COMMIT TO MASTER

Deliverable 3

CSCD01

Mahima Bhayana
Kalindu De Costa
Victor Lee
Harman Wadhwa
Leo Yao

Table of Contents

Feature: Lack of built-in Broken Axis support

Implementing

GitHub issue: https://github.com/matplotlib/matplotlib/issues/11682

Description

Users of matplotlib would like to see built-in support for Broken Axis. An axis break (Figure 1) is a disruption in the continuity of values on the axis on a figure. It is also known as a scale break or graph break and is shown on the chart as a wavy line or diagonal line on the axis and on the bars plotted on that axis. Often it is important to trim out irrelevant information from a plot to show regions of interest, which lie far apart from each other, in high detail, in a way that they can be easily compared.

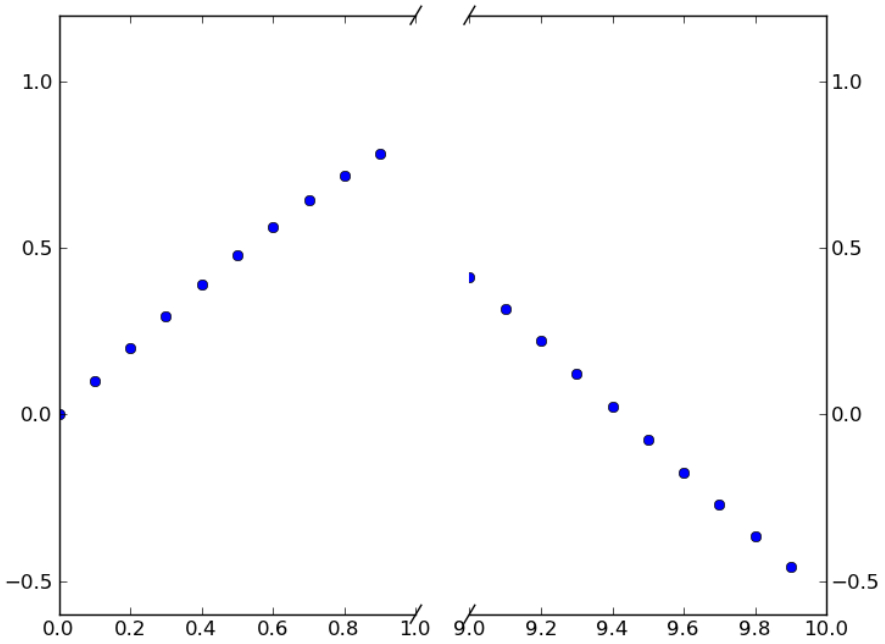

Figure 1: Axis Break

As described by the feature requestee, the solution to this problem is to put breaks in the axis lines (spines) to indicate the jump in magnitude.

The current solution to this feature is described as disappointing as it seems to be making additional subplots (Figure 2), erasing the spines and plotting fake lines to achieve the axis breaks.

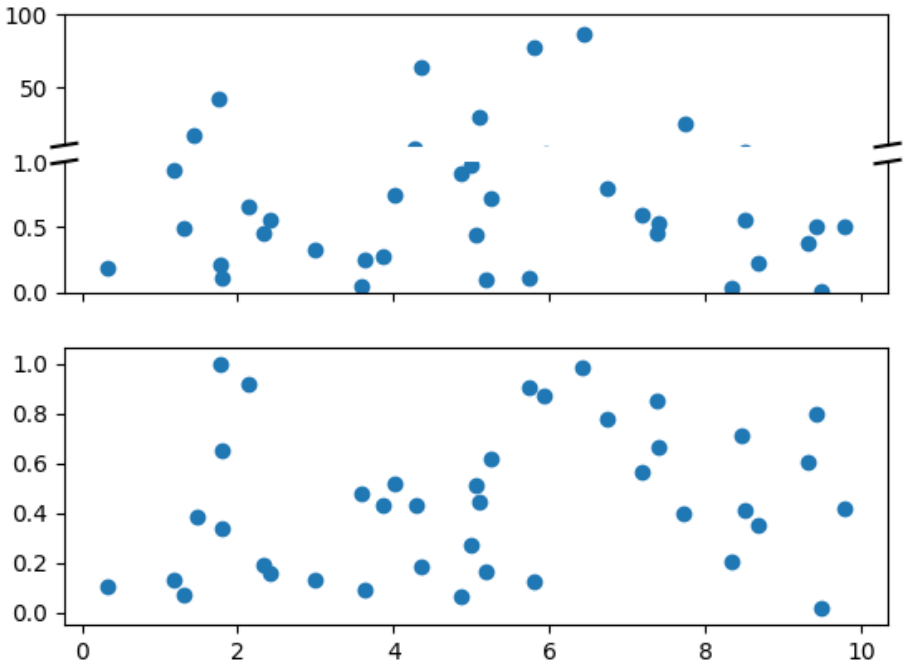

Figure 2: Additional Subplots

UML

A UML representation of the newly created AxisBreak class, along with the classes that would be needed to create a graph with breaks in between axes (Figure 3).

