Closed-Source Development Deliverable 3

Justin Abrokwah
Yi Shiun Chiu
Spencer McCoubrey
Parth Thakkar

Features / Bug Fixes to Examine

Feature / Bug Fix 1: Issue #14661: [Feature request] Show quartiles (box) in violin plot

Description

The feature requested by the poster is an enhancement to the violin plot ability. A violin plot is a more detailed version of the more common box plot. It presents the mean, median, and interquartile range of a dataset, while also including its full distribution. Currently, the interquartile range is not displayed on the violin plot even though it is a key component of the plot. This feature request is a call to implement the ability, to provide an option to the user to display the quartiles. We thought this would be an appropriate problem to tackle for multiple reasons. First, our supervising TA took this course last year and worked on a feature request for extended functionality for the violin plot on the Matplotlib project, as well. His insight and experience with the plotting function can prove to be invaluable to us if we need a question answered or want feedback on a potential solution to our problem. The person who posted the request proposed a parameter called *showquartiles*, similar to other parameters in the *violinplot* function like *showextrema* and *showmedians***.** The way the function handles these parameters gives us guidance into how we should handle our new parameter, as well. Also, there has been discussion in the comments section of the request, also giving us a more detailed description of what is desired.

Design

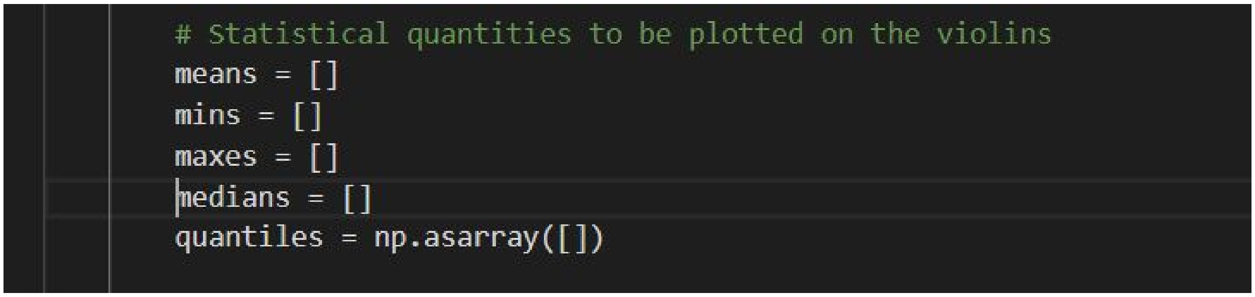
This feature would require modifying the existing violin plot function inside the *_axes.py* file to accept an additional parameter to allow for displaying the quartile values and interquartile range of the desired distribution. This feature currently exists in another library called seaborn which takes in a parameter called *inner* which can be set to “quart”, to display quartiles on the violin plot. A quartile is created by separating the dataset into four equal quarters and then mapping each of the quarters by using a dashed line in which the median is represented by a heavy dashed line. As a quartile is equal to a 4-quantile, we can use the current artist rendering of the quantiles and apply it to display the quartiles on the same graph.

Below we will list the assumed steps for implementation (assuming everything goes according to plan):

1. In lib\matplotlib\axes_axes.py, we will add a parameter to the *violinplot* function called *showquartiles* which will be a boolean value
2. In lib\site-packages\matplotlib\cbook\__init__.py, we will add another value to be returned in the *stats* dictionaries, kept in the *vpstats* list which is returned by the *violin_stats* function
    a. In the for loop where *X* (column of the dataset) is “zipped” to it’s *quantiles*, we will assign 2 new keys to our dictionary for *X*
        i. first_quartile = np.percentile(x, 25)
        ii. third_quartile = np.percentile(x, 75)

