vtkplotlib Documentation

Release 0.1

bwoodsend

Apr 13, 2020

Contents:

CHAPTER 1

Introduction

A simple library to make 3D graphics using easy. Built on top of VTK. Whilst VTK is a very versatile library, the learning curve is steep and writing in it is slow and painful. This library seeks to overcome that by wrapping the all ugliness into numpy friendly functions to create a 3D equivalent of matplotlib. All the VTK components/functionality are still accessible but by default are already setup for you.

1.1 Key features

- Clean and easy to install.
- Takes advantage of VTK's lesser known numpy support so that data can be efficiently copied by reference between numpy and VTK making it much faster than most of the VTK examples you'll find online.
- Has direct support for STL plotting.
- Can be embedded seamlessly into PyQt5 applications.
- Is freezable with pyinstaller.

1.2 Requirements for installing:

- numpy
- pathlib2
- matplotlib
- vtk
- future

It can take VTK a few weeks to release to PyPi for a new Python version so using the absolute latest Python is not recomended. There is no VTK version for Windows users with python 2.7 on PyPi. But you can get a .whl from here.

1.3 Installation:

To install run the following into shell/bash/terminal. The mandatory dependencies will installed automatically.

```
pip install vtkplotlib
```

1.4 Optional requirements:

Some features require you to install the following:

- numpy-stl or any other STL library if you want to plot STL files. numpy-stl is my STL library of choice.
- PyQt5 if you want to include your plots in a larger Qt GUI.
- namegenerator for fun.

1.5 Quickstart:

1.5.1 Scatter plots:

```
import vtkplotlib as vpl
import numpy as np

# In vtkplotlib coordinates are always expressed as numpy arrays with shape
# (3,) or (n, 3) or (..., 3).
# Create 30 random points.
points = np.random.uniform(-10, 10, (30, 3))

# Plot the points as spheres.
vpl.scatter(points)

# Show the plot.
vpl.show()
```

A window should open with lots of white sphere's in it.

You can add some color using the color argument.

Colors can be assigned to all the balls using rgb

```
vpl.scatter(points, color=(.3, .8, .8))
vpl.show()
```

or rgba

```
vpl.scatter(points, color=(.3, .8, .8, .2))
vpl.show()
```

or using any of matplotlib's named colors using strings.

```
vpl.scatter(points, color="r")
vpl.show()
```

See matplotlib or `vpl.colors.mpl_colors` for a full list of available colors.

Or colors can be given per point

```
colors = np.random.random(points.shape)
vpl.scatter(points, color=colors)
vpl.show()
```

1.5.2 Line plots:

```
import vtkplotlib as vpl
import numpy as np

# Create some kind of wiggly shape
# use ``vpl.zip_axes`` to combine (x, y, z) axes
t = np.linspace(0, 2 * np.pi, 300)
points = vpl.zip_axes(np.cos(2 * t),
                      np.sin(3 * t),
                      np.cos(5 * t) * np.sin(7 *t))

# Plot a line
vpl.plot(points,
         color="green",
         line_width=3)

vpl.show()
```

For plotting a polygon you can use join_ends=True to join the last point with the first.

```
# Create the corners of an octagon
t = np.arange(0, 1, 1 / 8) *  2 * np.pi
points = vpl.zip_axes(np.cos(t), np.sin(t), 0)

# Plot them
vpl.plot(points,
         join_ends=True)

vpl.show()
```

1.5.3 Mesh plots:

class vtkplotlib.plots.MeshPlot.**MeshPlot**(*mesh_data*, *tri_scalars=None*, *scalars=None*, *color=None*, *opacity=None*, *cmap=None*, *fig='gcf'*, *label=None*)

To plot STL files you will need some kind of STL reader library. If you don't have one then get this one numpy-stl. Their Mesh class can be passed directly to vpl.mesh_plot.

Parameters

- **mesh_data** (*An STL (like) object (see below)*) – The mesh to plot.
- **tri_scalars** (*np.ndarray, optional*) – Per-triangle scalar, texture-coordinates or RGB values, defaults to None.
- **scalars** (*np.ndarray, optional*) – Per-vertex scalar, texture-coordinates or RGB values, defaults to None.

- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of the whole plot, ignored if scalars are used, defaults to white.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **cmap** (*str, 2D np.ndarray, matplotlib colormap, vtkLookupTable, optional*) – Colormap to use for scalars, defaults to *rainbow*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns A meshplot object.

Return type *vtkplotlib.plots.MeshPlot.MeshPlot*

The following example assumes you have installed numpy-stl.

```
import vtkplotlib as vpl
from stl.mesh import Mesh

# path = "if you have an STL file then put it's path here."
# Otherwise vtkplotlib comes with a small STL file for demos/testing.
path = vpl.data.get_rabbit_stl()

# Read the STL using numpy-stl
mesh = Mesh.from_file(path)

# Plot the mesh
vpl.mesh_plot(mesh)

# Show the figure
vpl.show()
```

Unfortunately there are far too many mesh/STL libraries/classes out there to support them all. To overcome this as best we can, mesh_plot has a flexible constructor which accepts any of the following.

1. Some kind of mesh class that has form 2) stored in mesh.vectors. For example numpy-stl's stl.mesh.Mesh or pymesh's pymesh.stl.Stl
2. An np.array with shape (n, 3, 3) in the form:

```
np.array([[[x, y, z],  # corner 0                          [x, y, z],  #␣
↪corner 1  | triangle 0
          [x, y, z]], # corner 2  /
         ...
          [[x, y, z],  # corner 0                          [x, y, z],  #␣
↪corner 1  | triangle n-1
          [x, y, z]], # corner 2  /
        ])
```

   Note it's not uncommon to have arrays of shape (n, 3, 4) or (n, 4, 3) where the additional entries' meanings are usually irrelevant (often to represent scalars but as STL has no color this is always uniform). Hence to support mesh classes that have these, these arrays are allowed and the extra entries are ignored.

3. An np.array with shape (k, 3) of (usually unique) vertices in the form:

```
np.array([[x, y, z],
          [x, y, z],
          ...
          [x, y, z],
          [x, y, z],
          ])
```

And a second argument of an np.array of integers with shape (n, 3) of point args in the form

```
np.array([[i, j, k],  # triangle 0
          ...
          [i, j, k],  # triangle n-1
          ])
```

where i, j, k are the indices of the points (in the vertices array) representing each corner of a triangle.

Note that this form can be (and is) easily converted to form 2) using

```
vertices = unique_vertices[point_args]
```

Hopefully this will cover most of the cases. If you are using or have written an STL library (or any other format) that you want supported then let me know. If it's numpy based then it's probably only a few extra lines to support. Or you can have a go at writing it yourself, either *with mesh_plot* or with the `vpl.PolyData` class.

Mesh plotting with scalars:

To create a heat map like image use the *scalars* or *tri_scalars* options.

Use the *scalars* option to assign a scalar value to each point/corner:

```
import vtkplotlib as vpl
from stl.mesh import Mesh

# Open an STL as before
path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

# Plot it with the z values as the scalars. scalars is 'per vertex' or 1
# value for each corner of each triangle and should have shape (n, 3).
plot = vpl.mesh_plot(mesh, scalars=mesh.z)

# Optionally the plot created by mesh_plot can be passed to color_bar
vpl.color_bar(plot, "Heights")

vpl.show()
```

Use the *tri_scalars* option to assign a scalar value to each triangle:

```
import vtkplotlib as vpl
from stl.mesh import Mesh
import numpy as np

# Open an STL as before
path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

# `tri_scalars` must have one value per triangle and have shape (n,) or (n, 1).
# Create some scalars showing "how upwards facing" each triangle is.
```

(continues on next page)

(continued from previous page)

```
tri_scalars = np.inner(mesh.units, np.array([0, 0, 1]))

vpl.mesh_plot(mesh, tri_scalars=tri_scalars)

vpl.show()
```

Note: *scalars* and *tri_scalars* overwrite each other and can't be used simultaneously.

