D01LAR BILLS/Team 30

CSCD01 WINTER 2020

Deliverable 3

Submitted To:
Dr. Anya Tafliovich

Submitted By :
Joseph Sokolon
Wesley Ma
Raya Farhadi
Edgar Sarkisian

Contents

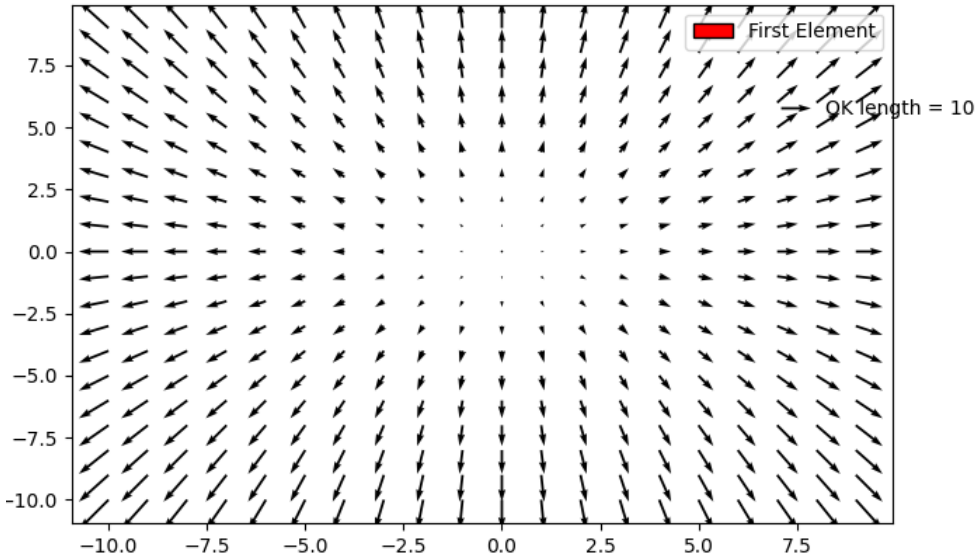
1 Features

1.1 Add quiverkey to legend

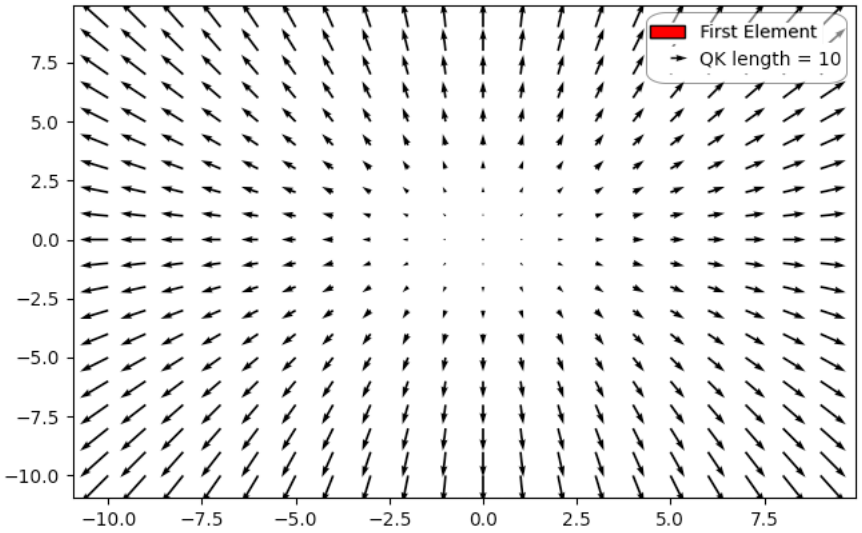
https://github.com/matplotlib/matplotlib/issues/16664

The requested feature, is to add support for QuiverKey Objects to be added to the legend for quiver plots. A quiver plot displays velocity vectors as arrows on a graph. The quiver key is used to give some description of the length of the arrows on the plot by showing an arrow and its corresponding size that you can then compare to the other arrows on the plot. Currently when using a quiver key the user specifies the x and y coordinate of where they want the quiver key to be displayed. Quiver plots do not typically have a legend, so this implementation makes sense, however when there is a legend, it makes sense to include the quiver key in that legend, as they are both descriptors of the plot and should be grouped together. Matplotlib does have the ability to create custom legend handlers using proxy artists to add non default information into your legends, however the main component of a quiverkey is the length of the arrow in the key.

