COMMIT TO MASTER

Deliverable 2

CSCD01

Mahima Bhayana
Kalindu De Costa
Victor Lee
Harman Wadhwa
Leo Yao

Table of Contents

ISSUES	**2**
#12911	2-5
#2341	5-7
#11746	8-10
#8532	10-12
DEVELOPMENT PROCESS	12

Issue #12911

https://github.com/matplotlib/matplotlib/issues/12911

Background

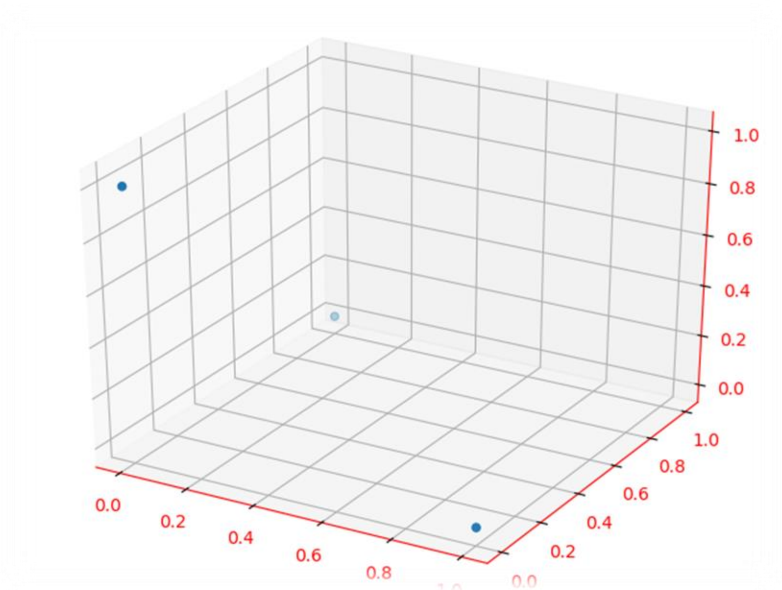
The tick_params method of Axes3D does not properly change the colour of the ticks--it (incorrectly) changes only the colour of the tick label, not of the tick itself. As such, setting the colour of the ticks of an Axes3D object would result in the figure having black ticks with labels in the indicated colour.

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot as plt

fig = plt.figure()
ax = Axes3D(fig)

ax.scatter((0, 0, 1), (0, 1, 0), (1, 0, 0))
ax.w_xaxis.line.set_color('red')
ax.w_yaxis.line.set_color('red')
ax.w_zaxis.line.set_color('red')
ax.xaxis.label.set_color('red')
ax.yaxis.label.set_color('red')
ax.zaxis.label.set_color('red')
ax.tick_params(axis='x', colors='red')  # only affects
ax.tick_params(axis='y', colors='red')  # tick labels
ax.tick_params(axis='z', colors='red')  # not tick marks

fig.show()
```



Solution

We found that the tick_params method was actually setting the colour of the ticks properly, but that this colour was being overwritten in the draw method of axis3D.
The solution for this problem ended up being rather simple--we had to stop the method from overwriting the tick color. Removing this line, which sets the tick colour to a hard-coded value ('k', for black) stored inside self._axinfo.

```
            tick_update_position(tick, (x1, x2), (y1, y2), (lx, ly))
            tick.tick1line.set_linewidth(info['tick']['linewidth'])
            tick.tick1line.set_color(info['tick']['color'])
            tick.set_label1(label)
            tick.set_label2(label)
            tick.draw(renderer)
```

Found in the class Axis in lib/mpl_toolkits/mplot3d/axis3d.py

This change helped remove dependence on a legacy, hard-coded value from the Axis3D class. A comment on the issue revealed that self._axinfo in the Axis3D class is just a dictionary created years ago to consolidate hard-coded values into one object. While this change didn't impact the design/code of matplotlib very heavily, it did prevent it from relying on legacy hard-coded values that are no longer relevant.

Acceptance test suite

We added image comparison tests to lib/mpl_toolkits/tests/test_mplot3d.py.

