University of Toronto – Scarborough
CSCD01 – Engineering Large Software Systems

Deliverable 04 – Report

Prepared by ***GoonSquad*** Team

Swarnajyoti Datta
Nikki L. Quibin
Junil Patel
Beiyang Liu
Laine London
Hajoon Choi
Leo Li

Date: March 16, 2018

Table of Contents

Bug #1: Legend Annotate (Fixed)

Bug/Issue: Legend does not show ‘annotate’ #8236

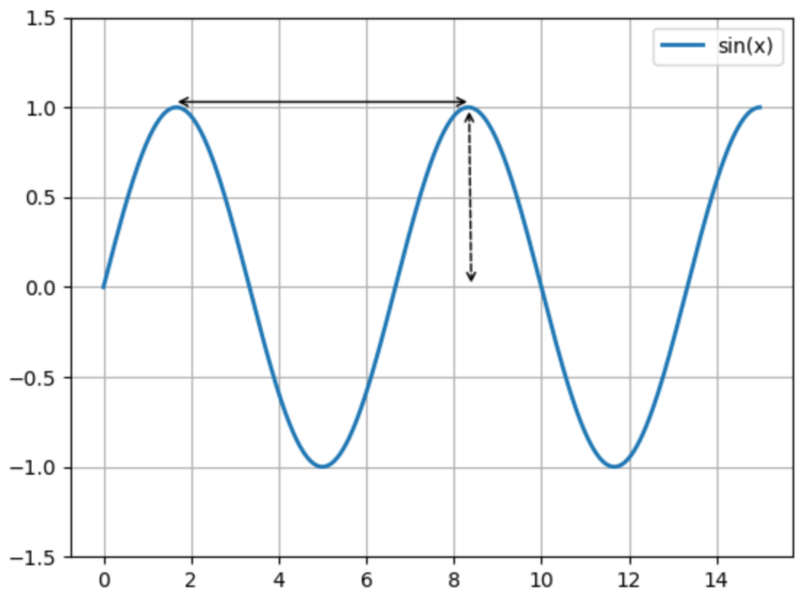
Estimated Hours:

- Explore and create a solution (7 h)
- Implement solution (20 h)
- Testing/validation (12 h)
- Code Review (3 h)
- Documentation (1 h)
- *Total: 43 h*

Description:

Legend items for annotations are currently not operational despite inputs currently being legal for them. Boxing the bi-directional arrows above into a legend to denote amplitude and wavelength is an example of a desirable use case.

When the legend() method on an Axes object gets called, the program eventually adds items to a list of handles to be inserted into the Legend for that Axes (see legend.py:1308, 1313). Annotations, stored in the Axes field texts (as Annotations are a sub-class of the Text class), are not currently added to this list. Additionally, the handler to construct the legend items for Annotations and Texts do not currently exist in the file legend_handler.py (and are subsequently not mapped in legend.py:805).



Solution:

Updated Forked Repo

The affected files are ***legend.py***, and ***legend_handler.py*** where the updated files are found in ***solutions/legend_annotate/***.

The solution to this bug follows precisely the original plan for it from deliverable 3. A handler for the legend Annotation was added and it was mapped to Annotation objects like the other legend items. It additionally makes use of handlers for Texts and Arrows (that we also added), because annotations can be composed of texts and/or arrows which are themselves separate entities. Text objects (super class of annotations) were added to the list of objects to be added to the legend and import statements were added where necessary.

As for composing the actual legend item, all types of annotations have been broken down into 3 cases: blank, text, arrow (with or without text). Blank annotations show up as blank on the legend. Making the legend omit adding this item by default would require going outside of this self-contained solution and this presents a niche use case, anyways. A user making a blank annotation constitutes a very specific action and if they wanted to hide it, they could just add the appropriate argument label onto the annotation. Otherwise, it would likely be intended behaviour on their part. Texts, again, are only omittable by user action for the same reasons. The legend item will be a replication of the text in the original font, colour, and style (scaled to an appropriate size). Texts that exceed the width of the legend item field will simply be cut off. This was chosen because sizing down texts was deemed to be more useless to the end user if they cannot read it in the first place. Annotations with arrows are perfectly replicated into the legend area. Colour, style, and thickness of different components are respected.

There is a competing pull request with regards to this bug. As explained, an unrelated contributor worked on this a year ago and left it inactive after failing to pass error tests. Recently, it was picked up by who we suspect are fellow D01 students. As such, we spent time running and profiling their solution to see what we were up against in terms of winning the pull request.

