PyMOLViz

Despite its age, PyMOL is still a popular tool to display (and manipulate) molecular files. Unfortunately its great graphical capabilities are often hidden by incomplete documentation and obscure syntax. This python module is an attempt to comprehensively consolidate my understanding of these capabilities.

Setup

Requirements

- `numpy`
- `scipy`
- `matplotlib`
- `gemmi` if you want to load .mtz files

Surprisingly PyMOL is not necessarily required to run PyMOLViz, only to display the generated files.

Installation

In order to install the module you should navigate to the `module` folder and type:

```
pip install -e .
```

```
~/PyMolViz$ cd module/
~/PyMolViz/module$ pip install -e .
```

This should install the module in your current environment and automatically load any updates to the module.

Introduction

The way that PyMOLViz is currently set up, it does not interact with PyMOL directly (although this is planned to be a possibility for a future update) but rather to generate python scripts that can then be loaded into PyMOL. This has the benefit that PyMOLViz does not have to run in the same environment (nor under the same operating system or even on the same computer) as PyMOL.

Simple Example

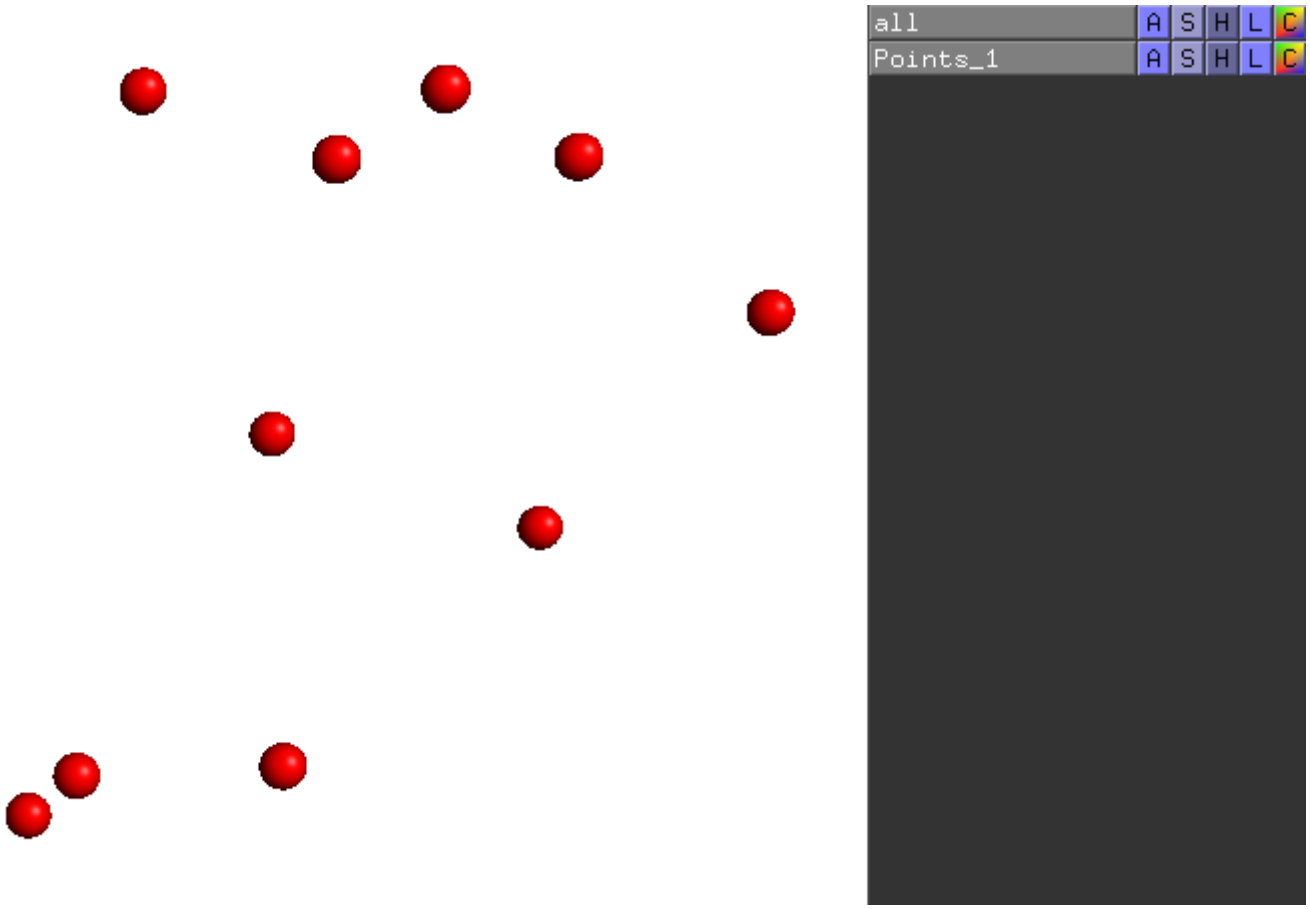
In a simple example we will create a set of points and display them in PyMOL.

```python
import numpy as np
import pymolviz as pmv

points = np.random.rand(10, 3) * 10
pmv.Points(points).write("out/points.py")
```

This generates a python script in out:

```
(base) highgarden@DESKTOP-5B09I4T:~/PyMolViz/test/00-Introduction/out$ ls
points.py
```

When we load this into PyMOL we can see 10 points:



Displayables

There are a couple of things of note here. PyMOLViz is based on so called *Displayables*. A *Displayable* is everything that gets a seperate entry in the objects side bar in PyMOL in this example case *Points_1*.

Displayables can always be written to a script via the *write* method and they always have a name. From the example we can see, that a dummy name will be generated if none is provided by the user.

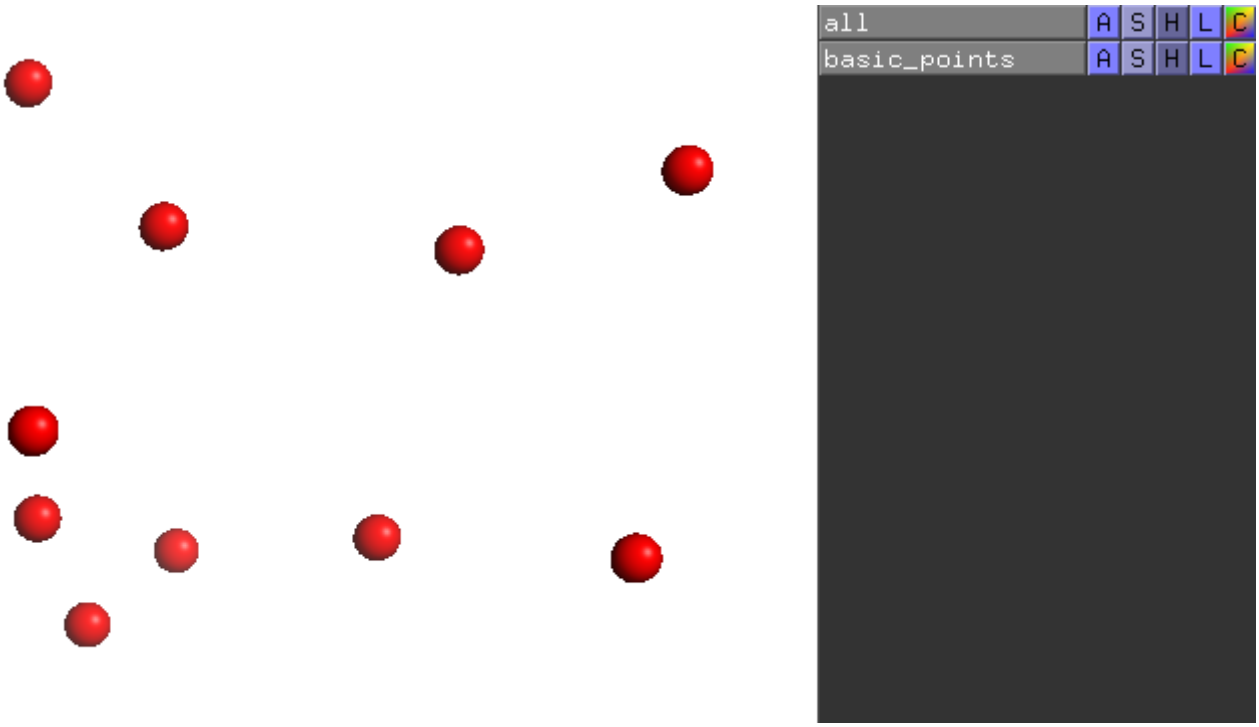
Points, Lines and Arrows

Some of the easiest to create and useful objects are points, lines and arrows (especially during debugging). These objects have the advantage over pseudo atoms or measurements that they are significantly less computationally intensive, allowing to display larger datasets.

Points

Points are defined by a set of *vertices* (positions) and *color*. As they are inheriting from *Displayables*, they also have a name and can be written to a script via *write*.