```python
@image_comparison(baseline_images=['draw_tick_colors_blue'], extensions=['png'])
def test_draw_tick_colors_blue():
    fig = plt.figure()
    ax = Axes3D(fig)

    ax.scatter((0, 0, 1), (0, 1, 0), (1, 0, 0))
    ax.tick_params(axis='x', color='blue')
    ax.tick_params(axis='y', color='blue')
    ax.tick_params(axis='z', color='blue')

@image_comparison(baseline_images=['draw_tick_colors'], extensions=['png'])
def test_draw_tick_colors():
    fig = plt.figure()
    ax = Axes3D(fig)

    ax.scatter((0, 0, 1), (0, 1, 0), (1, 0, 0))

@image_comparison(baseline_images=['draw_tick_colors_blue'], extensions=['png'])
def test_draw_tick_colors_multiple_times():
    fig = plt.figure()
    ax = Axes3D(fig)

    ax.scatter((0, 0, 1), (0, 1, 0), (1, 0, 0))
    ax.tick_params(axis='x', color='red')
    ax.tick_params(axis='y', color='red')
    ax.tick_params(axis='z', color='red')

    ax.tick_params(axis='x', color='blue')
    ax.tick_params(axis='y', color='blue')
    ax.tick_params(axis='z', color='blue')

```

These tests ensure that the tick colour is indeed being set when we use tick_params, and also ensure that figures look the way they are supposed to when the colour is not explicitly set. We added additional tests to set the colours multiple times and ensure that the last colour set is the one that displays on the figure.

Issue #2341

https://github.com/matplotlib/matplotlib/issues/2341

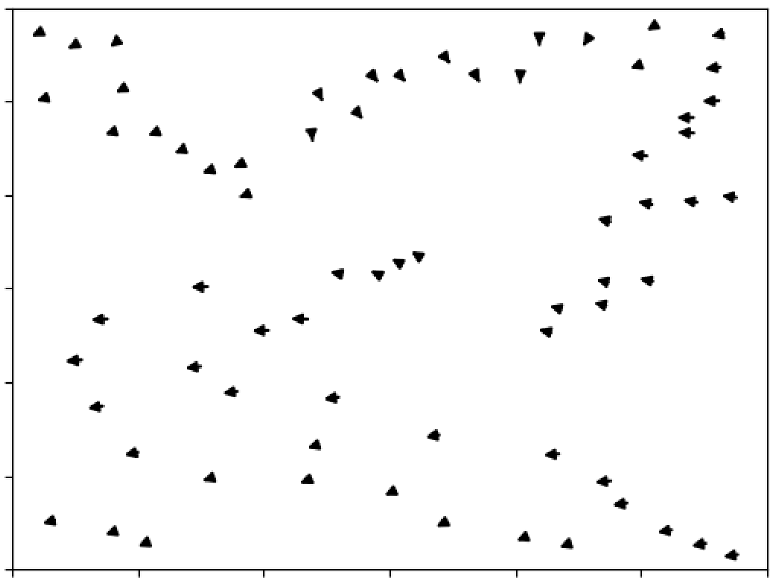
Background

PatchCollection cannot handle FancyArrowPatch patches, as the paths for FancyArrowPatch can't be evaluated during PatchCollection creation. The StreamplotSet.arrows PatchCollection appears to be entirely useless. In fact, streamplot() doesn't even add the collection to the axis, instead adding the individual patches, and then creating an unused PatchCollection to return. This means that things that should work, like, for a StreamplotSet s, doing s.arrows.set_alpha(0), will not work, nor, to my knowledge, will doing anything with StreamplotSet.arrows have the desired result.

```
import numpy as np
import matplotlib.pyplot as plt

w = 3
Y, X = np.mgrid[-w:w:100j, -w:w:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2

c = plt.streamplot(X, Y, U, V, color=(0,0,0,1))
c.lines.set_alpha(0)
c.arrows.set_alpha(0)
plt.show()
```



Solution

- Creating a new Collection: FancyArrowPatchCollection that is an extension of PatchCollection. This will allow FancyArrowPatches to be altered by attributes such as set_alpha and set_color. (lib/matplotlib/collections.py)
- FancyArrowPatchCollection will properly handle get_path and set_path for the Collection as well as _prepare_points to properly calculate the paths.
- Previously StreamPlot was manually adding all FancyArrowPatchs individually to the axes without using the collection, now with the new collection this step is unnecessary and the collection can be added directly to axes. (lib/matplotlib/streampot.py)

```
    ac = matplotlib.collections.FancyArrowPatchCollection(arrows)
    axes.add_collection(ac)
```

Acceptance test suite

Tests are based on image comparison (lib/matplotlib/tests/test_streamplot.py)

1. The first is to test set_alpha attribute, which creates a streamplot and triggers the alpha values of the lines and arrow.
   Expected outcome: A blank plot

2. The second is to test set_color attribute, which creates a streamplot and changes the colour of lines and arrows.
   Expected outcome: A plot with red lines and blue arrowheads

Since prior to this fix, none of the attributes for the arrows were working, here we are testing some attributes which are used commonly to verify that, arrows are affected by such attributes.

