

CSCD01 - Team Name:

'Scrum Till You Waterfall'

Deliverable 2

Team 13

Table of Contents

Plan of Action - Waterfall Software Development Process	3
Deliverable 2 Gantt Chart	4
Documented Tasks	5
Requirements Phase	7
Bug #1	7
Bug #2	7
Bug #3	7
Bug #4	8
Bug #5	9
Bugs That We Chose To Implement	9
System And Software Design Phase	11
Bug #2 Implementation	11
Bug #3 Implementation	12
Implementation and Unit Testing Phase	13
Acceptance Test Bug #2	13
Acceptance Test Bug #3	14
Technical Commentary on Code Changes	17
How we Followed the Waterfall Development Process	18

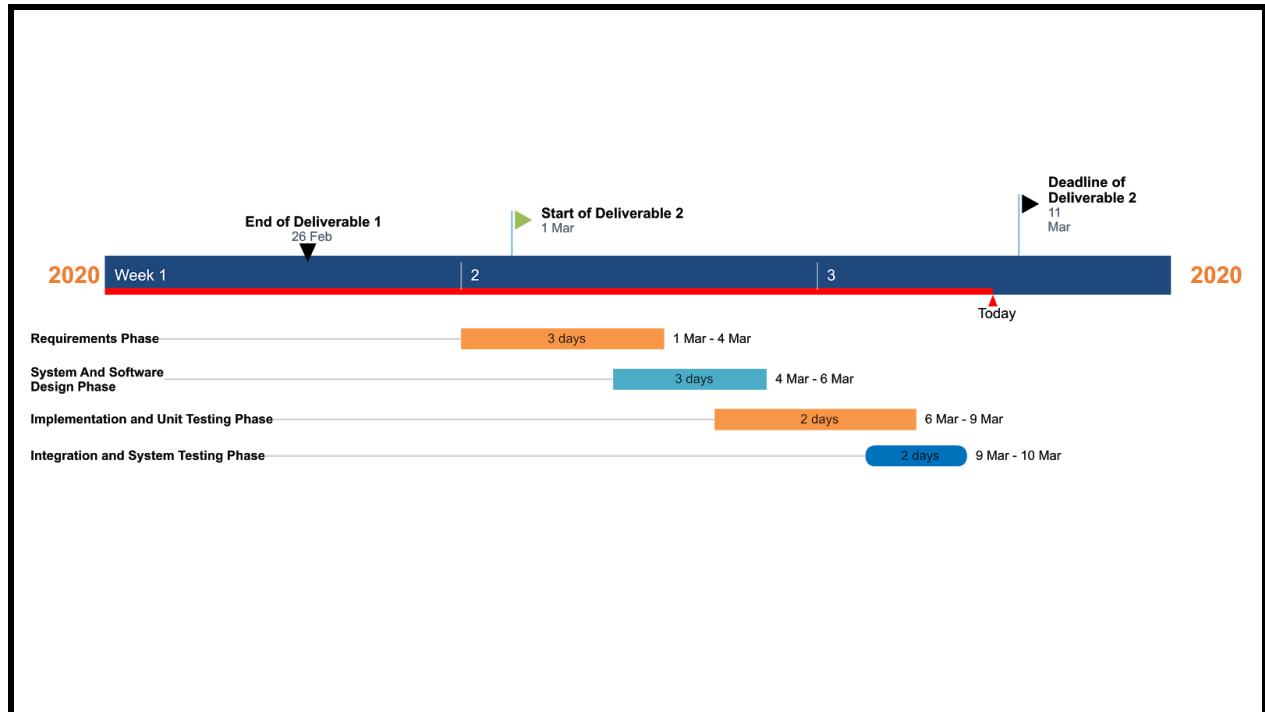
Plan of Action - Waterfall Software Development Process

Here are the five phases of Waterfall and how our team will follow them for this deliverable.