See also:

Having per-triangle-edge scalars doesn't fit well with VTK. So it got its own seperate function `vpl.mesh_plot_with_edge_scalar`.

Figure managing:

There are two main basic types in vtkplotlib.

- Figures are the window you plot into.
- Plots are the physical objects that go in the figures.

In all the previous examples the figure has been handled automatically. For more complex scenarios you may need to handle the figures yourself. The following demonstrates the figure handling functions.

```
import vtkplotlib as vpl
import numpy as np

# You can create a figure explicitly using figure()
fig = vpl.figure("Your Figure Title Here")

# Creating a figure automatically sets it as the current working figure
# You can get the current figure using gcf()
vpl.gcf() is fig # Should be True

# If a figure hadn't been explicitly created using figure() then gcf()
# would have created one. If gcf() had also not been called here then
# the plotting further down will have internally called gcf().

# A figure's properties can be edited directly
fig.background_color = "dark green"
fig.window_name = "A New Window Title"


points = np.random.uniform(-10, 10, (2, 3))

# To add to a figure you can either:

# 1) Let it automatically add to the whichever figure gcf() returns
vpl.scatter(points[0], color="r")

# 2) Explicitly give it a figure to add to
vpl.scatter(points[1], radius=2, fig=fig)

# 3) Or pass fig=None to prevent it being added then add it later
arrow = vpl.arrow(points[0], points[1], color="g", fig=None)
```

(continues on next page)

(continued from previous page)

```
fig += arrow
# fig.add_plot(arrow) also does the same thing


# Finally when your ready to view the plot call show. Like before you can
# do this one of several ways
# 1) fig.show()
# 2) vpl.show() # equivalent to gcf().show()
# 3) vpl.show(fig=fig)

fig.show() # The program will wait here until the user closes the window.


# Once a figure is shown it is gets placed in `vpl.figure_history` which
# stores recent figures. The default maximum number of figures is two. For
# convenience whilst console bashing, you can retrieve the last figure.
# But it will no longer be the current working figure.

vpl.figure_history[-1] is fig # Should be True
fig is vpl.gcf() # Should be False

# A figure can be reshown indefinitely and should be exactly as you left it
# when it was closed.
fig.show()
```

vtkplotlib Plots

vtkplotlib.scatter

vtkplotlib.plots.Scatter.**scatter**(*points*, *color=None*, *opacity=None*, *radius=1.0*, *use_cursors=False*, *fig='gcf'*, *label=None*)

Scatter plot using little spheres or cursor objects.

Parameters

- **points** (np.array with `points.shape[-1] == 3`) – The point(s) to place the marker(s) at.
- **color** (str, 3-tuple, 4-tuple, np.array with same shape as *points*, optional) – The color of the markers, can be singular or per marker, defaults to white.
- **opacity** (*float, np.array, optional*) – The translucencies of the plots, 0 is invisible, 1 is solid, defaults to solid.
- **radius** (*float, np.array, optional*) – The radius of each marker, defaults to 1.0.
- **use_cursors** (*bool, optional*) – If false use spheres, if true use cursors, defaults to False.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns The marker or an array of markers.

Return type vtkplotlib.plots.Scatter.Sphere or vtkplotlib.plots.Scatter.Cursor or np.array

vtkplotlib.plot

class vtkplotlib.plots.Lines.**Lines**(*vertices*, *color=None*, *opacity=None*, *line_width=1.0*, *join_ends=False*, *cmap=None*, *fig='gcf'*, *label=None*)

Plots a line passing through an array of points.

Parameters

- **vertices** (*np.ndarray of shape (n, 3)*) – The points to plot through.
- **color** (*str, 3-tuple, 4-tuple, np.ndarray optional*) – The color(s) of the lines, defaults to white.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **line_width** (*float, optional*) – The thickness of the lines, defaults to 1.0.
- **join_ends** (*bool, optional*) – If true, join the 1st and last points to form a closed loop, defaults to False.
- **cmap** (*str, 2D np.ndarray, matplotlib colormap, vtkLookupTable, optional*) – Colormap to use for scalars, defaults to *rainbow*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns A lines object.

Return type *vtkplotlib.plots.Lines.Lines*

If *vertices* is 3D then multiple seperate lines are plotted. This can be used to plot meshes as wireframes.

```
import vtkplotlib as vpl
from stl.mesh import Mesh

mesh = Mesh.from_file(vpl.data.get_rabbit_stl())
vertices = mesh.vectors

vpl.plot(vertices, join_ends=True, color="dark red")
vpl.show()
```

If *color* is an *np.ndarray* then a color per vertex is implied. The shape of *color* relative to the shape of *vertices* determines whether the colors should be interpreted as scalars, texture coordinates or RGB values. If *color* is either a list, tuple, or str then it is one color for the whole plot.

```
import vtkplotlib as vpl
import numpy as np

# Create an octagon, using `t` as scalar values.

t = np.arange(0, 1, .125) * 2 * np.pi
vertices = vpl.zip_axes(np.cos(t),
                        np.sin(t),
```

(continues on next page)

(continued from previous page)

```
                        0)

# Plot the octagon.
vpl.plot(vertices,
         line_width=6,    # use a chunky (6pt) line
         join_ends=True, # join the first and last points
         color=t,        # use `t` as scalar values to color it
         )

# use a dark background for contrast
fig = vpl.gcf()
fig.background_color = "grey"

vpl.show()
```

vtkplotlib.mesh_plot

class vtkplotlib.plots.MeshPlot.**MeshPlot**(*mesh_data*, *tri_scalars=None*, *scalars=None*, *color=None*, *opacity=None*, *cmap=None*, *fig='gcf'*, *label=None*)

To plot STL files you will need some kind of STL reader library. If you don't have one then get this one numpy-stl. Their Mesh class can be passed directly to vpl.mesh_plot.

Parameters

- **mesh_data** (*An STL (like) object (see below)*) – The mesh to plot.
- **tri_scalars** (*np.ndarray, optional*) – Per-triangle scalar, texture-coordinates or RGB values, defaults to None.
- **scalars** (*np.ndarray, optional*) – Per-vertex scalar, texture-coordinates or RGB values, defaults to None.
- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of the whole plot, ignored if scalars are used, defaults to white.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **cmap** (*str, 2D np.ndarray, matplotlib colormap, vtkLookupTable, optional*) – Colormap to use for scalars, defaults to *rainbow*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns A meshplot object.

Return type *vtkplotlib.plots.MeshPlot.MeshPlot*

The following example assumes you have installed numpy-stl.

```
import vtkplotlib as vpl
from stl.mesh import Mesh
```

(continues on next page)

(continued from previous page)

```
# path = "if you have an STL file then put it's path here."
# Otherwise vtkplotlib comes with a small STL file for demos/testing.
path = vpl.data.get_rabbit_stl()

# Read the STL using numpy-stl
mesh = Mesh.from_file(path)

# Plot the mesh
vpl.mesh_plot(mesh)

# Show the figure
vpl.show()
```

Unfortunately there are far too many mesh/STL libraries/classes out there to support them all. To overcome this as best we can, mesh_plot has a flexible constructor which accepts any of the following.