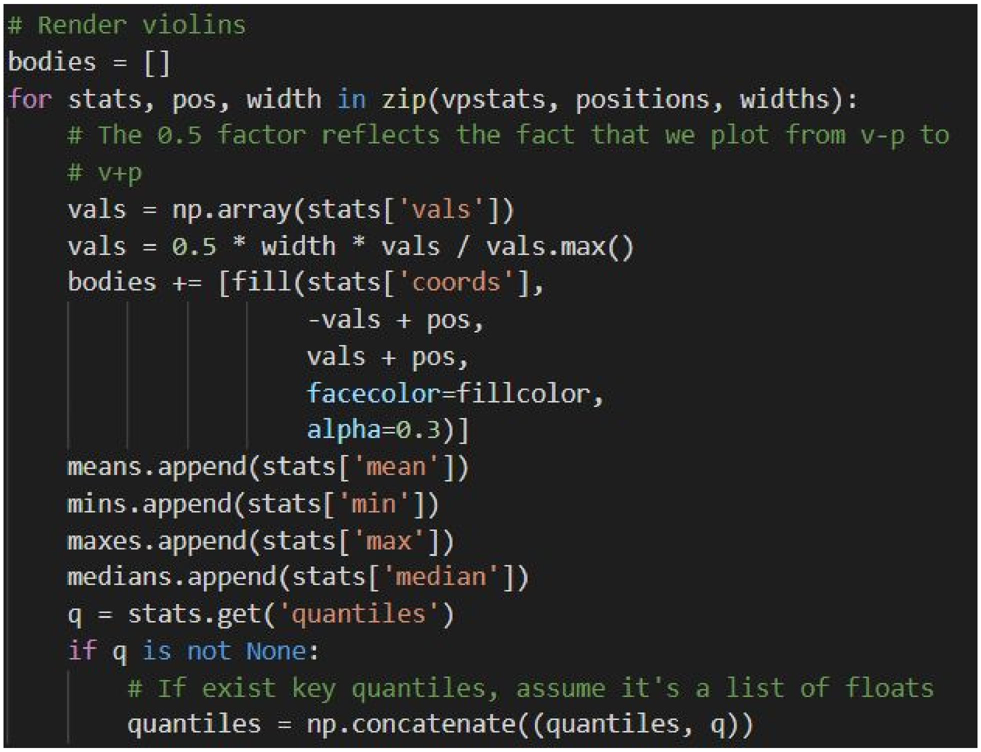
        iii. We don’t need to compute the median as that’s already a dictionary in the dictionary

3. When *violin_stats* returns our list of dictionaries (*vpstats*) to the *violinplot* function, we pass in *vpstats* to our *self.violin* function, as well as our *showquartiles* parameter to get the artists for the plot. We have to make these changes in *self.violin*:
    a. We must add a new empty array called quartiles to hold the quartiles for each column in the dataset



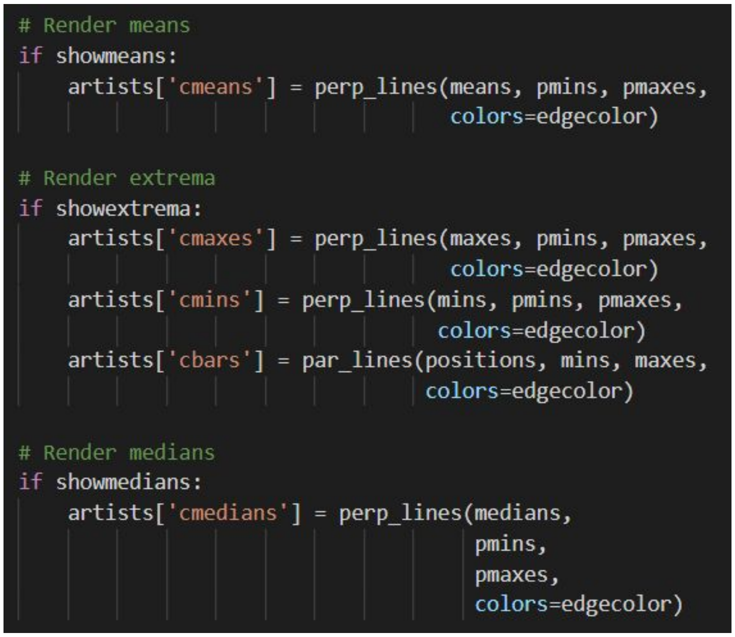
```
        # Statistical quantities to be plotted on the violins
        means = []
        mins = []
        maxes = []
        medians = []
        quantiles = np.asarray([])
```

    b. In the for loop, where we render the bodies of the violins, we must append the quartiles of the current column to our list, similarly done for means and medians:



```
# Render violins
bodies = []
for stats, pos, width in zip(vpstats, positions, widths):
    # The 0.5 factor reflects the fact that we plot from v-p to
    # v+p
    vals = np.array(stats['vals'])
    vals = 0.5 * width * vals / vals.max()
    bodies += [fill(stats['coords'],
                    -vals + pos,
                    vals + pos,
                    facecolor=fillcolor,
                    alpha=0.3)]
    means.append(stats['mean'])
    mins.append(stats['min'])
    maxes.append(stats['max'])
    medians.append(stats['median'])
    q = stats.get('quantiles')
    if q is not None:
        # If exist key quantiles, assume it's a list of floats
        quantiles = np.concatenate((quantiles, q))
```

    c. We check our *showquartiles* parameter and add a key to the artists dictionary being returned called ‘cquartiles’ (collection of quartiles) to remain consistent with the other keys and assign it a perp_line like in ‘cmedians’ in the screenshot below. We implement it as such based on the similar parameters, *showextrema*, and *showmedians*, also shown here:



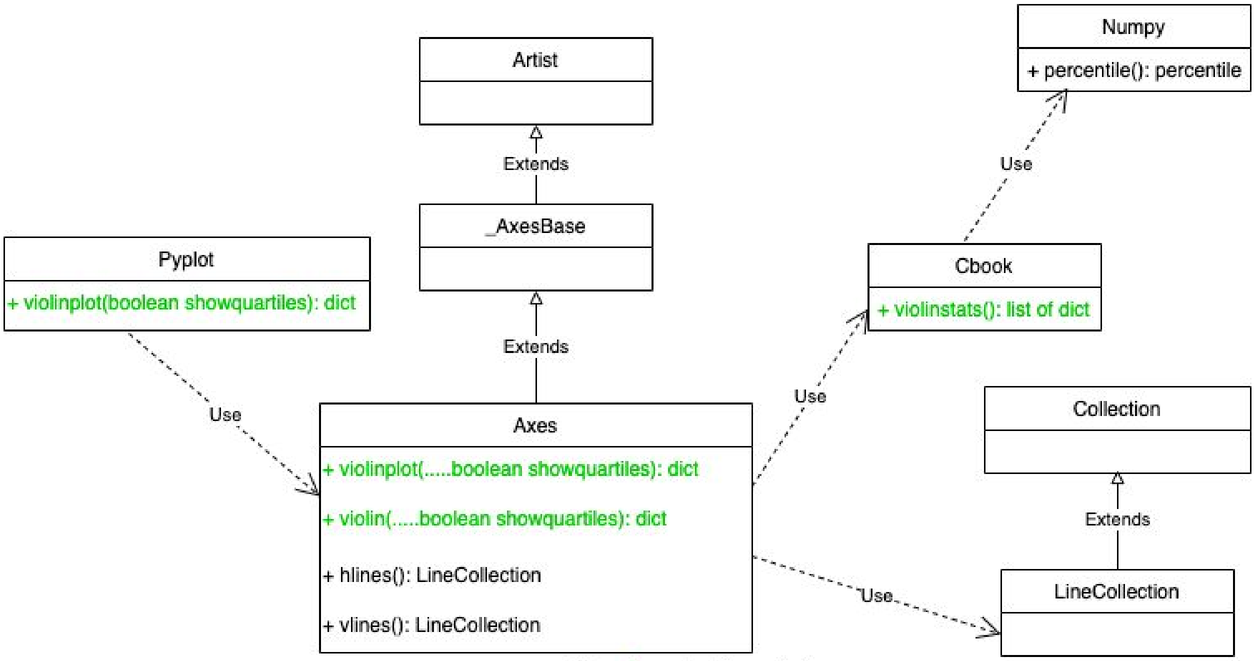
```
# Render means
if showmeans:
    artists['cmeans'] = perp_lines(means, pmins, pmaxes,
                                   colors=edgecolor)

# Render extrema
if showextrema:
    artists['cmaxes'] = perp_lines(maxes, pmins, pmaxes,
                                   colors=edgecolor)
    artists['cmins'] = perp_lines(mins, pmins, pmaxes,
                                  colors=edgecolor)
    artists['cbars'] = par_lines(positions, mins, maxes,
                                 colors=edgecolor)

# Render medians
if showmedians:
    artists['cmedians'] = perp_lines(medians,
                                     pmins,
                                     pmaxes,
                                     colors=edgecolor)
```