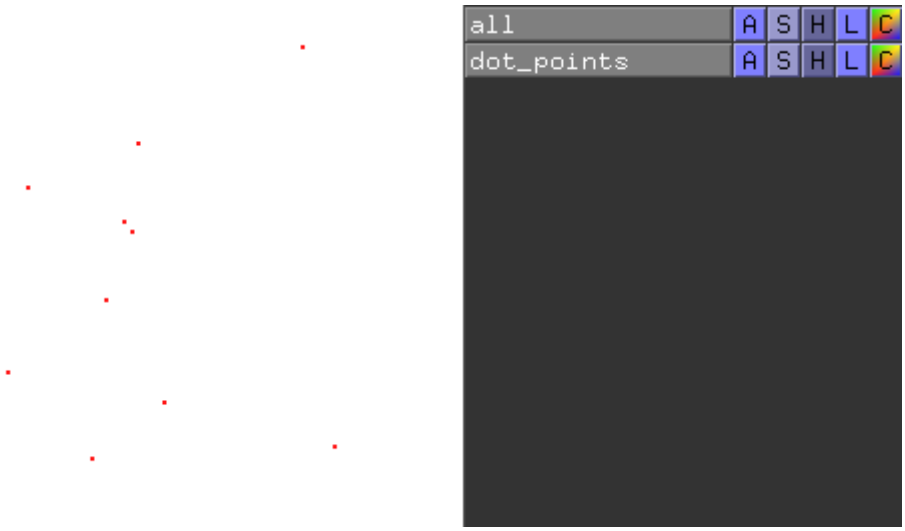
```
import numpy as np
import pymolviz as pmv

vertices = np.random.rand(10,3) * 10
p = pmv.Points(vertices, color = "red", name = "basic_points")

p.write("basic_points.py")
```



Render Types

By default, points are rendered as Spheres. Even more performant is rendering them as "Dots" (pixels):

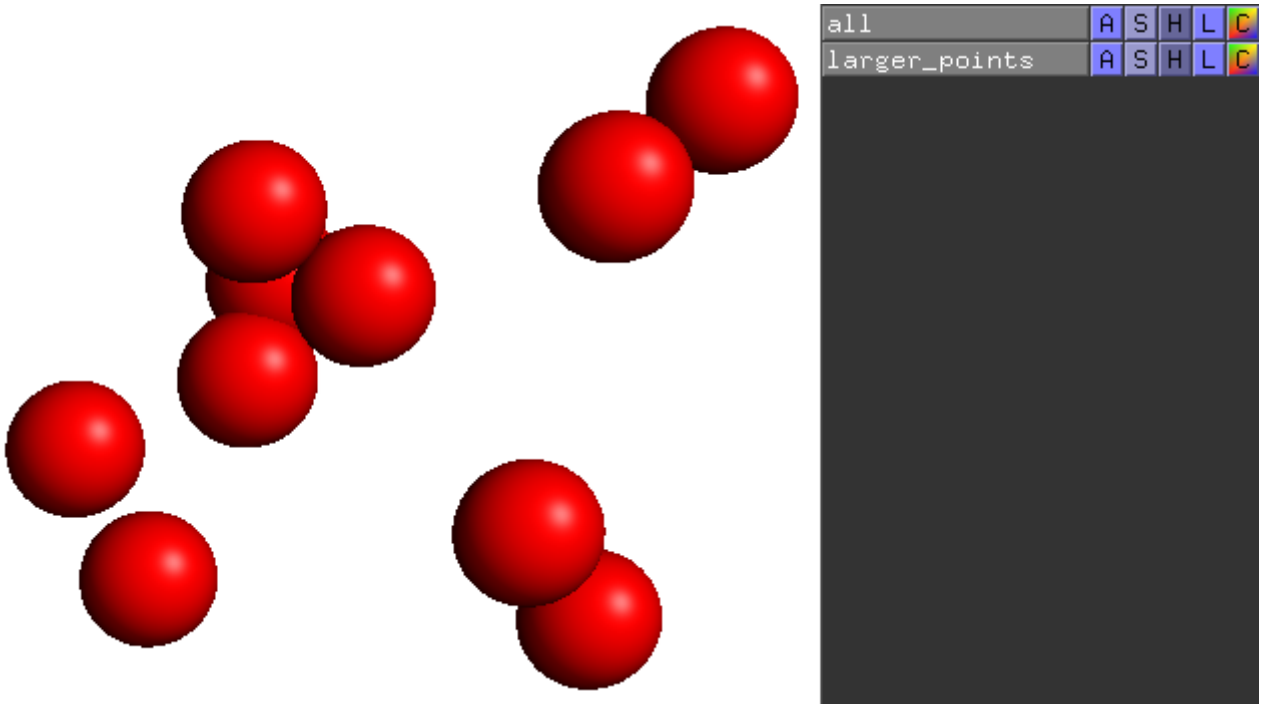
```
p2 = pmv.Points(vertices, render_as="Dots", name = "dot_points")
p2.write("out/dot_points.py")
```



When rendering as spheres you can pass the additional keyword *radius* which is 0.3 by default.

```
p3 = pmv.Points(vertices, radius = 1, name = "larger_points")
```

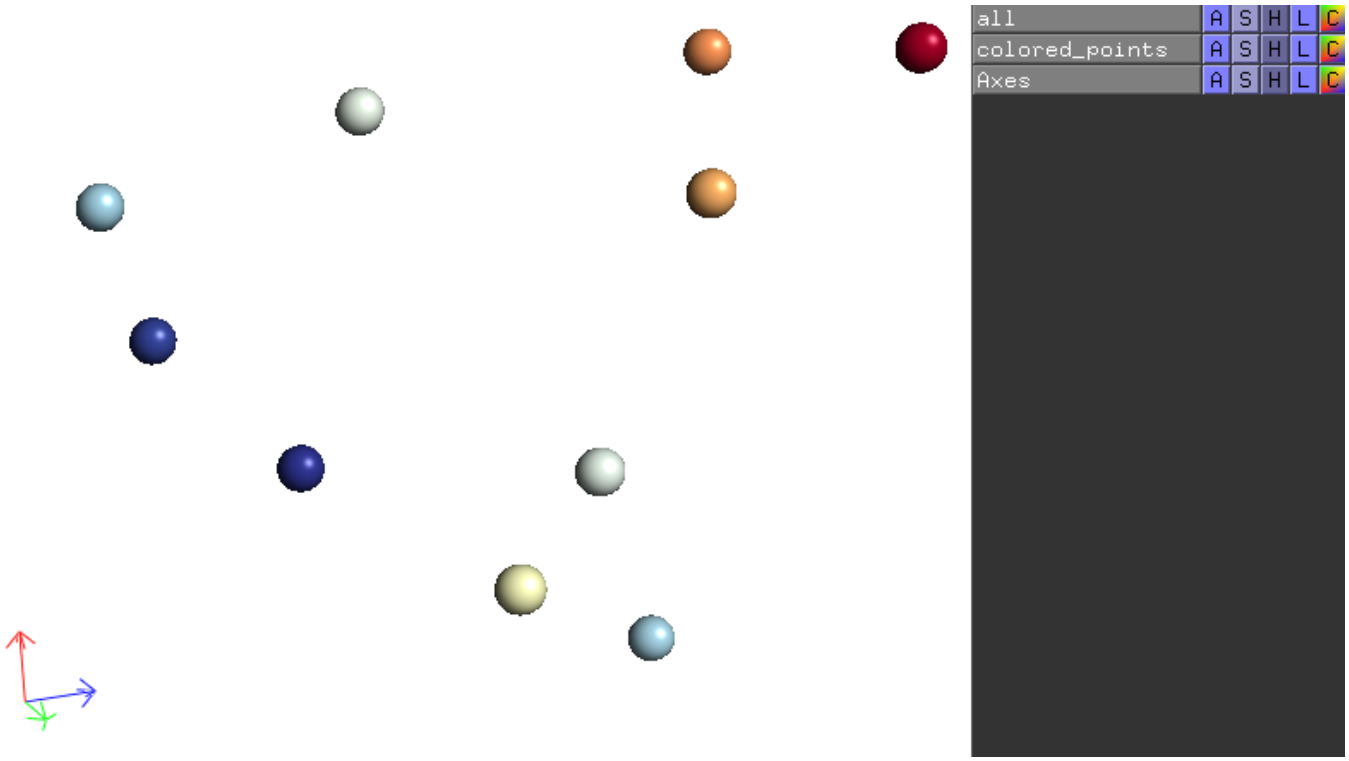
```
p3.write("out/larger_points.py")
```



Automatic color inference

Instead of passing a color as a string we can pass any value that is accepted by the *ColorMap* class explained below. Specifically we can just pass a set of values and they will be automatically colored. For an example, we will color the points based on their distance to the origin:

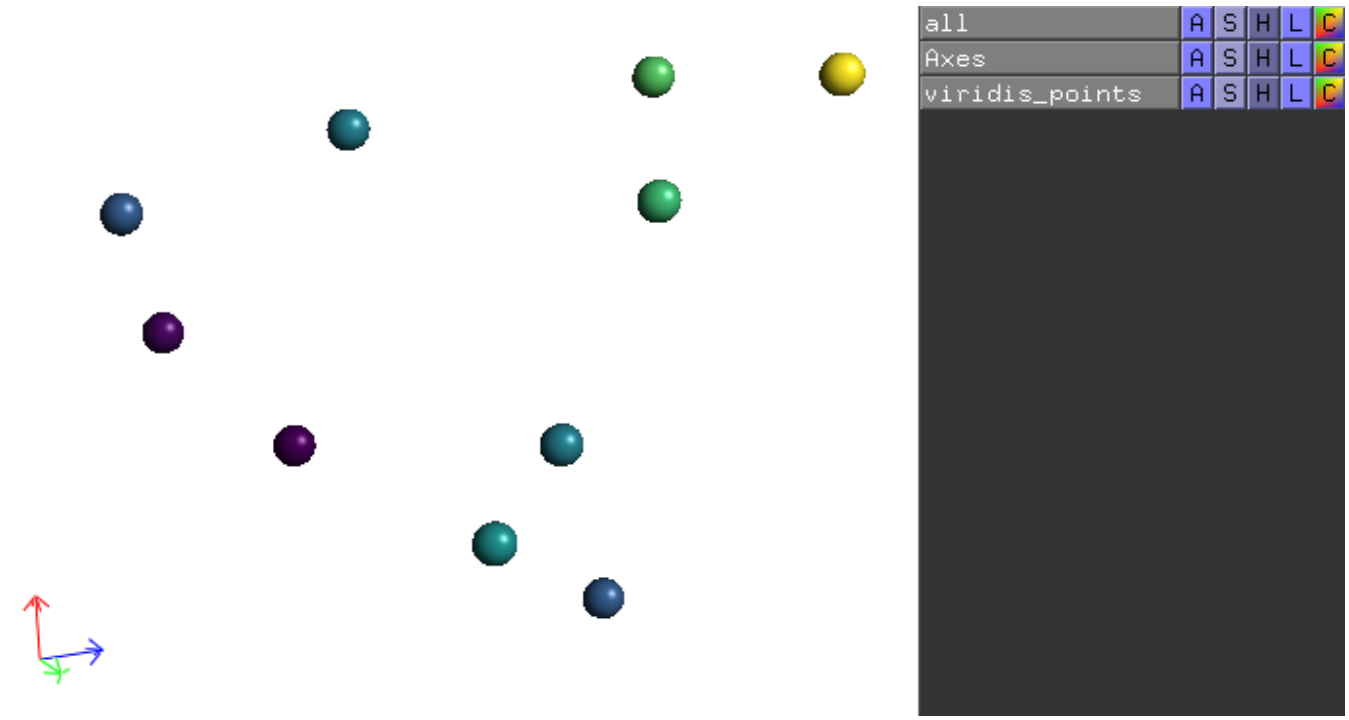
```
values = np.linalg.norm(vertices, axis = 1)
p2 = pmv.Points(vertices, color = values, name = "colored_points")
p2.write("out/colored_points.py")
```



