CSCD01 - Team Name:

‘Scrum Till You Waterfall’

Deliverable 4

Team 13

Table of Contents

Plan of Action - Waterfall Software Development Process

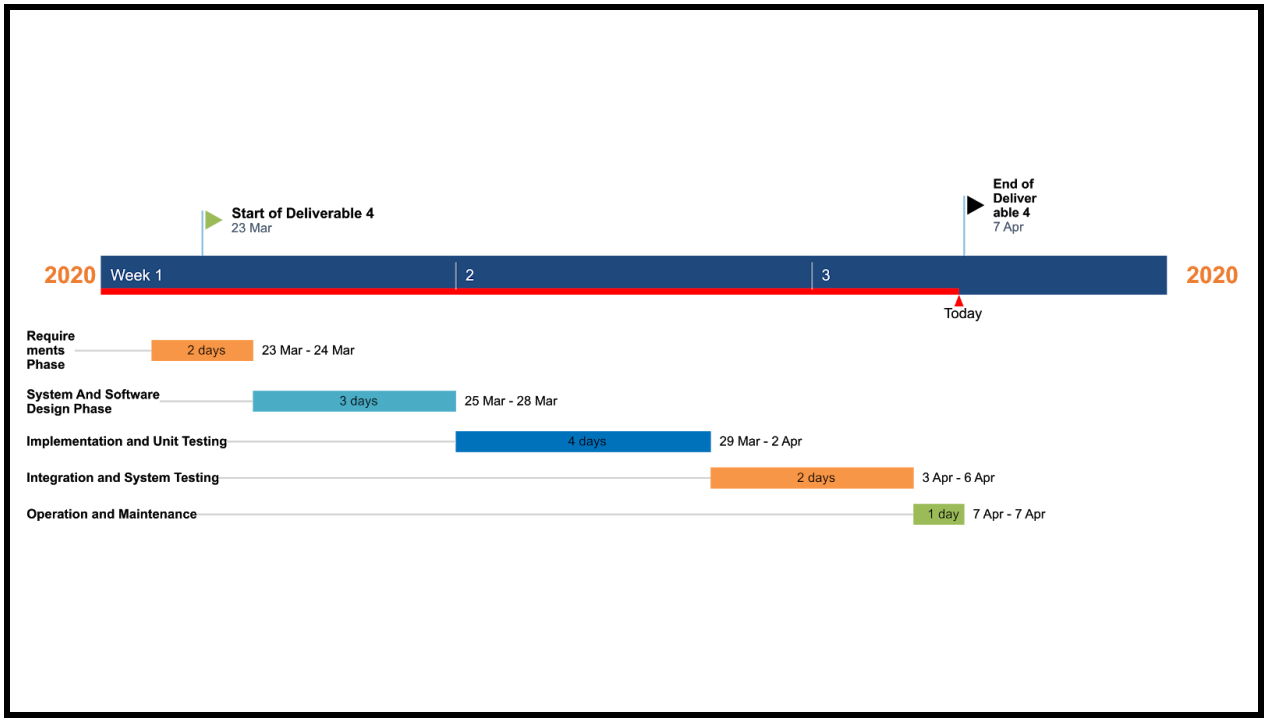
Here are the phases of the Waterfall software development process that we used and how our team will follow them for this deliverable.

- Requirements Phase (Deadline: Mar 24)
    - Strategize how exactly we will implement our bug, based off our planning phase in Deliverable 3
    - Provide description and link to our chosen feature
    - The outcome of this phase will be a section of the document that briefly goes over deliverable 3's planning stage and if any changes were made from then upon additional investigation
- System And Software Design Phase (Deadline: Mar 28)
    - Investigate more in-depth what classes and files need to be configured and any design patterns that should be used and why
    - Show existing classes and functions that will interact with this new feature
        - Provide a UML to help with this
    - The outcome of this phase will be a section of the document showcase how exactly we plan to design our feature
- Implementation and Unit Testing Phase (Deadline: Apr 1)
    - Implement our new feature based on the previous designing phase
        - Show files that were modified and new code that was added to them
        - Provide a description of what specific code is doing in case it isn't obvious at first glance
    - Create 2-4 unit tests for our features and place them in the appropriate test suite
    - Include acceptance tests from deliverable 3 and any changes that were made to them
    - The outcome of this phase will be a section of the document will to show our implementation of the new feature and the unit and acceptance tests that we created for it

- Integration and System Testing Phase (Deadline: Apr 2)
    - Run existing unit tests suites for Matplotlib to make sure no tests were broken and update code if they are somehow broken
    - Create documentation for this feature and a User Guide
    - Outcome of this phase will be a section of the document will be to show how the new code that we implemented interacted with the system and that there were no existing tests and features that were broken. Also it will provide instructions on how to run our feature
- Operation and Maintenance Phase (Deadline: Apr 3)
    - Review of our code of our implementation through code peer reviews
        - Showcase any changes that were made and any bugs caught from these reviews
    - Final check on UMLs, designs, and documentation to ensure its accuracy and correctness
    - Outcome of this phase will be improved documentation and code implementation via code review

Deliverable 4 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 deliverable 4. This chart has been updated based on the additional time given.



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
Requirements Phase: Description of Feature To Implement - Quiver key	Andrew	Monday, March 23	1 hour	40 min
Requirements Phase: Write result of phase	Andrew	Tuesday, March 24	10 min	10 min
System And Software Design Phase: Add UML and description of changes to design	Jason	Thursday, March 26	2 hours	2 hours
System And Software Design Phase: Write result of phase	Jason	Friday, March 27	20 min	10 min
Implementation and Unit Testing Phase: Implement Quiver key handler	Anandha, Venkada	Monday March 30	3 days	2 days
Implementation and Unit Testing Phase: Add feature to customize colour	Anandha	Wednesday, April 1	1 hour	1.5 hours
Implementation and Unit Testing Phase: Write and document unit test cases	Venkada	Thursday, April 2	40 min	1 hour

Implementation and Unit Testing Phase: Write result of phase	Andrew	Thursday, April 2	10 min	10 min
Integration and System Testing Phase: Acceptance test cases	Abithan	Saturday, April 4	2 hours	1.5 hours
Integration and System Testing Phase: Creating User Guide	Abithan	Sunday, April 5	4 hours	3 hours
Integration and System Testing Phase: Write result of phase	Abithan	Sunday, April 5	5 min	10 min
Operation and Maintenance Phase: Individual and Group Peer review of Documentation and Code Implementation	Whole Team	Wednesday, April 7	2 hours	2 hours
Operation and Maintenance Phase: Result of Phase and explanation of how we used Waterfall Software development process	Andrew	Wednesday, April 7	20 min	20 min

Requirements Phase

Description of Feature To Implement

The feature we decided to implement for this deliverable is a quiver key handler that is able to add a QuiverKey object to the legend of a plot.

The link to this feature can be found at:
https://github.com/matplotlib/matplotlib/issues/16664

As described in deliverable 3, this feature mainly revolves around creating a new class called QuiverKey from scratch that acts as a proxy artist.

More details about proxy artists can be found here:
http://matplotlib.org/users/legend_guide.html#creating-artists-specifically-for-adding-to-the-legend-aka-proxy-artists

The main purpose of this feature request was in order to display properties of a QuiverKey object in the legend which would be a consequence of passing the QuiverKey object from the proxy artist class we construct into the legend handler that Matplotlib uses. This feature will require substantial development as it involves investigating how proxy artists work and extending it to create a class that supports QuiverKey objects. Tackling this issue will consist of mainly developing a proxy artist class, documentation, as well as writing unit tests for it.

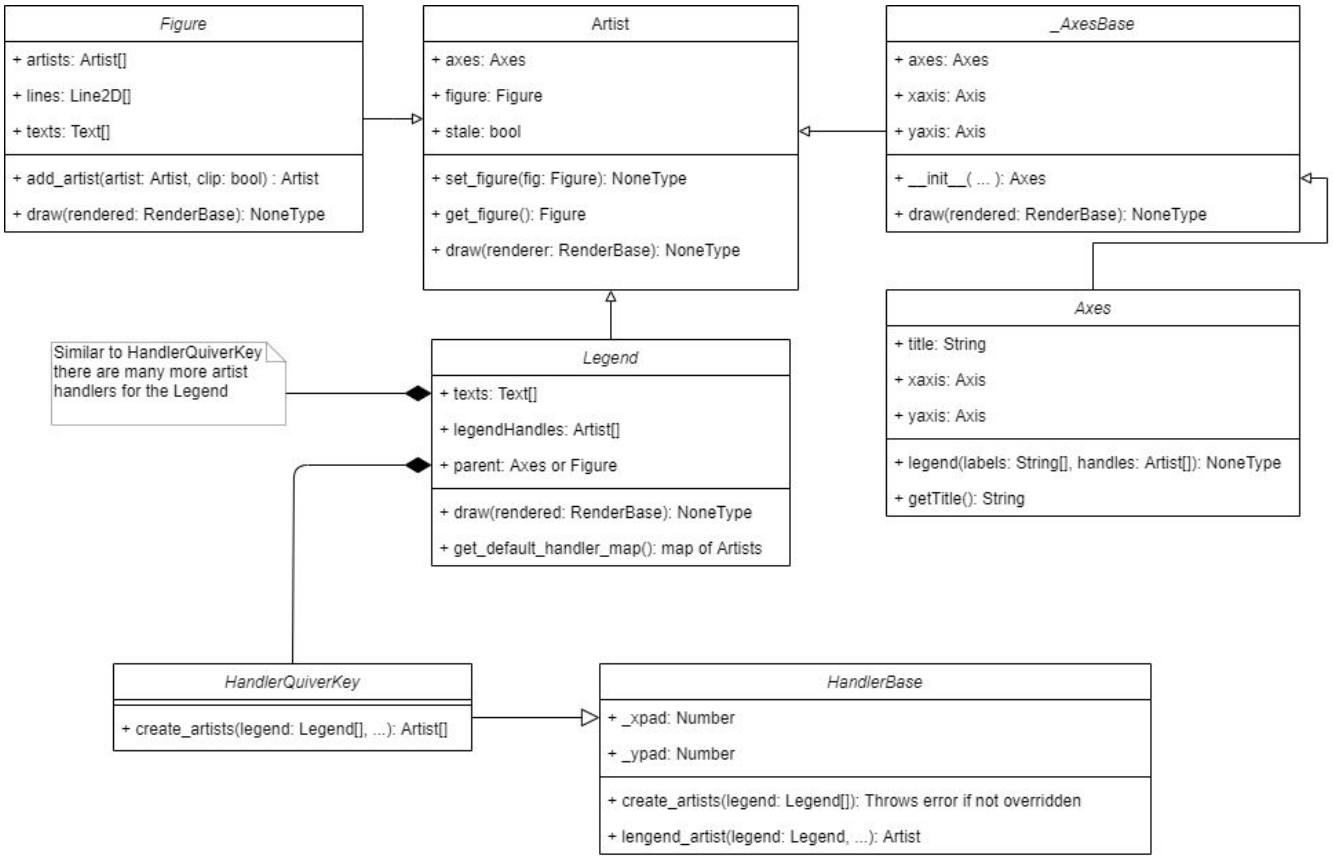
In terms of the overall design for our implementation, we have not decided to change our initial design strategy since deliverable 3 upon further investigation. This strategy to implement this feature is still to create a class named `*HandlerQuiverKey*` that must be implemented in `*legend_handler.py*` and composed within the Legend Class. However we also decide to add the additional feature of being able to change the color of the arrow of the quiver key and allow the user customize to the colour they prefer.

Result of Phase

Now that we have gathered all the requirements for our feature from our past deliverable along with additional investigation and strategizing, we are fully prepared to design exactly how we will design our new QuiverKey handler feature in the following phases, such that we won't have to look back and modify or add any new requirements to what we currently have.

System And Software Design Phase

UML Diagram of Added Feature with Existing Codebase



UML for Quiver Key legend feature

As initially planned, a case for the quiver key will be added to the legend_handler.py. A class called HandlerQuiverKey was created as a proxy artist for quiver keys in the legend_handler.py to draw the quiver key into the legend container. One difference from the original UML is that the HandlerPatch class was changed to be HandlerQuiverKey to highlight the implementation details specific to the feature.

Result of Phase

By using this UML diagram we have created, it will be a lot easier to know exactly how to implement our feature in the next phase. It helps to know exactly what classes need to interact with our new code, such that our feature will be fully functional and working with no new bugs created.

Implementation and Unit Testing Phase

Link To Implementation of the Feature

Having fully implemented our HandlerQuiverKey class in in the legend_handler.py, along with modifying the legend.py and creating unit tests in test_legend.py, our code additions can be found from the following link.

Link: https://github.com/CSCD01-team13/matplotlib/pull/4

Interactions Between New Code and Existing Code

The goal of this feature is to support a Quiver Key inside the legend and also include its label beside the quiver key arrow. The color of the quiver key inside the legend can also be specified to meet the user's requirements. The major component needed for all legend elements is the existence of its legend handler and this class needs to be inside `legend_handler.py`. We have implemented this HandlerQuiverKey class for this purpose and it will return the arrow, and the label when the quiver key is added as a handle for the legend and also remove the existing quiver key that is already plotted. The HandleQuiverKey class also needs to be added to the dictionary _default_handler_map which maps objects to its handler class. The mapping was added between the QuiverKey object and the HandlerQuiverKey class.

The HandlerQuiverKey class implements an init function and a create_artists method. The HandlerQuiverKey inherits the HandlerBase which requires it to override the create_artists method. This create_artists method was overridden to create an arrow icon using the FancyArrow class. The existing quiverkey is removed from the graph. The calculations of the tail and the head of the arrow were done using the xdescent and width variables and a Text was created to include the label that was required by the user. The FancyArrowPatch object along with its Text is returned which will be displayed on the legend.

Unit Tests

To unit test the implemented code, it was best to consult the previous examples involving the other legend handlers such as NumPoints. This includes creating a test that executes an image comparison. The following is the unit test for the quiver key legend handler.

```
@image_comparison(['quiverkey.png'])
def test_handler_quiverkey():
    """Test legend handler with a quiverkey as handler."""
    # related to #16664
    fig, ax = plt.subplots()
    X = np.arange(-10, 11, 1)
    Y = np.arange(-10, 11, 1)
    U, V = np.meshgrid(X, Y)

    q = ax.quiver(X, Y, U, V)

    qk = ax.quiverkey(q, X, Y, U, label='label', labelpos='E')

    legend_elements = [
        qk
    ]

    ax.legend(handles=legend_elements)
```

The decorator `@image_comparison` is used to provide the file name of the test image file that is used as the expected image when doing the image comparison. The test automatically locates the image in `/lib/matplotlib/tests/baseline_images/test_legend`. If at any point in the future a change affects the functionality of the quiver key legend handler, this unit test will fail and indicate that something went wrong or something needs to be updated.

Matplotlib also provides a detailed documentation on how to go about writing an image comparison test in the following web page:

https://matplotlib.org/3.2.1/devel/testing.html#writing-an-image-comparison-test

Result of Phase

Having fully implemented our feature and unit tests, we can now move on to verifying our acceptance test cases from deliverable 3, modify them if need be, and write our User Guide for this feature.

Integration and System Testing Phase

Acceptance Tests

The main goal of this feature is to address
https://github.com/matplotlib/matplotlib/issues/16664

We will design our acceptance criteria to encompass this issue as well as make the feature we develop as robust as possible.

Steps to reproduce this criteria are as follows:

Step 1:

Write a simple test script to test this feature, and let us call this quiver_legend.py with the following code:

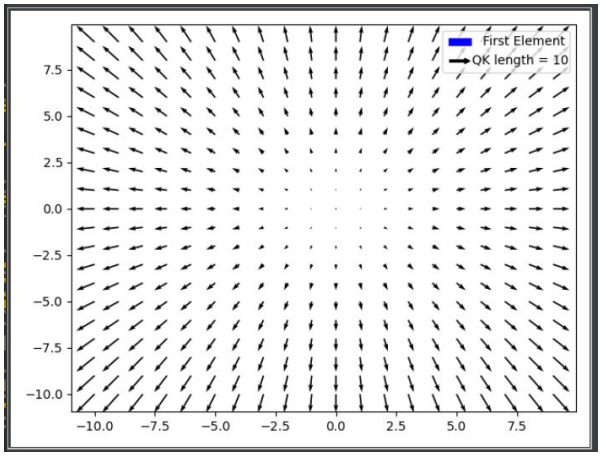
```
from matplotlib.patches import Patch
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)
q = ax.quiver(X, Y, U, V)
qk = ax.quiverkey(q, 0.9,0.8, U=10, label='QK length = 10', labelpos='E')

legend_elements = [
   Patch(facecolor="red", edgecolor="black", label="First Element"),
   qk
]

ax.legend(handles=legend_elements)
```

Step 2: Run this code

Prior to feature implementation:
In the console window a similar warning should appear, and the QuiverKey will not appear in the legend:

```
/Users/abithankumarasamy/Documents/CSCD01/feature.py:20: UserWarning: Legend does not support <matplotlib.quiver.QuiverKey object at 0x112b502d0> instances.
A proxy artist may be used instead.
See: http://matplotlib.org/users/legend_guide.html#creating-artists-specifically-for-adding-to-the-legend-aka-proxy-artists
  ax.legend(handles=legend_elements)

Process finished with exit code 0
```

After feature implementation:
There should be no warnings in the console window.



Acceptance Criteria:

- QuiverKey object should have a handler that allows it to be added to the legend
- QuiverKey object without the legend should support properties such as:
    - Label of QuiverKey
    - Label Position relative to QuiverKey
    - QuiverKey Colour
    - QuiverKey Edge colour
    - Etc.
- Error message/ warning regarding proxy artist class should no longer appear in the console window upon executing the script to plot

User Guide

The purpose of this feature is mainly to allow for QuiverKey objects to have properties displayed in the legend as well as modify properties of the QuiverKey object itself, and the following guide will provide you with a thorough guide on how to accomplish this.

There are six steps that need to be followed in order to properly use this feature, which are as follows.

Step 1:

Checkout our master branch of our forked repository or the official matplotlib library once the feature is merged.

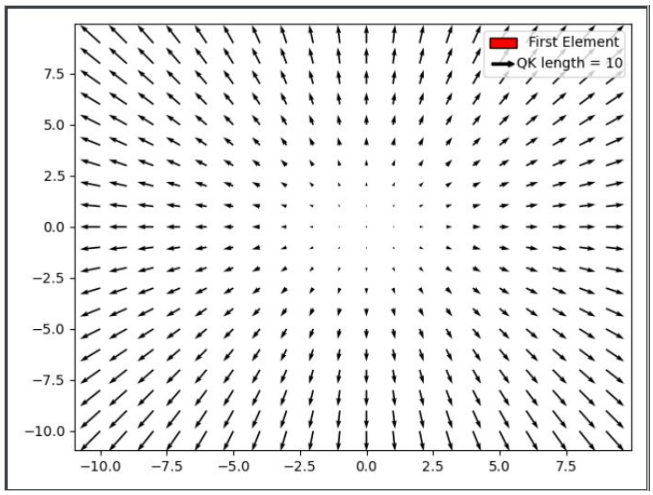
Step 2:

Write a simple test script to test this feature, and let us call this quiver_legend.py with the following code:

```
from matplotlib.patches import Patch
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)
q = ax.quiver(X, Y, U, V)
qk = ax.quiverkey(q, 0.9,0.8, U=10, label='QK length = 10', labelpos='E')

legend_elements = [
   Patch(facecolor="red", edgecolor="black", label="First Element"),
   qk
]

ax.legend(handles=legend_elements)
```

Step 3:

Once this script is ran, you should see the following output:



Step 4:

Now let us explain some ways to test using properties that are already outlined in Matplotlib's documentation.