- Requirements Phase (Deadline: Mar 4)
 - Team meetings on “date” to investigate the five bugs that we want to target for Matplotlib
 - For each bug, we explain why we chose the bug, the value of the bug, and the impact
 - Outcome of this phase will be a section of the document that highlights each of the 5 bugs chosen
- System And Software Design Phase (Deadline: Mar 6)
 - Determine which classes/modules will be related to the bug
 - Reproduce and trace the origin of the bug
 - Design a solution for the bug
 - Creating an acceptance test suite
 - Outcome of this phase will be a section of the document explaining why each of the 5 bugs happen, the difficulty of the solution, an acceptance test suite, and the decided 2 bugs that we will be implementing the fixes for
- Implementation and Unit Testing Phase (Deadline: Mar 9)
 - The 2 chosen bugs will have their fixes implemented
 - A set of unit tests and acceptance tests will be included
 - Outcome of this phase will be a set of unit tests, steps on how to run the tests, and the bug fixes in the project repository
- Integration and System Testing Phase (Deadline: Mar 10)
 - Code review
 - Outcome of this phase will be improved code via code review (work in process)
- Operation and maintenance (not within the scope of this deliverable)

Deliverable 2 Gantt Chart

This Gantt chart was based on our Plan of Action and is used to help us to keep track of our schedule for each phase of our software development process for this deliverable.



Created using <https://online.officetimeline.com/>

Documented Tasks

Task	Assigned Group Member	Start Time of Task	Estimated Time of the Task	Actual Completed Time of the Task
Description of Bug#3 - Default checkbox	Abithan	Friday, March 6	2 hours	1.5 hours
Description of Bug#1 - Font size bug	Jason	Friday, March 6	3 hours	3 hours
Description of Bug#2 - Polar subplot bug	Venkada	Thursday, March 5	3 hours	5 hours
Description of Bug#4 - hLines/vLines bug investigation and results description	Anandha	Wednesday, March 4	2 hours	3.5 hours
Description of Bug#5 - Multicolor errorbars caps bug	Andrew	Friday, March 6	2 hours	3 hours
Deciding which two bugs to implement	Whole Team	Friday, March 6	30 min	1 hour
Implementation of solution for Bug#3 - Default checkbox	Abithan	Saturday, March 7	1 hour	1 hour
Implementation of solution for Bug#2 - Polar subplot bug	Venkada	Sunday, March 8	2 hours	3 hours
Creating Acceptance Test of Bug#3	Abithan	Monday, March 9	2 hour	1.5 hour

Creating Acceptance Test of Bug#2	Venkada	Monday, March 9	2 hours	2.5 hours
Code Review of Two Bugs	Whole Team	Tuesday March 10	1 hour	1.5 hours
Explain How We Followed Waterfall	Andrew	Tuesday March 10	2 hours	3 hours

Requirements Phase

Here are the five bugs that each member chose for this deliverable.

Bug #1: <https://github.com/matplotlib/matplotlib/issues/16389>

The bug is an issue with setting the properties for the axes titles and text of the chart. The characters that should display on the figure ignore the “size” property if it is set before the “fontproperties”. A method is called to update the canvas but `update` is called twice, once with the size property and then the FontProperty instance in the second. This causes the size property to be overwritten with the default size in the FontProperty instance. It is expected that the font should keep the explicit size argument rather than being overwritten. Matplotlib’s text.py uses the update function to apply properties in a sequential order which is what causes the issue, therefore by applying the fontproperties changes first, the issue can be solved. An estimate for this bug would be about 3-5 hours in total.

Bug #2: <https://github.com/matplotlib/matplotlib/issues/16501>

This bug is a regarding the polar subplot and is reproducible by setting the min and max radians such that it meets the condition $\max - \min > 2 * \pi$. The polar subplot should throw an error in the case $\max - \min > 2 * \pi$ and if it’s within the 2π range then the plot must be showing that range. An estimate for this bug would be about 5 hours to fix the display and create test cases for the solution.