For reference axes were added at the origin, see below in the Arrows chapter.

We can change the colormap beeing used by passing a different argument to *colormap*:

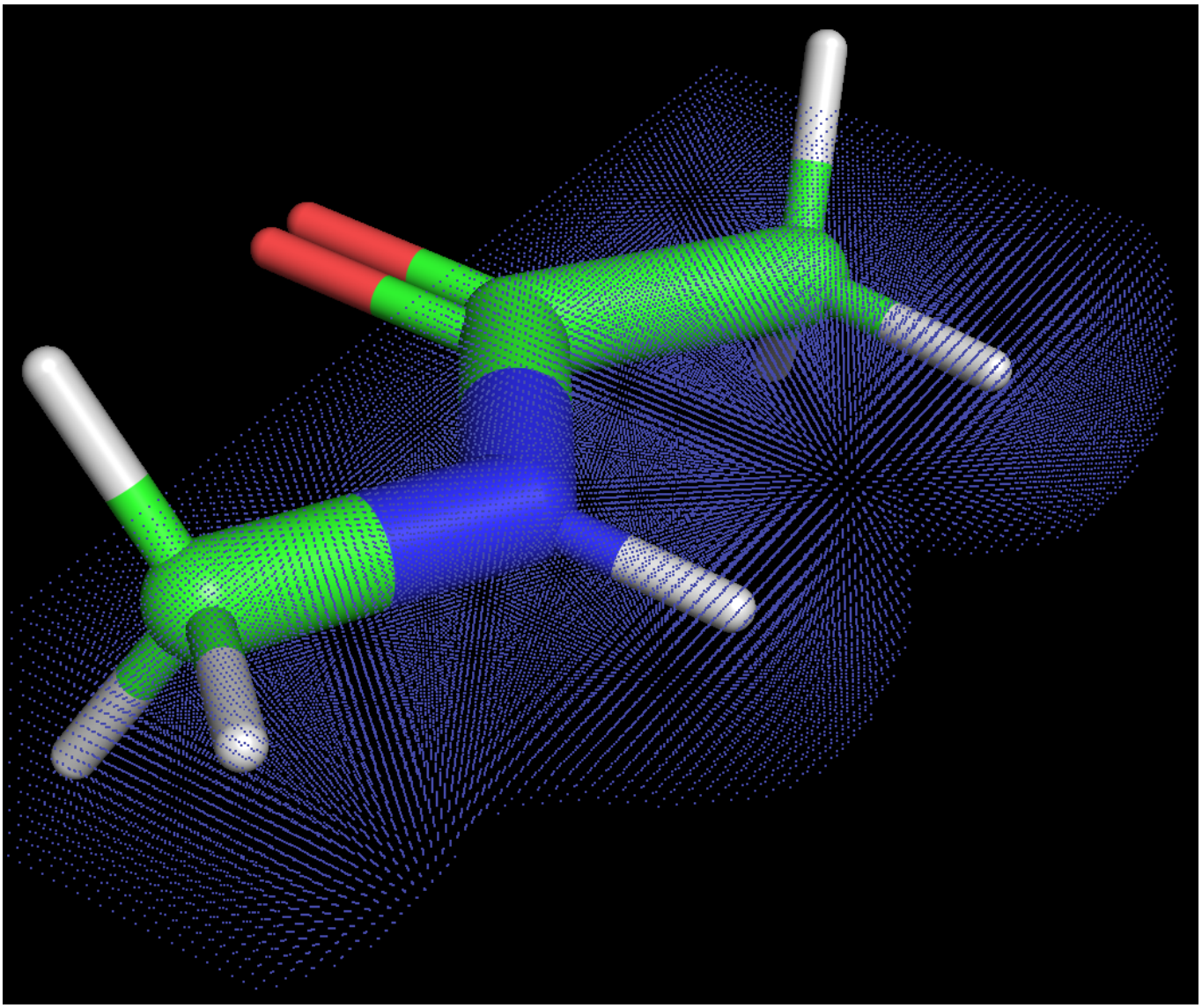
```
p3 = pmv.Points(vertices, color = values, name = "viridis_points",
colormap="viridis")
p3.write("out/viridis_points.py")
```



colormap can be a string describing a matplotlib colormap, a matplotlib colormap itself or a PyMOLViz.ColorMap. The latter is particularly useful if you want to customize your colormaps easily.

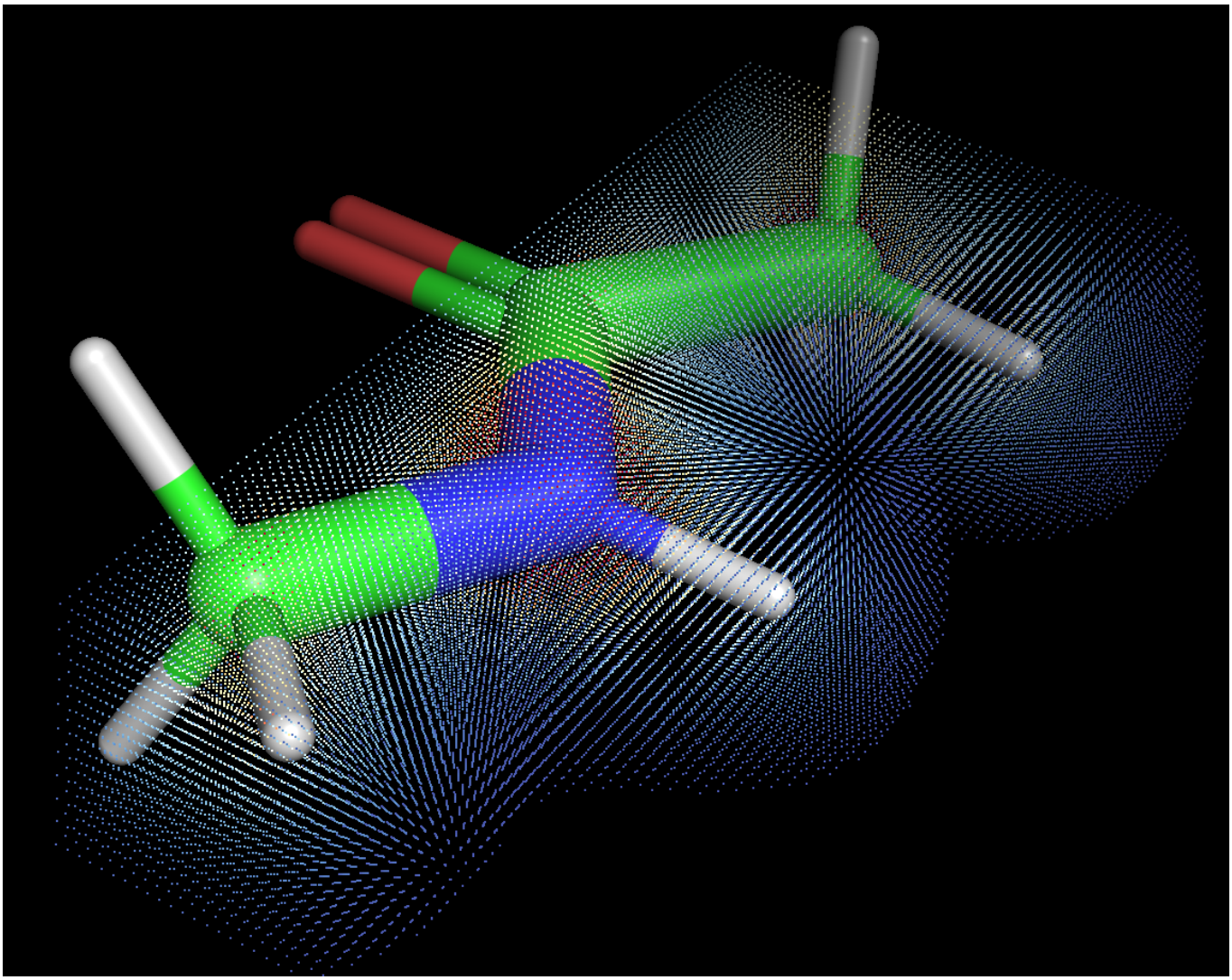
As an example we will plot results of a electron density calculation for N-methyl acetamide as points (**you should almost never actually do this! Instead you should use a volumetric display as explained below.**). To easily read in the data we make use of the PyMOLViz GridData class which will be explained below.

```
data = pmv.GridData.from_xyz("../data/td.xyz")
vertices = data.get_positions()
values = data.values

p4 = pmv.Points(vertices, color = values, render_as="Dots", name =
"td_points")
p4.write("out/td_points.py")
```



Almost all points get assigned a blue color. This is because the colors are normalized against the minimum and maximum values. The density exactly around the atoms is very large! In order to change the coloring into a more useful range, we pass a custom PyMOLViz colormap.

