – like size, but for selected glyphs
- **color** – color for each point/vertex/symbol, can be string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (N, 3) or (S, N, 3)
- **color_selected** – like color, but for selected glyphs
- **marker** – (currently only 'arrow' would make sense)
- **kwargs** – extra arguments passed on to the Scatter constructor

Returns *Scatter*

ipyvolume.pylab.**volshow**(*data, lighting=False, data_min=None, data_max=None, tf=None, stereo=False, ambient_coefficient=0.5, diffuse_coefficient=0.8, specular_coefficient=0.5, specular_exponent=5, downscale=1, level=[0.1, 0.5, 0.9], opacity=[0.01, 0.05, 0.1], level_width=0.1, controls=True, max_opacity=0.2*)

Visualize a 3d array using volume rendering.

Currently only 1 volume can be rendered.

Parameters

- **data** – 3d numpy array
- **lighting** (*bool*) – use lighting or not, if set to false, lighting parameters will be overriden
- **data_min** (*float*) – minimum value to consider for data, if None, computed using np.nanmin
- **data_max** (*float*) – maximum value to consider for data, if None, computed using np.nanmax
- **tf** – transfer function (or a default one)
- **stereo** (*bool*) – stereo view for virtual reality (cardboard and similar VR head mount)
- **ambient_coefficient** – lighting parameter
- **diffuse_coefficient** – lighting parameter
- **specular_coefficient** – lighting parameter
- **specular_exponent** – lighting parameter

- **downscale** (*float*) – downscale the rendering for better performance, for instance when set to 2, a 512x512 canvas will show a 256x256 rendering upscaled, but it will render twice as fast.
- **level** – level(s) for the where the opacity in the volume peaks, maximum sequence of length 3
- **opacity** – opacity(ies) for each level, scalar or sequence of max length 3
- **level_width** – width of the (gaussian) bumps where the opacity peaks, scalar or sequence of max length 3
- **controls** (*bool*) – add controls for lighting and transfer function or not
- **max_opacity** (*float*) – maximum opacity for transfer function controls

Returns

ipyvolume.pylab.**plot_surface**(*x*, *y*, *z*, *color='red'*, *wrapx=False*, *wrapy=False*)

Draws a 2d surface in 3d, defined by the 2d ordered arrays x,y,z

Parameters

- **x** – numpy array of shape (N,M) or (S, N, M) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z
- **color** – color for each point/vertex string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (2, N, 3) or (S, 2, N, 3)
- **wrapx** (*bool*) – when True, the x direction is assumed to wrap, and polygons are drawn between the end end begin points
- **wrapy** (*bool*) – simular for the y coordinate

Returns *Mesh*

ipyvolume.pylab.**plot_trisurf**(*x*, *y*, *z*, *triangles=None*, *lines=None*, *color='red'*, *u=None*, *v=None*, *texture=None*)

Draws a polygon/triangle mesh defined by a coordinate and triangle indices

The following example plots a rectangle in the z==2 plane, consisting of 2 triangles:

```
>>> plot_trisurf([0, 0, 3., 3.], [0, 4., 0, 4.], 2,
       triangles=[[0, 2, 3], [0, 3, 1]])
```

Note that the z value is constant, and thus not a list/array. For guidance, the triangles refer to the vertices in this manner:

```
^ ydir
|
2 3
0 1  ---> x dir
```

Note that if you want per face/triangle colors, you need to duplicate each vertex.

Parameters

- **x** – numpy array of shape (N,) or (S, N) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z

- **triangles** – numpy array with indices referring to the vertices, defining the triangles, with shape (M, 3)
- **lines** – numpy array with indices referring to the vertices, defining the lines, with shape (K, 2)
- **color** – color for each point/vertex/symbol, can be string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (N, 3) or (S, N, 3)
- **u** – numpy array of shape (N,) or (S, N) indicating the u (x) coordinate for the texture (can be a sequence)
- **v** – numpy array of shape (N,) or (S, N) indicating the v (y) coordinate for the texture (can be a sequence)
- **texture** – PIL.Image object or ipywebrtc.MediaStream (can be a seqence)

Returns *Mesh*

ipyvolume.pylab.**plot_wireframe**(*x*, *y*, *z*, *color='red'*, *wrapx=False*, *wrapy=False*)