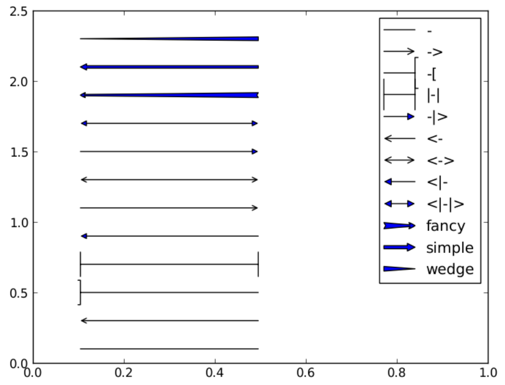
Our solution differs in that texts are represented based on the actual text string even when cut off while the other one replaces the text with a base string set (“Aa” or similar) in the same styling as the original when exceeding a certain limit character limit (the limit is often premature and depends on a character limit rather than the actual space available). We do not like this because there is no way to differentiate between long texts that have the same style in every other respect other than the actual string. Additionally, we have a strict improvement when handling arrows. The “linestyle” property of FancyArrows is not respected in the other solution and thus does not reflect certain aspects of arrows (like whether it is dashed). The other solution chooses to make “text + arrow” annotations a special case and displays both the text (usually in base text form “Aa”) and arrow within the small confines of the legend. We do not believe this is useful for anyone for the same reasons we do not like how texts are handled and because it takes away emphasis from the arrow (reducing its size and identifiability). Another difference is that we also decided not to legend Text and Arrow items on the plot outside of actual Annotation objects (by not mapping the handlers for them to legend creation). This was not the purpose of the bug fix nor do we see any useful functionality for it that wouldn’t interfere with normal operation. The default nature of entries being added to the legend unless explicitly argued against can prove to be annoying for users who simply want to add simple texts and arrows without having them showing up on the legend. We think that making it so that Texts/FancyArrows in Annotations being added to the legend by default and raw Texts/FancyArrows not being added to the legend by default offers the most flexibility without having to resort to omission arguments on the part of the user. If a user wants to make an arrow/text to show up on the legend, they just need make it an Annotation. However, this functionality, should anyone choose to do so, can be trivially added onto our solution with the addition of 2 lines of code in the ***_default_handler_map*** object in legend_handler.py with appropriate import statements. The handlers we implemented in for Texts and FancyArrows as a side effect of the Annotation handler will already suffice for this simple addition. Overall, we think we have a more complete and well-thought out solution.

Testing:

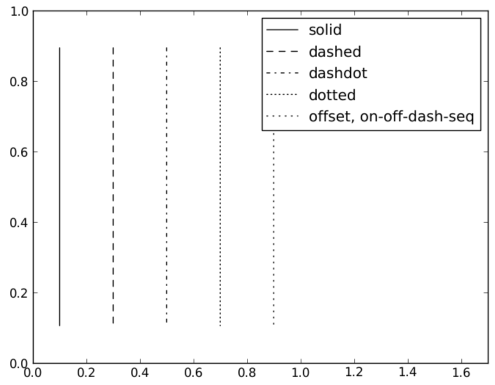
To test that the solution works, image comparison tests were used, as suggested by matplotlib when testing for changes to the graph figure. The associated tests are in ***solutions/legend_annotate/tests/***. The file ***test_legend_annotate.py*** contains the specific test cases for annotations appearing in the legend. The result images are found in the folder ***/test_legend/*** and should be copied over to ***lib/matplotlib/tests/baseline_images/test_legend*** in the actual matplotlib source directory when running tests. In addition, ***test_legend.py*** is the formal way to test the legend, thus it includes the existing tests for legend and as well as the newly added test cases in ***test_legend_annotate.py***. As a result, when testing, this ***test_legend.py*** should replace the existing matplotlib file.

Since the issue pertains to annotations not appearing in the legend, it is a rendering issue, so image comparison tests are used. Moreover, existing legend tests used image comparison tests for testing the labels, for example, ***test_various_labels()***.

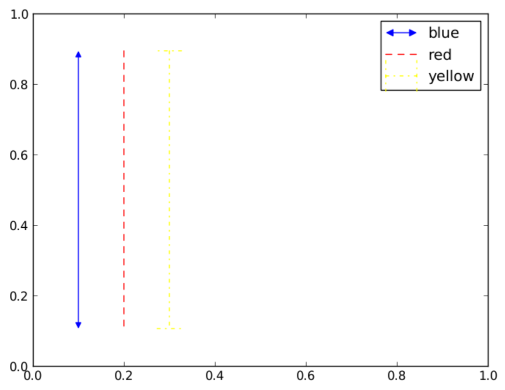
There were six tests used. Below are the images used as the baseline images. The test cases cover all the existing linestyles, arrowstyles, couple of colours and texts, no arrow, text, or both, and a practical example of using annotations. Also, the existing tests were ran and have passed.



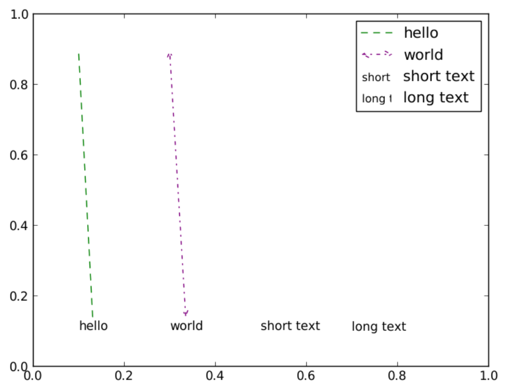
test_all_arrowstyles.png



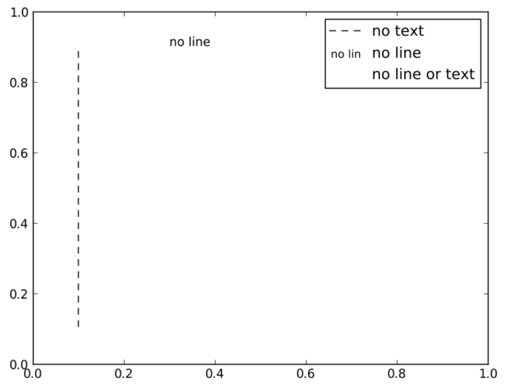
test_all_linestyles.png



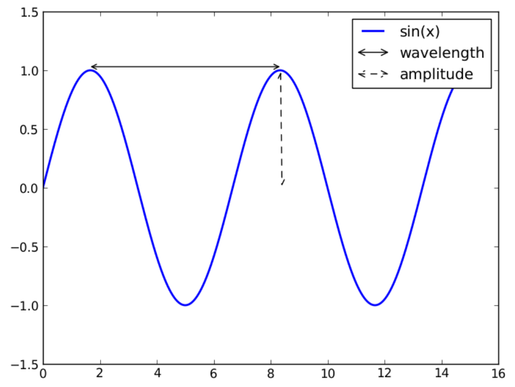
test_annotation_colours.png



test_annotation_text.png



test_annotation_no_line_text.png



test_simple_annotation.png

Confidence in Solution:

The proposed solution works very well since existing and new tests have passed. In addition, there are only two affected files and are only contained with respect to the legend, thus other parts of the matplotlib code are unaffected. Moreover, the implemented solution is a valid and practical fix as opposed to a “hack” fix. The solution consists of using the ***legend_handler*** class to add the annotation to the legend, which was also the procedure for adding existing labels. Guidelines for contributing to matplotlib were also followed accordingly, such as proper documentation and following PEP8 guidelines. All in all, there is high confidence that the proposed solution works.

Bug #2: Bbox Tight Legend (Fixed)

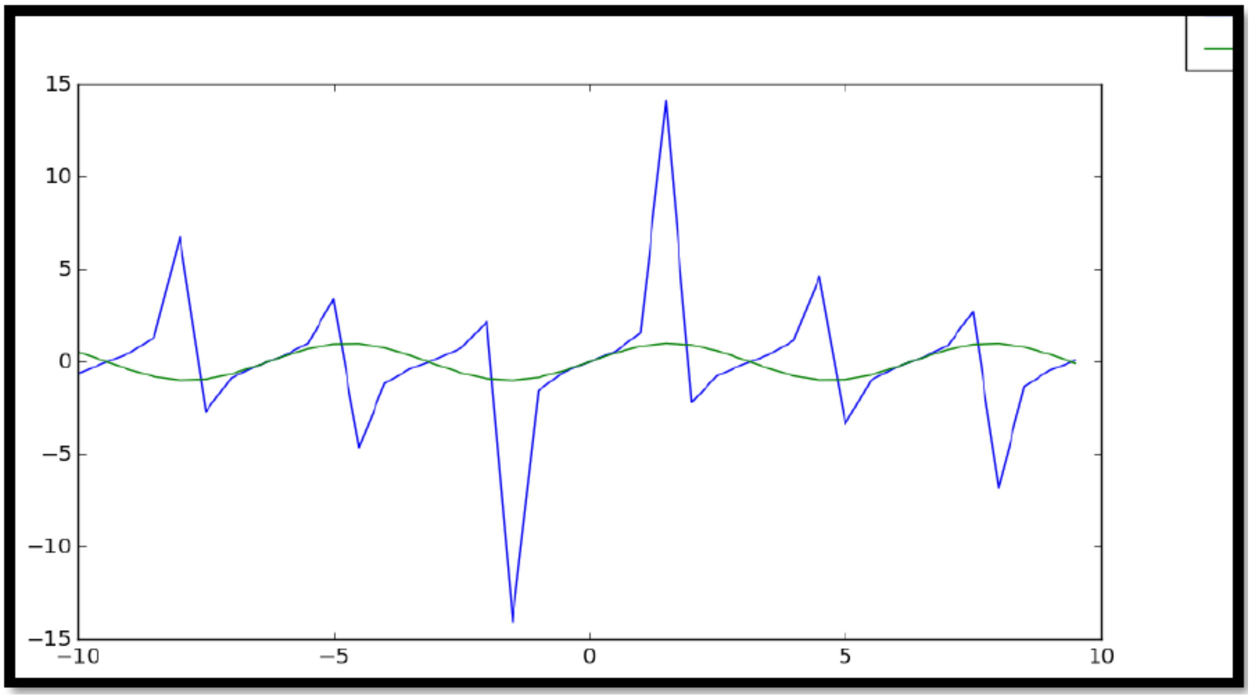
Bug/Issue: Legend is not present in the generated image if I use “tight” for bbox_inches #10194

Estimated Hours:

- Explore and create a solution (30 h)
- Implement solution (10 h)
- Testing/validation (2 h)
- Code Review (4 h)
- Documentation (2 h)
- *Total: 48 h*

Description:

As shown in the figure below, when ***bbox_anchor*** is used along with the ‘tight’ property of ***bbox_inches***, the created legend gets cut off in the outputted file. The ‘tight’ property enables users to reduce the size of the whitespace in the outputted figure but in the process, the figure’s legend is omitted. This results in a figure that lacks important information. This rendering is also inconsistent with using the ***loc*** property, wherein the legend is not omitted in the outputted file, and a detailed figure is produced.