Current behaviour:



Proposed feature:

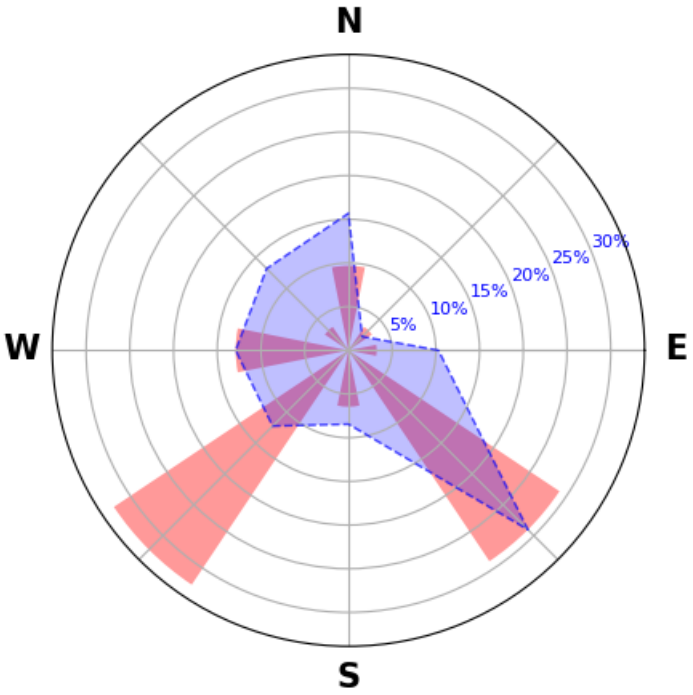


1.2 Add twin axis to polar graph

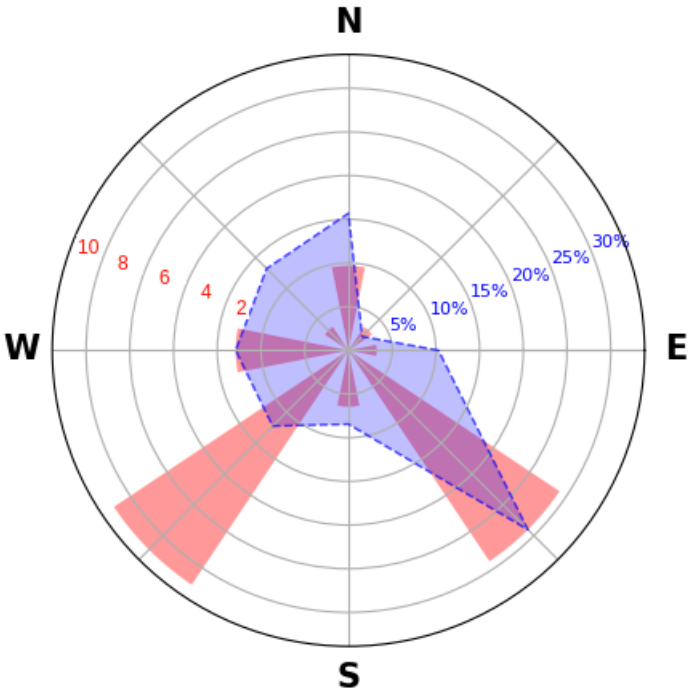
https://github.com/matplotlib/matplotlib/issues/12506

The feature proposed, is to add support for a secondary radial axes in polar graphs. Currently there is only support for a single radial axis in polar graphs. Users can plot multiple types of data on a polar graph, so it would be useful to represent the data separately with multiple radial axes labels. Currently, while plotting on a Cartesian grid, users can call .twinx(), which creates a secondary y-axis that share the same x axes. Our proposed changes would add a twintheta() function, so users could use polar graphs with multiple radial axes that share the same theta space.