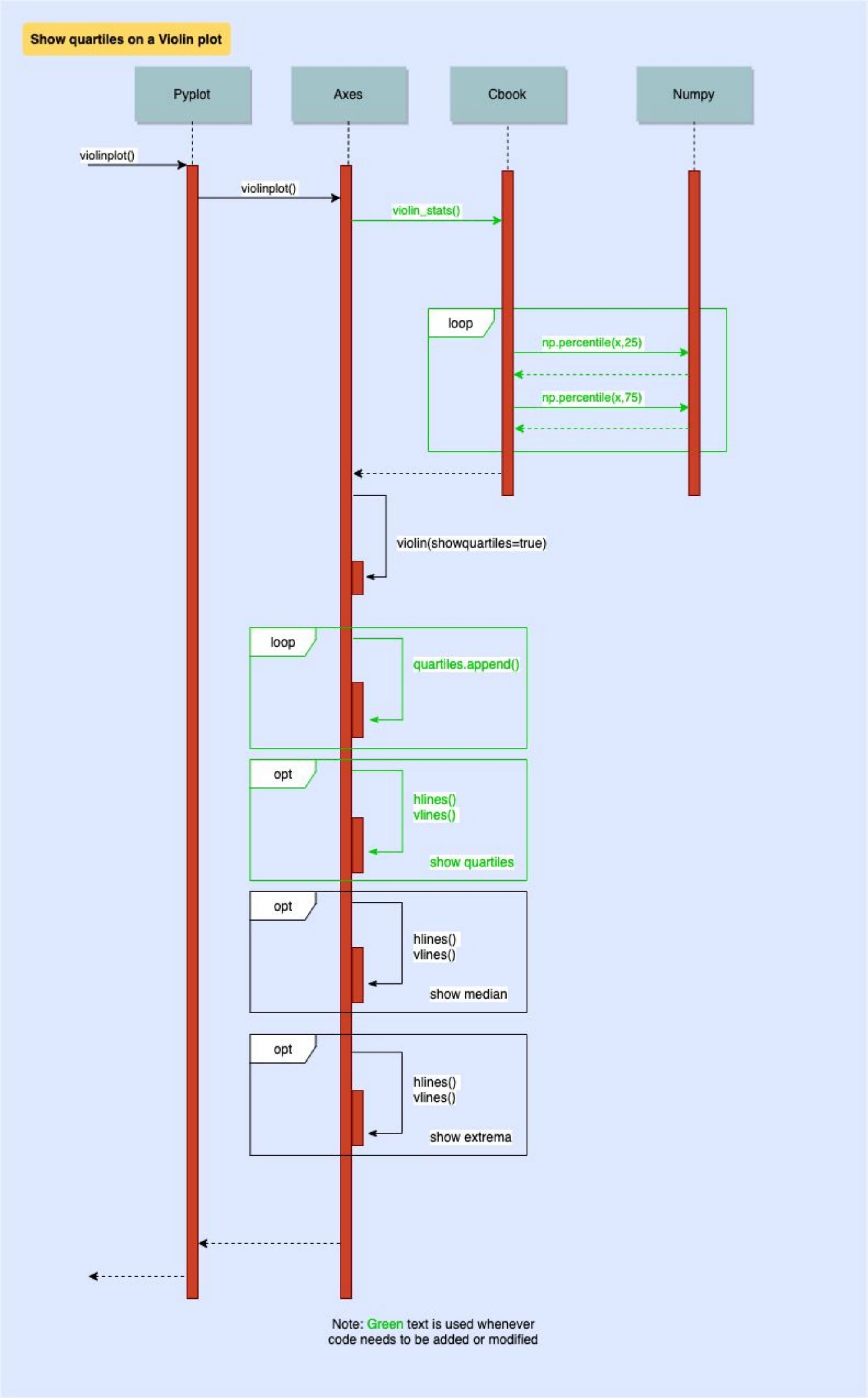
4. The PyPlot interface must be updated because of its strong relationship with the Axes class that the *violinplot* function resides in. Since PyPlot and Axes share an interface, PyPlot must be updated to acknowledge our changes.
5. Implement error checking on inputs appropriately and where needed
6. Implement test cases according to the acceptance test suite