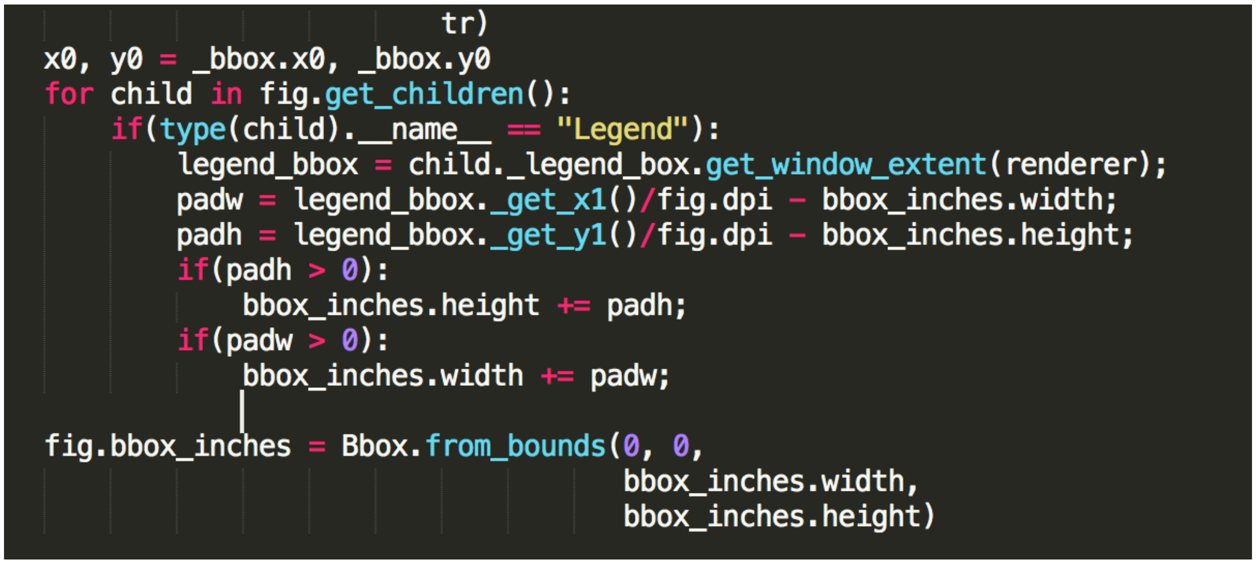


Solution:

Updated Forked Repo

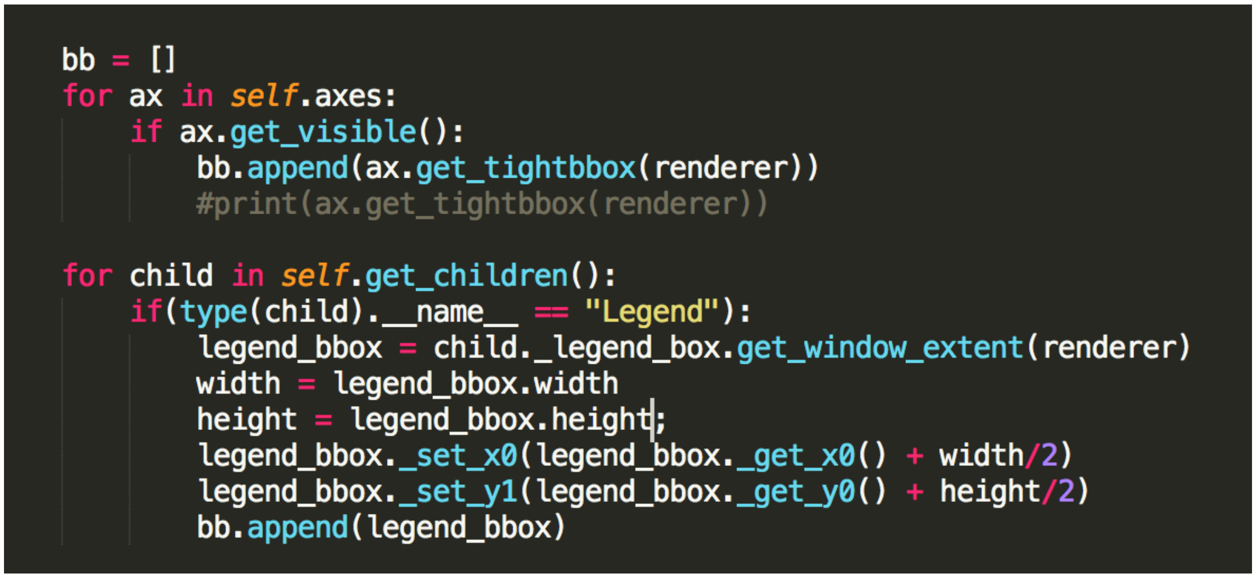
For the solution we had to alter 1 file, ***legend.py*** and specifically the function ***set_bbox_to_anchor***. The updated affected file ***legend.py***, can be found in ***solutions/bbox_tight_legend***/.

While the final solution seems extremely simple, we spent a majority of the time attempting to create a solution by adding functionality that did not previously exist before settling on our final solution. We initially implemented a solution to pad additional space onto the image based on where the legend was:



```
                                    tr)
x0, y0 = _bbox.x0, _bbox.y0
for child in fig.get_children():
    if(type(child).__name__ == "Legend"):
        legend_bbox = child._legend_box.get_window_extent(renderer);
        padw = legend_bbox._get_x1()/fig.dpi - bbox_inches.width;
        padh = legend_bbox._get_y1()/fig.dpi - bbox_inches.height;
        if(padh > 0):
            bbox_inches.height += padh;
        if(padw > 0):
            bbox_inches.width += padw;

fig.bbox_inches = Bbox.from_bounds(0, 0,
                                   bbox_inches.width,
                                   bbox_inches.height)
```

However, we realized that this was defeating the purpose of the 'tight' constraint. This took 10 initial hours spread between 2 developers. We then noticed that in the code, the legend properties of a figure were not added to the bbox properties. Only legends associated with an axis was accounted for. We then attempted to create a solution by adding the legend to the tight bbox:



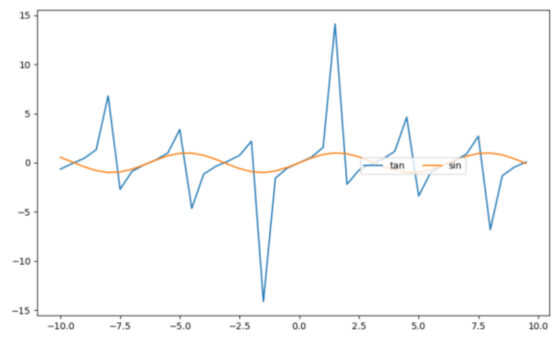
```
bb = []
for ax in self.axes:
    if ax.get_visible():
        bb.append(ax.get_tightbbox(renderer))
        #print(ax.get_tightbbox(renderer))

for child in self.get_children():
    if(type(child).__name__ == "Legend"):
        legend_bbox = child._legend_box.get_window_extent(renderer)
        width = legend_bbox.width
        height = legend_bbox.height;
        legend_bbox._set_x0(legend_bbox._get_x0() + width/2)
        legend_bbox._set_y1(legend_bbox._get_y0() + height/2)
        bb.append(legend_bbox)
```

This in theory could have solved the problem, however we realized that adding the legend to tight bbox could already be done using the ***extra_artists*** property when creating the legend. As such we considered our solution redundant and investigated why the ***extra_artists*** property was not adding the appropriate padding. We finally realized after much investigation that there was an issue with how the legend was creating its own bounding box when none was supplied. The issues were strange in that bbox instances that looked identical when compared and printed, yielded different results as they are used. Ultimately, we created a solution that takes the already constructed ***bboxTransTo*** from the legends parent rather than constructing a new one from the parent's bbox.

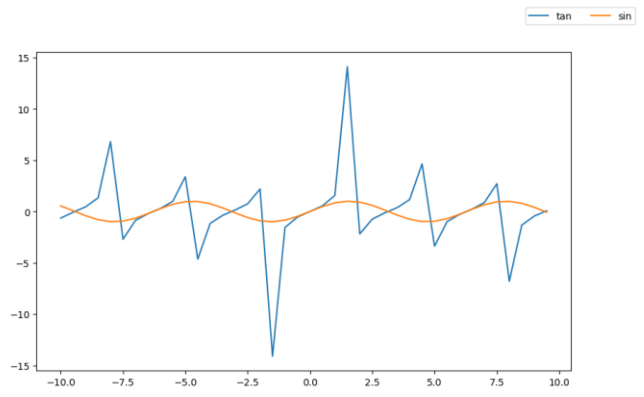
Testing:

To test that the solution works, we needed to visually check the outputted images. The associated tests are in ***solutions/bbox_tight_legend/tests/***. The file ***test_bbox_tight.py*** contains the specific test cases for the bbox issue that cut off the legend from the outputted file.

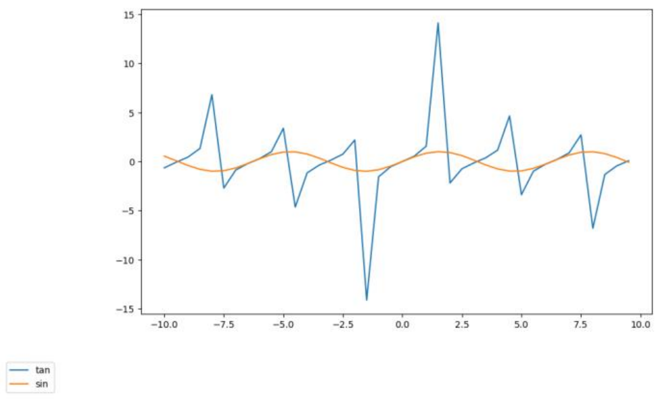
There are six tests used. Below are the outputted images for the test cases. Since our issue is affected by the location of the legend, we tested all the 4 corners and a random location to ensure the legend does not get cut off in any scenario.