Current behaviour:



Proposed feature:



2 Proposed Architectural Changes

2.1 Add quiverkey to legend

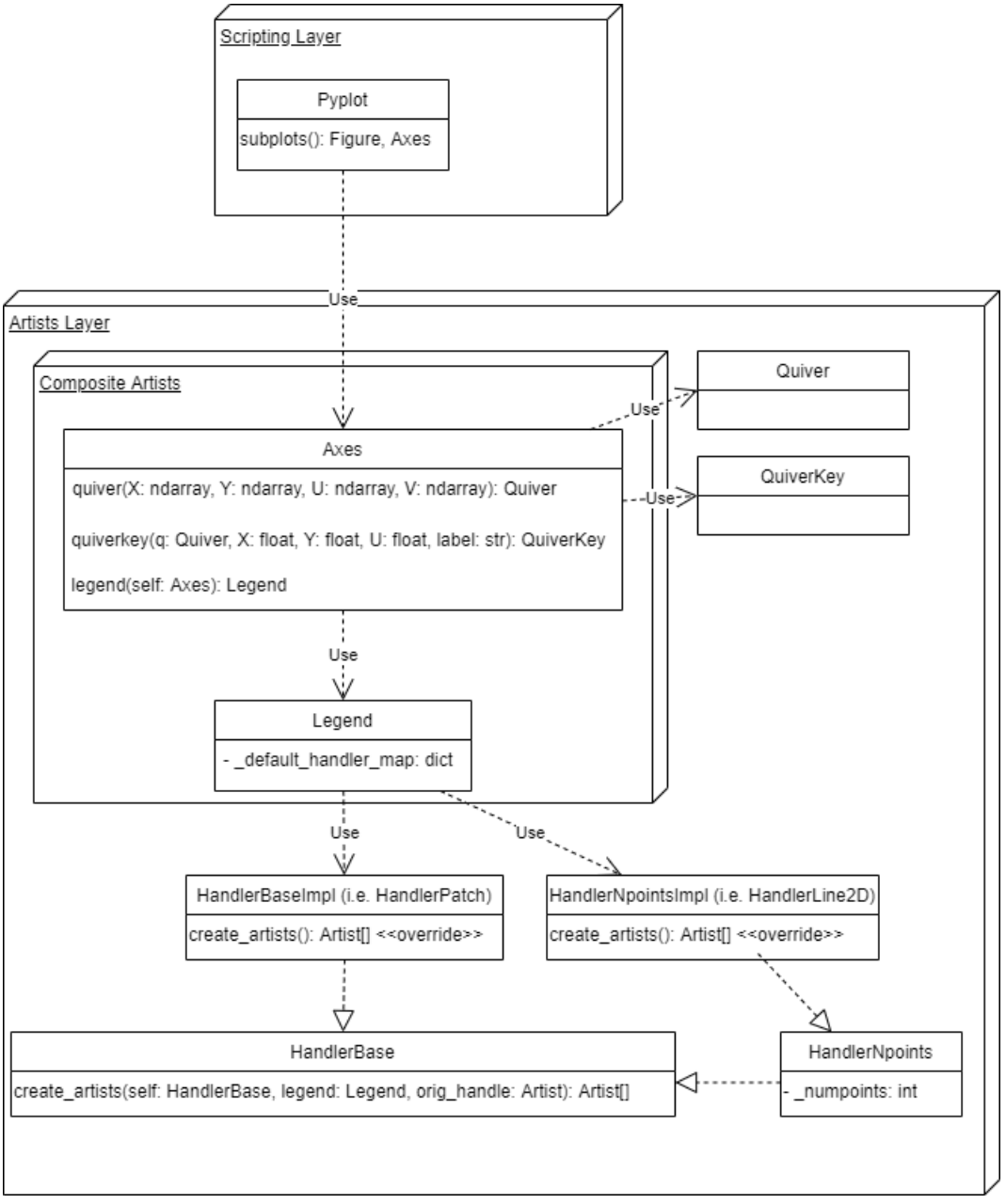
When legend([handler1, handler2]) is called with a list of artists as an argument, legend.py goes through this list and maps a `legend_handler` to each, that defines how to draw the legend key (the marker next to each legend label) in the legend box for that specific artist. The mapping of `artists` to `legend_handler` is defined in a dictionary in legend.py, `_default_handler_map`.

Currently there does not exist a handler for the `Quiverkey` object, so the fix would be to add a handler, as defined in legend_handler.py, that defines how to draw Quiverkey in the legend box. It will respect the length, color, and all other modifications of the original Quiverkey instance except those that deal with positioning since that will be taken care of at the legend box level.