Draws a 2d wireframe in 3d, defines by the 2d ordered arrays x,y,z

See also *ipyvolume.pylab.plot_mesh*

Parameters

- **x** – numpy array of shape (N,M) or (S, N, M) with x positions (can be a sequence)
- **y** – idem for y
- **z** – idem for z
- **color** – color for each point/vertex string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0), or rgb array of shape (2, N, 3) or (S, 2, N, 3)
- **wrapx** (*bool*) – when True, the x direction is assumed to wrap, and polygons are drawn between the begin and end points
- **wrapy** (*bool*) – idem for y

Returns *Mesh*

ipyvolume.pylab.**plot_mesh**(*x*, *y*, *z*, *color='red'*, *wireframe=True*, *surface=True*, *wrapx=False*, *wrapy=False*, *u=None*, *v=None*, *texture=None*)

Draws a 2d wireframe+surface in 3d: generalization of *plot_wireframe* and *plot_surface*

Parameters

- **x** – {x2d}
- **y** – {y2d}
- **z** – {z2d}
- **color** – {color2d}
- **wireframe** (*bool*) – draw lines between the vertices
- **surface** (*bool*) – draw faces/triangles between the vertices
- **wrapx** (*bool*) – when True, the x direction is assumed to wrap, and polygons are drawn between the begin and end points
- **wrapy** (*boool*) – idem for y
- **u** – {u}
- **v** – {v}

- **texture** – {texture}

Returns *Mesh*

ipyvolume.pylab.**plot_isosurface**(*data*, *level=None*, *color='red'*, *wireframe=True*, *surface=True*, *controls=True*)

Plot a surface at constant value (like a 2d contour)

Parameters

- **level** (*float*) – value where the surface should lie
- **color** – color of the surface, although it can be an array, the length is difficult to predict beforehand, if per vertex color are needed, it is better to set them on the returned mesh afterwards.
- **wireframe** (*bool*) – draw lines between the vertices
- **surface** (*bool*) – draw faces/triangles between the vertices
- **controls** (*bool*) – add controls to change the isosurface

Returns *Mesh*

ipyvolume.pylab.**xlim**(*xmin*, *xmax*)

Set limits of x axis

ipyvolume.pylab.**ylim**(*ymin*, *ymax*)

Set limits of y axis

ipyvolume.pylab.**zlim**(*zmin*, *zmax*)

Set limits of zaxis

ipyvolume.pylab.**xyzlim**(*vmin*, *vmax=None*)

Set limits or all axis the same, if vmax not given, use [-vmin, vmax]

ipyvolume.pylab.**xlabel**(*label*)

Set the labels for the x-axis

ipyvolume.pylab.**ylabel**(*label*)

Set the labels for the y-axis

ipyvolume.pylab.**zlabel**(*label*)

Set the labels for the z-axis

ipyvolume.pylab.**xyzlabel**(*labelx*, *labely*, *labelz*)

Set all labels at once

ipyvolume.pylab.**view**(*azimuth*, *elevation*)

Sets camera angles

Parameters

- **azimuth** (*float*) – rotation around the axis pointing up in degrees
- **elevation** (*float*) – rotation where +90 means 'up', -90 means 'down', in degrees

ipyvolume.pylab.**figure**(*key=None*, *width=400*, *height=500*, *lighting=True*, *controls=True*, *controls_vr=False*, *debug=False*, ***kwargs*)

Create a new figure (if no key is given) or return the figure associated with key

Parameters

- **key** – Python object that identifies this figure
- **width** (*int*) – pixel width of WebGL canvas

- **height** (*int*) –
- **lighting** (*bool*) – use lighting or not
- **controls** (*bool*) – show controls or not
- **controls_vr** (*bool*) – show controls for VR or not
- **debug** (*bool*) – show debug buttons or not

Returns *Figure*

ipyvolume.pylab.**gcf**()
Get current figure, or create a new one

Returns *Figure*

ipyvolume.pylab.**gcc**()
Return the current container, that is the widget holding the figure and all the control widgets, buttons etc.

ipyvolume.pylab.**clear**()
Remove current figure (and container)

ipyvolume.pylab.**show**(*extra_widgets=[]*)
Display (like in IPython.display.dispay(...)) the current figure

ipyvolume.pylab.**save**(*filepath*, *makedirs=True*, *title=u'IPyVolume Widget'*, *all_states=False*, *offline=False*, *scripts_path='js'*, *drop_defaults=False*, *template_options=(('extra_script_head', ''), ('body_pre', ''), ('body_post', ''))*, *devmode=False*, *offline_cors=False*)
Save the current container to a HTML file

By default the HTML file is not standalone and requires an internet connection to fetch a few javascript libraries. Use offline=True to download these and make the HTML file work without an internet connection.