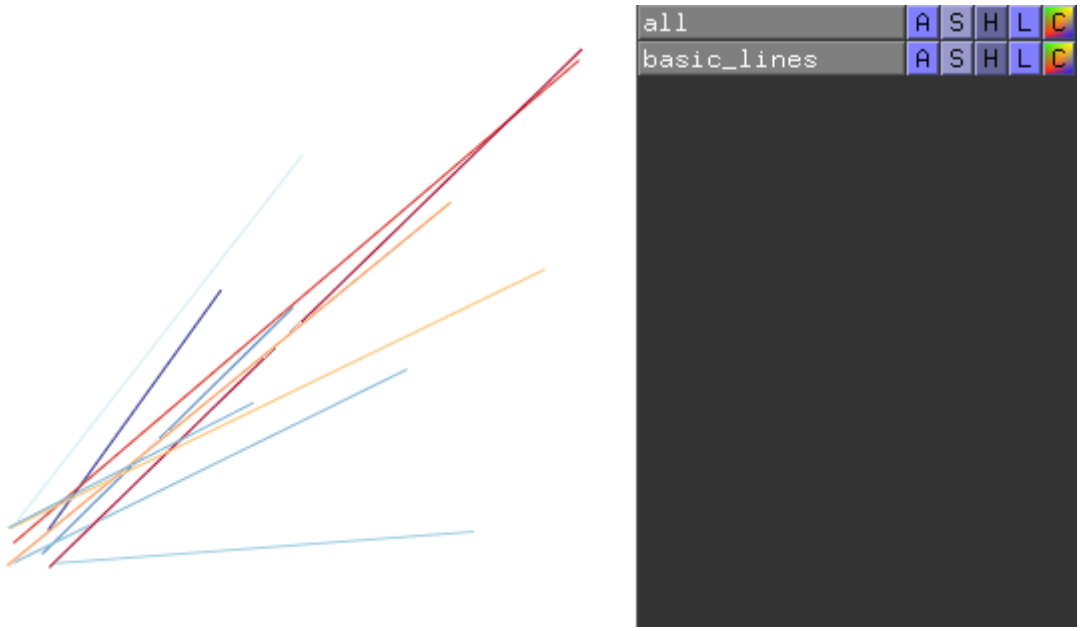
```
p6 = pmv.Points(vertices, color = values, render_as="Dots", name =
"td_points", colormap = pmv.ColorMap([0, 0.5]))
p6.write("out/td_points.py")
```



Lines

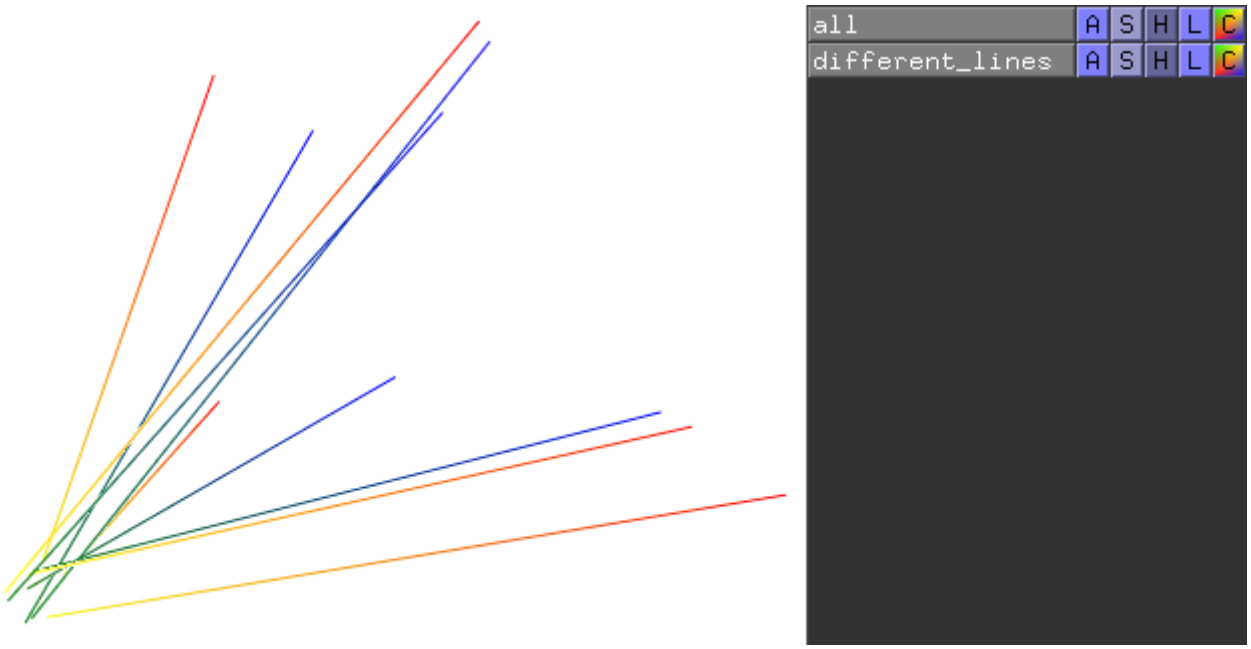
Instead of points lines can also be generated. For this the *Lines* constructor can be used. Lines inherits from Points and also accepts a set of vertices as its first argument. Lines are then drawn between each 2 consecutive points. If the color argument is an array-like, it can either specify a color for each single line or each point.

```
starts = np.random.rand(10,3) * 1
ends = np.random.rand(10,3) * 10
values = np.linalg.norm(ends, axis = 1)

l = pmv.Lines(np.hstack([starts, ends]), name = "basic_lines", color = values)
l.write("out/basic_lines.py")
```



With a color value for each point:

```
values = np.full((5,4), ["yellow", "red", "green", "blue"]).flatten()
l2 = pmv.Lines(np.hstack([starts, ends]), name = "different_lines", color =
values)
l2.write("out/different_lines.py")
```

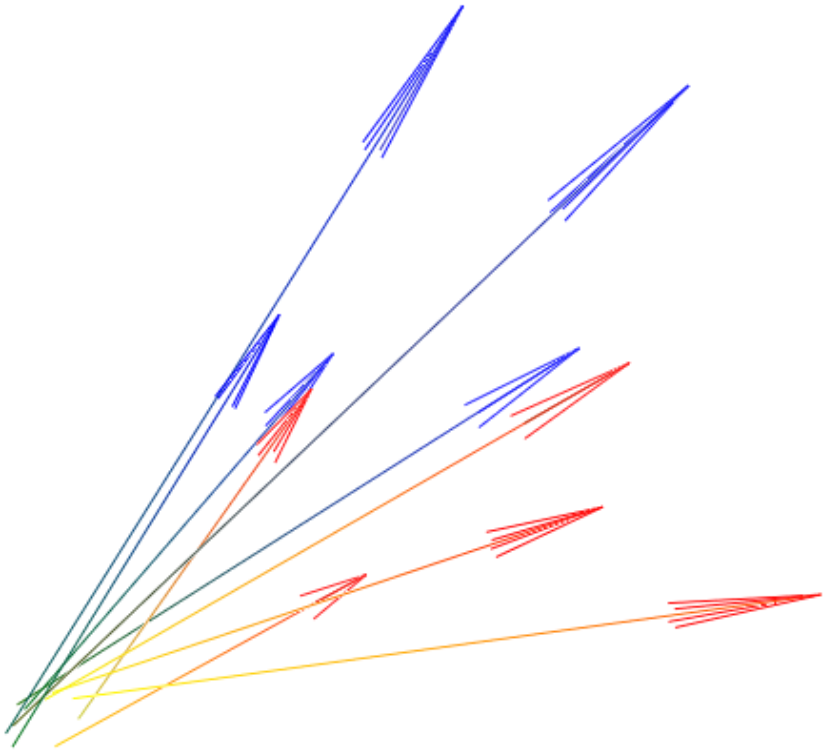


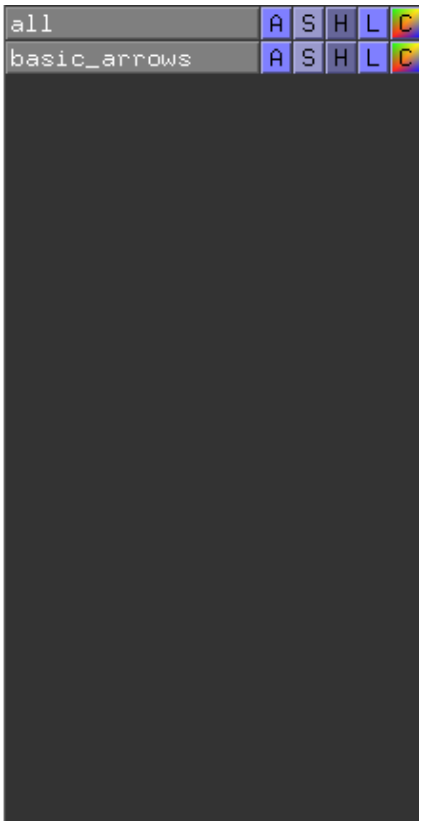
Arrows

Instead of just creating lines, the direction can be indicated via Arrows. Arrows inherit from lines.
Arrows get 2 additional keywords, indicating how their head should be shaped:
head_length and *head_width* both are relative values w.r.t. their length and default to .2.