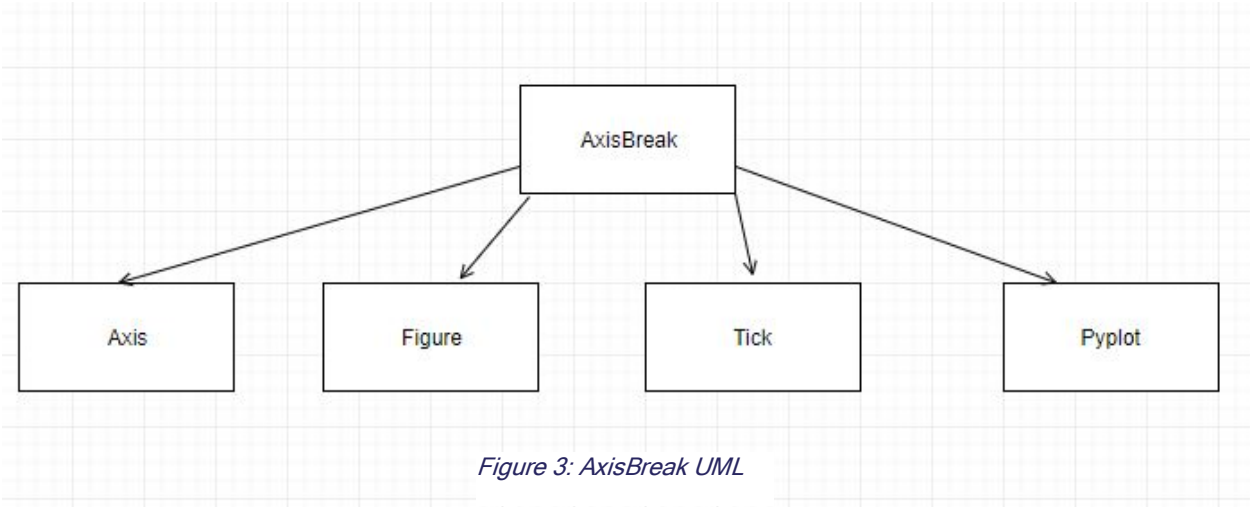

Figure 3: AxisBreak UML

Implementation strategy

The proposed solution by the requestee involves creating a custom Axis subclass with its own Transform subclass. It will need to introduce breaks in the axis (spine) which is the critical element to this feature (Figure 4). Furthermore, any solution that goes into the matplotlib codebase would need to make sure not only the axis is broken, but also any content in it. For instance, the current method used by many users for this issue is to use two axes to create such plots, because then the clipping is done automatically.

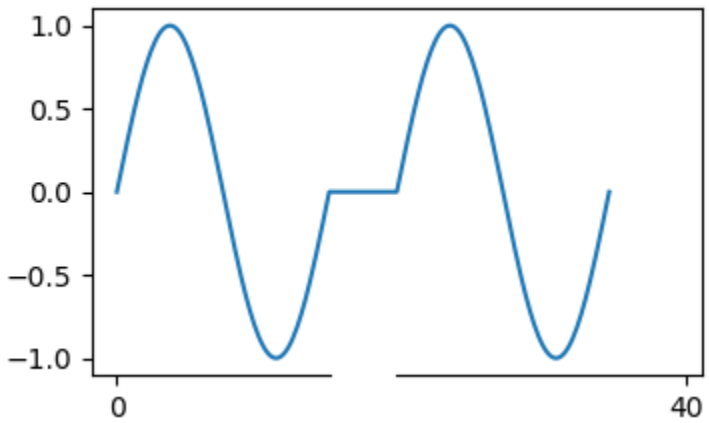

Figure 4: Breaks in Axis

Going forward, much of the strategy to implement this feature would rely on creating two subplots while maintaining the parent container-axes and making sure that it is passed on to the figure as an axes. This would make sure that tight_layout/contrained_layout to behave normally.