1. Some kind of mesh class that has form 2) stored in mesh.vectors. For example numpy-stl's stl.mesh.Mesh or pymesh's pymesh.stl.Stl
2. An np.array with shape (n, 3, 3) in the form:

```
np.array([[[x, y, z],  # corner 0                          [x, y, z],  #␣
↪corner 1  | triangle 0
          [x, y, z]], # corner 2  /
         ...
          [[x, y, z],  # corner 0                          [x, y, z],  #␣
↪corner 1  | triangle n-1
          [x, y, z]], # corner 2  /
         ])
```

   Note it's not uncommon to have arrays of shape (n, 3, 4) or (n, 4, 3) where the additional entries' meanings are usually irrelevant (often to represent scalars but as STL has no color this is always uniform). Hence to support mesh classes that have these, these arrays are allowed and the extra entries are ignored.

3. An np.array with shape (k, 3) of (usually unique) vertices in the form:

```
np.array([[x, y, z],
          [x, y, z],
          ...
          [x, y, z],
          [x, y, z],
          ])
```

   And a second argument of an np.array of integers with shape (n, 3) of point args in the form

```
np.array([[i, j, k],  # triangle 0
          ...
          [i, j, k],  # triangle n-1
          ])
```

   where i, j, k are the indices of the points (in the vertices array) representing each corner of a triangle.

   Note that this form can be (and is) easily converted to form 2) using

```
vertices = unique_vertices[point_args]
```

Hopefully this will cover most of the cases. If you are using or have written an STL library (or any other format) that you want supported then let me know. If it's numpy based then it's probably only a few extra lines to support. Or you can have a go at writing it yourself, either *with mesh_plot* or with the `vpl.PolyData` class.

Mesh plotting with scalars:

To create a heat map like image use the *scalars* or *tri_scalars* options.

Use the *scalars* option to assign a scalar value to each point/corner:

```
import vtkplotlib as vpl
from stl.mesh import Mesh

# Open an STL as before
path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

# Plot it with the z values as the scalars. scalars is 'per vertex' or 1
# value for each corner of each triangle and should have shape (n, 3).
plot = vpl.mesh_plot(mesh, scalars=mesh.z)

# Optionally the plot created by mesh_plot can be passed to color_bar
vpl.color_bar(plot, "Heights")

vpl.show()
```

Use the *tri_scalars* option to assign a scalar value to each triangle:

```
import vtkplotlib as vpl
from stl.mesh import Mesh
import numpy as np

# Open an STL as before
path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

# `tri_scalars` must have one value per triangle and have shape (n,) or (n, 1).
# Create some scalars showing "how upwards facing" each triangle is.
tri_scalars = np.inner(mesh.units, np.array([0, 0, 1]))

vpl.mesh_plot(mesh, tri_scalars=tri_scalars)

vpl.show()
```

Note: *scalars* and *tri_scalars* overwrite each other and can't be used simultaneously.

See also:

Having per-triangle-edge scalars doesn't fit well with VTK. So it got its own seperate function `vpl.mesh_plot_with_edge_scalar`.

---

vtkplotlib.mesh_plot_with_edge_scalars

vtkplotlib.plots.MeshPlot.**mesh_plot_with_edge_scalars**(*mesh_data*, *edge_scalars*, *centre_scalar='mean'*, *opacity=None*, *cmap=None*, *fig='gcf'*, *label=None*)

Like mesh_plot but able to add scalars per triangle's edge. By default, the scalar value at centre of each triangle is taken to be the mean of the scalars of its edges, but it can be far more visually effective to use the `centre_scalar=` number option.

Parameters

- **mesh_data** (*An STL (like) object (see below)*) – The mesh to plot.
- **edge_scalars** (*np.ndarray*) – Per-edge scalar, texture-coordinates or RGB values.
- **centre_scalar** (*str, optional*) – Scalar value(s) for the centre of each triangle, defaults to 'mean'.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **cmap** (*str, 2D np.ndarray, matplotlib colormap, vtkLookupTable, optional*) – Colormap to use for scalars, defaults to *rainbow*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns A meshplot object.

Return type *vtkplotlib.plots.MeshPlot.MeshPlot*

Edge scalars are very much not the way VTK likes it. Infact VTK doesn't allow it. To overcome this, this function triple-ises each triangle. See the diagram below to see how this is done:

```
(The diagrams's tacky, I know)

           p1

         //|\\          Double lines represent the original triangle.
        // | \\         The single lines represent the division lines that
  l0   //  |  \\  l1    split the triangle into three.
      //  / \  \\       The annotations show the order in which the
     // /     \ \\      scalar for each edge must be provided.
    ///~~~~~~~~~\\\
p0  ~~~~~~~~~~~~~~~  p2
          l2

(reST doesn't like it either)
```

Here is a usage example:

```
import vtkplotlib as vpl
from vtkplotlib import geometry
from stl.mesh import Mesh
import numpy as np
```

(continues on next page)

(continued from previous page)

```
path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

# This is the length of each side of each triangle.
edge_scalars = geometry.distance(mesh.vectors[:, np.arange(1, 4) % 3] - mesh.
↪vectors)

vpl.mesh_plot_with_edge_scalars(mesh, edge_scalars, centre_scalar=0, cmap="Greens
↪")

vpl.show()
```

I wrote this orginally to visualise curvature. The calculation is ugly, but on the off chance someone needs it, here it is.

```
import vtkplotlib as vpl
from vtkplotlib import geometry
from stl.mesh import Mesh
import numpy as np


path = vpl.data.get_rabbit_stl()
mesh = Mesh.from_file(path)

def astype(arr, dtype):
    return np.frombuffer(arr.tobytes(), dtype)

def build_tri2tri_map(mesh):
    """This creates an (n, 3) array that maps each triangle to its 3
    adjacent triangles. It takes advantage of each triangles vertices
    being consistently ordered anti-clockwise. If triangle A shares an
    edge with triangle B then both A and B have the edges ends as
    vertices but in opposite order. Looking for this helps reduce the
    complexity of the problem.
    """

    # The most efficient way to make a pair of points hashable is to
    # take its binary representation.
    dtype = np.array(mesh.vectors[0, :2].tobytes()).dtype

    vectors_rolled = mesh.vectors[:, np.arange(1, 4) % 3]

    # Get all point pairs going one way round.
    pairs = np.concatenate((mesh.vectors, vectors_rolled), -1)

    # Get all point pairs going the other way round.
    pairs_rev = np.concatenate((vectors_rolled, mesh.vectors), -1)

    bin_pairs = astype(pairs, dtype).reshape(-1, 3)
    bin_pairs_rev = astype(pairs_rev, dtype).reshape(-1, 3)

    # Use a dictionary to find all the matching pairs.
    mapp = dict(zip(bin_pairs.ravel(), np.arange(bin_pairs.size) // 3))
    args = np.fromiter(map(mapp.get, bin_pairs_rev.flat), dtype=float, count=bin_
↪pairs.size).reshape(-1, 3)
```

(continues on next page)

(continued from previous page)

```
    # Triangles with a missing adjacent edge come out as nans.
    # Convert mapping to ints and nans to -1s.
    mask = np.isfinite(args)
    tri2tri_map = np.empty(args.shape, int)
    tri2tri_map[mask] = args[mask]
    tri2tri_map[~mask] = -1

    return tri2tri_map


tri2tri_map = build_tri2tri_map(mesh)

tri_centres = np.mean(mesh.vectors, axis=1)
curves = np.cross(mesh.units[tri2tri_map], mesh.units[:, np.newaxis])
displacements = tri_centres[tri2tri_map] - tri_centres[:, np.newaxis]
curvatures = curves / geometry.distance(displacements, keepdims=True)

curvature_signs = np.sign(geometry.inner_product(mesh.units[:, np.newaxis],
                                                 displacements)) * -1

signed_curvatures = geometry.distance(curvatures) * curvature_signs

# And finally, to plot it.
plot = vpl.mesh_plot_with_edge_scalars(mesh, signed_curvatures)

# Curvature must be clipped to prevent anomalies overwidenning the
# scalar range.
plot.scalar_range = -.1, .1

# Red represents an inside corner, blue represents an outside corner.
plot.cmap = "coolwarm_r"

vpl.show()
```

vtkplotlib.polygon

class vtkplotlib.plots.Polygon.**Polygon**(*vertices*, *scalars=None*, *color=None*, *opacity=None*, *fig='gcf'*, *label=None*)

Creates a filled polygon(s) with *vertices* as it's corners. For a 3 dimensional *vertices* array, each 2d array within *vertices* is a seperate polygon.