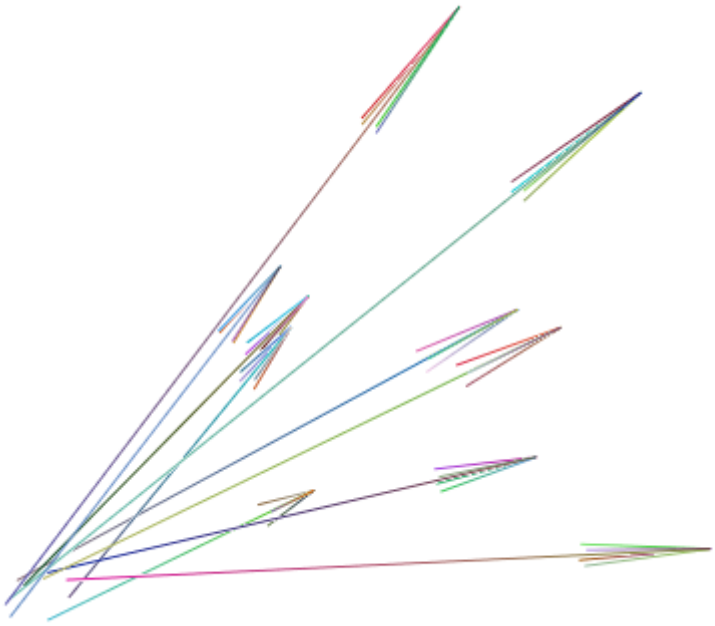
If color is an array, it can either be passed as with lines, i.e. each line, or each end, or for each point of the arrow (first two for the line, next 8 for each line constituting the head).

```
a1 = pmv.Arrows(np.hstack([starts, ends]), name = "basic_arrows", color =
values)
a1.write("out/basic_arrows.py")
```





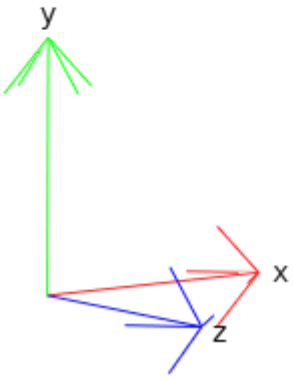

```
values = np.random.rand(10,10, 3).reshape(-1, 3)
a2 = pmv.Arrows(np.hstack([starts, ends]), name = "random_arrows", color =
values)
a2.write("out/random_arrows.py")
```



Coordinate Axes

Using the *Arrows* and the *Labels* classes we can create coordinate axes:

```
start = np.zeros((3,3))
end = np.eye(3)
a3 = pmv.Arrows(np.hstack([start, end]), name = "coordinate_axes_arrows",
color = end)
labels = pmv.Labels(end * 1.1, name = "coordinate_axes_labels", labels = ["x",
"y", "z"])
pmv.Group([a3, labels], "coordinate_axes").write("out/coordinate_axes.py")
```



Colors

Colors in PyMOLViz are exclusively determined via the *ColorMap* class.

ColorMap

One of the most reoccuring tasks is to apply a colormap to a set of values. In order to simplify this as much as possible, PyMolViz has implemented an auxiliary ColorMap class which automatically tries to gracefully interpret different types of input.

Display

The colormap class implements two functions to display them. One to show them as a matplotlib figure:

```
cmap.get_figure(orientation = "horizontal", figsize=(10, 1))
```

As in:

```
def plot_colors(cmap, x):
    # plotting given points
    test_colors = cmap.get_color(x)
    print("\n\nColored points:")
    plt.scatter(x, np.full_like(x, 0.5), color = test_colors)
    plt.show()

    #plotting the colorbar
    print("Colorbar:")
    cmap.get_figure(orientation = "horizontal", figsize=(10, 1))
```

The other way is to show them as a ColorRamp in PyMOL:

```
values = [-2, 0, 0, 3]
color_values = [0, 0.4, 0.65, 1]
cmap = pmv.ColorMap(list(zip(values, color_values)), name ="colormap")
cmap.write("colormap.py")
```

Inputs

In the following every possible *color* input to the ColorMap class is discussed.

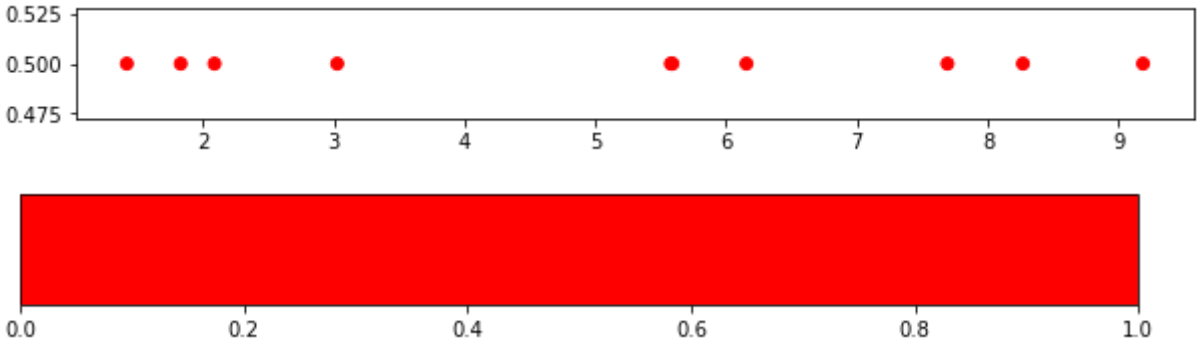
Single Color Input

Sometimes instead of a color for every point / line / mesh, you just want a single color (this is in fact the default if no color is passed). For this the colormap can be passed only a single color and will assign this color to any value that is passed to it.

Single Color Inputs can be:

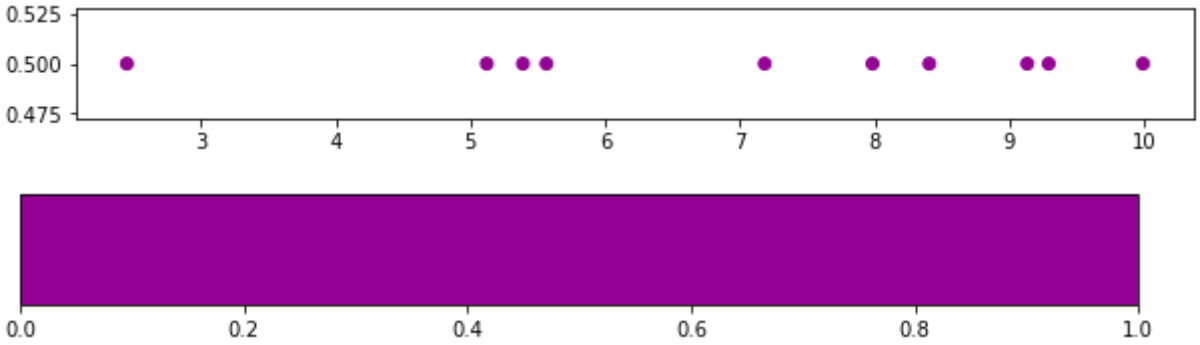
- **A colors name**

```
cmap = pmv.ColorMap("red")
plot_colors(cmap, np.random.rand(10) * 10)
```



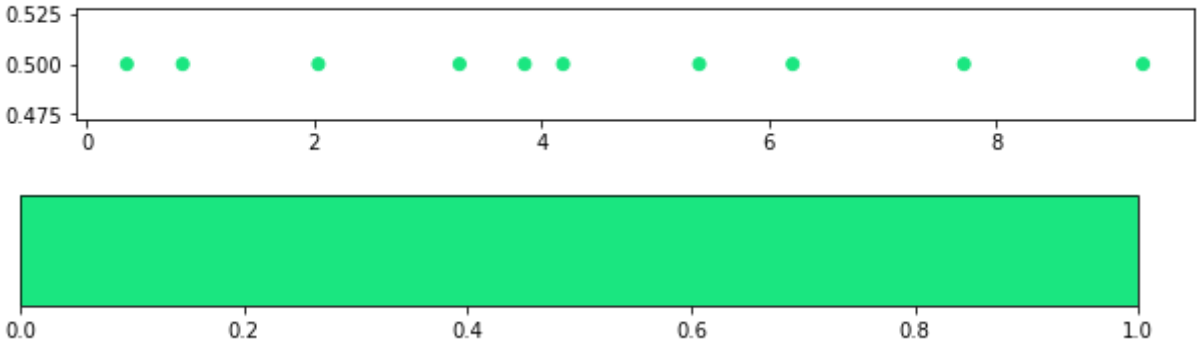
- **An elements name (which then gets converted based on pymols element colors)**

```
cmap = pmv.ColorMap("I")
plot_colors(cmap, np.random.rand(10) * 10)
```



- **An array-like with 3 or 4 entries as rgb (a), either indicating values as float between 0 and 1 or as integer between 0 and 255.**

```
cmap = pmv.ColorMap([0.1, 0.9, 0.5])
plot_colors(cmap, np.random.rand(10) * 10)
```