Our implementation of this feature will create a wrapper class with the use of the Facade design pattern to generate a plot with broken axis. The class **AxisBreak** will accept 3 parameter, ***xbreak,*** ***ybreak*** and ***color***. The parameters ***xbreak*** and ***ybreak*** will specify the range of values to break on a given axis (tuple), they are optional parameters. Finally the ***color*** parameter will define the color of the break lines on the axis, it is also an optional parameter. Since the class is a wrapper class, all inner methods of pyplot and axes will be delegated to through a backwards method caller to avoid unnecessary attributes for the **AxisBreak** class.

Acceptance Tests

Blackbox testing from the point of view of the user. Since our implementation relies on the Facade design pattern to create a wrapper class around existing matplotlib features, many of the unit tests that rely on image comparisons can be treated as the acceptance test suite. Tests such as the *One broken Axis (x-axis/y-axis)* is essentially written from the point of view of the user to determine whether an axis is properly broken. Much of the implementation relies of visual features and so its data structure cannot be tested directly without some sort of visual comparison.

Unit tests

We have 6 unit tests in total to test the our implementation (more may be added as the implementation grows)

1. **One broken Axis (x-axis/y-axis)**

The purpose of this test is to ensure that the broken axis feature is functional for the both x-axis and the y-axis independently. The test is using the image comparison feature of matplotlib to compare a correct representation of a broken axis plot to one that is generated using our implementation. The test will first create a figure and break the x-axis/y-axis for a certain range on the axis.

```
15  @image_comparison(baseline_images=["axis_break_yaxis"])
16  def test_axis_break_yaxis():
17      fig = plt.figure()
18      br_ax = AxisBreak(ybreak=(30,100))
19      x = np.linspace(0, 1, 100)
20      br_ax.plot(x, np.sin(10 * x))
21      br_ax.plot(x, np.cos(10 * x))
22
```

2. **Multiple broken Axis (both x-axis and y-axis)**

The purpose of this test is to ensure that the broken axis feature works when both the x-axis and the y-axis have a break in them. The test uses image comparison to compare an expected output of a broken axis plot to the one that the feature generates. The test creates a figure and breaks both the x and y axes for certain ranges, respectively.

```
31  @image_comparison(baseline_images=["axis_break_x_y_axes"])
32  def test_axis_break_x_y_axes():
33      fig = plt.figure()
34      br_ax = AxisBreak(xbreak=(30,100), ybreak=(0,0.25))
35      x = np.linspace(0, 1, 100)
36      br_ax.plot(x, np.sin(10 * x), label='sin')
37      br_ax.plot(x, np.cos(10 * x), label='cos')
```

3. Different types of plots (bar/scatter)

The purpose of these tests is to ensure that the broken axis feature works for different types of graphs, and that it performs correctly for different types of graphs (e.g. only breaking the x axis in vertical bar graphs, and the y axis in horizontal bar graphs). The tests use image comparison to compare the expected output to the one generated by our implementation. The tests create a figure and break axes for certain ranges.

```
39  @image_comparison(baseline_images=["axis_break_bar_vert"])
40  def test_axis_break_bar_vert():
41      fig = plt.figure()
42      br_ax = AxisBreak(ybreak=(30,100))
43      objects = ('test1', 'test2', 'test3', 'test4')
44      y_pos = np.arange(len(objects))
45      data = [300,10,5,25]
46
47      br_ax.bar(y_pos, data, align='center', alpha=0.5)
48      br_ax.xticks(y_pos, objects)
49
50  @image_comparison(baseline_images=["axis_break_bar_horz"])
51  def test_axis_break_bar_horz():
52      fig = plt.figure()
53      br_ax = AxisBreak(ybreak=(30,100))
54      objects = ('test1', 'test2', 'test3', 'test4')
55      y_pos = np.arange(len(objects))
56      data = [300,10,5,25]
57
58      br_ax.barh(y_pos, data, align='center', alpha=0.5)
59      br_ax.yticks(y_pos, objects)
```

4. Color of the axis break line

The purpose of this test is to ensure that the color of the break lines can be changed through the **AxisBreak** class. For instance calling **AxisBreak** with *color='blue'* should change the break lines become blue in color. The test uses image comparison to compare the expected output of a colored break line to one that is generated through our implementation. The test creates a figure, breaks it's axis and changes the color of the break lines.

```
@image_comparison(baseline_images=["axis_break_color"])
def test_axis_break_color():
    fig = plt.figure()
    br_ax = AxisBreak(xbreak=(0.1,0.5), ybreak=(0,0.25), color='blue')
    x = np.linspace(0, 1, 100)
    br_ax.plot(x, np.sin(10 * x), label='sin')
    br_ax.plot(x, np.cos(10 * x), label='cos')
```

