ipyvolume Documentation

Release 0.3.2

Maarten A. Breddels

May 29, 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.volume.quickvolshow*. From a numpy array, we create two boxes, using slicing, and visualize it.

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

[widget]

Quiver plot

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

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

[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.pylab as p3
import numpy as np
x, y, z, u, v, w = np.random.random((6, 1000))*2-1
selected = np.random.randint(0, 1000, 100)
p3.figure()
quiver = p3.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([p3.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.pylab as p3
        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]: p3.figure()
        quiver = p3.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")
        p3.xyzlim(-30, 30)
        p3.show()
```

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 [6]: import ipywidgets
```

```
In [7]: out = ipywidgets.Output()
        with out:
            show(p)
```

```
In [8]: ipywidgets.HBox([out, p3.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 [9]: from bokeh.resources import CDN
        from bokeh.embed import components

        script, div = components((p))
        ipyvolume.embed.embed_html("bokeh.html",
               [p3.current.container, ipyvolume.bokeh.wmh], all=True,
               extra_script_head=script + CDN.render_js() + CDN.render_css(),
               body_pre="<h2>Do selections in 2d (bokeh)<h2>"+div+"<h2>And see the selection in ipyvo
```

```
In [ ]:
```

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()
```

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()
```

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
```

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 [11]: ipyvolume.embed.embed_html("bqplot.html", hbox)
```

```
In [ ]:
```

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 [1]:
```
import pylab
import scipy.optimize as op
import emcee
import numpy as np
%matplotlib inline
```

In [2]:
```
# 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
```

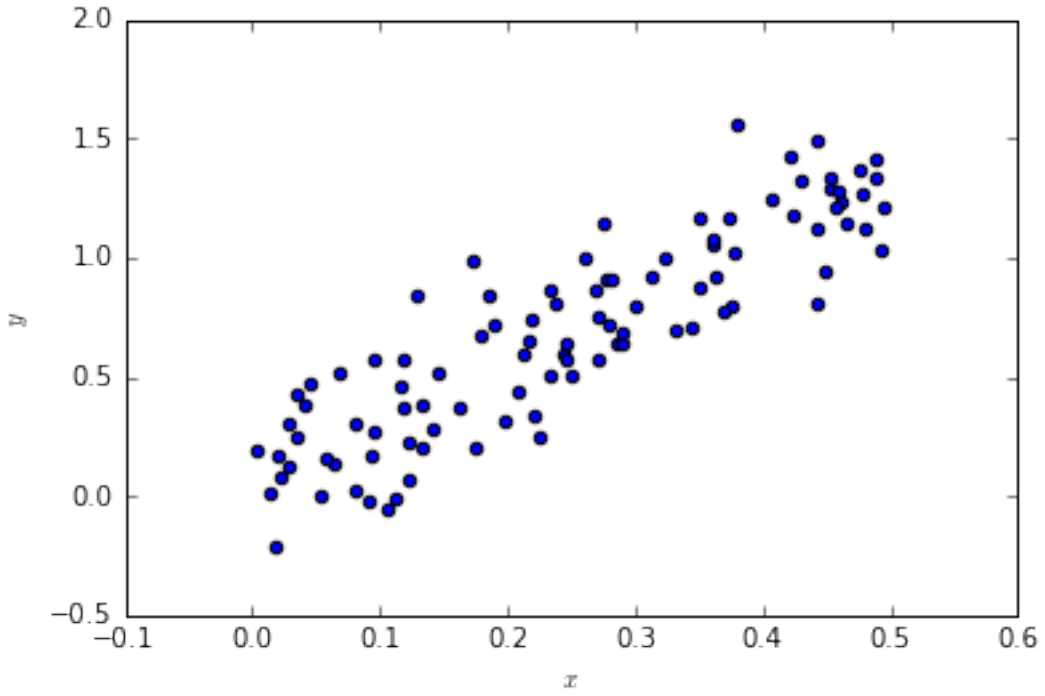
In [3]:
```
N = 100
a_true, b_true, c_true = -1., 2., 1.5

# 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[3]: <matplotlib.text.Text at 0x10d1cdb70>



In [4]:
```
# 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"]
```

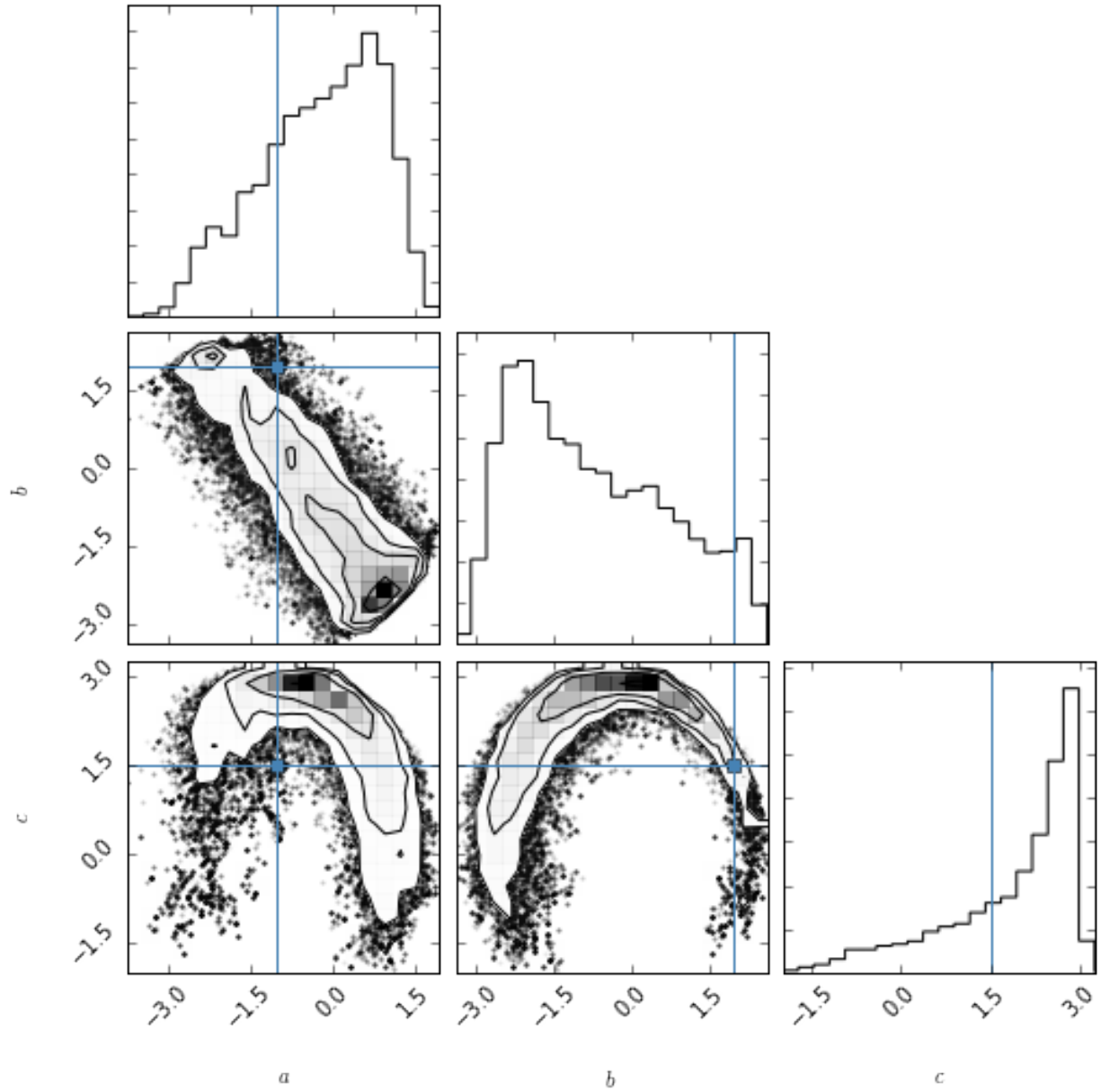
```
estimates 0.140490426416 -1.27278566041 2.56885371889
true values -1.0 2.0 1.5
```