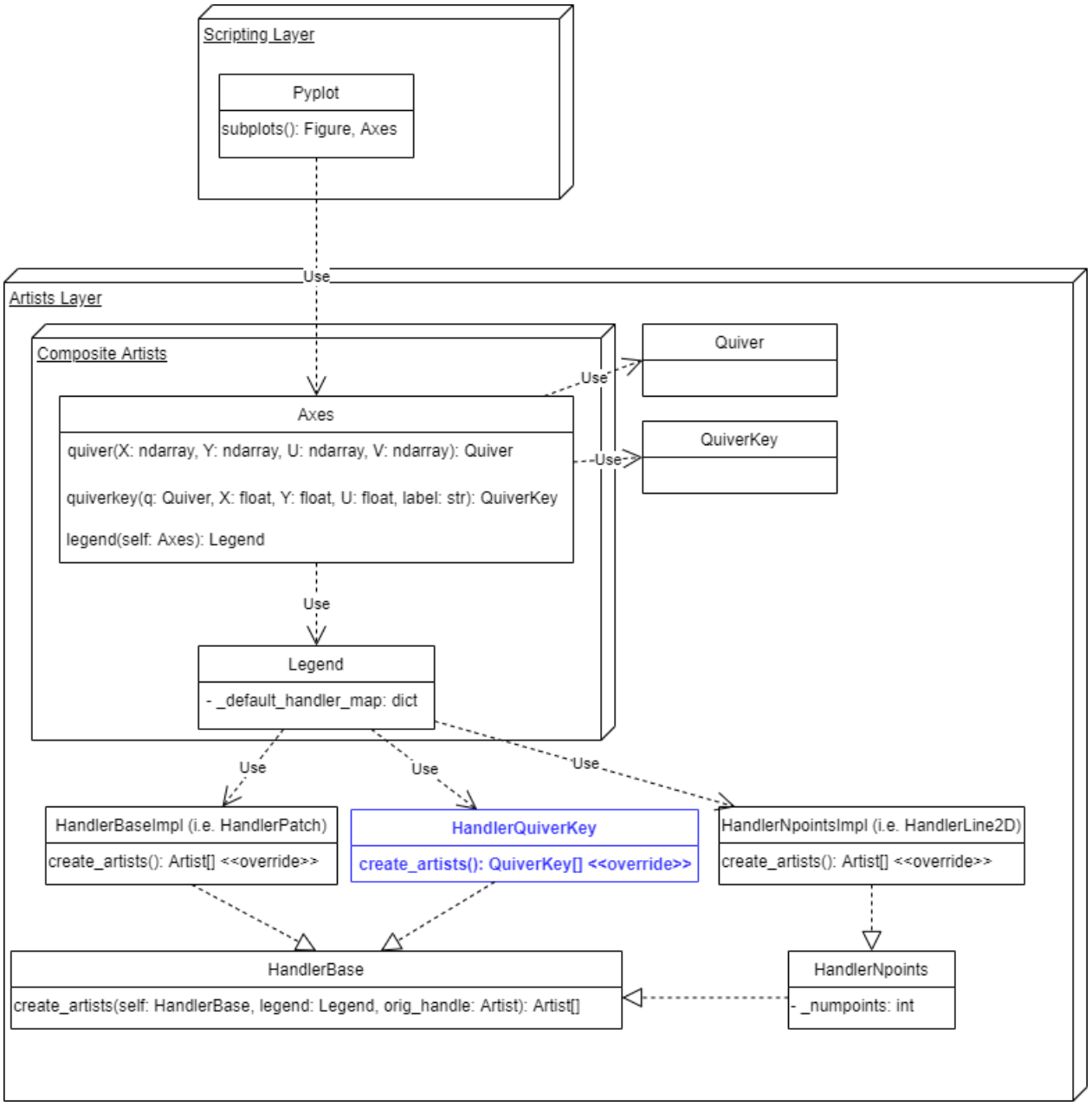
Another issue we will have to deal with is the fact that just calling `ax.quiverkey` adds a quiver key to the axes, and if `legend()` is called afterwards we don't want

the key to be drawn twice; once inside the legend box and once at the quiver key's provided location. Since the `legend_handler` gets the original quiver key instance passed to it, we should be able to just call its remove() to remove it from the graph, leaving us with just the one quiver key in the legend box.