```python

@image_comparison(baseline_images=['streamplot_arrows_attributes_alpha'],
                  extensions=['png'], remove_text=True, style='mpl20')
def test_arrows_alpha():
    w = 3
    Y, X = np.mgrid[-w:w:100j, -w:w:100j]
    U = -1 - X**2 + Y
    V = 1 + X - Y**2

    c = plt.streamplot(X, Y, U, V, color=(0,0,0,1))
    c.lines.set_alpha(0)
    c.arrows.set_alpha(0)

@image_comparison(baseline_images=['streamplot_arrows_attributes_color'],
                  extensions=['png'], remove_text=True, style='mpl20')
def test_arrows_color():
    w = 3
    Y, X = np.mgrid[-w:w:100j, -w:w:100j]
    U = -1 - X**2 + Y
    V = 1 + X - Y**2

    c = plt.streamplot(X, Y, U, V, color=(0,0,0,1))
    c.lines.set_color('red')
    c.arrows.set_color('blue')
```

Issue #11746

https://github.com/matplotlib/matplotlib/issues/11746

<u>Background</u>

The arrowheads of arrows in 3D space appears to converge and disappear as the scale of the axis increases. When calling quiver(*args, length=1, arrow_length_ratio=0.3, pivot='tail', normalize=False, **kwargs) from mpl_toolkits.mplot3d.axes3d.Axes3D, there is a predefined angle which calculates 2 (fixed) points where the arrowhead ends. These 2 points are then connected to the tip of the arrow itself, to create the triangular point.

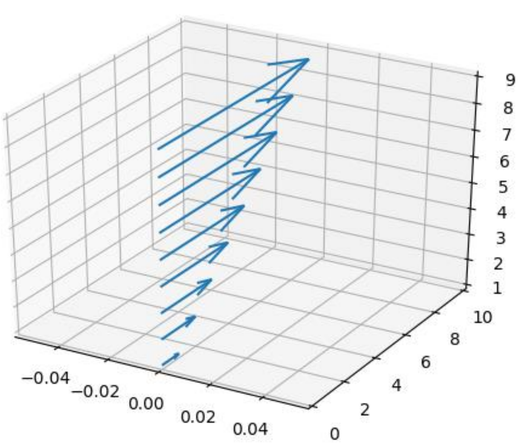
However, as the scale of the axis increase (z-axis in this example), the same 2 points will appear closer to the body of the arrow itself and look as if it had disappeared.

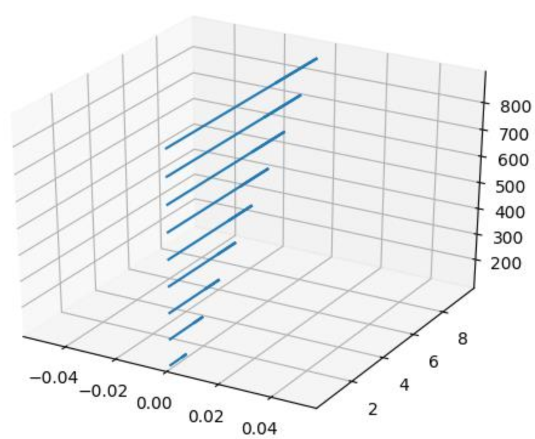
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.zeros(10)
y = np.zeros(10)
z = np.arange(10)*100 # remove *100 and the arrow heads will reappear.
dx = np.zeros(10)
dy = np.arange(10)
dz = np.zeros(10)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.quiver(x, y, z, dx, dy, dz)
ax.set_ylim(0,10)

plt.show()
```





Notice the difference in the z-axis scale between both diagrams. The distance between the 2 points are exactly the same on both diagrams, just that the perceived distance between them is skewed from the difference in scale.

Solution

Users are now able to pass in an extra parameter, headwidth, to set the headwidth of an arrow in 3D space. Headwidth is used the same way as in 2D space: it is the width of the arrowhead relative to the axis it is drawn in.