```
Out[4]: 'Optimization terminated successfully.'
```

```
In [5]: # 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 [7]: # 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 [8]: import vaex
        import scipy.ndimage
        import ipyvolume
```

```
In [9]: ds = vaex.from_arrays(a=samples[...,0], b=samples[...,1], c=samples[...,2])
        # 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)
```

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

In [ ]:

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.animate_glyphs(s) # shows controls for animation controls
        p3.show()
```

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.animate_glyphs(s, interval=200)
        p3.ylim(-3,3)
        p3.show()
```

```
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.animate_glyphs(s, interval=200)
        p3.ylim(-3,3)
        p3.show()
```

```
/Users/maartenbreddels/anaconda3/lib/python3.5/site-packages/traitlets/traitlets.py:565: FutureWarnir
  silent = bool(old_value == new_value)
```

Performance

By default, data is transferred from Python to the Notebook (browser), using JSON. This may be slow for large volumes of data. To get better performance, change the following:

```
In [ ]: import ipyvolume.serialize
        ipyvolume.serialize.performance = 1 # 1 for binary, 0 for JSON
```

This will transfer all the data (numpy arrays) in binary, leading to a huge increase in performance. The downside (that is why it is not the default) is that embedding it in html (i.e. `p3.save(..)`) is not supported at the moment.

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 [10]: 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 [11]: 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.animate_glyphs(q, interval=200)
         p3.show()
```

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

```
In [ ]:
```

API docs

ipyvolume.volume

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

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

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

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

ipyvolume.pylab

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

Create a scatter 3d plot with

Parameters

- **x** – 1d numpy array with x positions
- **y** –
- **z** –
- **color** – string format, examples for red:'red', '#f00', '#ff0000' or 'rgb(1,0,0)
- **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** – array with indices of x,y,z arrays of the selected markers, which can have a different size and color
- **kwargs** –

Returns

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** – {x}
- **y** –
- **z** –
- **u** – {u}
- **v** –
- **w** –
- **size** – {size}
- **size_selected** – like size, but for selected glyphs
- **color** – {color}
- **color_selected** – like color, but for selected glyphs
- **marker** – (currently only 'arrow' would make sense)
- **kwargs** –

Returns

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*)

Visualize a 3d array using volume rendering.

Currently only 1 volume can be rendered.

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 (or a default one)
- **stereo** – 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** – 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** – add controls for lighting and transfer function or not

Returns

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*)

Set limits or all axis the same

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.**figure**(*key=None*, *width=400*, *height=500*, *lighting=True*, *controls=True*, *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** – pixel width of WebGL canvas
- **height** –
- **lighting** – use lighting or not
- **controls** – show controls or not
- **debug** – show debug buttons or not

Returns

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

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**(*filename*, *copy_js=True*, *makedirs=True*)
Save the figure/visualization as html file, and optionally copy the .js file to the same directory

ipyvolume.pylab.**savefig**(*filename*, *timeout_seconds=10*)
Save the current figure to an image (png or jpeg) to a file

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

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
aqa2 = ipyvolume.datasets.aquariusA2.fetch()
ipyvolume.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

- 0.3
  - 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