Present architecture:



Proposed changes: (Changes outlined in blue)



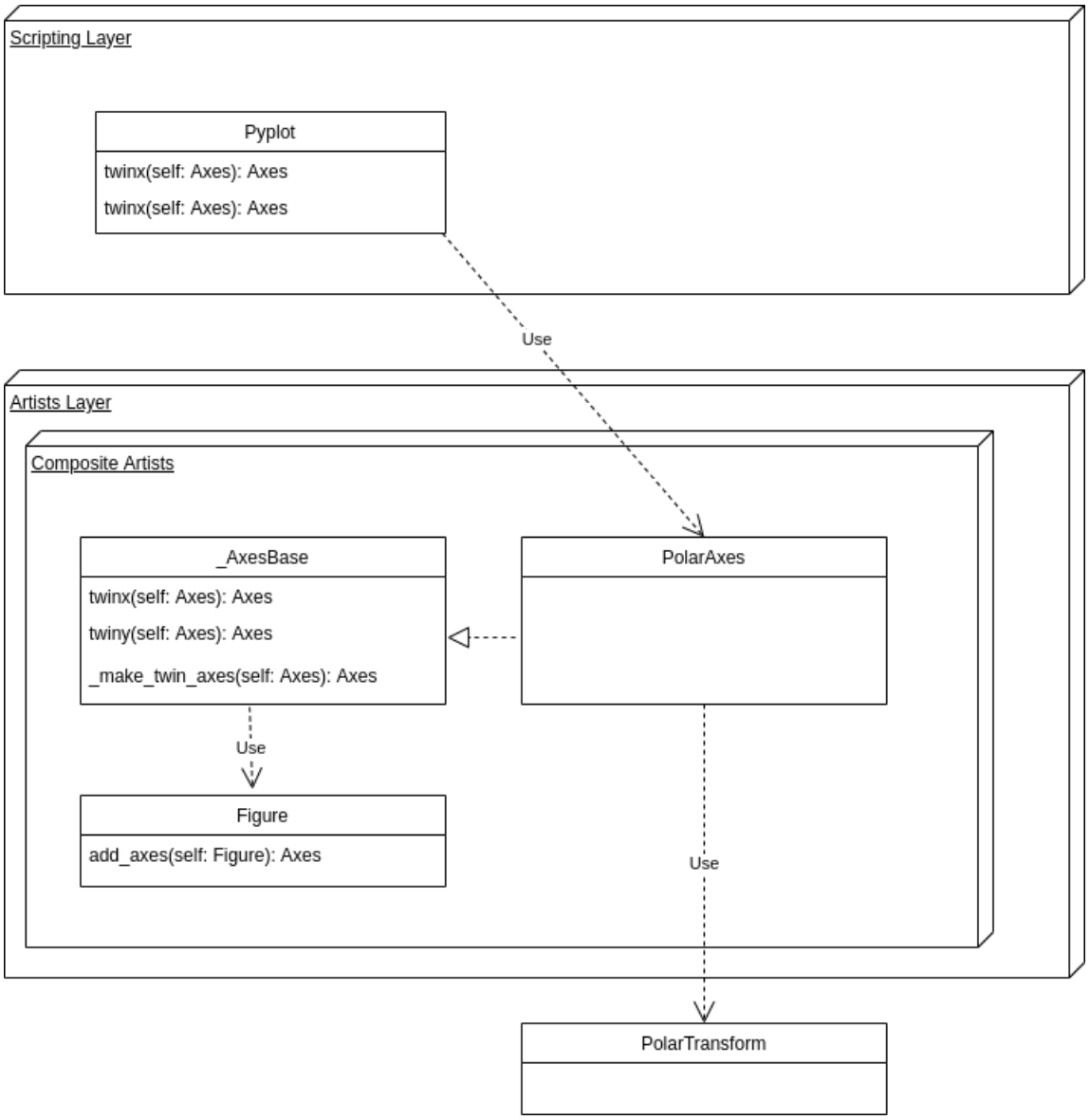
2.2 Add twin axis to polar graph

The main modification we would make would be to add a `twinpolar()` function in `pyplot.py` and that would call a new `twinpolar()` in PolarAxes.