Bug #3: <https://github.com/matplotlib/matplotlib/issues/5496>

This bug deals with an inconvenient UX issue for users, where when they open up the properties for a plot, they have to manually set the checkbox “(Re-)Generate automatic legend” every time. UX issues may not be hard in terms of complexity to fix, but they do provide a great deal of user satisfaction, as this bug could gradually happen multiple times, causing a major nuisance for users of Matplotlib, like the reporter of this issue. Furthermore, we foresee this bug taking minimal development time which means it is reasonable and within the scope of this deliverable. An estimate for this bug would be a half business day, so roughly 4 hours, as it encompasses identifying the root cause, designing a fix, implementation, and acceptance tests.

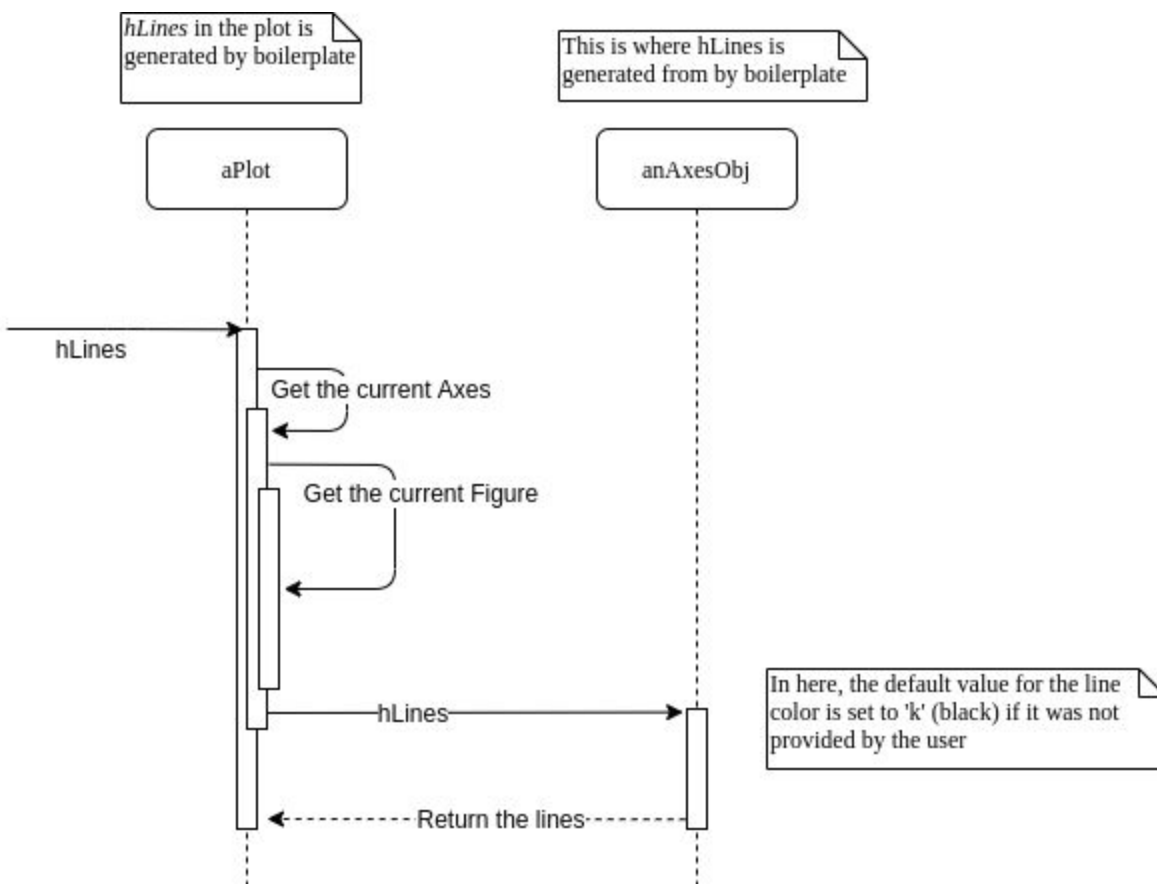
Bug #4: <https://github.com/matplotlib/matplotlib/issues/16482>

This is a bug that is related to inconsistent configuration options for default values.