Parameters

- **vertices** (*np.ndarray with shape ([number_of_polygons,] points_per_polygon, 3)*) – Each corner of each polygon.
- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of whole the plot, defaults to white.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().

- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns A polygon object

Return type *vtkplotlib.plots.Polygon.Polygon*

VTK renders everything as only triangles. Polygons with more than 3 sides are broken down by VTK into multiple triangles. For non-flat polygons with many sides, the fragmentation doesn't look too great.

---

vtkplotlib.scalar_bar

class vtkplotlib.plots.ScalarBar.**ScalarBar**(*plot*, *title=''*, *fig='gcf'*)

Create a scalar bar. Also goes by the alias *colorbar*.

Parameters

- **plot** – The plot with scalars to draw a scalarbar for.
- **title** (*str, optional*) – , defaults to ''.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().

Returns The scalarbar object.

Return type *vtkplotlib.plots.ScalarBar.ScalarBar*

The *plot* argument can be the output of any `vpl.***` command that takes *scalars* as an argument.

---

vtkplotlib.arrow

vtkplotlib.plots.Arrow.**arrow**(*start*, *end*, *length=None*, *width_scale=1.0*, *color=None*, *opacity=None*, *fig='gcf'*, *label=None*)

Draw (an) arrow(s) from *start* to *end*.

Parameters

- **start** (*np.ndarray*) – The starting point(s) of the arrow(s).
- **end** (*np.ndarray*) – The end point(s) of the arrow(s).
- **length** (*number, np.ndarray, optional*) – The length of the arrow(s), defaults to None.
- **width_scale** (*number, np.ndarray, optional*) – How fat to make each arrow, is relative to its length, defaults to 1.0.
- **color** (*str, 3-tuple, 4-tuple, np.ndarray of shape(n, 3)*) – The color of each arrow, defaults to white.
- **opacity** (*float*) – The translucency of each arrow, 0 is invisible, 1 is solid, defaults to solid.
- **fig** (*vpl.figure, vpl.QtFigure*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns arrow or array of arrows

Return type vtkplotlib.plots.Arrow.Arrow, np.array of Arrows

The shapes of *start* and *end* should match. Arrow lengths are automatically calculated via pythagoras if not provided but can be overwritten by setting *length*. In this case the arrow(s) will always start at *start* but may not end at *end*. *length* can either be a single value for all arrows or an array of lengths to match the number of arrows.

Note: Arrays are supported only for convenience and just use a python for loop. There is no speed bonus to using numpy or trying to plot in bulk here.

See also:

`vpl.quiver` for field plots.

---

vtkplotlib.quiver

`vtkplotlib.plots.Arrow.`**`quiver`**(*point*, *gradient*, *length=None*, *length_scale=1.0*, *width_scale=1.0*, *color=None*, *opacity=None*, *fig='gcf'*, *label=None*)

Create arrow(s) from 'point' towards a direction given by 'gradient' to make field/quiver plots. Arrow lengths by default are the magnitude of 'gradient but can be scaled with 'length_scale' or frozen with 'length'. See arrow's docs for more detail.

Parameters

- **`point`** (*`np.ndarray`*) – The starting point of the arrow(s).
- **`gradient`** (*`np.ndarray`*) – The displacement / gradient vector.
- **`length`** (*`NoneType, optional`*) – A frozen length for each arrow, defaults to None.
- **`length_scale`** (*`int, optional`*) – A scaling factor for the length of each arrow, defaults to 1.0.
- **`width_scale`** (*`int, optional`*) – How fat to make each arrow, is relative to its length, defaults to 1.0.
- **`color`** (*`str, 3-tuple, 4-tuple, np.ndarray of shape(n, 3)`*) – The color of each arrow, defaults to white.
- **`opacity`** (*`float`*) – The translucency of each arrow, 0 is invisible, 1 is solid, defaults to solid.
- **`fig`** (*`vpl.figure, vpl.QtFigure`*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **`label`** (*`str, optional`*) – Give the plot a label to use in legends, defaults to None.

Returns arrow or array of arrows

Return type vtkplotlib.plots.Arrow.Arrow, np.array of Arrows

See also:

`vpl.arrow` to draw arrows from a start point to an end point.

---

vtkplotlib.text

class vtkplotlib.plots.Text.**Text**(*text_str*, *position=(0, 0)*, *fontsize=18*, *use_pixels=False*, *color=(1, 1, 1)*, *opacity=None*, *fig='gcf'*)

2D text at a fixed point on the window (independent of camera position / orientation).

Parameters

- **text_str** (*str, object*) – The text, converts to string if not one already.
- **position** (*2-tuple of ints, optional*) – The (x, y) position in pixels on the screen, defaults to (0, 0) (left, bottom).
- **fontsize** (*int, optional*) – Text height (ignoring tails) in pixels, defaults to 18.
- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of the text, defaults to white.
- **opacity** (*float, optional*) – The translucency of the text, 0 is invisible, 1 is solid, defaults to solid.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().

Returns The text plot object.

Return type *vtkplotlib.plots.Text.Text*

The text doesn't resize or reposition itself when the window is resized. It's on the todo list.

See also:

vpl.text3D

---

vtkplotlib.text3d

class vtkplotlib.plots.Text3D.**Text3D**(*text*, *position=(0, 0, 0)*, *follow_cam=True*, *scale=1*, *color=None*, *opacity=None*, *fig='gcf'*, *label=None*)

Create floating text in 3D space. Optionally can be set to orientate itself to follow the camera (defaults to on) with the *follow_cam* argument.

Parameters

- **string** – The text to be shown.
- **position** (*tuple, optional*) – The position of the start of the text, defaults to (0, 0, 0).
- **follow_cam** (*bool, optional*) – Automatically rotates to follow the camera, defaults to True.
- **scale** (*number or 3-tuple of numbers, optional*) – The height of one line of text, can have 3 values, defaults to 1.0.
- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of the text, defaults to white.
- **opacity** (*float, optional*) – The translucency of the text, 0 is invisible, 1 is solid, defaults to solid.

- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns text3D plot object

Return type *vtkplotlib.plots.Text3D.Text3D*

Warning: This can't be passed about between figures if `follow_cam=True` (the default). The figure who's camera it follows is frozen to the figure given to it on first construction.

See also:

`vpl.text` for 2D text at a fixed point on the screen.

See also:

`vpl.annotate` for a convenient way to label features with text and an arrow.

---

vtkplotlib.annotate

vtkplotlib.plots.Text3D.**annotate**(*points*, *text*, *direction*, *text_color='w'*, *arrow_color='k'*, *distance=3.0*, *text_size=1.0*, *fig='gcf'*)

Annotate a feature with an arrow pointing at a point and a text label on the reverse end of the arrow. This is just a convenience call to `vpl.arrow` and `vpl.text3d`. See there for just one or the other.