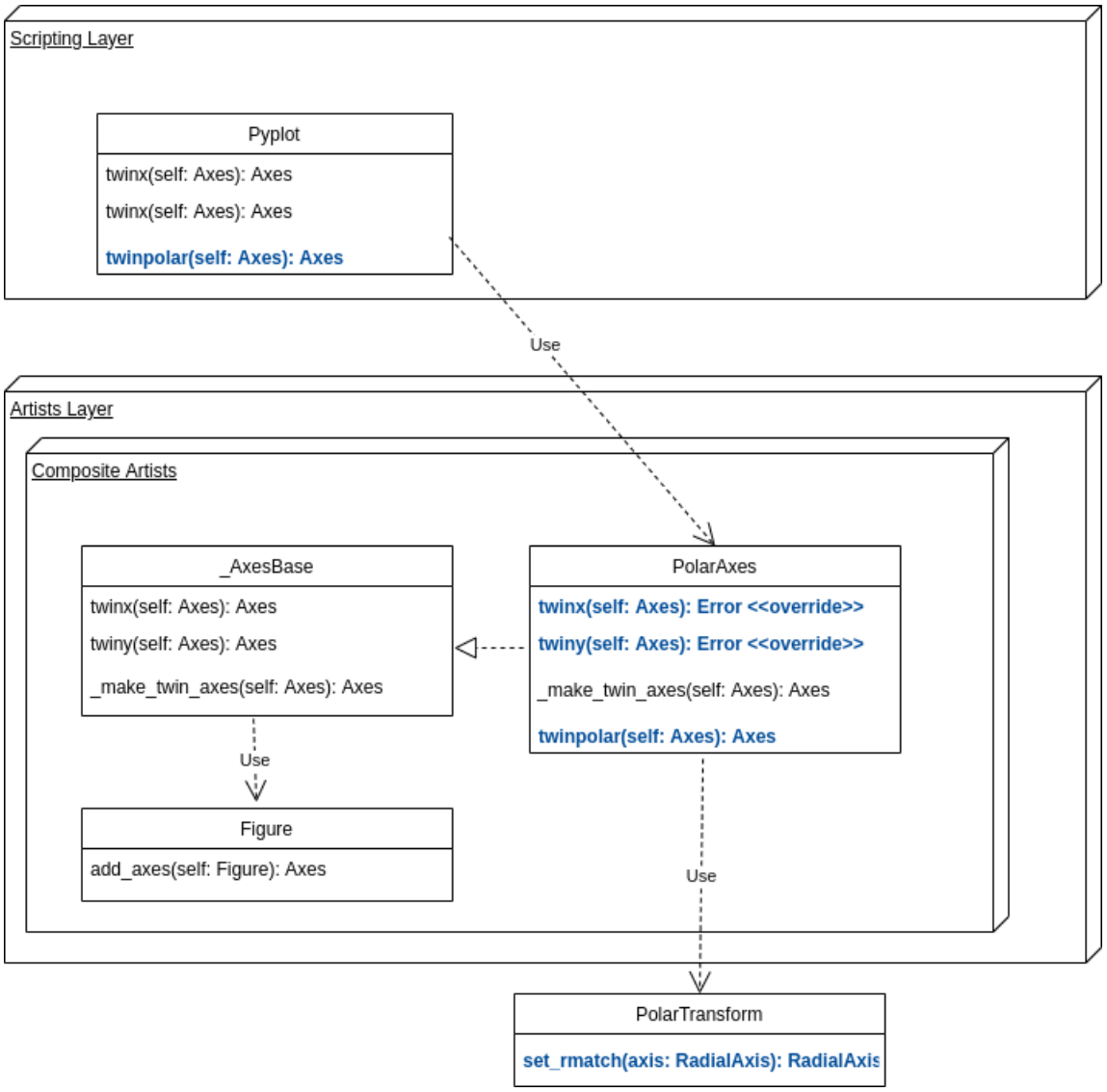
Inside of `PolarAxes.twinpolar()`, we would call `figure.add_axes()` making sure to pass the correct arguments so that it generates another radial axes for the polar graph. Then we would hide the xaxis and rotate the `_r_label_position` so its angled in a different direction then the original. Finally we would need to adjust the `r_lim` of the new radial axis so the grid lines aligned with the original. To do this we would create a new function in `projections/polar.py` called `set_rmatch()` which would take in the original r-axis and would align the new r-axis gridlines to the old ones.

Additionally, we would override the two methods, `twinx` and `twiny` inside of PolarAxes so that they would throw errors when called, since it doesn't make sense to add an additional x or y axis when the graph doesn't have an x or y axis.