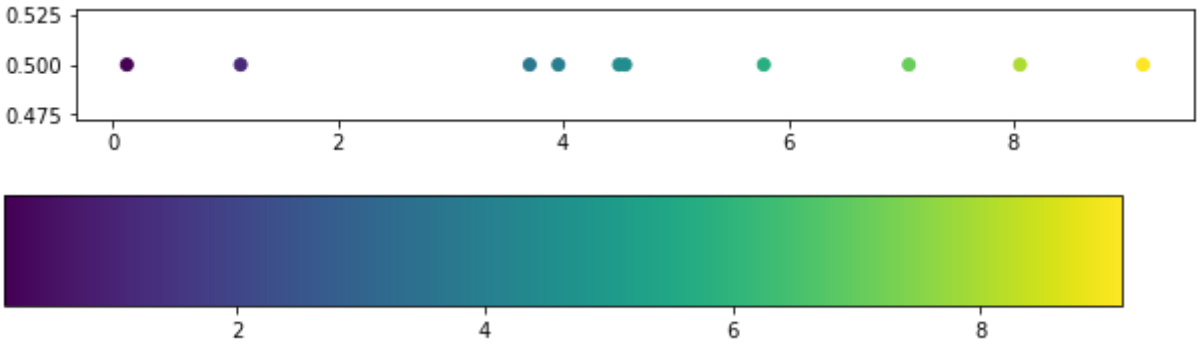
Multiple Color Input

The more interesting case is if you want to assing different colors to different values. For

this the ColorMap class can infer the colormap scaling from a set of values passed to it. In this case the passed *colormap* keyword is used to infer the colormaps colors. Alternatively if a list of single colors (see above) is passed to it, it will map each index to the corresponding color.
Multiple Color Inputs can be:

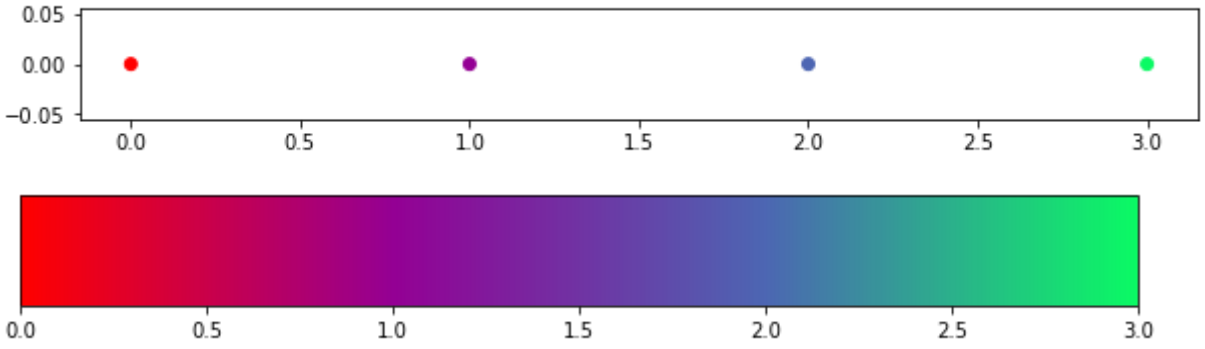
- **An array-like containing float-like values (colors are taken from a matplotlib colormap corresponding to the passed *colormap* argument. Defaults to *"RdYlBu_r"*.)**

```
values = np.random.rand(10) * 10
cmap = pmv.ColorMap(values, colormap="viridis")
plot_colors(cmap, values)
```



- **An array-like containing single colors (see above)**

```
cmap = pmv.ColorMap(["red", "I", [0.3, 0.4, 0.7], [10, 250, 100]])
plot_colors(cmap, range(4))
```

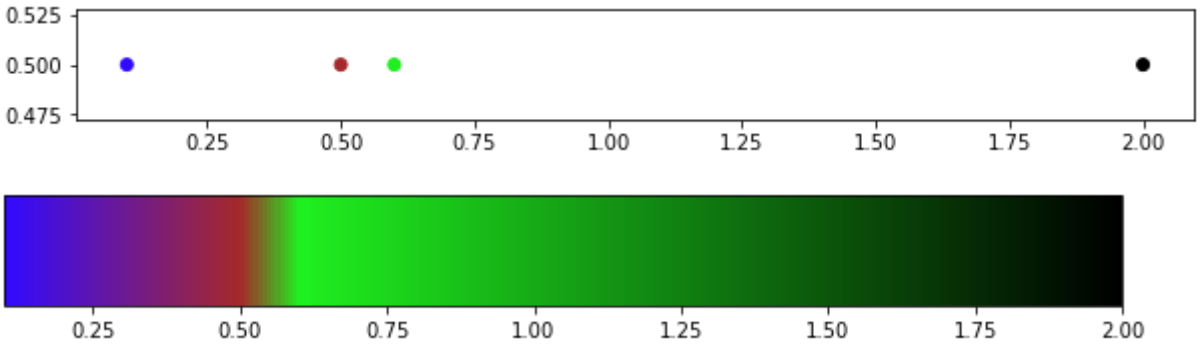


Linear Segmented Color Input
If you want to assign certain colors to certain values, you can pass a list of tuples to the ColorMap, with each tuple (value, color) containing the value and the corresponding single color (see above).

Alternatively tuples of (value, color_value) pairs can be passed. In this case color_value may be a float between 0 and 1 or an integer between 0 and 255 indicating the color based on the passed *colormap* keyword.
Linear Segmented Color Inputs can be:

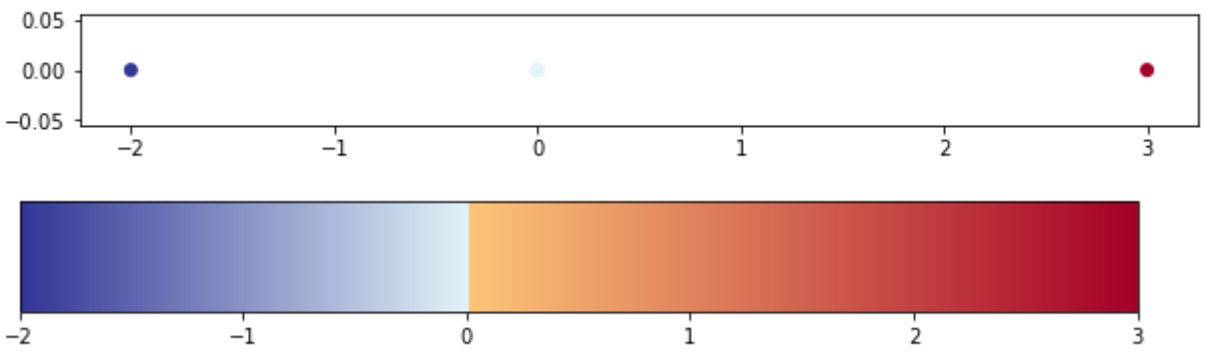
- **An array-like containing tuples of (float-like, single color (see above)).**

```
values = [0.1, 0.5, 0.6, 2]
colors = [(50, 10, 255), "Br", "Cl", "black"]
cmap = pmv.ColorMap(list(zip(values, colors)))
plot_colors(cmap, values)
```



- **An array-like containing tupes of (float-like, color-indicator) with color-indicator being a float between 0 and 1. The colors are then determined by passing the color-indicator to the matplotlib colormap corresponding to the passed *colormap* argument (Defaults to *"RdYlBu_r"*).**

```
values = [-2, 0, 0, 3]
color_values = [0, 0.4, 0.65, 1]
cmap = pmv.ColorMap(list(zip(values, color_values)))
plot_colors(cmap, values)
```



Meshes and Primitives

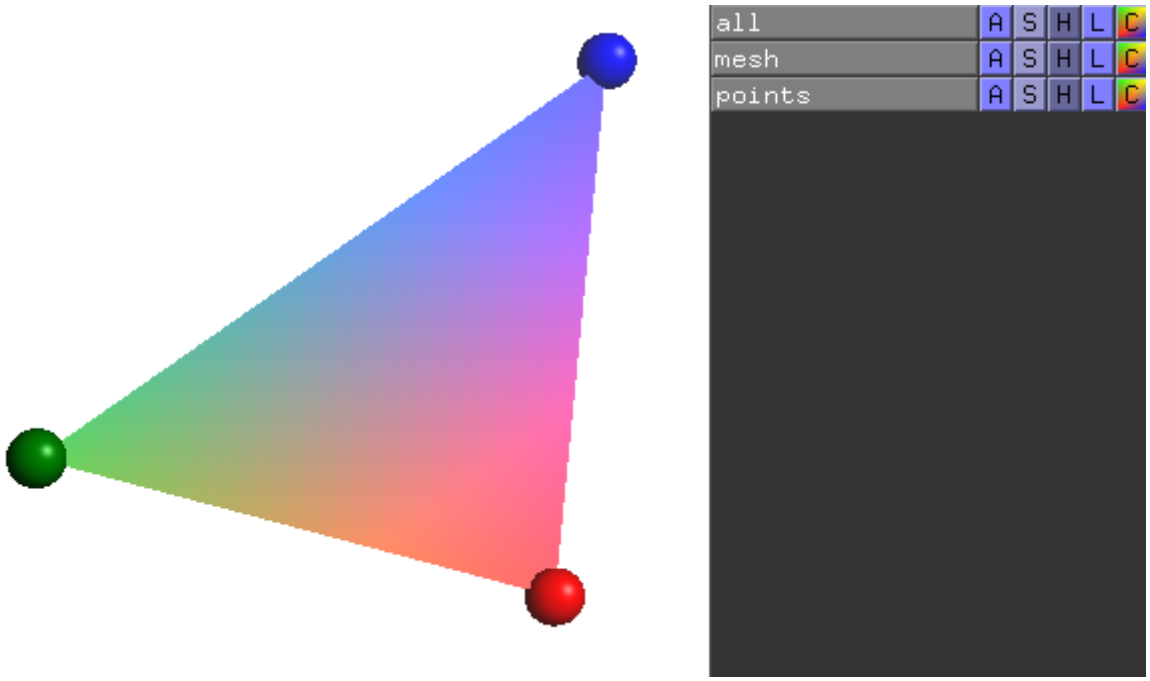
Meshes

In addition to the basic data representions presented thus far, PyMOLViz also provides an Interface to the Mesh representation in PyMOL. Such a mesh has, in addition to points and colors also faces and vertex normals.

Faces should be a list or array of 3 dimensional indices into vertices. Faces indicate which 3 vertices should be connected to generate an opaque triangle. If faces are not given, every 3 consecutive points are assumed to form a triangle.

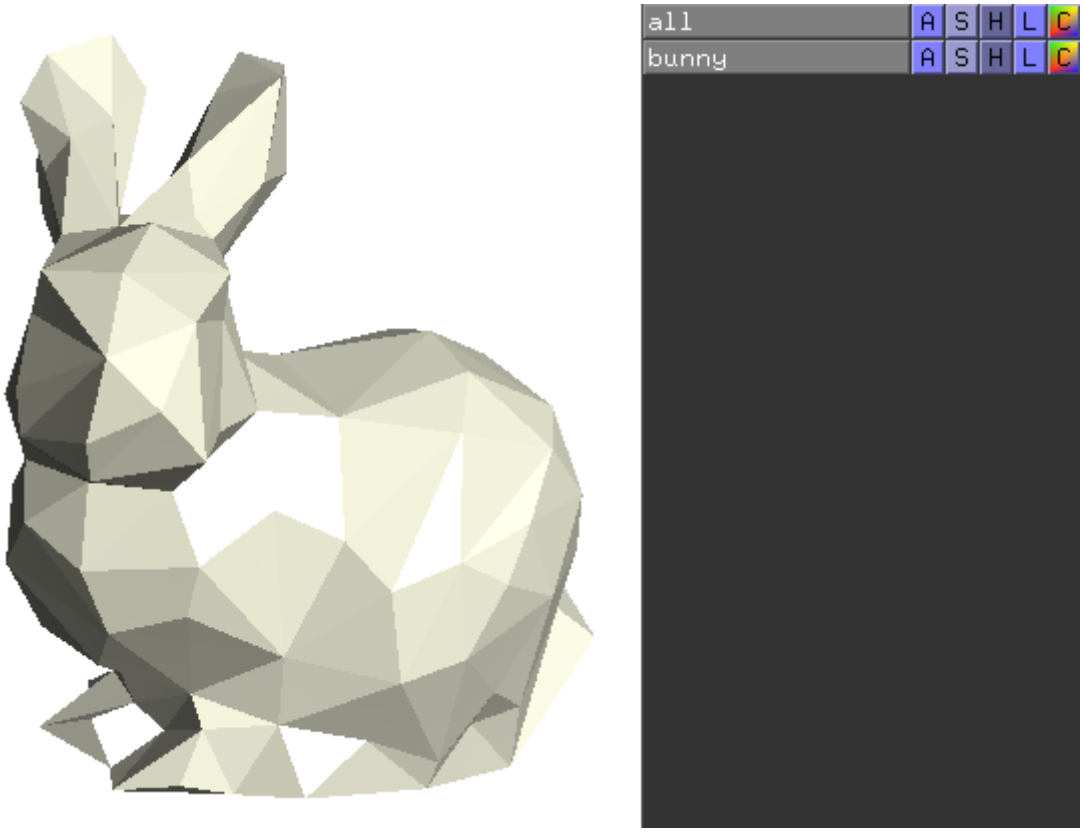
As a mesh will display an actual surface the additional information of the direction of the surface is used to compute lighting on the surface. For this, **vertex normals** can be provided which should for every provided vertex indicate the surface normal at that vertex. If you have no information of vertex normals you can either compute them from surrounding vertices or just pass zero vectors.

```
import pymolviz as pmv
import numpy as np

# creating dummy points
points = np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0]])
colors = np.array(["red", "green", "blue"])

# creating a point mesh of the dummy points, rendering the points as spheres
p = pmv.Points(points, color = colors, name ="points")

# creating the actual mesh. As we dont know the vertex normals, we set them to
0 vectors. The faces input means, the points at index 0, 1 and 2 should be
connected to create a triangle. Any color between vertices is interpolated (by
PyMOLs shaders).
m = pmv.Mesh(points, faces = [[0, 1, 2]], normals = np.zeros_like(points),
color = colors, name = "mesh")

pmv.Script([m, p]).write("out/test_mesh.py")
```



We can also load larger meshes into PyMOL, to display arbitrary objects:

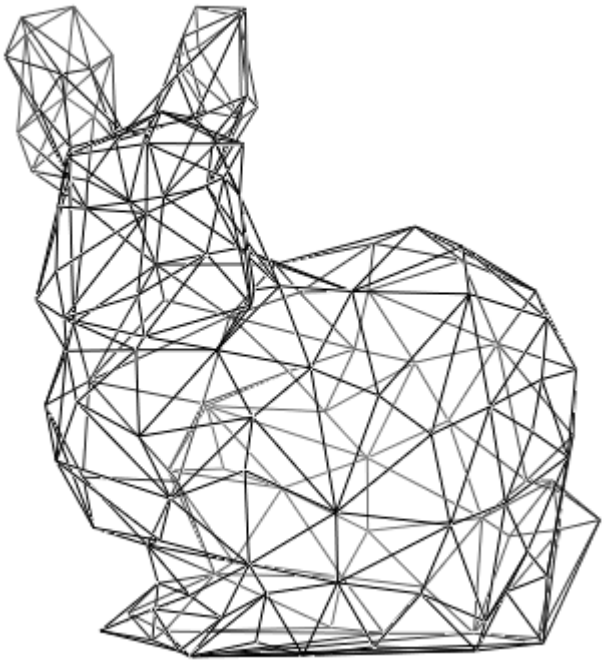
```
bunny_points = np.load('../data/Bunny.npy')
m2 = pmv.Mesh(bunny_points, color = "beige", name = "bunny")
m2.write("out/bunny.py")
```



Wireframes

Any mesh can be turned into a wireframe via the *to_wireframe* method. Note that this is different from an IsoMesh that is created using volumetric data, which is explained further below.

```
bunny_points = np.load('../data/Bunny.npy')
l = m2.to_wireframe(name = "bunny_wireframe")
l.colormap = pmv.ColorMap("black")
l.write("out/bunny_wireframe.py")
```

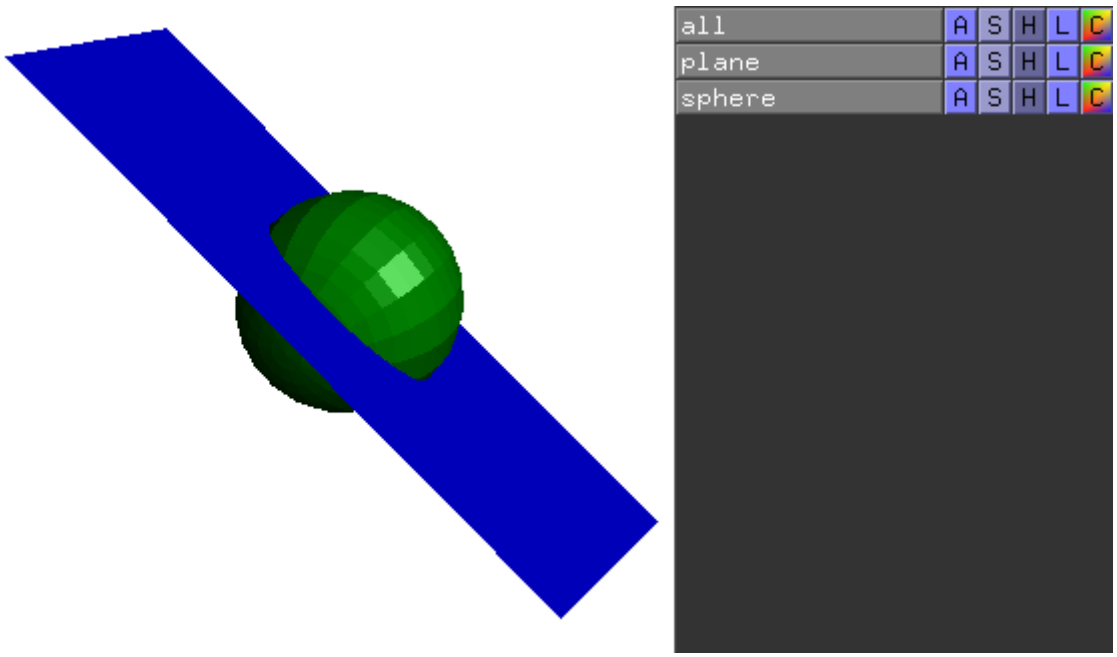


Mesh Primitives

PyMOLViz currently also provides the *Plane* and *Sphere* classes, which allow to instantiate plane and sphere meshes more easily.

- The *Plane* class represents a mesh consisting of two triangles which are placed at a given `position` and with a given plane `normal`. The size of the triangles is controlled by the `scale` parameter.
- The *Sphere* class allows to create a sphere directly as a mesh (allowing to e.g. extract its wireframe representation). If you want to represent many points as spheres, you should use the *Points* class instead. The *sphere* is defined by its `position`, `radius` and its `resolution`, determining how many points are approximating a circle.

```
start_point = np.array([1,1,1])

# creating plane
normal = np.array([1,1,-1])
plane = pmv.Plane(start_point, normal, scale = 2, color = "blue", name =
"plane")

# creating sphere
sphere = pmv.Sphere(start_point, 0.3, color = "green", name = "sphere")

# joining meshes into a script
s = pmv.Script([plane, sphere])
s.write("test_primitives.py")
```



Collections, Groups, Scripts and States

Up to now we basically always created a *Displayable* and used its *write* method to generate a corresponding script. In reality we will usually want to combine different *Displayables* and write them to a single script. The classes *CGOCollection*, *Group*, and *Script* all allow to gather different *Displayables* but have distinct use cases.

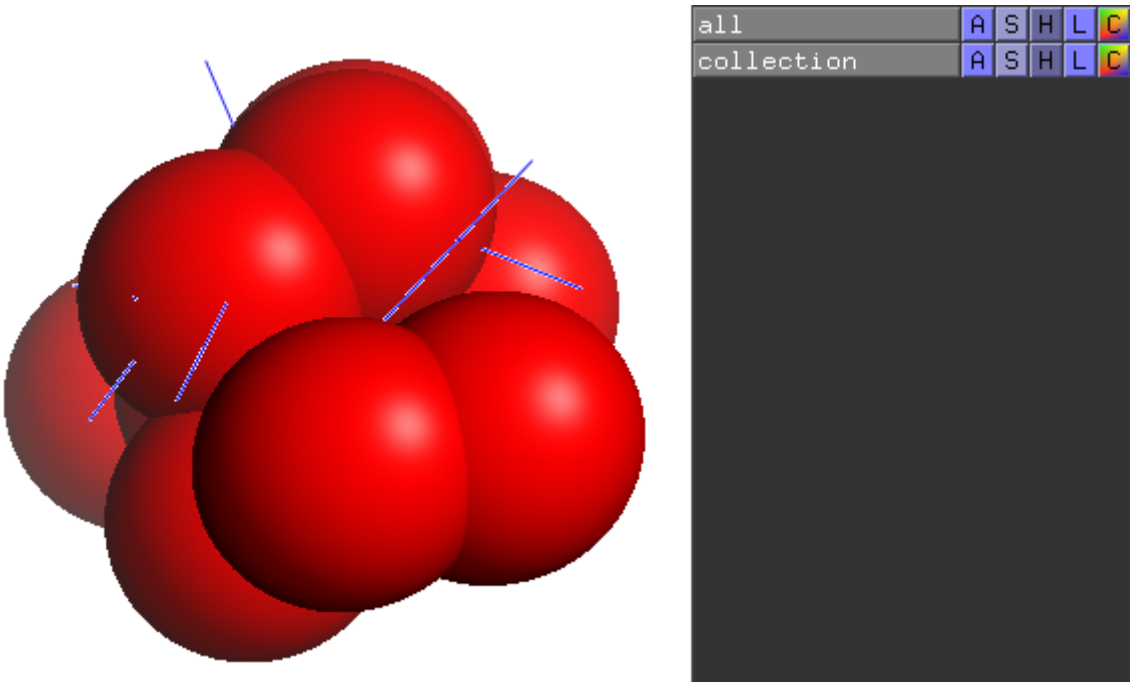
CGOCollection

CGOCollections, as the name indicates are collections of CGOs, **C**ompiled **G**raphics **O**bjects. In PyMOLViz this corresponds to every class that inherits from *Points*. This

includes every class except for the ColorMap that has been presented thus far. However it specifically excludes any volumetric class.

CGOCollections are *Displayables* and have therefore a *name* and a *write* function. They are special in that all *Displayables* that are part of a *CGOCollection* will be combined and appear in PyMOL as the **same** *Displayable* under the name of the Collection (unless they are also added by different means).

```
points = pmv.Points(np.random.rand(10,3), color = (1,0,0), name = "points")
lines = pmv.Lines(np.hstack([np.random.rand(10,3), np.random.rand(10,3)]),
color = (0,1,0), name = "lines")
collection = pmv.CGOCollection([points, lines], name = "collection")
collection.write("basic_collection.py")
```

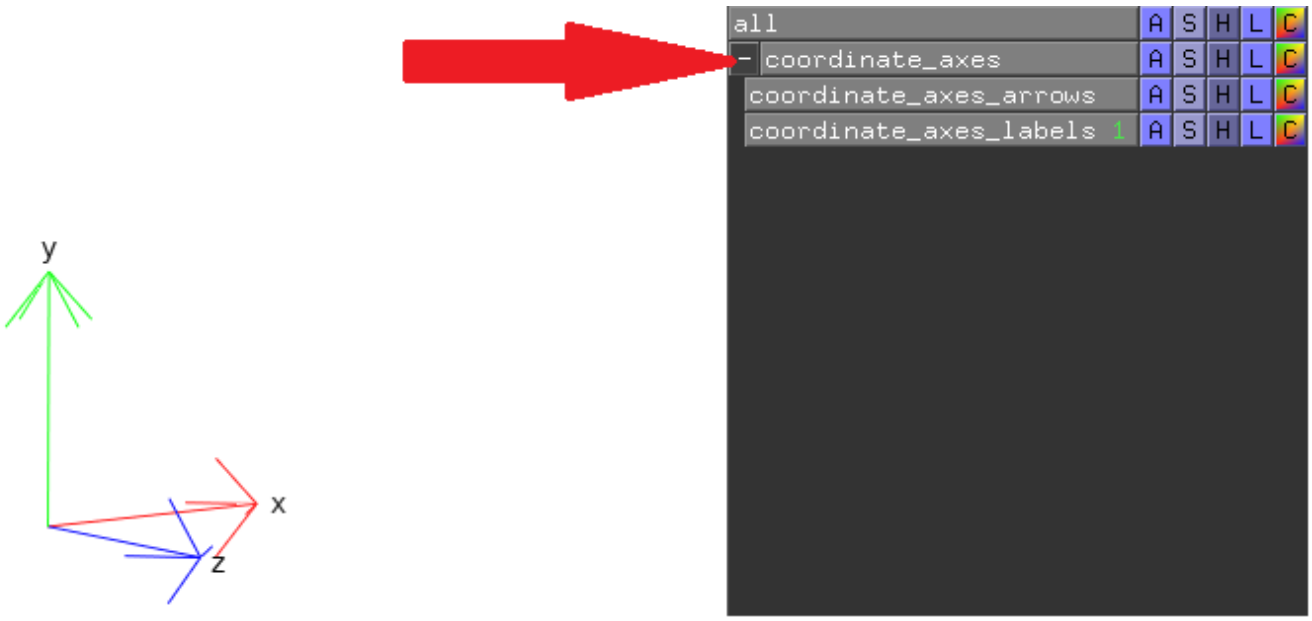


Groups

Groups are used to gather multiple *Displayables*. Other than with *CGOCollections*, any other *Displayable* **that is not a *Group*** can be gathered. They are part of PyMOLs GUI. They are themselves also *Displayables* and therefore a *name* and a *write* function. Any *Displayable* that is part of a *Group* will also be loaded into PyMOL itself. We made use of the *Group* class when implementing the coordinate axes.

```
start = np.zeros((3,3))
end = np.eye(3)
a3 = pmv.Arrows(np.hstack([start, end]), name = "coordinate_axes_arrows",
```

```
color = end)
labels = pmv.Labels(end * 1.1, name = "coordinate_axes_labels", labels = ["x",
"y", "z"])
g = pmv.Group([a3, labels], "coordinate_axes")
g.write("out/coordinate_axes.py")
```

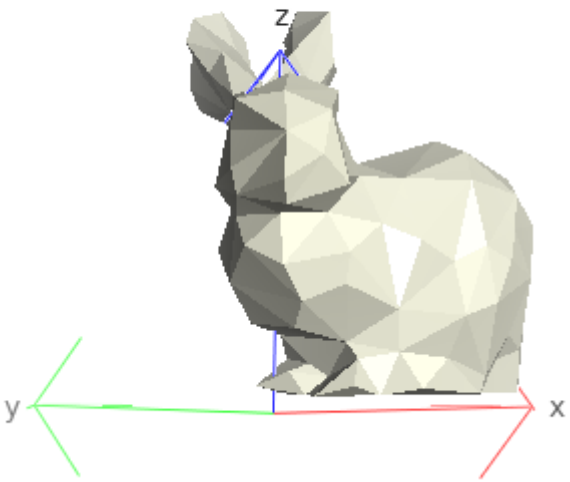


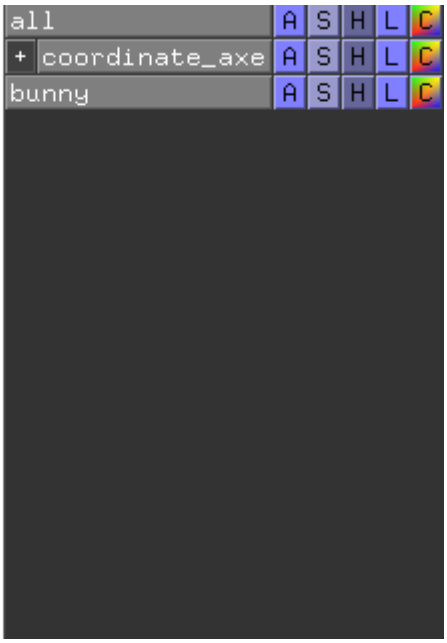
Scripts

Scripts lie at the root of the implementation of PyMOLViz as any *Displayable* is converted to a *Script* when its *write* function is called. The *Script* class is used to actually convert its *Displayables* into runnable python scripts. It can be used to gather different *Displayables* which should be loaded with a single script. ***Scripts* themselves are not *Displayables* but rather represent a collection of *Displayables* that should be converted into a single python script.**

We could use the *Script* class to write the coordinate axes and the bunny to a single script:

```
bunny_points = np.load('../data/Bunny.npy')
m2 = pmv.Mesh(bunny_points, color = "beige", name = "bunny")
s = pmv.Script([g, m2])
s.write("out/bunny_with_coordinates.py")
```


z
y
x


all
A S H L C
+ coordinate_axe
A S H L C
bunny
A S H L C