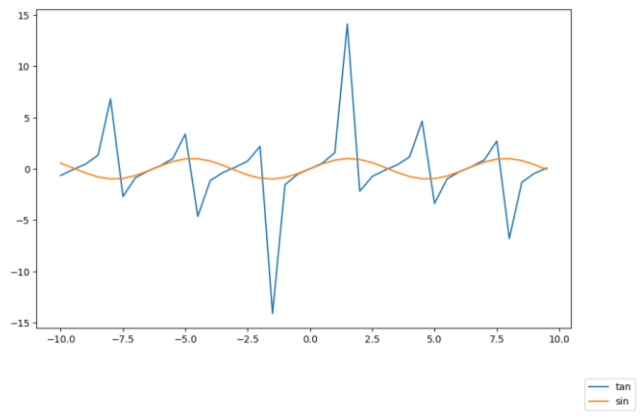
simple_randomLocation.png



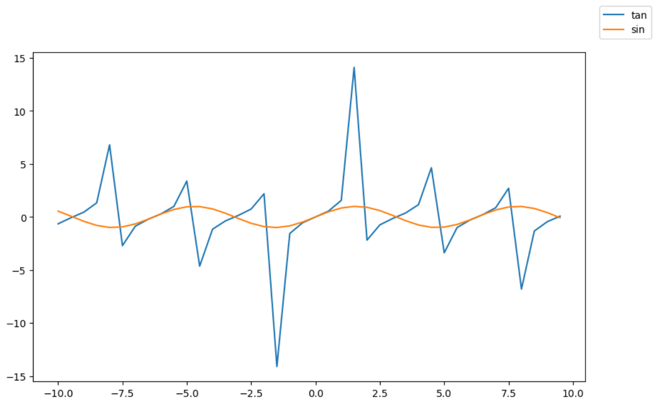
simple_multipleColumns.png



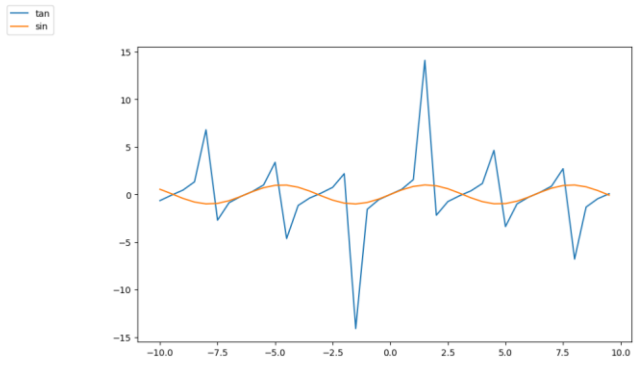
simple_bottomLeft.png



simple_bottomRight.png



simple_topRight.png



simple_topLeft.png

Confidence in Solution:

I believe our solution is the best possible solution because we have the same ideology as the previous developer that developed this functionality. We essentially needed to transform the bbox after adding extra artist properties but for some reason the previous way was not working so we used the transformation that was already created and outputted that as our bbox. We tried to do ad hoc fixes before but as time passed we realized this was the best way to go because there is no shortcut involved, making our solution the best.

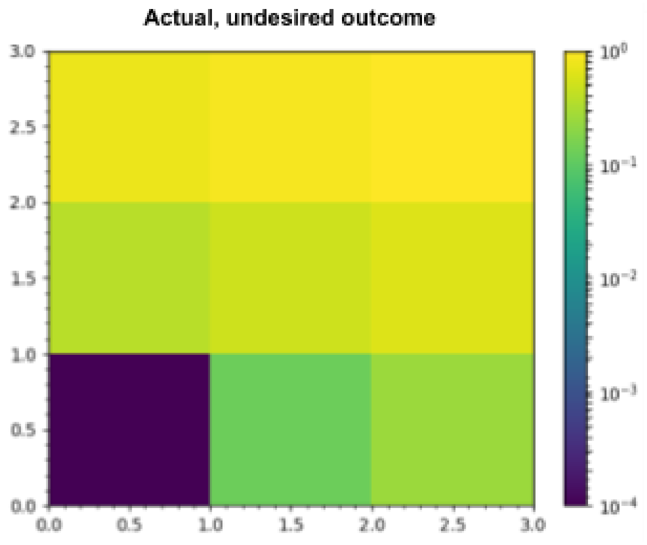
Bug #3: Log Scale (Fixed)

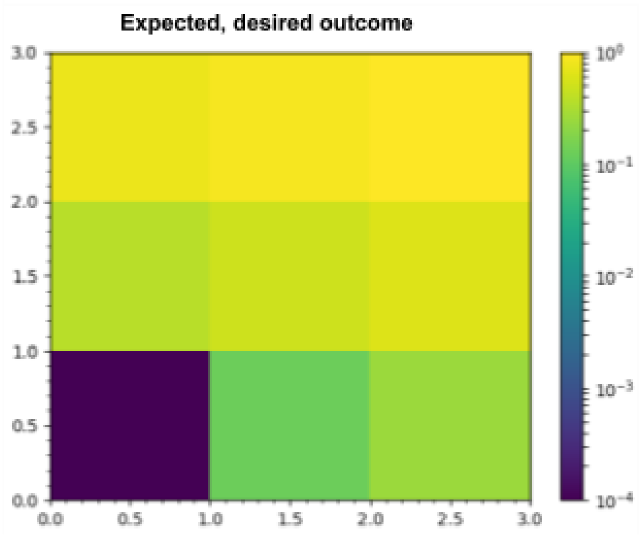
Bug/Issue: Minor ticks on log-scale colorbar are not cleared #8358

Estimated Hours:
- *Explore and create a solution (22 h)*
- *Implement solution (11 h)*
- *Testing/validation (12 h)*
- *Code Review (4 h)*
- *Documentation (1 h)*
- *Total: 50 h*

Description:
When creating a Colorbar with a logarithmic scale, its minor ticks are already shown by default. However, if the user's rcParams values 'xtick.minor.visible' or 'ytick.minor.visible' is true so that minor ticks are always visible by default, then linear scale minor ticks will unexpectedly be present in addition to the log scale ticks.