UML Diagram


Artist
Extends
_AxesBase
Extends
Pyplot
+ violinplot(boolean showquartiles): dict
Use
Axes
+ violinplot(.....boolean showquartiles): dict
+ violin(.....boolean showquartiles): dict
+ hlines(): LineCollection
+ vlines(): LineCollection
Numpy
+ percentile(): percentile
Use
Cbook
+ violinstats(): list of dict
Use
Collection
Extends
LineCollection
Use

Note: Green text is used whenever code needs to be added or modified

Sequence Diagram / Code Trace



Note: Green text is used whenever code needs to be added or modified

Feature / Bug Fix 2: Issue #15066: [Feature request] stem3

Description

This feature request proposes the addition of a functionality in the area of 3d plotting, where a user will be able to plot a sequence of discrete data in 3D as a stem plot. Currently, Matplotlib only has this feature in 2D. Matlab, another technology, has this operation and it's called stem3. Again in the comments section, seasoned Matplotlib developers have provided insight on where to start with the feature and the poster provided a current workaround, also giving us a good look at what we would have to do.

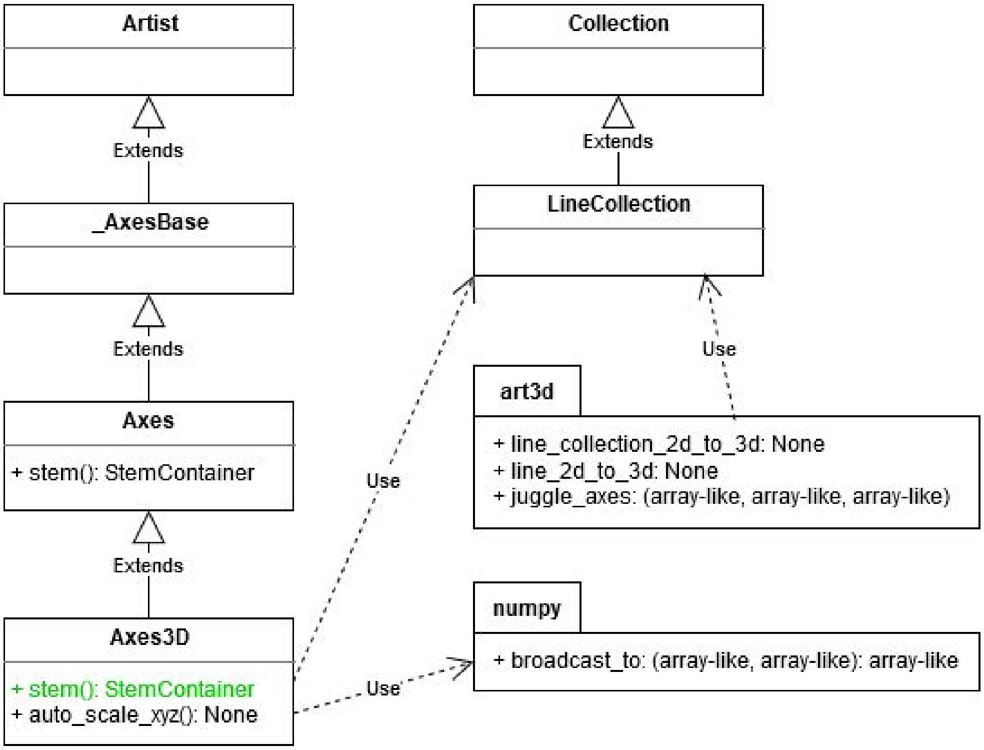
Design

The implementation design for adding a stem3D plot will follow very closely to the already existing implementations for plotting 3D scatter, bar graphs, and general plotting so that we aim to maintain consistency in the existing codebase. Luckily for us, there are several examples of porting 3D render logic while piggy-backing off of the 2D plotting interfaces as well as recommendations in the github issue itself. Below we will list the assumed steps for implementation (assuming everything goes according to plan):

1. Add a new *stem* method to the Axes3D class in **lib/mpl_toolkits/mplot3d/axes3d.py** whose parameters will be very similar to other 3D implementations. These parameters will include x-coordinates, y-coordinates, z-coordinates, plot orientation and any other arguments required to pass along to subsequent calls as is common in matplotlib (using kwargs and rags, etc)
2. Format the coordinate inputs such that they are of the same dimension for proper plotting using **numpy** and most likely the ***broadcast_to*** method in order to align the inputs accordingly
3. Delegate the parameters and inputs to the inherited ***Axes.stem*** method
4. For each returned line from the ***Axes.stem*** call, transmute them into 3D lines using the **art3d** package. This will either use the ***line_2d_to_3d*** or ***line_collection_2d_to_3d*** method depending on implementation choices
5. Call the ***Axes3D.auto_scale_xyz*** and ***art3d.juggle_axes*** methods to properly format the outputted line segments
6. Implement error checking on inputs appropriately and where needed
7. Implement test cases according to the acceptance test suite

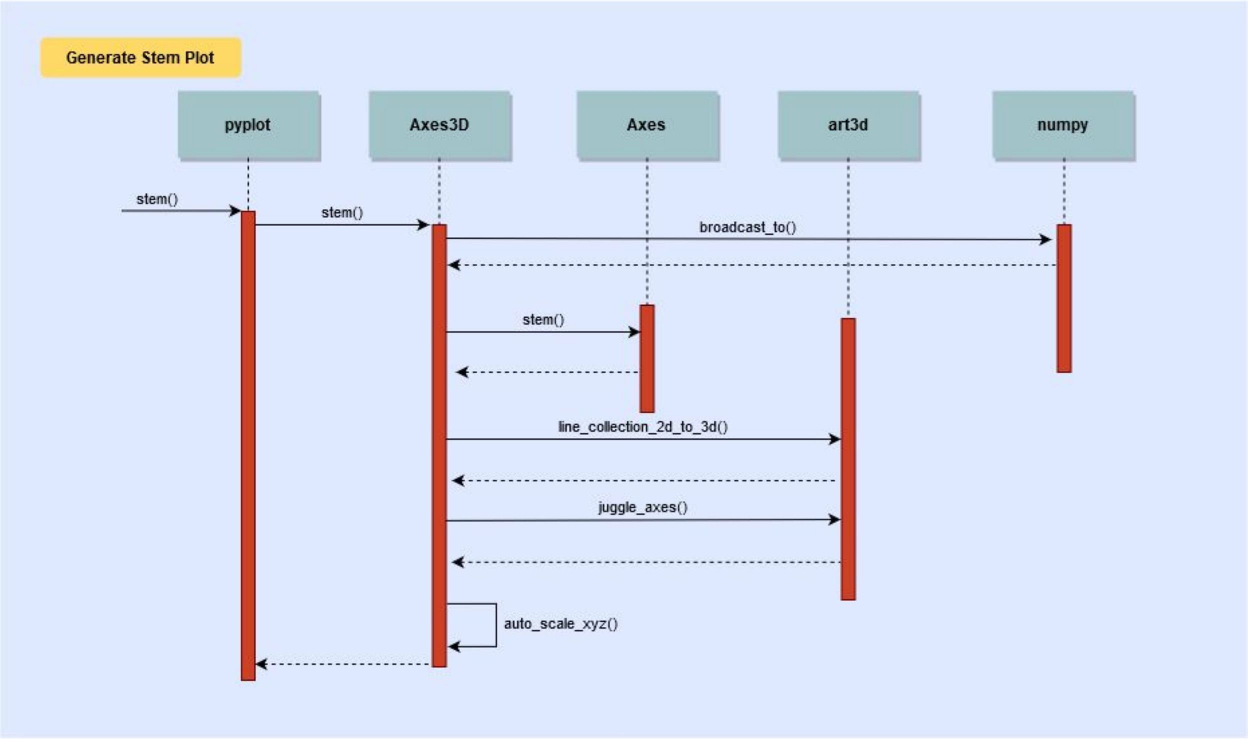
UML Diagram

This diagram contains the low-level architecture for the areas of matplotlib that will be used or affected by the change in implementing issue 15066. Note that not all methods an relationships are shown, only the ones that are relevant or will be used or midified during the implementation



Note: Green text is used whenever code needs to be added or modified

Sequence Diagram / Code Trace



Chosen Feature to Implement

Issue #14661: [Feature request] Show quartiles (box) in violin plot

We have decided to move forward and implement issue #14661: showing quartiles on a violin plot. Our reasons for doing so are as follows:

- The feature-request is clearly defined and outlined in the issue which enables us to design accurate and concise acceptance tests; this will lead to a more testable and polished final product that is more likely to meet matplotlib's requirements and be accepted
- There has been some discussion in the issue itself in terms of implementation details and requirements, further strengthening our understanding and acceptance criteria of the feature
- The feature is fairly self-contained to the *violinplot* interface and its 2 dependent functions; this implies that regression issues will be less of a problem when it comes time to test compatibility
- The functionality itself mirrors existing functionality such as the *showextrema* parameter; this gives us a solid foundation as to how we should go about implementing the issue according to matplotlib's style
- Our TA for this project actually opened and had a PR accepted in the exact same set of functions and interfaces; when it comes time to make a PR and submit to matplotlib, our TA will be a valuable asset in giving us advice; also if we run into issues while implementing or are confused by the code, he might be able to clear up our issues to allow us to continue moving forward
- There is an existing comprehensive test suite exactly for this set of interfaces, as such testing regression will easily allow us to determine if we broke existing functionality

Implementation plans for our selected issue can be found in the above section detailing this specific issue.

Acceptance Test Suite

For our implementation, we have to perform a number of tests to validate our code's correctness. For all our tests, we have to construct datasets and provide positioning to the *violinplot* function. We will pass in our new parameter *showquartiles* as True. Using the *violinplot* function, we will render our plot and check whether the quartiles are drawn in the right places. To verify, in another violin plot with the same dataset and positioning, we set our *showquartiles* to False, and we pass in quantiles of 0.25, 0.5, and 0.75. Then we can compare the 2 graphs using the image comparison feature used in numerous Matplotlib tests. Pre-existing test cases using the functions we modify should still pass, as well. Since horizontal and vertical violin plots are possible, we must test our feature against both orientations of a violin plot. We also change the *violin* and *violin_stats* function so we will make test cases to verify that the right values are being returned

Test Case Format

- [2x for vertical and horizontal violin plots] Construct random dataset with 1 column and create 2 violin plots to see how the 2 plots compare
    - 1 with showquartiles set to true, quantiles = []
    - 1 with showquartiles set to false, quantiles = [0.25, 0.5, 0.75]
- [2x for vertical and horizontal violin plots] Construct random dataset with 1 column and create violin plot to see how it compares when the quantiles are set to the same quartiles
    - showquartiles set to true, quantiles = [0.25, 0.5, 0.75]
- Construct random dataset and input into the *violin_stats* function
    - Verify the list of dictionaries returns the first and third quartiles in each entry of the dictionary and that the values are correct
- Construct random dataset and input into the *violin* function with *showquartiles* being set to true
    - Verify the artists dictionary returned, consists of an entry called 'cquartiles' and it contains the correct values