5. Invalid input

The purpose of this test is to ensure that some wrong plots do not register as successes due to some unexpected cases. The success case for these tests is the expected throwing of the ImageComparisonFailure exception. (i.e. If the ImageComparisonFailure exception is thrown, the test case will pass. If any other exception OR no exception is thrown, the test will fail.) This action is achieved by checking the @pytest.mark.xfail(raises=ImageComparisonFailure) check above the function.

```
# Wrong Output Tests

@pytest.mark.xfail(raises=ImageComparisonFailure)
@image_comparison(baseline_images=["axis_break_yaxis"])
def test_axis_break_yaxis():
    fig = plt.figure()
    br_ax = AxisBreak(ybreak=(0,0.5))
    x = np.linspace(0, 1, 100)
    br_ax.plot(x, np.sin(10 * x))
    br_ax.plot(x, np.cos(10 * x))

@pytest.mark.xfail(raises=ImageComparisonFailure)
@image_comparison(baseline_images=["axis_break_xaxis"])
def test_axis_break_x_axis():
    fig = plt.figure()
    br_ax = AxisBreak(xbreak=(60, 100))
    x = np.linspace(0, 1, 100)
    br_ax.plot(x, np.sin(10 * x), label='sin')
    br_ax.plot(x, np.cos(10 * x), label='cos')

@pytest.mark.xfail(raises=ImageComparisonFailure)
@image_comparison(baseline_images=["axis_break_x_y_axes"])
def test_axis_break_x_y_axes():
    fig = plt.figure()
    br_ax = AxisBreak(xbreak=(50,100), ybreak=(0,0.5))
    x = np.linspace(0, 1, 100)
    br_ax.plot(x, np.sin(10 * x), label='sin')
    br_ax.plot(x, np.cos(10 * x), label='cos')
```

6. Acceptance tests

These tests are meant to test real world cases instead of individual components of the feature. Hence, these cases test different types of commonly used plots like histograms, streamplots and scatter plots. These are meant to follow a black box testing style where we only care about the appropriate output for any given input.

```python
@image_comparison(baseline_images=["axis_break_streamplot"])
def text_axis_break_streamplot():

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

    # streamplot of data
    fig = plt.figure()
    br_ax = AxisBreak(ybreak=(30,100))
    br_ax.streamplot(X, Y, U, V, color=U, density=0.6, linewidth=2, cmap=plt.cm.autumn)
```

```python
@image_comparison(baseline_images=["axis_break_historam"])
def test_axis_break_hist():

    # data
    np.random.seed(19680801)
    mu, sigma = 100, 15
    x = mu + sigma * np.random.randn(10000)

    # the histogram of the data
    fig = plt.figure()
    br_ax = AxisBreak(ybreak=(30,100))
    n, bins, patches = br_ax.hist(x, 50, density=True, facecolor='g', alpha=0.75)
    plt.grid(True)
```

```python
@image_comparison(baseline_images=["axis_break_scatter"])
def test_axis_break_scatter():
    # Data
    N = 500
    x = np.random.rand(N)
    y = np.random.rand(N)
    colors = (0,0,0)
    area = np.pi*3

    # Plot
    fig = plt.figure()
    br_ax = AxisBreak(ybreak=(50,100), ybreak=(0,0.25))
    br_ax.scatter(x, y, s=area, c=colors, alpha=0.5)
```

Feature: more operations on collections

Not Implementing

GitHub issue: https://github.com/matplotlib/matplotlib/issues/7600

Description

Users of matplotlib would like to see more operations that can be carried out on the Collections class. Specifically, operations such as merge/union between other sets of collections which can utilize the '+' operator of python. Grouping is another operation which will allow users to group several collections together and apply the same operation across all of them. Such operations include but are not limited to, setting properties, transformations, etc.

UML

A UML representation of the Collection class, along with all its subclasses and how it is used by the Axes class (Figure 5).