Example:
```python
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

fig = plt.figure()
plt.rcParams['xtick.minor.visible'] = True
plt.rcParams['ytick.minor.visible'] = True
data = [[.0001,.125,.25],
        [.375,.5,.625],
        [.75, .875, 1]]

plt.pcolormesh(data, norm=LogNorm())
cbar = plt.colorbar()

plt.show()
```

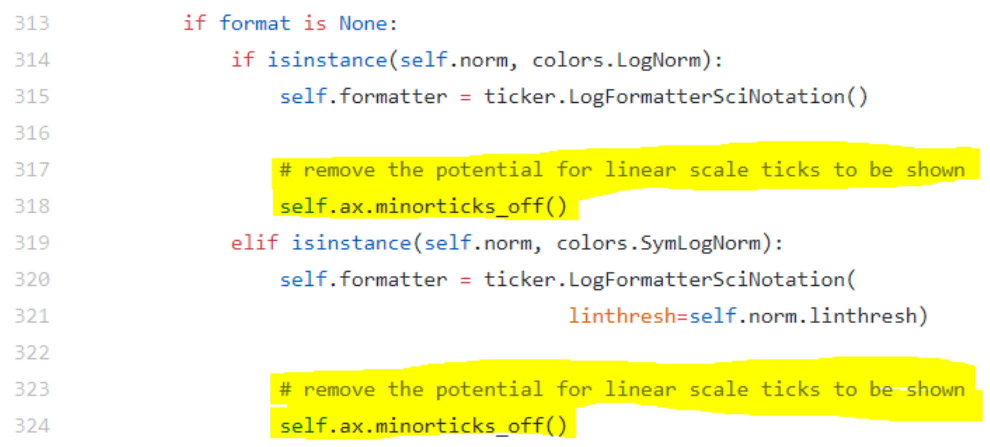




Solution:
Updated Forked Repo
The affected file is ***colorbar.py*** and its updated version with our solution can be found in ***solutions/logscale/colorbar.py***.

For our solution, we spent a lot of time thinking of an effective way of combating the issue. We started by looking into fixing the original implementation of LogLocator. In the responses to the issue, a contributor mentioned that the ticks produced by LogLocator are emulated using major ticks, which is the cause of its odd behaviour. We wanted to edit the class such that log scale minor ticks are treated like the others (i.e. hidden by default, affected by Axes.minorticks_on() or Axes.minorticks_off()). However, we decided that this solution could break existing uses and dependencies, as its implementation has been around for years.

Instead, we looked into removing the minor ticks while the Colorbar is being created. Although our solution was simple, we came to it through hours of trial and error. Our goal was to remove the unwanted ticks, so we started by appending different methods of the Axis, Axes and Colorbar classes that were related to minor ticks to the end of our test script to see which methods did the job. While browsing the axes/_base.py, we noticed that it had a method called minorticks_off() which says it removes minor ticks from the axes. So, we tested this method, and it produced the expected result. Our next goal was to find a way to identify which types of ticks were being used. We tried checking if the locator was LogLocator, or if the scale was 'log', but due to the odd implementation of log scale ticks, neither of these worked. Then, we noticed that there was an if-statement in the __init__ method of ColorbarBase that checked if it was given a log norm. We plugged minorticks_off() into there, and it produced the desired result. After successfully testing it over a series of existing and new test cases to make sure that it does not break existing functionality, we decided that it is a good solution, which is show below.

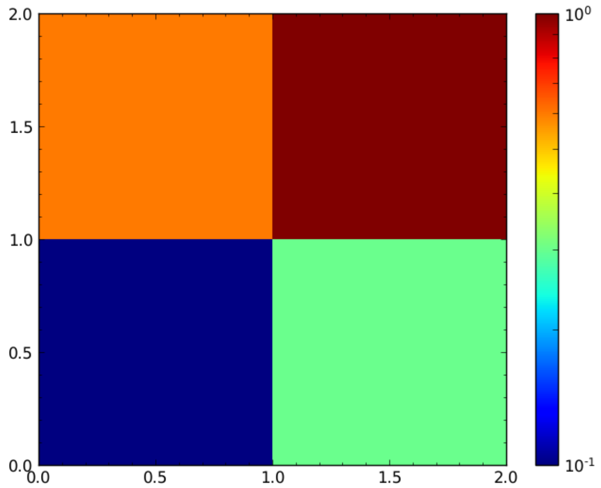


```
        if format is None:
            if isinstance(self.norm, colors.LogNorm):
                self.formatter = ticker.LogFormatterSciNotation()

                # remove the potential for linear scale ticks to be shown
                self.ax.minorticks_off()
            elif isinstance(self.norm, colors.SymLogNorm):
                self.formatter = ticker.LogFormatterSciNotation(
                                        linthresh=self.norm.linthresh)

                # remove the potential for linear scale ticks to be shown
                self.ax.minorticks_off()
```

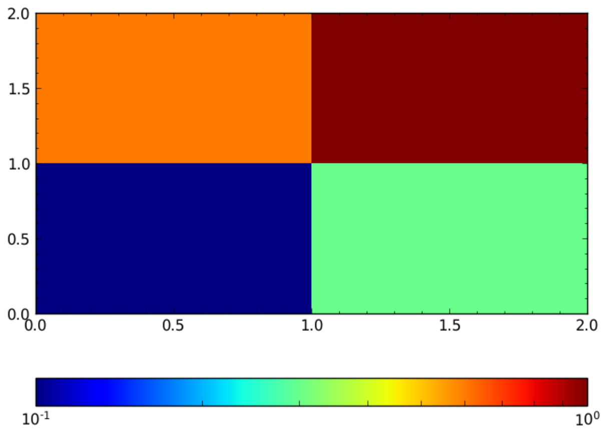
Testing:
Once again, image comparison was used to test the solution. The new tests are located in ***solutions/log_scale/tests/***. There, the file ***test_logscale.py*** contains the new test cases written for our solution. Since the bug and its solution is rooted in the Colorbar, these tests should be added to matplotlib's existing ***test_colorbar.py***. This updated ***test_colorbar.py*** can be found alongside ***test_logscale.py***, in ***D4/solutions/log_scale/tests/***, and should replace matplotlib's ***test_colorbar.py*** in the event of a merge.