When a user tries to plot lines on a non-default background, for example dark, without specifying the color of the line in the methods `hlines` or `vlines`, the backend defaults the color of line to black (`'k'`). The expected behavior for this scenario should be that the backend resorts to `rcParams` to get the color value for the lines that it plots.

Furthermore, when the user provides the `color` argument as `None` to the methods `hlines` or `vlines`, the backend then defaults to the `lines.color` value from `rcParams`.

The sequence of events could be illustrated like the following:



To proceed to implement the fix would involve multiple tasks, which includes:

- 1) Changing the documentation of the methods `hlines` and `vlines` to mention that the default color of the lines would be retrieved from `rcParams` instead of a hardcoded black value.
- 2) Modify code to retrieve `rcParams['lines.color']` and set that as the default value
 - a) This would just be replacing `colors='k'` in `hlines`
`[lib/matplotlib/axes/_axes.py]` to `colors=None`

- 3) Add unit tests that verify that ``lines.color`` is used as default when the ``color`` argument is not provided.

Addressing the bug and providing a production-ready fix would take at most 2 hours as finding and replacing the documentation, and writing unit test for this could be time-consuming relative to the code changes this bug has.

Bug #5: <https://github.com/matplotlib/matplotlib/issues/14480>

This is a bug where when trying to display a set of subplots of different colors, where the colors are set in the `'edgecolor'` and `'ecolor'` parameters for each of the subplots, if `"errorbar.capsize"` in `'rcParams'` is set at a value that is not equal to 0, then the app crashes stating the error, `"ValueError: Invalid RGBA argument."` along the list of whatever colors that were given. Upon further investigation, no matter what the colors are set to, the application still crashes and gives the same error. However, if we just remove the colors altogether and do not set anything for the `'edgecolor'` and `'ecolor'` parameters, the `capsize` can be set normally to any value you want and not just 0. From the error messages, this bug is focused on the `colors.py` file which has to do with the Artist layer that deals with the frontend component of Matplotlib. From this, this bug could take a reasonable amount of time to understand the architecture of the Artist layer starting from `colors.py` and to develop a working solution. Realistically within the scope of this deliverable, I estimate that fixing this bug would take approximately 4-5 hours to achieve the whole process of studying the architecture of the Artist layer, designing a solution, implementing that solution, and creating an acceptance test for it.

Bugs That We Chose to Implement

The two bugs that we chose to implement for this deliverable are, **#2** and **#3**. The reason for this choice is as follows:

Bug #2: <https://github.com/matplotlib/matplotlib/issues/16501>

We chose this bug as it is one that was simple and a bug that we can fix within the given time limits. We also thought the bug would positively impact matplotlib users and prevent them from producing deformed graphs by raising errors and through this, save their precious time. We had an intuition that we would be able to use phases of the waterfall process to carefully plan each stage from requirements to integration and system testing.

Bug #3: <https://github.com/matplotlib/matplotlib/issues/5496>

The reason we chose this bug is mainly for simplicity and value of impact. Our goal when choosing a bug was to be confident that we could accurately design and implement the bug according to our waterfall process, and this bug met those needs, as the requirements were laid out. Also, this bug was being a nuisance to the users of the software, due to a UX inconvenience, so this could provide great satisfaction to the end-user. The bug's simplicity made it stand out as we could assure ourselves that we would meet the deliverable deadline and complete all stages of testing, all while bringing value to the consumer.

System and Software Design Phase

The files where these implementation changes were committed can be found in <https://github.com/CSCD01-team13/matplotlib/>. For further details, see the “Technical Commentary on Code Changes” section

Bug #2 Implementation: <https://github.com/matplotlib/matplotlib/issues/16501>

The fix for this bug can be implemented in file:

lib/matplotlib/projections/polar.py

```
def set_thetalim(self, *args, **kwargs):
    """
    Set the minimum and maximum theta values.

    Can take the following signatures:

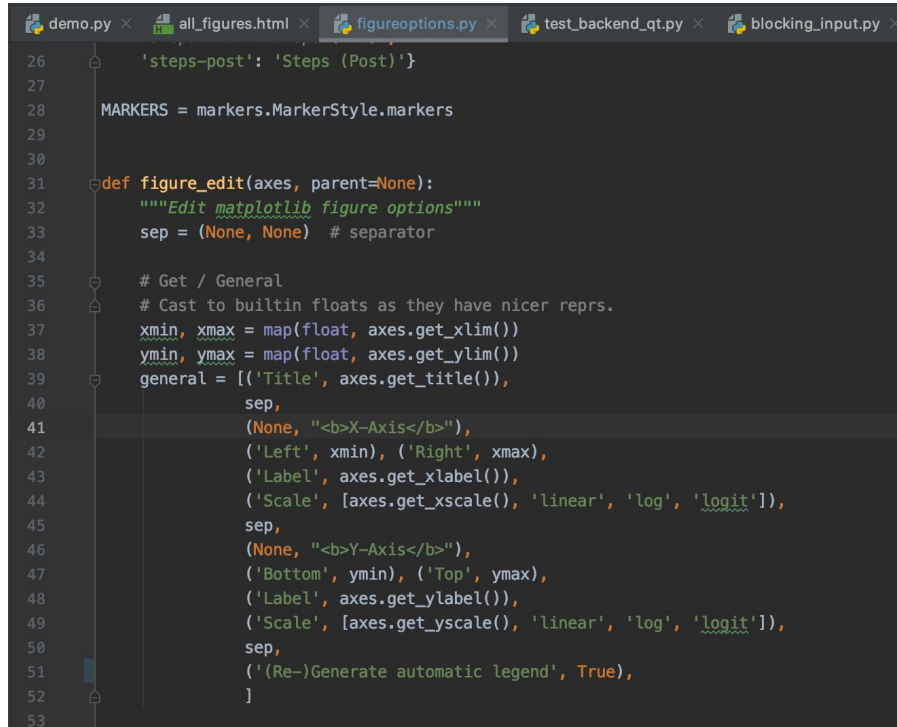
    - ``set_thetalim(minval, maxval)``: Set the limits in radians.
    - ``set_thetalim(thetamin=minval, thetamax=maxval)``: Set the limits
      in degrees.

    where minval and maxval are the minimum and maximum limits. Values are
    wrapped in to the range :math:`[0, 2\pi]` (in radians), so for example
    it is possible to do ``set_thetalim(-np.pi / 2, np.pi / 2)`` to have
    an axes symmetric around 0. If not within range :math:`[0, 2\pi]` a
    ValueError is raised.
    """
    if 'thetamin' in kwargs:
        kwargs['xmin'] = np.deg2rad(kwargs.pop('thetamin'))
    if 'thetamax' in kwargs:
        kwargs['xmax'] = np.deg2rad(kwargs.pop('thetamax'))
    return tuple(np.rad2deg(self.set_xlim(*args, **kwargs)))
```

The fix is to check for the parameters that are sent to method `set_thetalim(self, *args, **kwargs)` and to check whether the range of ``thetamin`` and ``thetamax`` are within the range `[0, 2pi]` radians or `[0, 360]` in terms of degrees.

Bug #3 Implementation: <https://github.com/matplotlib/matplotlib/issues/5496>

So the fix for this bug can be implemented in the following file:
matplotlib/lib/matplotlib/backends/qt_editor/figureoptions.py

A screenshot of a code editor window showing the file figureoptions.py. The editor has several tabs at the top: demo.py, all_figures.html, figureoptions.py (active), test_backend_qt.py, and blocking_input.py. The code is in Python and shows the figure_edit function. Line 51 has been modified to set the default value for 'Generate automatic legend' to True. The code is as follows:

```
26     'steps-post': 'Steps (Post)'}
27
28     MARKERS = markers.MarkerStyle.markers
29
30
31 def figure_edit(axes, parent=None):
32     """Edit matplotlib figure options"""
33     sep = (None, None) # separator
34
35     # Get / General
36     # Cast to builtin floats as they have nicer reprs.
37     xmin, xmax = map(float, axes.get_xlim())
38     ymin, ymax = map(float, axes.get_ylim())
39     general = [('Title', axes.get_title()),
40               sep,
41               (None, "<b>X-Axis</b>"),
42               ('Left', xmin), ('Right', xmax),
43               ('Label', axes.get_xlabel()),
44               ('Scale', [axes.get_xscale(), 'linear', 'log', 'logit']),
45               sep,
46               (None, "<b>Y-Axis</b>"),
47               ('Bottom', ymin), ('Top', ymax),
48               ('Label', axes.get_ylabel()),
49               ('Scale', [axes.get_yscale(), 'linear', 'log', 'logit']),
50               sep,
51               ('(Re-)Generate automatic legend', True),
52               ]
53
```

The simple fix is to simply set the boolean default value from False to True on Line 51. This figure_edit method controls the portion of the UI that appears when you go to edit a model when using a backend, such as “Qt5Agg”.

Implementation and Unit Testing Phase

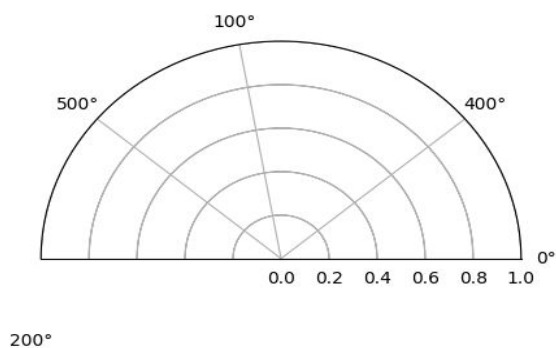
The following acceptance tests can be run in
<https://github.com/CSCD01-team13/matplotlib/>.

Acceptance test Bug #2: <https://github.com/matplotlib/matplotlib/issues/16501>

The code to reproduce the bug is as follows:

```
1  import sys
2  sys.path.insert(0, '/home/venkada/Documents/CSCD01/matplotlib/lib/matplotlib/')
3  import matplotlib
4  from matplotlib import pyplot as plt
5  # This import registers the 3D projection, but is otherwise unused.
6  from matplotlib import cbook
7  from matplotlib import cm
8  from matplotlib.colors import LightSource
9  import numpy as np
10
11  ax = plt.subplot(111, projection='polar')
12  ax.set_thetalim(thetamin = 0, thetamax= 361)
13  # or ax.set_thetalim(0, 3 * np.pi)
14
15  plt.show()
16
```

This will produce a subplot as below:



So this shows a deformed graph where there is a 100 degree in between and a floating 200 degree label which is not the expected outcome.

After Bug fix:

```
raise ValueError('Cannot pass angle range > 2 pi')
ValueError: Cannot pass angle range > 2 pi
```

A ValueError is raised whenever a range of larger than 2π is being done

Acceptance test Bug #3: <https://github.com/matplotlib/matplotlib/issues/5496>

Write a python script with the following code:

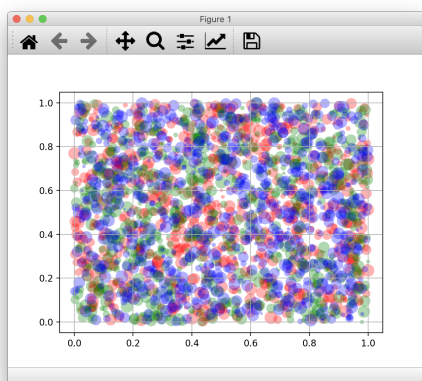
Note that the plot being used is just an example, but the fix applies to all plots using the qt_editor GUI component

```
import matplotlib.pyplot as plt
from numpy.random import rand
import matplotlib as mpl

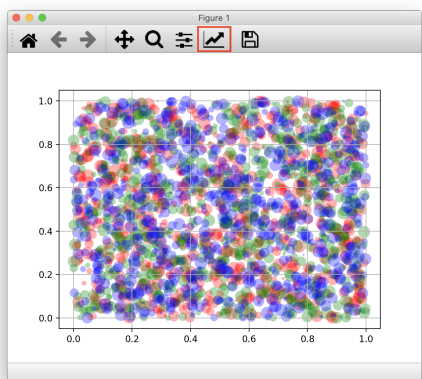
mpl.use('Qt5Agg')

fig, ax = plt.subplots()
for color in ['red', 'green', 'blue']:
    n = 750
    x, y = rand(2, n)
    scale = 200.0 * rand(n)
    ax.scatter(x, y, c=color, s=scale, label=color,
               alpha=0.3, edgecolors='none')
ax.grid(True)
plt.show()
```

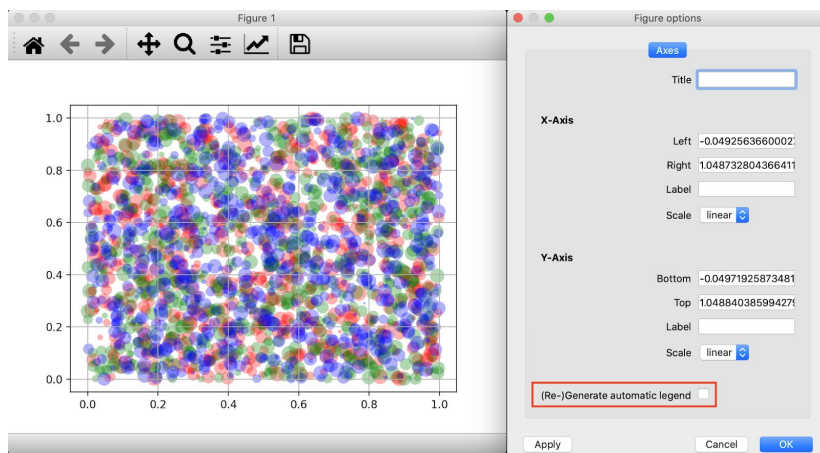
The following plot should appear:



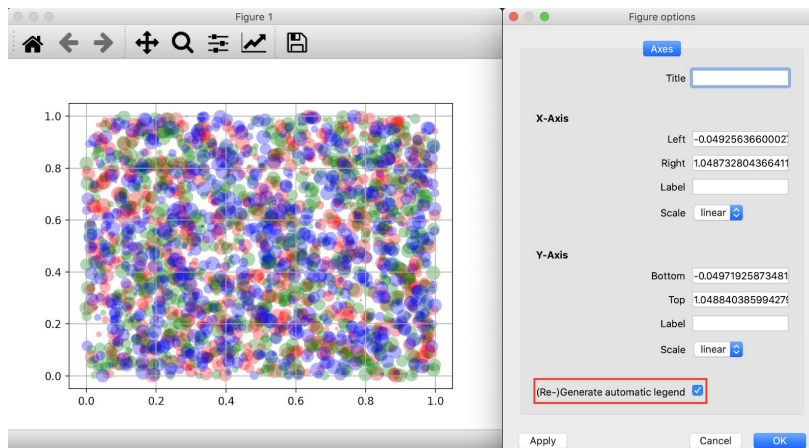
Now select the “Edit axes, curve, and image parameters” button



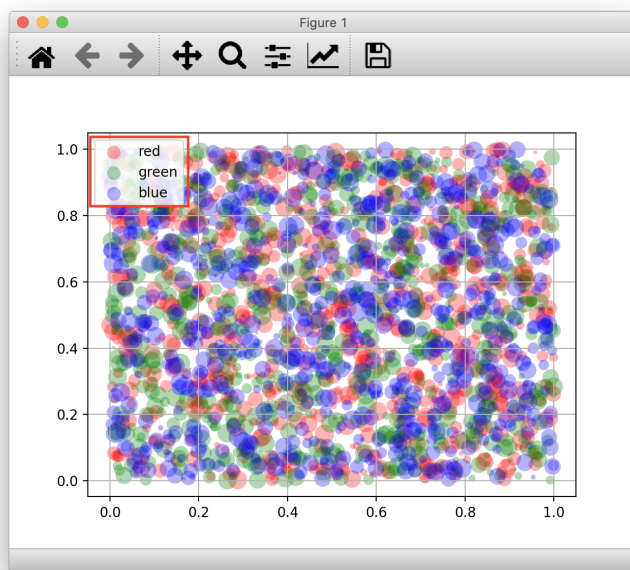
Before bug fix:



After bug fix:



Select OK in the dialog box and verify that the legend is actually being generated in the top left corner.



After the bug fix, the default behaviour of this dialog box should be that the “(Re-)Generate automatic legend” is always checked by default, and this will prevent users from having to manually check it every time after making a change to the figure.

Technical Commentary on Code Changes

Bug # 2

<https://github.com/matplotlib/matplotlib/issues/16501>

The source files that needed changes to solve issue:

lib/matplotlib/projections/polar.py

lib/matplotlib/tests/test_subplot.py

The PR is merged into master branch of our team fork

PR: <https://github.com/CSCD01-team13/matplotlib/pull/2>

The fix for the bug was to simply raise a ValueError whenever invalid theta parameters were sent into the method to create a polar plot. Along with the fix are two unit tests to properly use the different method signatures and detect the ValueError that are raised when invalid theta parameters are sent. Users will get errors shown instead of a deformed graph and this will make them enter valid parameters.

Bug # 3

<https://github.com/matplotlib/matplotlib/issues/5496>

The only source file that was modified is:

matplotlib/lib/matplotlib/backends/qt_editor/figureoptions.py

The PR has been merged to the master branch of our fork, and the PR can be found here: <https://github.com/CSCD01-team13/matplotlib/pull/1>

The code changes made for this bug fix is fairly straightforward, as it involves just changing the default behaviour of the checkbox for the “(Re-)Generate automatic legend” in the GUI. This bug fix is primarily a UX enhancement, as previously users had to manually select the checkbox to generate the legend every time after they made a change to the figure. Now with this UX enhancement, they can make changes to the figure, and not have to worry about checking this checkbox before exiting the dialog box.

How we Followed the Waterfall Software Development Process

Throughout this deliverable 2, our group has followed the incremental workflow of the Waterfall design process that we planned out in the beginning during our 'Plan of Action' phase. This can be seen where we set deadlines for each of the phases which were Requirements Phase, System And Software Design Phase, Implementation and Unit Testing Phase, and Integration and System Testing Phase.

For each of these phases, as shown by our documented log of tasks, each more or less member completed the work required by them for that phase within the timeframe for that phase. By keeping on time with each phase deadline, it allowed the work for each phase that followed to run very smoothly and be finished at an efficient pace since all the required work and investigation was done in the previous phase(s).

For the decision of using a table to keep track of our work instead of the Trello board specified in deliverable 1, we decided that it was much easier to meet with each other to discuss the tasks that needed to be done for each phase and just individually record the tasks that were assigned as we finished them. With Trello, we decided that it was inefficient to manage all the tasks and who was doing what unless we were very detailed in documenting each of our tasks before and after the task was completed, such as recording the start and finish dates for it. Rather than having a Trello board that was too messy to manage, a simple table within the deliverable document was much easier to work with.

Overall, our group collectively was able to follow the Waterfall software process in order to incrementally document the five bugs from Matplotlib, choose and implement two of those bugs, and create acceptance tests for them. Although some of us completed our tasks after the deadline we set due to the work from other courses, we still did our best to follow the process of Waterfall in a way that best fits this deliverable. One thing that could have possibly been improved was setting deadlines that were realistic for each of the team members as a whole, where we properly factor in the amount of work that each member has for their other courses and plan accordingly. Other than that, each of the phases went smoothly with no overlapping work that conflicted with each of the team members in the group. For example, there was no time wasted trying to understand the architecture involving our bugs during the implementation phase since we already investigated that during the requirements phase. Additionally, this helped each of us communicate what we did with each other since we all worked on the same type of work during each phase instead all of us do different tasks at our own pace. In other words, everyone was always up to speed with each other, making it easy to work.