Parameters

- **points** (*np.ndarray*) – The position of the feature where the arrow's tip should be.
- **text** – The text to put in the label.
- **direction** (*np.ndarray with shape (3,)*) – The direction from the feature to the text position as a unit vector.
- **text_color** (*optional*) – The color of the label, defaults to 'w'.
- **arrow_color** (*optional*) – The color of the arrow, defaults to 'k'.
- **distance** (*number, optional*) – The distance from the feature to the label, defaults to 3.0.
- **text_size** (*number or 3-tuple of numbers, optional*) – The height of one line of text, can have 3 values, defaults to 1.0.
- **fig** (*vpl.figure, vpl.QtFigure*) – The figure to plot into, can be None, defaults to vpl.gcf().

Returns (arrow, text) 2-tuple

Return type (Arrow, *Text3D*)

The arrow points to the highest point and the text is placed at a point *distance* above (where above also is determined by direction).

If *text* is not a str then it is automatically converted to one.

```
import vtkplotlib as vpl
import numpy as np

# Create a ball at a point in space.
point = np.array([1, 2, 3])
vpl.scatter(point)

vpl.annotate(point,
             "This ball is at {}".format(point),
             np.array([0, 0, 1]))
vpl.show()
```

If multiple points are given the farthest in the direction *direction* is selected. The idea is to try to prevent the annotations ending up in amongst the plots or, when plotting meshes, inside the mesh.

```
import vtkplotlib as vpl
import numpy as np

# Create several balls.
points = np.random.uniform(-30, 30, (30, 3))
vpl.scatter(points, color=np.random.random(points.shape))

vpl.annotate(points,
             "This ball is the highest",
             np.array([0, 0, 1]),
             text_color="k",
             arrow_color="orange"
             )

vpl.annotate(points,
             "This ball is the lowest",
             np.array([0, 0, -1]),
             text_color="rust",
             arrow_color="hunter green"
             )

vpl.show()
```

vtkplotlib.surface

class vtkplotlib.plots.Surface.**Surface**(*x*, *y*, *z*, *scalars=None*, *color=None*, *opacity=None*, *texture_map=None*, *cmap=None*, *fig='gcf'*, *label=None*)

Create a parametrically defined surface. This is intended for visualising mathematical functions of the form

$$z = f(x, y)$$

or

$$x = f(\phi, \theta) \quad y = g(\phi, \theta) \quad z = h(\phi, \theta)$$

See also:

`vtkplotlib.mesh_plot` for a surface made out of triangles

See also:

`vtkplotlib.polygon` for a surface made out of polygons.

Parameters

- **x** (*2D np.ndarray with shape (m, n)*) – x components.
- **y** (*2D np.ndarray with shape (m, n)*) – y components.
- **z** (*2D np.ndarray with shape (m, n)*) – z components.
- **scalars** (*np.ndarray with shape (m, n [, 1 or 2 or 3]) NoneType, optional*) – per-point scalars / texturemap coordinates / RGB colors, defaults to None.
- **color** (*str, 3-tuple, 4-tuple, optional*) – The color of the plot, defaults to white.
- **opacity** (*float, optional*) – The translucency of the plot, 0 is invisible, 1 is solid, defaults to solid.
- **cmap** (*str, 2D np.ndarray, matplotlib colormap, vtkLookupTable, optional*) – Colormap to use for scalars, defaults to *rainbow*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **label** (*str, optional*) – Give the plot a label to use in legends, defaults to None.

Returns The surface object.

Return type *vtkplotlib.plots.Surface.Surface*

This is the only function in *vtkplotlib* that takes it's (x, y, z) components as seperate arguments. `x`, `y` and `z` should be 2D arrays with matching shapes. This is typically achieved by using `phi, theta = np.meshgrid(phis, thetas)` then calculating x, y and z from `phi` and `theta`. Here is a rather unexciting example.

```
import vtkplotlib as vpl
import numpy as np

phi, theta = np.meshgrid(np.linspace(0, 2 * np.pi, 1024),
                         np.linspace(0, np.pi, 1024))


x = np.cos(phi) * np.sin(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(theta)

vpl.surface(x, y, z)

vpl.show()
```

See also:

A parametrically constructed object plays well with a TextureMap.

vtkplotlib.PolyData

class vtkplotlib.plots.polydata.**PolyData**(*vtk_polydata=None*, *mapper=None*)

The polydata is a key building block to making customised plot objects. A lot of vtkplotlib plot commands use one of these under the hood.

Parameters **vtk_polydata** (*vtk.vtkPolyData, optional*) – An original vtkPolyData to build on top of, defaults to None.

This is a wrapper around VTK's vtkPolyData object. This class is incomplete. I still need to sort colors/scalars properly. And I want to add functions to build from vtkSource objects.

A polydata consists of:

1. points
2. lines
3. polygons
4. scalar, texturemap coordinates or direct color information

or combinations of the four.

Parameters

- **self.points** (np.ndarray of floats with shape (*number_of_vertices*, *3*)) – All the vertices used for all points, lines and polygons. These points aren't visible. To create some kind of points plot use vpl.scatter.
- **self.lines** (np.ndarray of ints with shape (*number_of_lines*, *points_per_line*)) – The arg of each vertex from *self.points* the line should pass through. Each row represents a seperate line.
- **self.polygons** (np.ndarray of ints with shape (*number_of_polygons*, *corners_per_polygon*)) – Each row represents a polygon. Each cell contains the arg of a vertex from *self.points* that is to be a corner of that polygon.

Lines and polyons can be interchanged to switch from solid surface to wire-frame.

Here is an example to create a single triangle

```
import vtkplotlib as vpl
import numpy as np


polydata = vpl.PolyData()

polydata.points = np.array([[1, 0, 0],
                            [0, 1, 0],
                            [0, 0, 1]], float)

# Create a wire-frame triangle passing points [0, 1, 2, 0].
polydata.lines = np.array([[0, 1, 2, 0]])

# Create a solid triangle with points [0, 1, 2] as it's corners.
polydata.polygons = np.array([[0, 1, 2]])

# The polydata can be quickly inspected using
polydata.quick_show()

# When you are happy with it, it can be turned into a proper plot
```

(continues on next page)

(continued from previous page)

```
# object like those output from other ``vpl.***()`` commands. It will be
# automatically added to ``vpl.gcf()`` unless told otherwise.
plot = polydata.to_plot()
vpl.show()
```

The original vtkPolyData object is difficult to use, can't directly work with numpy and is full of pot-holes that can cause unexplained crashes if not carefully avoided. This wrapper class seeks to solve those issues by acting as an intermediate layer between you and VTK. This class consists mainly of properties that

- handle the numpy-vtk conversions
- ensure proper shape checking
- hides VTK's rather awkward compound array structures
- automatically sets scalar mode/range parameters in the mapper

A *vpl.PolyData* can be constructed from scratch or from an existing *vtkPolyData* object.

It also provides convenience methods `self.to_plot()` and `self.quick_show()` for quick one-line visualising the current state.

---

vtkplotlib.legend

class vtkplotlib.plots.Legend.**Legend**(*plots_source='fig'*, *fig='gcf'*, *position=(0.7, 0.7)*, *size=(0.3, 0.3)*, *color='grey'*, *opacity=None*, *allow_non_polydata_plots=False*, *allow_no_label=False*)

Creates a legend to label plots.