How it works: headwidth (new parameter), length of the arrow body, and the ratio of the arrowhead to the arrow body are considered to determine the angle necessary to achieve it. The angle is a crucial part of construction of an arrow, and is the only parameter that changes the behaviour of the arrowhead. Using simple trigonometry, the following formula is used to calculate the angle:

$$tan^{-1}\left(\frac{headwidth}{2(arrowlength \cdot ratio)}\right)$$

This eliminates the concern of arrowheads "touching" the body of the arrow itself when a large axis is used. For the above example, a user would construct the arrows in the second diagram by calling quiver with the additional parameter, headwidth, at 100. The changes made allow the 3D implementations of arrow and its functions to be capable of performing those of the 2D implementation. The new implementation allows the user to pass in an extra parameter in calc_arrow, (headwidth), to set the desired width of the arrowhead relative to the axis.

```
def calc_angle(headwidth):
    """
    Calculate a new angle to for the arrowhead to match the given headwidth
    new angle = arctan (headwidth / (2* length of line covered by arrow))
    """

    x1, y1, z1= input_args[:3]
    x2,y2,z2 = input_args[3:argi]
    linelength = ((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)**0.5

    # length of line overlapping the arrow
    arrowlength = linelength * arrow_length_ratio
    angle = math.degrees(math.atan(headwidth/(2*arrowlength)))
    return angle

def calc_arrow(uvw, angle=15, headwidth=None):
    """
    To calculate the arrow head. uvw should be a unit vector.
    We normalize it here:
    """

    # if headwidth is privided, calculate arrowhead angle wrt the headwidth
    if headwidth:
        angle = calc_angle(headwidth)
```
Code snippet from mpl_toolkits.mplot3d.axes3d.Axes3D

Acceptance test suite

Tests are based on image comparison (lib/matplotlib/tests/test_quiver.py)

The test creates a single arrow with the custom headwidth as an input parameter.

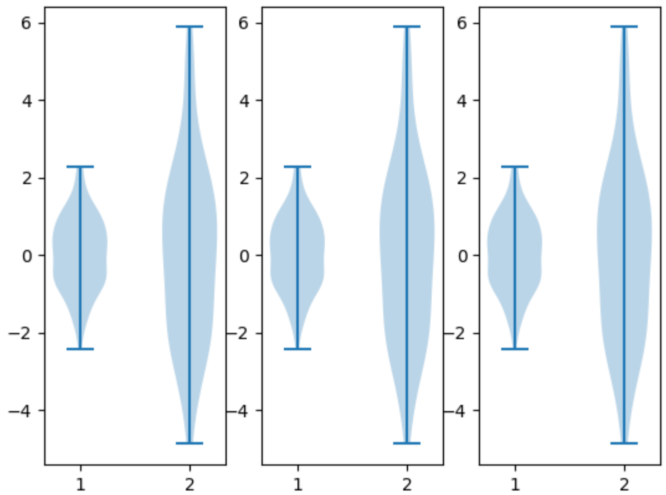
```
@image_comparison(baseline_images=['quiver_arrow_headwidth_3d'],
                  extensions=['png'], remove_text=True, tol=20)
def test_quiver_arrow_headwidth_3d():
    # sngle arrow with a custom headwidth in a 3d plot
    x,y,z = [0,0,10]
    dx,dy,dz = [0,10,0]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.quiver(x, y, z, dx, dy, dz, headwidth=0.1)
    ax.set_ylim(0,10)
```

Issue #8532

https://github.com/matplotlib/matplotlib/issues/8532

Background

This is a new feature. Matplotlib can create violin plots. Perpendicular lines can be drawn on these plots to mark the min, max, mean and median. This new feature will add the ability to draw lines on a user provided percentiles. Example of a violin plot marking the min and max is shown below.



Solution

Users can now pass in a Boolean “showpercentiles” (following the “show*” convention for mean, median etc…) and a list of percentiles to plot on the violin.