Since the issue bug appears with a log scale colorbar where the 'xtick.minor.visible' or 'ytick.minor.visible' rcParam values are set to true, we made a number of graphs reproducing this situation with different parameters. The test cases used different ranges of data sets and also made sure to verify that the bug is fixed for horizontal colorbars as well as the default vertical ones. Additionally, both the LogNorm() and SymLogNorm() parameters for the pcolormesh graph are tested, since they both implement logarithmic scales.

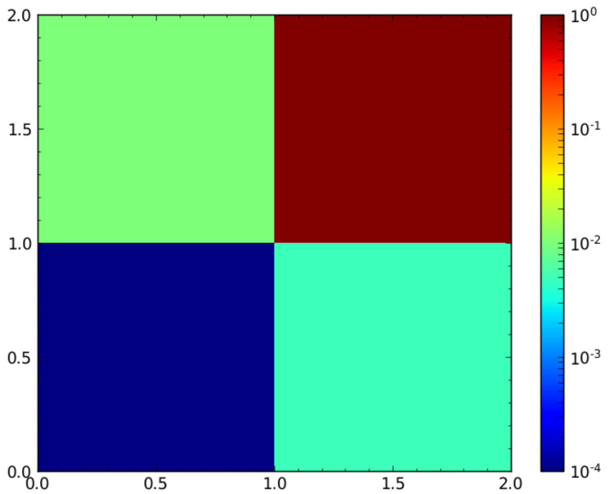
The resulting images of our tests, shown below, can be found in ***solutions/log_scale/tests/test_colorbar/*** and should be added to matplotlib's ***lib/matplotlib/tests/baseline_images/test_colorbar/*** during a merge. As one can see, there are no longer any undesired linear ticks on the colorbar despite the visibilities of the minor ticks being set to true (this is evident because all of the colorbar ticks are uniform in length – that is, there are no minor ticks). The updated ***test_colorbar.py*** was also run on our fix and all the tests have passed.



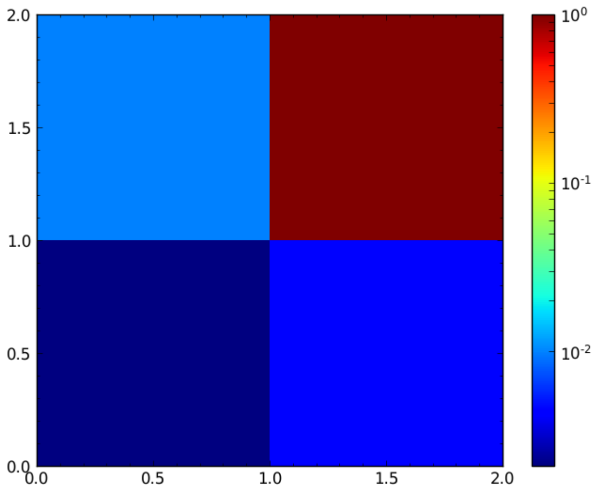
simple_log_cbar.png



simple_log_cbar_horizontal.png



dense_log_cbar.png



dense_log_cbar_symlognorm.png

Confidence in Solution:

We believe our solution works well because it solves the bug that the user was faced with. The purpose of rcParams is to add customizability to matplotlib so that users can easily set default styles for all their plots. In colorbars with many minor ticks, it is difficult to notice ticks that should not be there. The user that encountered this bug, and possibly many others have their ‘xtick.minor.visible’ and ‘ytick.minor.visible’ values set to true, and by fixing this bug, they will not see the unexpected linear scale ticks, and produce plots with log scale minor ticks correctly displayed on their colorbars.

Lessons Learned

There are a lot of lessons that our group has learned during our time working with matplotlib and fixing existing issues. Even though we're not sure if any of our efforts will result in any accepted pull requests, it was a valuable experience working with open source projects. Listed below are just some of the various lessons we've learned and agreed upon.

- Working on an open source and large-scale project requires a lot of coordination. Therefore, it's imperative that matplotlib had a guideline and standard for contributing.
- Quality of tests are important to validate that a solution works as opposed to just hoping it works.
- Fixing bugs in an open source project shouldn't be a full-time thing, rather, something that should be considered a hobby or honing programming skills.
- Something that looks simple to fix can be really complicated; looks can be deceiving, just like the matplotlib code.
- Debugging is an unbelievably valuable skill to have as a developer, and it can be just as, if not more important than the developer's ability to write code.
- Having good test cases is very important to verify that the code functions according to the expectations and is also useful for finding errors and defects.
- Many of the issues have a common cause/background. (e.g. there were several bugs related to the legend and tight bbox option).
- Good in-depth analysis and documentation of code structures has an amplifying effect on development. The more specific the analysis was (as was the case with the legend annotation bug) the more immediately work was able to start, and progression speed was further amplified.
- An understanding of various design patterns is applicable to big projects and that it helps with the overall understanding of how the code structure works. This can be applied and transferred to future projects.
- Whenever you are coming up with a solution you need to consider all cases, and ensure that you come out with the best solution. This is key for open source projects because you don't want to cause other bugs in other parts of the project.
- Simply doing google searches every time your stuck won't help, you have to go thoroughly understand the objects and functions surrounding the issue.
- If someone else works on the same issue, we must be quick and fast so that our update can be pulled first or be considered for other collaborators to work on. An example is the legend annotate bug pull request being closed because of another similar pull request addressing the same issue.