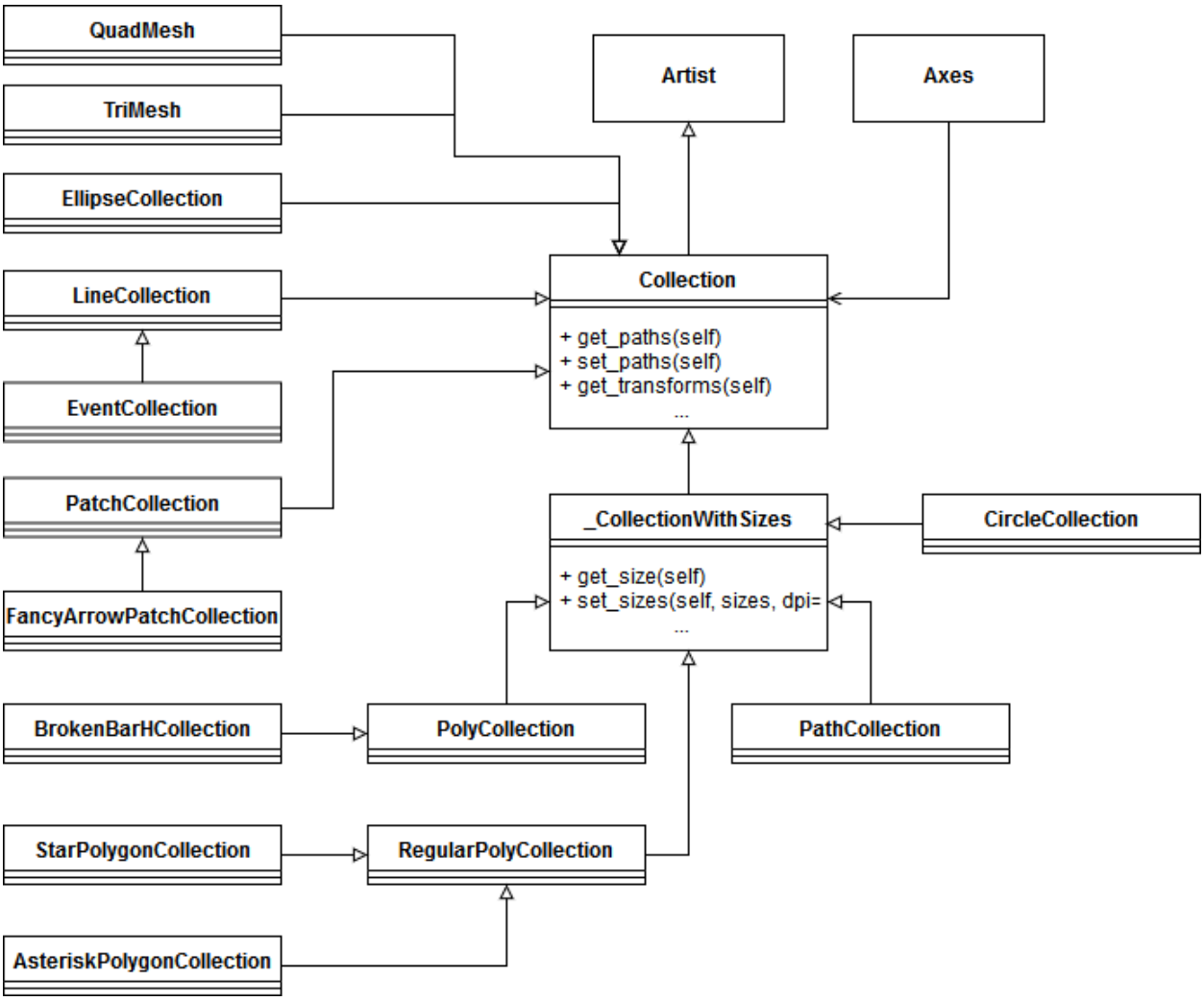

Figure 5: Collection Class UML

Use case: The following use case was described by the Feature requestee.

We currently work around these issues as follows:

1. Create Polygon for background, set face color, set clip path
2. Create Patches via routine for background, add to Collection, set same face color, set clip same path
3. Create Patches via routine for foreground, add to Collection, set other face color, set clip same path
4. (Shallow) copy **background (1.), collection (2.) and collection (3.)** three times each
5. Apply **same transform to all shapes (foreground, background)** for every copy
6. Set clip path **for all shapes** for every copy
7. Change face color to **distinct colors** for every copy

What we would like to do instead

1. Create Polygon for background
2. Create Patches via routine for background, add to Collection
3. Create Patches via routine for foreground, add to Collection, set other face color
4. **Merge Polygon and Collection** for Background, set same face color
5. **Group collections (3., 4.)**, set clip path
6. (Shallow) copy **groups**
7. Apply transform for every copy
8. **Invert colors** for every copy

Implementation strategy

Although at first this may seem relatively straightforward to implement within the Collections classes, it is much more complicated due to how the draw method of Artists are implemented within matplotlib. To provide an exact API for this feature, a new Artist class will need to be written where the internal representation and what gets sent to the renderer in the backend is much more adaptive to such operations either by deferring to the draw methods of children or by directly calling the render methods.