Based on the input data, we simple calculate the percentiles for each distribution and draw a perpendicular line at the calculated point. The functionality to draw a calculated point already exists, so the newer code calculates the percentiles and utilises the existing functionality to draw the lines

```
# Render percentiles
if showpercentiles:
    # Reorder the values to group via percentile rather than plot

    plt_percentiles = {}
    for single_plot in percentiles:
        counter = 0
        for single_val in single_plot:
            if counter not in plt_percentiles:
                plt_percentiles[counter] = []
            plt_percentiles[counter] += [single_val]
            counter+=1

    sorted_keys = list(plt_percentiles)
    sorted_keys.sort()

    # Once reordered, draw all n'th percentile lines
    counter = 0
    for key in sorted_keys:
        perc = plt_percentiles[key]
        artists['cpercentiles_' + str(counter)] = perp_lines(perc, pmins, pmaxes,colors=edgecolor)
        counter+=1
```

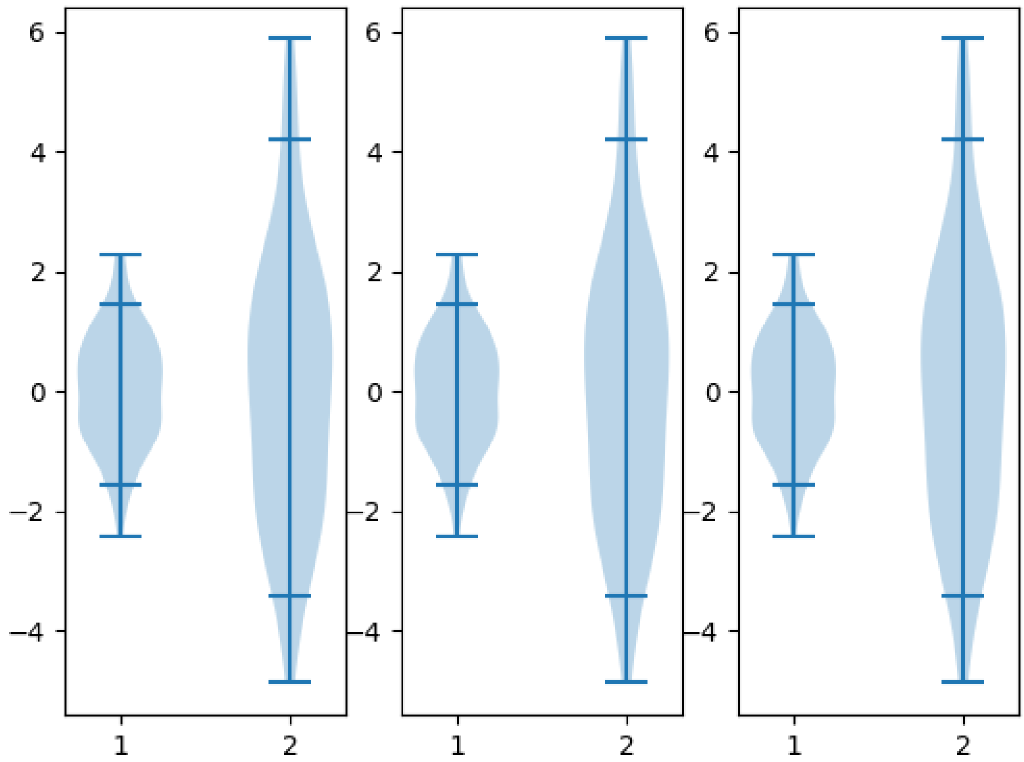
Code snippet from matplotlib.axes._axes.py

Acceptance test suite

Tests are based on image comparison (lib/matplotlib/tests/test_quiver.py)

The test creates a violinplot with random data and marks the 5th and 95th percentiles on the plot.

```
@image_comparison(baseline_images=['violinplot_percentiles_5_95'], remove_text=True, extensions=['png'], tol=1000)
def test_violinplot_percentile():
    pos = [1, 2]
    data = [[-0.63721969,  0.05942559,  0.19014301,  1.74227729,  0.84520328,   0.46434926,  0.91365678,  1.20994273,  0.87662343
    percentiles=[5,95]
    fig, axes = plt.subplots(nrows=1, ncols=3)
    axes[0].violinplot(data, pos, showextrema=True, showpercentiles=True, percentiles=percentiles)
    axes[1].violinplot(data, pos,showextrema=True, showpercentiles=True, percentiles=percentiles)
    axes[2].violinplot(data, pos, showextrema=True, showpercentiles=True, percentiles=percentiles)
```



Development Process

1. Each issue was created on the Team repo on Github and assigned to one of more team members.
2. Fixes for each issue were coded on separate branches.
3. Upon completion of coding a fix and creating test cases, pull requests were made for each branch.
4. The code was reviewed by at least one other team member and requested changes, if any, were incorporated.
5. Upon approval, pull requests were merged into the master branch.

Evidence of the above process can be found on the team repo (https://github.com/CSCD01/team14-project)