Parameters

- **filepath** (*str*) – The file to write the HTML output to.
- **makedirs** (*bool*) – whether to make directories in the filename path, if they do not already exist
- **title** (*str*) – title for the html page
- **all_states** (*bool*) – if True, the state of all widgets know to the widget manager is included, else only those in widgets
- **offline** (*bool*) – if True, use local urls for required js/css packages and download all js/css required packages (if not already available), such that the html can be viewed with no internet connection
- **scripts_path** (*str*) – the folder to save required js/css packages to (relative to the filepath)
- **drop_defaults** (*bool*) – Whether to drop default values from the widget states
- **template_options** – list or dict of additional template options
- **devmode** (*bool*) – if True, attempt to get index.js from local js/dist folder
- **offline_cors** (*bool*) – if True, sets crossorigin attribute of script tags to anonymous

ipyvolume.pylab.**savefig**(*filename*, *width=None*, *height=None*, *fig=None*, *timeout_seconds=10*, *output_widget=None*)
Save the figure to an image file.

Parameters

- **filename** (*str*) – must have extension .png, .jpeg or .svg
- **width** (*int*) – the width of the image in pixels
- **height** (*int*) – the height of the image in pixels
- **fig** (*ipyvolume.widgets.Figure or None*) – if None use the current figure
- **timeout_seconds** (*float*) – maximum time to wait for image data to return
- **output_widget** (*ipywidgets.Output*) – a widget to use as a context manager for capturing the data

ipyvolume.pylab.**screenshot**(*width=None*, *height=None*, *format='png'*, *fig=None*, *timeout_seconds=10*, *output_widget=None*)

Save the figure to a PIL.Image object.

Parameters

- **width** (*int*) – the width of the image in pixels
- **height** (*int*) – the height of the image in pixels
- **format** – format of output data (png, jpeg or svg)
- **fig** (*ipyvolume.widgets.Figure or None*) – if None use the current figure
- **timeout_seconds** (*int*) – maximum time to wait for image data to return
- **output_widget** (*ipywidgets.Output*) – a widget to use as a context manager for capturing the data

Returns PIL.Image

ipyvolume.pylab.**movie**(*f='movie.mp4'*, *function=<function _change_y_angle>*, *fps=30*, *frames=30*, *endpoint=False*, *cmd_template_ffmpeg='ffmpeg -y -r {fps} -i {tempdir}/frame-%5d.png -vcodec h264 -pix_fmt yuv420p {filename}'*, *cmd_template_gif='convert -delay {delay} {loop} {tempdir}/frame-*.png {filename}'*, *gif_loop=0*)

Create a movie (mp4/gif) out of many frames

If the filename ends in *.gif*, *convert* is used to convert all frames to an animated gif using the *cmd_template_gif* template. Otherwise *ffmpeg is assumed to know the file format*.

Example:

```
>>> def set_angles(fig, i, fraction):
>>>     fig.angley = fraction*np.pi*2
>>> # 4 second movie, that rotates around the y axis
>>> p3.movie('test2.gif', set_angles, fps=20, frames=20*4,
        endpoint=False)
```

Note that in the example above we use *endpoint=False* to avoid to first and last frame to be the same

Parameters

- **f** (*str*) – filename out output movie (e.g. 'movie.mp4' or 'movie.gif')
- **function** – function called before each frame with arguments (figure, framenr, fraction)
- **fps** – frames per seconds
- **frames** (*int*) – total number of frames
- **endpoint** (*bool*) – if fraction goes from [0, 1] (inclusive) or [0, 1) (endpoint=False is useful for loops/rotatations)

- **cmd_template_ffmpeg** (*str*) – template command when running ffmpeg (non-gif ending filenames)
- **cmd_template_gif** (*str*) – template command when running imagemagick's convert (if filename ends in .gif)
- **gif_loop** – None for no loop, otherwise the framenumber to go to after the last frame

Returns the temp dir where the frames are stored

ipyvolume.pylab.**animation_control**(*object*, *sequence_length=None*, *add=True*, *interval=200*)

Animate scatter, quiver or mesh by adding a slider and play button.

Parameters

- **object** – *Scatter* or *Mesh* object (having an sequence_index property), or a list of these to control multiple.
- **sequence_length** – If sequence_length is None we try try our best to figure out, in case we do it badly, you can tell us what it should be. Should be equal to the S in the shape of the numpy arrays as for instance documented in *scatter* or *plot_mesh*.
- **add** – if True, add the widgets to the container, else return a HBox with the slider and play button. Useful when you want to customise the layout of the widgets yourself.
- **interval** – interval in msec between each frame

Returns If add is False, if returns the ipywidgets.HBox object containing the controls

ipyvolume.pylab.**transfer_function**(*level=[0.1, 0.5, 0.9], opacity=[0.01, 0.05, 0.1], level_width=0.1, controls=True, max_opacity=0.2*)

Create a transfer function, see volshow

class ipyvolume.pylab.**style**

'Static class that mimics a matplotlib module.

Example: >>> import ipyvolume.pylab as p3 >>> p3.style.use('light']) >>> p3.style.use('seaborn-darkgrid']) >>> p3.style.use(['seaborn-darkgrid', {'axes.x.color':'orange'}])

Possible style values:

- figure.facecolor: background color
- axes.color: color of the box around the volume/viewport
- xaxis.color: color of xaxis
- yaxis.color: color of xaxis
- zaxis.color: color of xaxis

static use(*style*)

Set the style of the current figure/visualization

Parameters **style** – matplotlib style name, or dict with values, or a sequence of these, where the last value overrides previous

Returns

ipyvolume.widgets

ipyvolume.widgets.**quickvolshow**(*data, lighting=False, data_min=None, data_max=None, tf=None, stereo=False, width=400, height=500, ambient_coefficient=0.5, diffuse_coefficient=0.8, specular_coefficient=0.5, specular_exponent=5, downscale=1, level=[0.1, 0.5, 0.9], opacity=[0.01, 0.05, 0.1], level_width=0.1, **kwargs*)

Visualize a 3d array using volume rendering

Parameters

- **data** – 3d numpy array
- **lighting** – boolean, to use lighting or not, if set to false, lighting parameters will be overriden
- **data_min** – minimum value to consider for data, if None, computed using np.nanmin
- **data_max** – maximum value to consider for data, if None, computed using np.nanmax
- **tf** – transfer function (see ipyvolume.transfer_function, or use the argument below)
- **stereo** – stereo view for virtual reality (cardboard and similar VR head mount)
- **width** – width of rendering surface
- **height** – height of rendering surface
- **ambient_coefficient** – lighting parameter
- **diffuse_coefficient** – lighting parameter
- **specular_coefficient** – lighting parameter
- **specular_exponent** – lighting parameter
- **downscale** – downscale the rendering for better performance, for instance when set to 2, a 512x512 canvas will show a 256x256 rendering upscaled, but it will render twice as fast.
- **level** – level(s) for the where the opacity in the volume peaks, maximum sequence of length 3
- **opacity** – opacity(ies) for each level, scalar or sequence of max length 3
- **level_width** – width of the (gaussian) bumps where the opacity peaks, scalar or sequence of max length 3
- **kwargs** – extra argument passed to Volume and default transfer function

Returns

ipyvolume.widgets.**quickscatter**(*x*, *y*, *z*, ***kwargs*)

ipyvolume.widgets.**quickquiver**(*x*, *y*, *z*, *u*, *v*, *w*, ***kwargs*)

ipyvolume.widgets.**volshow**(**args*, ***kwargs*)

Deprecated: please use ipyvolume.quickvolshow or use the ipyvolume.pylab interface

class ipyvolume.widgets.**Figure**(***kwargs*)

Bases: ipywebrtc.webrtc.MediaStream

Widget class representing a volume (rendering) using three.js

ambient_coefficient

A float trait.

angle_order
A trait for unicode strings.

anglex
A float trait.

angley
A float trait.

anglez
A float trait.

animation
A float trait.

animation_exponent
A float trait.

camera_center
An instance of a Python list.

camera_control
A trait for unicode strings.

camera_fov
A casting version of the float trait.

data_max
A casting version of the float trait.

data_min
A casting version of the float trait.

diffuse_coefficient
A float trait.

downscale
A casting version of the int trait.

eye_separation
A casting version of the float trait.

height
A casting version of the int trait.

matrix_projection
An instance of a Python list.

matrix_world
An instance of a Python list.

meshes
An instance of a Python list.

on_lasso(*callback*, *remove=False*)

on_screenshot(*callback*, *remove=False*)

project(*x*, *y*, *z*)

render_continuous
A boolean (True, False) trait.

scatters
An instance of a Python list.

screenshot(*width=None*, *height=None*, *mime_type='image/png'*)

show
A trait for unicode strings.

specular_coefficient
A float trait.

specular_exponent
A float trait.

stereo
A boolean (True, False) trait.

style
An instance of a Python dict.

tf
A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

volume_data
A numpy array trait type.

width
A casting version of the int trait.

xlabel
A trait for unicode strings.

xlim
An instance of a Python list.

ylabel
A trait for unicode strings.

ylim
An instance of a Python list.

zlabel
A trait for unicode strings.

zlim
An instance of a Python list.

class ipyvolume.widgets.**Scatter**(***kwargs*)
Bases: ipywidgets.widgets.domwidget.DOMWidget

color
A numpy array trait type.

color_selected
A trait type representing a Union type.

connected
A casting version of the boolean trait.

geo
A trait for unicode strings.

selected
A numpy array trait type.

sequence_index
An integer trait.

Longs that are unnecessary (<= sys.maxint) are cast to ints.

size
A trait type representing a Union type.

size_selected
A trait type representing a Union type.

visible
A casting version of the boolean trait.

visible_lines
A casting version of the boolean trait.

visible_markers
A casting version of the boolean trait.

vx
A numpy array trait type.

vy
A numpy array trait type.

vz
A numpy array trait type.

x
A numpy array trait type.

y
A numpy array trait type.

z
A numpy array trait type.

class ipyvolume.widgets.**Mesh**(***kwargs*)
Bases: ipywidgets.widgets.domwidget.DOMWidget

color
A numpy array trait type.

lines
A numpy array trait type.

sequence_index
An integer trait.

Longs that are unnecessary (<= sys.maxint) are cast to ints.

texture
A trait type representing a Union type.

triangles
A numpy array trait type.

u
A numpy array trait type.

v
A numpy array trait type.

visible
A casting version of the boolean trait.

visible_faces
A casting version of the boolean trait.

visible_lines
A casting version of the boolean trait.

x
A numpy array trait type.

y
A numpy array trait type.

z
A numpy array trait type.

ipyvolume.headless

Virtual reality

Ipyvolume can render in stereo, and go fullscreen (not supported for iOS). Together with Google Cardboard or other VR glasses (I am using VR Box 2) this enables virtual reality visualisation. Since mobile devices are usually less powerful, the example below is rendered at low resolution to enable a reasonable framerate on all devices.

Open this page on your mobile device, enter fullscreen mode and put on your glasses, looking around will rotate the object to improve depth perception.

```
import ipyvolume as ipv
aqa2 = ipv.datasets.aquariusA2.fetch()
ipv.quickvolshow(aqa2.data.T, lighting=True, level=[0.16, 0.25, 0.46], width=256,␣
↪height=256, stereo=True, opacity=0.06)
```

[widget]

CHAPTER 5

Changelog

- 0.4
  - plotting
    - lines
    - wireframes
    - meshes/surfaces
    - isosurfaces
    - texture (animated) support, gif image and MediaStream (movie, camera, canvas)
  - camera control (angles from the python side), FoV
  - movie creation
  - eye separation for VR
  - better screenshot support (can be to a PIL Image), and higher resolution possible
  - mouse lasso (a bit rough), selections can be made from the Python side.
  - icon bar for common operations (fullscreen, stereo, screenshot, reset etc)
  - offline support for embedding/saving to html
  - Jupyter lab support
  - New contributors
    - Chris Sewell
    - Satrajit Ghosh
    - Sylvain Corlay
    - stonebig
    - Matt McCormick
    - Jean Michel Arbona

- 0.3
  - new
    - axis with labels and ticklabels
    - styling
    - animation (credits also to https://github.com/jeammimi)
    - binary transfers
    - default camera control is trackball
  - changed
    - s and ss are now spelled out, size and size_selected

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

R

S

T

U

V

W

X

Y

Z