Let us consider 3 simple properties of the QuiverKey.
facecolor: the color of the QuiverKey
label: the label to display around the QuiverKey
labelpos: positioning of the label with respect to the QuiverKey ie. N, S, W, E

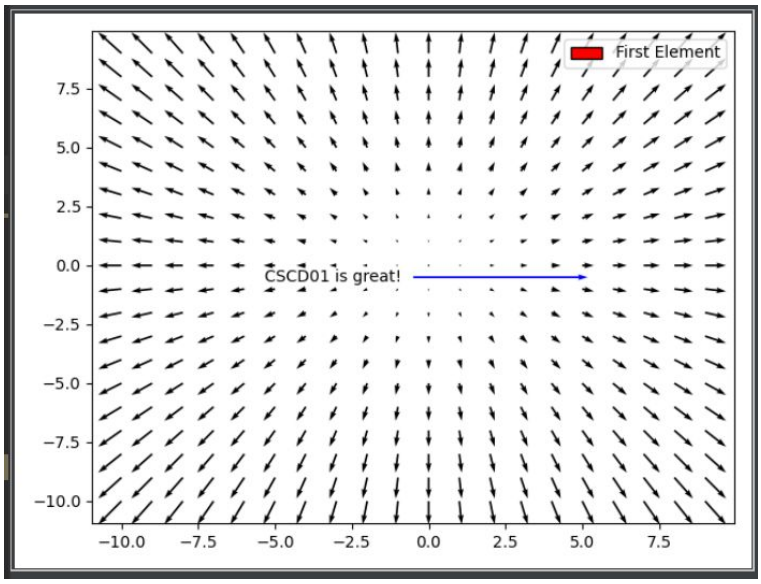
Step 5:

Consider the following code with properties modified to make a blue QuiverKey, with a label "CSCD01 is great!", positioned to the left of the object.

```
from matplotlib.patches import Patch

import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

X = np.arange(-10, 10, 1)
```

```
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)

q = ax.quiver(X, Y, U, V)

qk = ax.quiverkey(q, 0.5,0.5, U=75,facecolor="blue", label='CSCD01 is great!', labelpos='W')

legend_elements = [
   Patch(facecolor="red", edgecolor="black", label="First Element")
]

ax.legend(handles=legend_elements)

plt.show()
```

Once this script is ran, you should see the following:



Step 6:

Continue to experiment and use the enhancements we have implemented in the QuiverKeyHandler to better aesthetically support the QuiverKey and its new properties.

Result of Phase

With our acceptance tests and User Guide fully completed, we have finally completed all work for this deliverable. All that needs to be done is regression testing and verification through peer code reviews and checking this document for any errors.

Operation and Maintenance Phase

All team members have peer reviewed all of our collective code and documentation for this deliverable 4.

In order to accomplish this with care and precision, we first each individually reviewed this document ourselves and peer reviewed the code for our feature implementation. After that, we held a team meeting within a Facebook group chat, where each of us stated if there were any outstanding problems or if there were minor changes they wanted to make. From our discussion, we concluded that there wasn't anything that we really wanted to change and we were all generally happy with the result we have accomplished from this deliverable.

While we might have been about to implement a more complex feature if given more time within each of our own personal schedules, as we all have full course loads of 5-6 courses for this semester as stated from our deliverable 0, being able to put this much time and effort in for this deliverable is something we have accepted and are fine with.

Overall, besides a few typos and communication errors in our writing for this document, we have fully checked and reviewed our documentation and feature implementation and ensured its accuracy and correctness.

Result of Phase and How the Waterfall Software Development Process Was Used

With the completion of this phase to validate and verify our documentation and implementation, we have fully went through all the phases for our Waterfall Software Development process, which are the 'Requirements Phase', 'System And Software Design Phase', 'Implementation and Unit Testing Phase', 'Integration and System Testing Phase', and 'Operation and Maintenance Phase'. Going through each of these phases incrementally, we have been able to easily transition from one stage to the next as the work done in the previous phase helped to build on top of the next phase's work. This can be seen, for example, when we needed to understand the documentation and UML diagrams for the feature we wanted to implement in order to actually properly implement our feature without error or confusion. Thus, our whole development process was actually easy to comprehend and understand by all group members with little to no risk or uncertainty. It was also easy to measure the progress through each of our phases through how we documented each of tasks on page 6 of this document.