D01LAR BILLS/Team 30

CSCD01 WINTER 2020

Deliverable 4

Submitted To:
Dr. Anya Tafliovich

Submitted By :
Joseph Sokolon
Wesley Ma
Raya Farhadi
Edgar Sarkisian

Contents

1 User Guide

An interactive demo for adding a quiverkey into a legend can be found at examples/images_contours_and_fields/quiver_legend_demo.py and will be shown in the documentation in matplotlib's website at https://matplotlib.org/gallery/images_contours_and_fields/quiver_legend_demo.html

To create a legend containing a quiver key entry, you must create a quiverkey instance and call the legend() method from Axes or Figure.

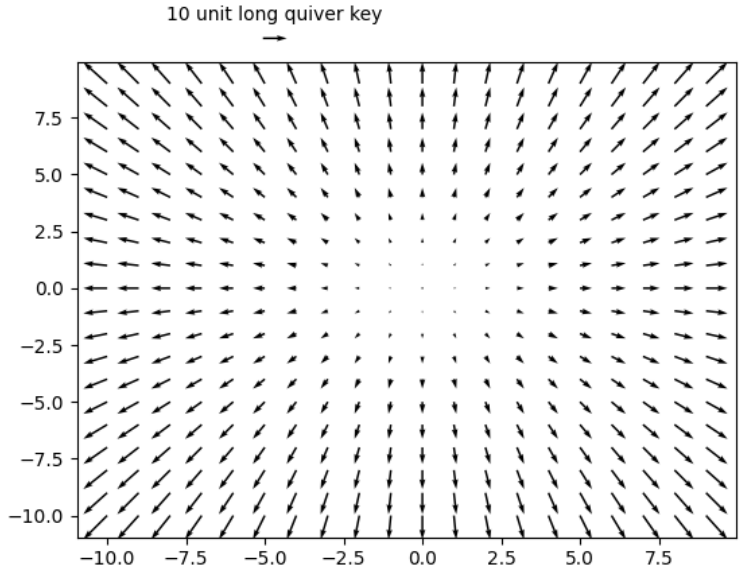
First draw a quiver plot and create a quiver key object like so:

```
from matplotlib import pyplot as plt


...
# X, Y, U, V is the data of the arrows to be plotted
q = plt.quiver(X, Y, U, V)

# to draw a quiver key, you must provide a label
plt.quiverkey(q, X=0.3, Y=1.1, U=10, label='10 unit long quiver key')
```

This will produce a figure with a field of arrows and a "key" arrow, like so:



1.1 Customizing the quiver key

The quiver key object can be customized when it is drawn to the graph as per the documentation, but for our purposes, since we will be adding it to the legend, only the customization that makes sense is supported and will be carried over to the key shown in the legend.
Here are some of the options:

1.1.1 `U`: float

Define the length of the quiver key. This will reflect the same scale as in the graph. If you specify 10 units, it will be the same length as an arrow 10 units long plotted on the graph.

1.1.2 `angle`: float

Default: 0. Define the angle of the quiver key, in degrees counter-clockwise from the x-axis.

1.1.3 `color`: color

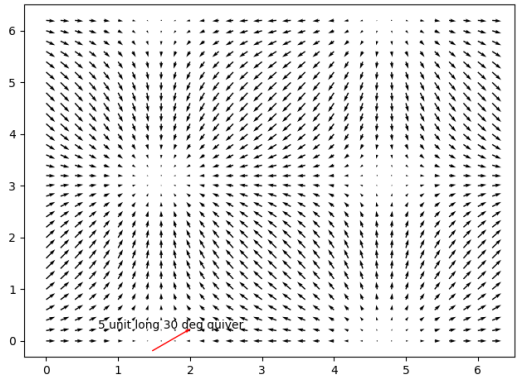
Define the face and edge colors of the quiver key. When its not provided, the face and edge colors of `q` is used.

Example:

```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
U = np.cos(X)
V = np.sin(Y)
q = plt.quiver(X, Y, U, V, units='width')
qk = plt.quiverkey(q, 0.3, 0.05, 5, '5 unit long 30 deg quiver',
                   angle=30, color='red')
```

produces the following:



Next, call `legend(handles)` or `legend(handles, labels)`.

1.2 `legend()`

Calling `legend()` draws a legend and adds an entry to it for each of the elements in your `handles` array.

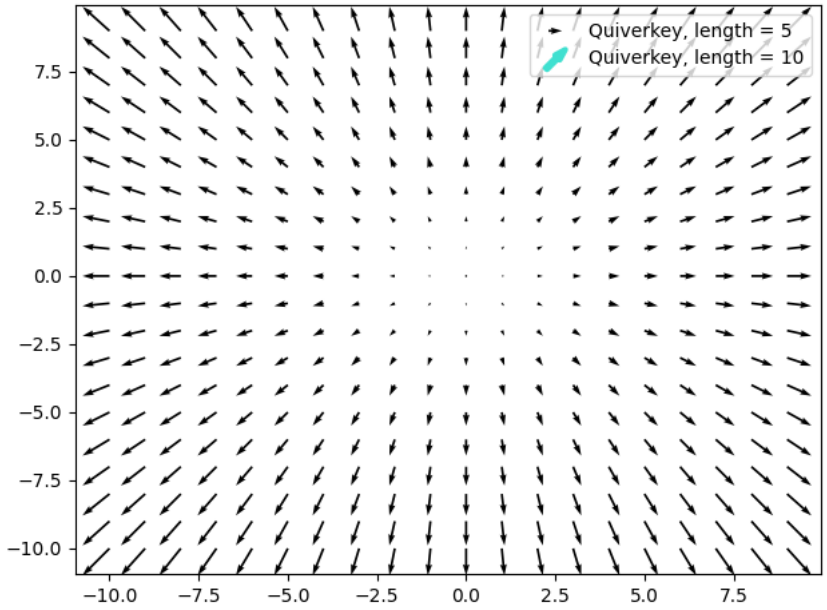
The `legend()` method can be called with or without the `labels` array. Include it if you'd like to override the existing labels of the elements in `handles`.

1.2.1 `legend(handles)`

```
# clear current axes
plt.cla()

qk1 = plt.quiverkey(q, X=0.3, Y=1.1, U=5, label='Quiverkey, length = 5')
qk2 = plt.quiverkey(q, X=0.6, Y=1.1, U=10, angle=45.0, color='turquoise',
                    linewidth=3, label='Quiverkey, length = 10')

plt.legend(handles=[qk1, qk2])
```



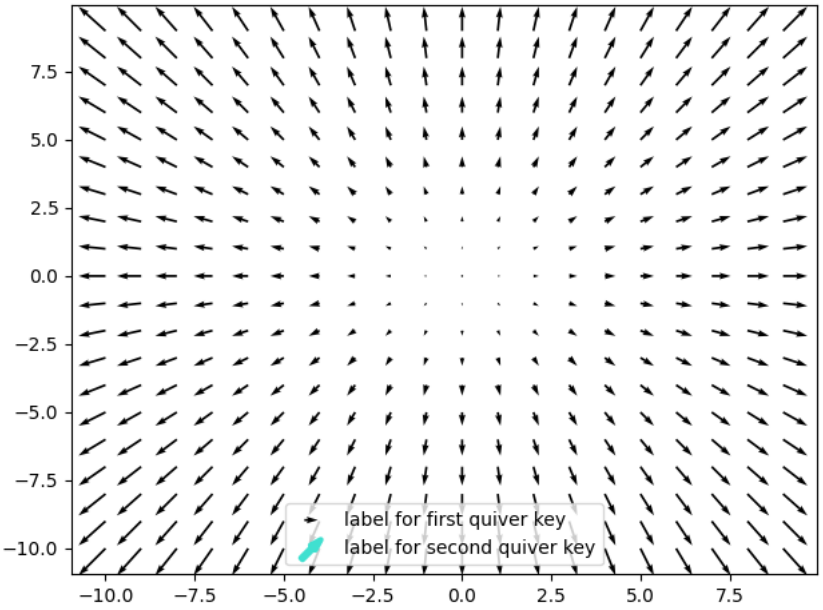
1.2.2 `legend(handles, labels)`

To override the existing labels of the elements in `handles`, pass a list of labels:

```
# clear current axes
plt.cla()

qk1 = plt.quiverkey(q, X=0.3, Y=1.1, U=5, label='Quiverkey, length = 5')
qk2 = plt.quiverkey(q, X=0.6, Y=1.1, U=10, angle=45.0, color='turquoise',
                    linewidth=3, label='Quiverkey, length = 10')

plt.legend([qk1, qk2], ['label for first quiver key',
                        'label for second quiver key'])
```



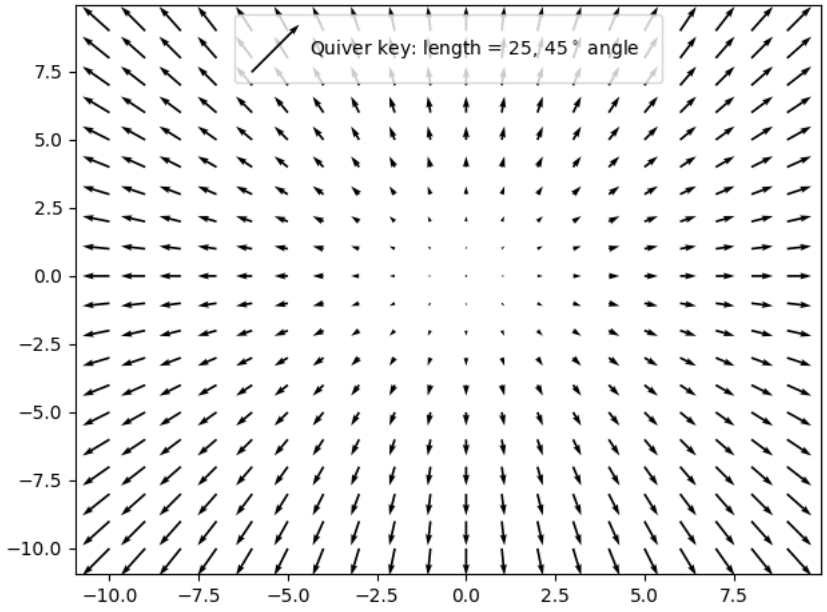
As you can see, either of the signatures will preserve the customizations to the quiverkey except the coordinates, since those are taken care of by the positioning of the legend.

Inserting large quiver keys into the legend

When adding long quiver keys to a legend, you may need to tweak the handlelength, borderpad or other parameters of legend() to make the arrow fit in the legend.

```
#clear current axes
plt.cla()
qk = plt.quiverkey(q, X=0, Y=0, U=25, angle=45.0,
                   label=r'Quiver key: length = 25, 45$^\circ$ angle')

plt.legend(handles=[qk], loc='upper center', handlelength=2, borderpad=1.2)
```


Quiver key: length = 25, 45 ° angle
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

2 Design

The implementation of this feature was more work than originally anticipated.
We encountered challenges throughout the three steps for implementing this feature:

1. Creating a new `QuiverKey` to be put in the legend
2. Positioning the new `QuiverKey` in the Legend
3. Making the label for the legend entry

These steps were meant to be carried out within the new `QuiverkeyHandler` class, in the `create_artists()` method.

2.1 Creating a new `QuiverKey` to be put in the legend

We first needed to create a copy of the original QuiverKey inside . This copy had to preserve all its original properties, and was meant to be re-positioned into the Legend's frame. To accomplish this, we added and overrode `Artist.update_from()` in the QuiverKey class to also copy over QuiverKey's own attributes.
Copying the original object was not trivial to figure out for two main reasons:

1. The super method that calls `Artist.update_from()` (ie. `Handler.update_prop()`), was overridden in most of the existing legend handler implementations. In fact, in most cases, the functionality to copy attributes over was implemented in these overridden functions.
2. Most of the artists used in the legend handler implementations don't even override their `.update_from()` methods.

For this step, the design of the Artists made it difficult for us to fully understand how to create copies of them. We believe a better design would be that all Artist implementations should override their `update_from()` method, and the legend handlers should be able to call existing inherited methods to minimize the amount of extra logic. Creating a new legend handler for Artists should not require devs to be familiar with the implementation of said Artist.

2.2 Positioning the new `QuiverKey` in the Legend

We needed to remap the coordinates of the original QuiverKey (which could be based off the axes) to that of the legend's frame. This Transform is provided through the `trans` param in `create_artists()`. Typically it's sufficient to simply call `new_artist.set_transform(trans)` directly in `create_artists()`, however for QuiverKey, the coordinate system needed to be set internally.

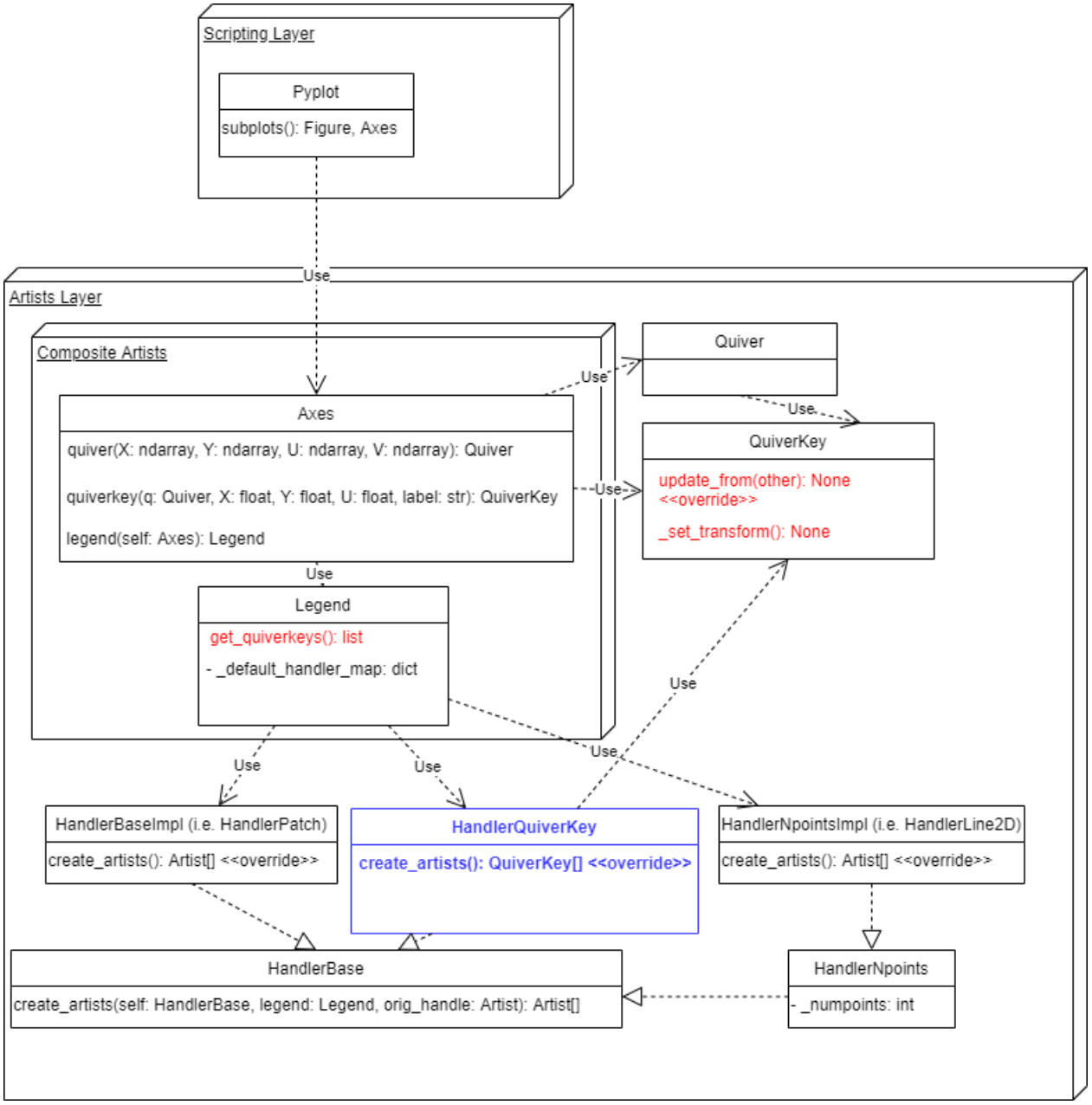
On init, QuiverKey takes in a parameter, `coord` ('Inches', 'Axes', ...) and calls it's own private method `_set_transform()` to set a transform to based off the given `coord` String. Our fix was to modify the `_set_transform()` method to also check if the `coord` attribute was an AffineTransform, and to use it if it is. Thus, in `create_artists()`, we simply set the new QuiverKey's `coord` attribute to be `trans`. The actual position of the `QuiverKey` was done by setting the X and Y attributes based off the given position variables.

2.3 Making the label for the legend entry

One of our issues was that the legend wasn't able to detect Quiverkeys label. The legend gets the label for each label element from either a passed in parameter or an inherited `get_label()` method from artists. The problem was that the inherited method was expecting a `._label` property while in QuiverKey the label was saved as a `.label` property. We saved the label in both properties so that the artists method could detect the label properly.

In the legend, legend elements are drawn in two parts, the graphic and the label. This was a problem for the current `QuiverKey` we were adding since it already drew its own label. Since the displaying of the label should be handled by the legend, we had to stop `HandlerQuiverKey` from drawing a label. For our implementation we created a new Quiverkey with an empty label. We then saved this label text in a variable, and after copying over all the properties from the original QuiverKey using `update_from()` we re assigned the label text to be the empty one rather then the label text from the original QuiverKey.

The following is the UML of the implementation (Blue - anticipated in Deliv. 3, Red - new changes):


Scripting Layer
Pyplot
subplots(): Figure, Axes
Use
Artists Layer
Composite Artists
Axes
quiver(X: ndarray, Y: ndarray, U: ndarray, V: ndarray): Quiver
quiverkey(q: Quiver, X: float, Y: float, U: float, label: str): QuiverKey
legend(self: Axes): Legend
Use
Legend
get_quiverkeys(): list
- _default_handler_map: dict
Quiver
Use
QuiverKey
update_from(other): None
<<override>>
_set_transform(): None
HandlerBaseImpl (i.e. HandlerPatch)
create_artists(): Artist[] <<override>>
HandlerQuiverKey
create_artists(): QuiverKey[] <<override>>
HandlerNpointsImpl (i.e. HandlerLine2D)
create_artists(): Artist[] <<override>>
HandlerBase
create_artists(self: HandlerBase, legend: Legend, orig_handle: Artist): Artist[]
HandlerNpoints
- _numpoints: int

3 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, Y = np.meshgrid(np.arange(0, 2 * np.pi, 1),
                   np.arange(0, 2 * np.pi, 1))
U = np.cos(X)
V = np.sin(Y)

Q = ax.quiver(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:

```
qk = ax.quiverkey(Q, 0.9, 0.8, U=10, label='QK length = 10', labelpos='E')
```

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([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.

4 Unit Tests

Setup

All the unit tests will use the same set up as the acceptance test suite, with the exception of adding a colour to the quiverkey:

```
qk = ax.quiverkey(Q, 0.9, 0.8, U=10, label='QK length = 10', labelpos='E',
                  color='red')
```

Tests

The following tests will ensure that the quiver key marker drawn in the legend respects 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([qk])` should draw a legend that has an entry with a quiverkey arrow, with the same length as the arrow passed in, as the marker.
- calling `ax.legend([qk])` should draw a legend that has an entry with a quiverkey arrow, with the same colour as the arrow passed in, as the marker.
- calling `ax.legend([qk])` should draw a legend that has an entry with a quiverkey arrow, with the same angle as the arrow passed in, as the marker.

- calling `ax.legend([qk])` should draw a legend that has an entry with a quiverkey arrow, with the same distance between the arrow and the label as the arrow passed in, as the marker.