Parameters

- **plots_source** (*iterable of plots or None, optional*) – Plots to use in the legend, can be None, defaults to *fig.plots*.
- **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().
- **position** (*tuple pair of floats, optional*) – Position (relative to the size of the figure) of the bottom left corner, defaults to (0.7, 0.7).
- **size** (*tuple pair of floats, optional*) – Size (relative to the size of the figure) of the legend, defaults to (0.3, 0.3).
- **color** (*str, 3-tuple, 4-tuple, optional*) – The background color of the legend, defaults to 'grey'.
- **allow_no_label** (*bool, optional*) – Allow plots that have no label to have label None, only applicable if entries are added automatically, defaults to False.
- **allow_non_polydata_plots** (*bool, optional*) – Allow plots that have no polydata to represented with a box, only applicable if entries are added automatically, defaults to False.

Returns The legend created.

Return type *vtkplotlib.plots.Legend.Legend*

Elements can be added to the legend automatically or explicitly. Most plot commands have an optional *label* argument. If this is used then they will be added automatically. Multiple plots with the same label will be grouped.

```
import vtkplotlib as vpl
import numpy as np

# Create some plots with labels
vpl.scatter([0, 0, 0], color="red", label="Red Ball")
vpl.scatter(vpl.zip_axes(5, np.arange(0, 10, 4), 0), color="yellow", label=
↪"Yellow Ball")

# A plot without a label will not be included in the legend.
vpl.scatter([10, 0, 0], color="green")

# Create the legend
vpl.legend()
# Unless told otherwise this internally calls
# legend.add_plots(vpl.gcf().plots)

vpl.show()
```

The legend uses a tabular format.

Symbol	Icon	Text Label
Symbol	Icon	Text Label
Symbol	Icon	Text Label

- A *symbol* is a 2D representation of the original plot. VTK can generate these flattened snapshots from a polydata object which most plots either have or can generate. These are accessible via `plot.polydata`.
- An *icon* is just a 2D image. Note that VTK only supports greyscale icons in legends.
- The *text label*, as the name suggests, is just a piece of text.

All three columns are optional. If a column is never used throughout the legend then the contents adjusts to close the space. Color can only be applied per-row. i.e the symbol, icon and text of an entry are always the same color.

The following example shows how to set the entries explicitly.

```
import vtkplotlib as vpl

# Create a legend and don't allow it to fill itself.
legend = vpl.legend(plots_source=None)


# A labelled plot contains all the information it needs to add itself.
sphere = vpl.scatter([0, 5, 10], color="g", label="Green Ball")
# Add it like so.
legend.set_entry(sphere)

# Written explicitly the above is equivalent to:
# legend.set_entry(symbol=sphere.polydata, color=sphere.color, label=sphere.label)


# Not all plots can have a polydata. If one isn't provided then a by
```

(continues on next page)

(continued from previous page)

```
# default a square is used.
legend.set_entry(label="Blue Square", color="blue")

# Alternatively, if explicitly given ``symbol=None``, then the symbol
# space is left blank.
legend.set_entry(symbol=None, label="Just Text")


# Most plots can be used.
legend.set_entry(vpl.mesh_plot(vpl.data.get_rabbit_stl()), label="Rabbit")


# To use an icon, pass a string path, array, PIL image or vtkImageData
# to the `icon` argument. The image is converted to greyscale
# automatically.
legend.set_entry(None, label="Shark", icon=vpl.data.ICONS["Right"])


vpl.show()
```

`legend.set_entry` has an optional argument *index* which can be used to overwrite rows. Otherwise it defaults to appending a row.

Some caveats / potential sources of confusion:

- Long and thin plots such as vpl.quiver tend to mess up the spacing (which) is seemingly non configurable.
- Be careful of vpl.scatter and vpl.quiver which return an array of plots rather than a single plot.
- Plots based on lines such as the output of vpl.plot tend not to show well as the lines are less than one pixel wide.
- Automatic setting of color can only work for uniformly colored plots. any colors derived from scalars are ignored.

To some extent, the text labels can be customised via `legend.text_options` which holds the vtkTextProperty (bucket class for settings like font). However, a lot of its methods do nothing. Most notebely, `legend.text_options.SetFontSize(size)` has no effect.

vtkplotlib Figures

Figures are the window that you plot into. This section outlines:

- Their creation.
- General figure management.
- Functions for controlling the camera position.
- Screenshotting the figure contents to a frozen image (file).
- Embedding a figure into PyQt5.

Overview

Some of this is handled automatically. There is a global "current working figure". This can be accessed using `vpl.gcf()`. If it doesn't exist then it is automatically created. Each plot command will add itself to the current working figure unless explicitly told not to using the *fig* option. Either use `fig=alternative_figure` to plot into a

different one or `fig=None` to not use any. The figure is shown using `vpl.show()` or `fig.show()`. After the shown figure is closed the current working figure is deleted.

---

vtkplotlib.show

vtkplotlib.figures.figure_manager.**show**(*block=True*, *fig='gcf'*)

Shows a figure. This is analogous to matplotlib's show function. After your plot commands call this to open the interactive 3D image viewer.

Parameters

- **block** (*bool, optional*) – , defaults to True.
- **fig** (*vpl.figure, vpl.QtFigure*) – The figure to show, defaults to vpl.gcf().

If 'block' is True then it enters interactive mode and the program is held until window exit. Otherwise the window is opened but not monitored. i.e an image will appear on the screen but it wont respond to being clicked on. By editing the plot and calling fig.update() you can create an animation but it will be non-interactive. True interactive animation hasn't been implemented yet - it's on the TODO list.

Note: You might feel tempted to run show in a background thread. It doesn't work. If anyone does manage it then please let me know.

Warning: A window can not be closed by the close button until it is in interactive mode. Otherwise it'll just crash python. Use vtkplotlib.close() to kill a non interactive window.

The current figure is reset on **exit** from interactive mode.

---

vtkplotlib.figure

class vtkplotlib.figures.figure.**Figure**(*name=''*)

Create a new figure. This will automatically be set as the current working figure (returned by `vpl.gcf()`).

Parameters **name** (*str, optional*) – The window title, defaults to 'vtk figure'.

---

vtkplotlib.gcf

vtkplotlib.figures.figure_manager.**gcf**(*create_new=True*)

Gets the current working figure.

Parameters **create_new** (*bool, optional*) – Allow a new one to be created if none exist, defaults to True.

Returns The current figure or None.

Return type vpl.figure, vpl.QtFigure

If none exists then a new one gets created by `vpl.figure()` unless `create_new=False` in which case None is returned. Will always return None if `auto_figure(False)` has been called.

---

vtkplotlib.scf

vtkplotlib.figures.figure_manager.**scf**(*figure*)

Sets the current working figure.

Parameters **figure** (*vpl.figure, vpl.QtFigure*) – The figure or None.

---

vtkplotlib.reset_camera

vtkplotlib.figures.figure_manager.**reset_camera**(*fig='gcf'*)

Reset the position and zoom of the camera so that all plots are visible.

Parameters **fig** (*vpl.figure, vpl.QtFigure*) – The figure, defaults to vpl.gcf().

This does not touch the orientation. It pushes the camera without rotating it so that, whichever direction it is pointing, it is pointing into the middle of where all the plots are. Then it adjusts the zoom so that everything fits on the screen.

---

vtkplotlib.save_fig

vtkplotlib.figures.figure_manager.**save_fig**(*path*, *magnification=1*, *pixels=None*, *trim_pad_width=None*, *fig='gcf'*, ***imsave_plotargs*)

Take a screenshot and saves it to a file.

Parameters

- **path** (*str or Pathlike*) – The path, including extension, to save to.
- **magnification** (*int or a (width, height) tuple of ints, optional*) – Image dimensions relative to the size of the render (window), defaults to 1.
- **pixels** (*int or a (width, height) tuple of ints, optional*) – Image dimensions in pixels, defaults to None.
- **fig** (*vpl.figure, vpl.QtFigure*) – The figure screenshot, defaults to vpl.gcf().

This just calls `vpl.screenshot_fig` then passes it to matplotlib's `pylab.imsave` function. See those for more information.

The available file formats are determined by matplotlib's choice of backend. For JPEG, you will likely need to install PILLOW. JPEG has considerably better file size than PNG.

---

vtkplotlib.view

vtkplotlib.figures.figure_manager.**view**(*focal_point=None*, *camera_position=None*, *camera_direction=None*, *up_view=None*, *fig='gcf'*)

Set/get the camera position/orientation.

Parameters

- **focal_point** (*np.array([x, y, z]), optional*) – A point the camera should point directly to, defaults to None.
- **camera_position** (*np.array([x, y, z]), optional*) – The point the camera is situated, defaults to None.
- **camera_direction** (*np.array([eX, eY, eZ]), optional*) – The direction the camera is pointing, defaults to None.
- **up_view** (*np.array([eX, eY, eZ]), optional*) – Roll the camera so that the *up_view* vector is pointing towards the top of the screen, defaults to None.
- **fig** (*vpl.figure, vpl.QtFigure*) – The figure to plot into, can be None, defaults to vpl.gcf().

Returns A dictionary containing the current configuration.

Return type dict

Note: This function sucks. You may be better off manipulating the vtk camera directly (stored in fig.camera). If you do choose this route, start experimenting by calling `print(fig.camera)`. If anyone makes a better version of this function then please share.

There is an unfortunate amount of implicit chaos going on here. Here are some hidden implications. I'm not even sure these are all true.

1. If *forwards* is used then *focal_point* and *camera_position* are ignored.
2. If *camera_position* is given but *focal_point* is not also given then *camera_position* is relative to where VTK determines is the middle of your plots. This is equivalent to setting *camera_direction* as `-camera_position`.

The following is well behaved:

```
vpl.view(camera_direction=...,
         up_view=...,)        # set orientations first
vpl.reset_camera()            # auto reset the zoom
```

---

vtkplotlib.close

vtkplotlib.figures.figure_manager.**close**(*fig='gcf'*)

Close a figure.

Parameters **fig** (*vpl.figure, vpl.QtFigure*) – The figure to close, defaults to vpl.gcf().

If the figure is the current figure then the current figure is reset.

---

vtkplotlib.figure_history

vtkplotlib._history.**figure_history = <FigureHistory deque([], maxlen=2)>**

---

vtkplotlib.auto_figure

vtkplotlib.figures.figure_manager.**auto_figure**(*auto=None*)

Enables/disables automatic figure management. If no parameters are provided then it returns the current state.

Parameters **auto** (*bool, optional*) – Defaults to None.

On by default. Disabling causes `vpl.gcf()` to always return None. Meaning that all plot commands will not add to a figure unless told to explicitly using the `fig=a_figure` argument. This can be useful for complex scenarios involving multiple figures.

---

vtkplotlib.QtFigure

class vtkplotlib.figures.QtFigure.**QtFigure**(*name='qt vtk figure', parent=None*)

The VTK render window embedded into a PyQt5 QWidget. This can be embedded into a GUI the same way all other QWidgets are used.

Parameters

- **name** (*str, optional*) – The window title of the figure, only applicable is parent is None, defaults to 'qt vtk figure'.
- **parent** (*NoneType, optional*) – Parent window, defaults to None.

Note: If you are new to Qt then this is a rather poor place to start. Whilst many libraries in Python are intuitive enough to be able to just dive straight in, Qt is not one of them. Preferably familiarise yourself with some basic Qt before coming here.

This class inherits both from QWidget and a vtkplotlib BaseFigure class. Therefore it can be used exactly the same as you would normally use either a *QWidget* or a *vpl.figure*.

Care must be taken when using Qt to ensure you have **exactly one** QApplication. To make this class quicker to use the qapp is created automatically but is wrapped in a

```
if QApplication.instance() is None:
    self.qapp = QApplication(sys.argv)
else:
    self.qapp = QApplication.instance()
```

This prevents multiple QApplication instances from being created (which causes an instant crash) whilst also preventing a QWidget from being created without a qapp (which also causes a crash).

On `self.show()`, `self.qapp.exec_()` is called automatically if `self.parent() is None` (unless specified otherwise). If the QFigure is part of a larger window then `larger_window.show()` must also show the figure. It won't begin interactive mode until `qapp.exec_()` is called.

If the figure is not to be part of a larger window then it behaves exactly like a regular figure. You just need to explicitly create it first.

```
import vtkplotlib as vpl

# Create the figure. This automatically sets itself as the current
# working figure. The qapp is created automatically if one doesn't
# already exist.
vpl.QtFigure("Exciting Window Title")

# Everything from here on should be exactly the same as normal.

vpl.quick_test_plot()

# Automatically calls ``qapp.exec_()``. If you don't want it to then
# use ``vpl.show(False)``.
vpl.show()
```

However this isn't particularly helpful. A more realistic example would require the figure be part of a larger window. In this case, treat the figure as you would any other QWidget. You must explicitly call `figure.show()` however. (Not sure why.)

```
import vtkplotlib as vpl
from PyQt5 import QtWidgets
import numpy as np
import sys

# python 2 compatibility
from builtins import super


class FigureAndButton(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        # Go for a vertical stack layout.
        vbox = QtWidgets.QVBoxLayout()
        self.setLayout(vbox)

        # Create the figure
        self.figure = vpl.QtFigure()

        # Create a button and attach a callback.
        self.button = QtWidgets.QPushButton("Make a Ball")
        self.button.released.connect(self.button_pressed_cb)

        # QtFigures are QWidgets and are added to layouts with `addWidget`
        vbox.addWidget(self.figure)
        vbox.addWidget(self.button)


    def button_pressed_cb(self):
        """Plot commands can be called in callbacks. The current working
        figure is still self.figure and will remain so until a new
        figure is created explicitly. So the ``fig=self.figure``
        arguments below aren't necessary but are recommended for
        larger, more complex scenarios.
        """

        # Randomly place a ball.
```

(continues on next page)

(continued from previous page)

```
        vpl.scatter(np.random.uniform(-30, 30, 3),
                    color=np.random.rand(3),
                    fig=self.figure)

        # Reposition the camera to better fit to the balls.
        vpl.reset_camera(self.figure)

        # Without this the figure will not redraw unless you click on it.
        self.figure.update()


    def show(self):
        # The order of these two are interchangeable.
        super().show()
        self.figure.show()




qapp = QtWidgets.QApplication.instance() or QtWidgets.QApplication(sys.argv)

window = FigureAndButton()
window.show()
qapp.exec_()
```

See also:

QtFigure2 is an extension of this to provide some standard GUI elements, ready-made.

---

vtkplotlib.QtFigure2

class vtkplotlib.figures.QtGuiFigure.**QtFigure2**(*name='qt vtk figure'*, *parent=None*)

This is intended to be used as/for a more sophisticated GUI when one is needed. By providing some common features here, hopefully this can speed up the tedious process of building a GUI. Any contributions here would be very welcome. I want to write this so that each extra feature is optional allowing custom GUIs can be built quickly.

This is still under development. Currently it has:

1. A screenshot button.
2. A panel for preset camera views.
3. An actor table to show / hide / color plots interactively (although it needs some way to group them).

I hope/intend to add:

1. Suggestions welcome here...

Use this class the same way you would use `vpl.QtFigure` (see there first.) Each feature is added with a `fig.add_***()` method.

```
import vtkplotlib as vpl
import numpy as np

# Create the figure. This as-is looks like just a QtFigure.
```

(continues on next page)

(continued from previous page)

```
fig = vpl.QtFigure2()

# Add each feature you want. Pass arguments to customise each one.
fig.add_screenshot_button(pixels=1080)
fig.add_preset_views()
fig.add_show_plot_table_button()
# Use ``fig.add_all()`` to add all them all.

# You will likely want to dump the above into a function. Or a class
# inheriting from ``vpl.QtFigure2``.

# The usual, plot something super exciting.
vpl.polygon(np.array([[1, 0, 0],
                      [1, 1, 0],
                      [0, 1, 1],
                      [0, 0, 1]]), color="grey")

# Then either ``vpl.show()`` or
fig.show()
```

vtkplotlib Extras

vtkplotlib.zip_axes

vtkplotlib.nuts_and_bolts.**zip_axes**(**axes*)

Convert vertex data from seperate arrays for x, y, z to a single combined points array like most vpl functions require.

Parameters **axes** (*array_like or scalar*) – Each seperate axis to combine.

All *axes* must have the matching or broadcastable shapes. The number of axes doesn't have to be 3.

```
import vtkplotlib as vpl
import numpy as np

vpl.zip_axes(np.arange(10),
             4,
             np.arange(-5, 5))

# Out: array([[ 0,  4, -5],
#             [ 1,  4, -4],
#             [ 2,  4, -3],
#             [ 3,  4, -2],
#             [ 4,  4, -1],
#             [ 5,  4,  0],
#             [ 6,  4,  1],
#             [ 7,  4,  2],
#             [ 8,  4,  3],
#             [ 9,  4,  4]])
```

See also:

vpl.unzip_axes for the reverse.

vtkplotlib.unzip_axes

vtkplotlib.nuts_and_bolts.**unzip_axes**(*points*)

Seperate each component from an array of points.

Parameters **points** (*np.ndarray*) – Some points.

Returns Each axis separately as a tuple.

Return type tuple of arrays

---

vtkplotlib.TextureMap

class vtkplotlib.colors.**TextureMap**(*array*, *interpolate=False*)

Use a 2D image as a color lookup table.

> **Warning:** This is still very much under development and requires a bit of monkey-wrenching to use. Currently only `vpl.surface` and `vpl.PolyData` have any support for it.

Parameters

- **array** (*str path, os.PathLike, np.ndarray with shape (m, n, 3 or 4), PIL Image*) – The image data. It is converted to an array if it isn't one already.
- **interpolate** (*bool, optional*) – , defaults to False.

Returns A callable texturemap object.

Return type vtkplotlib.colors.TextureMap

The TextureMap object can be called to look up the color at a coordinate(s). Like everything else in vktplotlib, texture coordinates should be zipped into a single array of the form:

```
np.array([[x0, y0],
          [x1, y1],
          ...,
          [xn, yn]])
```

Unlike typical images, texture-map coordinates traditionally use the conventional (x, y) axes. i.e Right is x-increasing and up is y-increasing. Indices are always between 0 and 1 and are independent of the size of the image. To use texture-coordinates, pass an array with `array.shape[-1] == 2` as the scalar argument to a plot command.

Typically texture-maps are found in some 3D file formats but integrating those is still under development. Texture-maps also play very well with parametric plots, namely `vpl.surface` using the 2 independent variables as the texture coordinates.

```
import vtkplotlib as vpl
import numpy as np

# Define the 2 independent variables
phi, theta = np.meshgrid(np.linspace(0, 2 * np.pi, 1024),
                         np.linspace(0, np.pi, 1024))
```

(continues on next page)

(continued from previous page)

```
# Calculate the x, y, z values to form a sphere
x = np.cos(phi) * np.sin(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(theta)

# You can play around with this. The coordinates must be zipped
# together into one array with ``shape[-1] == 2``, hence the
# ``vpl.zip_axes``. And must be between 0 and 1, hence the ``% 1.0``.
texture_coords = (vpl.zip_axes(phi * 3, theta * 5) / np.pi) % 1.0

# Pick an image to use. There is a picture of a shark here if you
# don't have one available.
path = vpl.data.ICONS["Right"]
texture_map = vpl.TextureMap(path, interpolate=True)


# You could convert ``texture_coords`` to ``colors`` now using.
# colors = texture_map(texture_coords)
# then pass ``colors`` as the `scalars` argument instead.

vpl.surface(x, y, z,
            scalars=texture_coords,
            texture_map=texture_map)

vpl.show()
```

vtkplotlib.quick_test_plot

vtkplotlib.**quick_test_plot**(*fig='gcf'*)

A quick laziness function to create 30 random spheres.

Parameters **fig** (*vpl.figure, vpl.QtFigure, optional*) – The figure to plot into, can be None, defaults to vpl.gcf().

```
import vtkplotlib as vpl

vpl.quick_test_plot()
vpl.show()
```

Contributing to vtkplotlib

For feedback or suggestions for either the API or the docs use the github issues page and mark it with an appropriate label.

For a bug report or to point out undesirable behaviour, again please use the issues page. Please include:

- vtkplotlib version.
- Versions of all libraries listed under *Requirements for installing* and *Optional requirements* in our quickstart page.
- Python version.
- Whether or not you use Anaconda.

- OS.
- Along with the usual, what did you do? And what did vtkplotlib do wrong?

For random queries send me an email bwoodsend@gmail.com.

To request a feature, either create an issue or just ping me at bwoodsend@gmail.com. This library has rather grown organically as I need new features for other projects. Other features are generally not included because I haven't needed them yet rather than because they would be too much hassle. There are loads of VTK classes which I haven't ported over but would be easy (couple of hours work max) from me to do so. You can search online for `vtk [feature_name] example` (it'll likely be in C++ which is OK) or rummage around in the VTK namespace to see if one exists. If it doesn't then submit an issue anyway and I'll see what I can do. If you want to be specific or just to help you can sketch out the function. Something like the following would help. It doesn't need to be properly typeset.

```
def exciting_new_feature(some, key, parameters, or, options, you, think, youll, need):
    # Some vague info about the types, desired effect, defaults of each parameter from
    # above. Doesn't have to be complete.

    # Any handy little (type) abstractions you can would like. Such as:
    if not isinstance(text_parameter, str):
        text_parameter = str(text_parameter)
    # It doesn't have to be proper code.

    # leave the implementation to me...
    # unless you have any suggestions

    return # What, if anything, should come out?
```

Or whatever you feel makes the point…

To suggest a documentation improvement, or to point out a function or component that is poorly documented or completely undocumented, submit an issue along with the url or function / class name or attribute that needs changing along with any suggested text.

To write your own feature, you are welcome to, but if it's a one off and you're not already familiar with VTK, then you're probably better off just to request it. The learning curve for VTK is steep, and the internals of vtkplotlib are pretty unorganized. But if you're determined then don't let me stop you. One day I'll get round to documenting all the core base classes used to add new features, but until then, feel free to ping me bwoodsend@gmail.com first for guidance. I have a few features half-prepped hidden away and unexposed so you may not need to start from scratch.

If you do know VTK, then you may find it easier to write a pure VTK proof of concept script and I'll handle the rest.

You can either create a fork, and then a pull request, or you can just write a separate script that runs, using VTK and/or vtkplotlib (or anything else) and I'll turn it into a vtkplotlib feature.

If you want to become a collaborator then cool - you're hired! Again, email me. You will need a fairly good understanding of VTK or be willing to learn it. I'll eventually write a proper description of the library's internals to guide you.

To make a donation, I don't need your money, but others do. If this library has helped you then please say thank you by helping those who do need it https://www.trusselltrust.org/make-a-donation/.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index