Present architecture:



Proposed changes: (Changes outlined in blue)



3 Reasons for choosing quiverkey feature

We decided to choose the quiverkey feature for several reasons. The first reason why we choose quiverkey over the twin polar axes is that we believe quiverkeys scope is much more realistic for our project timeline. The quiverkey feature is confined to a single area (the legend) and its clear what changes we would have to make to add the feature. Another reason why we selected quiverkey is that we feel that the architecture surrounding `legend_handler` is high quality and extensibility was a first-class citizen in its design. The system was made with the idea that new artists would be added to the legend, and creating `HandlerQuiverKey` is a clean efficient way to implement our feature. Additionally, since this feature is confined to a single portion of the graph, we believe we will be able to create a thorough test suite with complete coverage of our feature. The combination scope, architectural design, and testability of quiverkey give us the best chance of producing a useful, high-quality feature, and this is why we selected it as our focus for deliverable 4.

We didn't choose twin polar axes for our feature due to the scope of implementing a change, and the risk associated with making large changes to a complicated graph type. The implementation of twin axes seemed sub-optimal to our team because it essentially creates a clone of the graph and then hides all the parts except the single axes we want to use. It would be much better if only the single-axis we wanted was cloned and it was added to the original graph instead of having two separate sets of axes and trying to keep them synced up. We realized the scope of overhauling the entire twin axes system was too large for our timeline and we didn't want to implement a sub-optimal solution. Another reason we didn't choose twin polar axes is that we believed the feature carried a large amount of risk, by adding twin axes support to a polar graph we would be affecting all parts of the graph. Since it would be a new feature we wouldn't be able to rely on previous regression tests to catch bugs in our code and we hypothesized that there was a high likely-hood of our team missing something in our test suite and therefore introducing bugs into the feature.

4 Acceptance Tests

Setup

All the acceptance tests require there to be a Quiver graph plotted, as well as a quiverkey, as described below, without which a quiverkey can't be added to the legend.

1. Plot a 2D field of arrows

Create the data required and plot it using `plt.quiver()`, where `plt` is pyplot or an instance of Axes. The data is four 2D arrays representing the x and y coordinates of the arrows and the x and y components of their direction vectors. To easily generate this data for the purposes of this acceptance test you can import numpy and call:

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

2. Plot a quiver key

Plot the quiver key by passing in the Quiver instance `q`, x and y coordinates for where to place it in the figure, a length and a label for the key that conveys its length:

```
ax.quiverkey(q, X=0.3, Y=1.1, U=10, label='10 unit long quiver key')
```

Quiverkey also accepts optional parameters like angle and width as well as all the Quiver optional arguments like pivot, headwidth, etc., to customize the key.

At this point the graph will display a field of 2D arrows and a quiverkey at the coordinates and with the properties you specified.

The following will test our feature.

Tests

For each of the following tests, the quiver key marker drawn in the legend should respect all the properties that customize the arrow that were provided in the arguments to the call to quiverkey(), except those that control its location or affect the positioning of the label, which will be handled by the legend itself.

- calling `ax.legend()` or `ax.legend([qk])` should draw a legend that has an entry with the quiverkey arrow as the marker and its label as the label. No other quiverkey should be drawn anywhere on the graph.
- calling `ax.legend([qk], ['other label'])` should draw a legend that has an entry with the quiverkey arrow as the marker and "other label" as the label. No other quiverkey should be drawn anywhere on the graph.
- if you make neither of those function calls the quiver key should still be drawn, at the the x,y coordinates that were specified and with the label positioning properties that were specified.

5 Architecture Revisted

As we have investigated various issues and potential features, we can confirm that matplotlib clearly follows a three layered open architecture. This is evidenced by the fact that all the issues we've investigated reside in the Artist layer, and required no detailed knowledge of any of the adjacent layers.

In particular, in the Artist layer, we noticed many instances of the interface segregation principle. Taking a look at what we've learned from the two issues above, although `Legend` uses the various `HandlerBase` implementations, and `Pyplot` uses various `Axes` implementations, in reality, the dependency is really on the abstract classes the implementations extend. `HandleQuiverKey` inherits `_HandlerBase`, and `PolarAxes` inherits `Axes` which inherits `_AxesBase`.
If we look at `Pyplot` for the second issue, we can see that it calls `ax.twinx()`, which calls the `twinx()` function defined in `_AxesBase`. Although functions/attributes are added and overridden as we go down the inheritance tree, regarding Axes, `Pyplot` is only aware of the methods available in the `_AxesBase` interface. As long as the different types of Axes respect the interface, the Scripting layer and the Axes components will remain decoupled.
We can also see that interface segregation is also used within the Backend layer, where Artists in the Artist layer depend on the the functionalities defined in `RendererBase`, `FigureCanvaseBase`, etc. in order to draw an output.

System architecture: