Team Anyalgorithms - Major Feature Planning

Deliverable 3

Ahmed Sirajuddin

Jon An

Ricky Chen

Robert Peng

Table of Contents

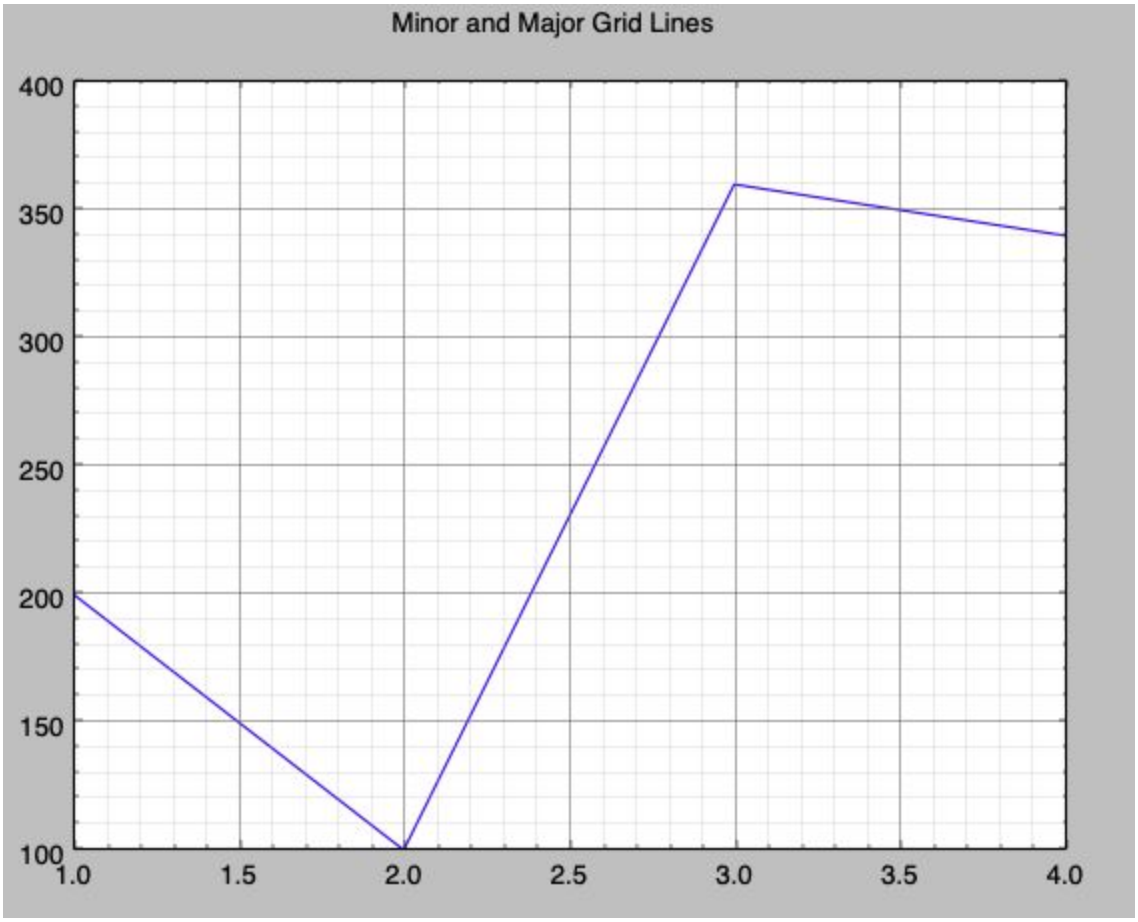
Issue 1

#13919: Impossible to configure minor/major grid line style independently in rcParams

Link: https://github.com/matplotlib/matplotlib/issues/13919

Description

As of now, users can customize the appearance of grids in Matplotlib in several ways: by changing its color, its linewidth, its linestyle and its opacity. On a grid, there are major grid lines and minor grid lines. The image below demonstrates the difference between major and minor grid lines:



The minor gridlines are seen as light gray, and have a lower opacity. The major gridlines are seen as dark gray and can be seen more visibly. In order to customize the look of the major

and minor gridlines, the Matplotlib user needs to do it programatically as there are no exact keys for them. For example, to create the grid above, the following code was used:

```
import matplotlib.pyplot as plt

# The Data
x = [1, 2, 3, 4]
y = [200, 100, 360, 340]

# Create the figure and axes objects
fig, ax = plt.subplots(1, figsize=(8, 6))
fig.suptitle('Minor and Major Grid Lines')

# Plot the data
ax.plot(x,y)

# Show the major grid lines with dark grey lines
plt.grid(b=True, which='major', color='#666666', linestyle='-')

# Show the minor grid lines with very faint and almost transparent
grey lines
plt.minorticks_on()
plt.grid(b=True, which='minor', color='#999999', linestyle='-',
alpha=0.3)

plt.show()
```

As seen in the above code, the Matplotlib user has to pass through the `which` argument either a `major`, `minor`, or `both` string, and then change the style of the respective grid lines. The code highlighted in blue changes the major grid line color to `#666666` which is dark gray, and the red code changes the minor grid line color to `#999999` which is light gray.

Another way to customize grid lines (which only applies to major gridlines) is to use the `rcParams` key:

```
from matplotlib import rcParams
rcParams['grid.color'] = '#666666'
rcParams['grid.linestyle'] = '-'
```

The feature requested in this issue is for Matplotlib users to be able to set the major and minor grid line styles using keys, rather than doing it programmatically through these functions.

For example, after implementing this feature, users should be able to change the color and style of both major and minor grid lines like so:

```
from matplotlib import rcParams
rcParams['grid.major.color'] = '#666666'
rcParams['grid.major.linestyle'] = '-'

rcParams['grid.minor.color'] = '#999999'
rcParams['grid.major.linestyle'] = '-'
rcParams['grid.minor.alpha'] = '0.3'
```

This is beneficial because it makes the code more consistent, and, when dealing with multiple grids and lines, it will be more efficient and clean to customize them using keys rather than doing it programatically.

Relevant Code

Certain parts of the code will need to be modified to implement this feature. First and foremost, the `rcParams` will need to be modified. This is the file relevant to rcParams: https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/__init__.py#L632

It is in `matplotlib/__init__.py`. This file uses `rcsetup.py`. This is where the `rcParams` dictionary is created. For a specific example, these lines of code will be relevant to implementing the new feature: https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/rcsetup.py#L1356-L1359

Currently, the above lines show the 4 ways to customize grid line styles. The feature will modify / add the following:

```
'grid.minor.color':
'grid.major.color':
'grid.minor.linestyle':
'grid.major.linestyle':
'grid.minor.linewidth':
'grid.major.linewidth':
'grid.minor.alpha':
'grid.major.alpha':
```

The above code will be in the `defaultParams` dictionary, which is used to initialize `rcParams`.

Next, the source code that directly changes major and minor grid line styles using `rcParams` will need to be modified. As the Matplotlib documentation mentions here: https://matplotlib.org/tutorials/intermediate/artists.html#tick-containers

"The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The `Tick` contains the tick and grid line instances,"

From this documentation, we located the source code that handles gridlines to be here: https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/axis.py#L135-L143

Therefore, `matplotlib.axis.py` will also need to be modified to implement this feature. More specifically, this file will need to be modified to include

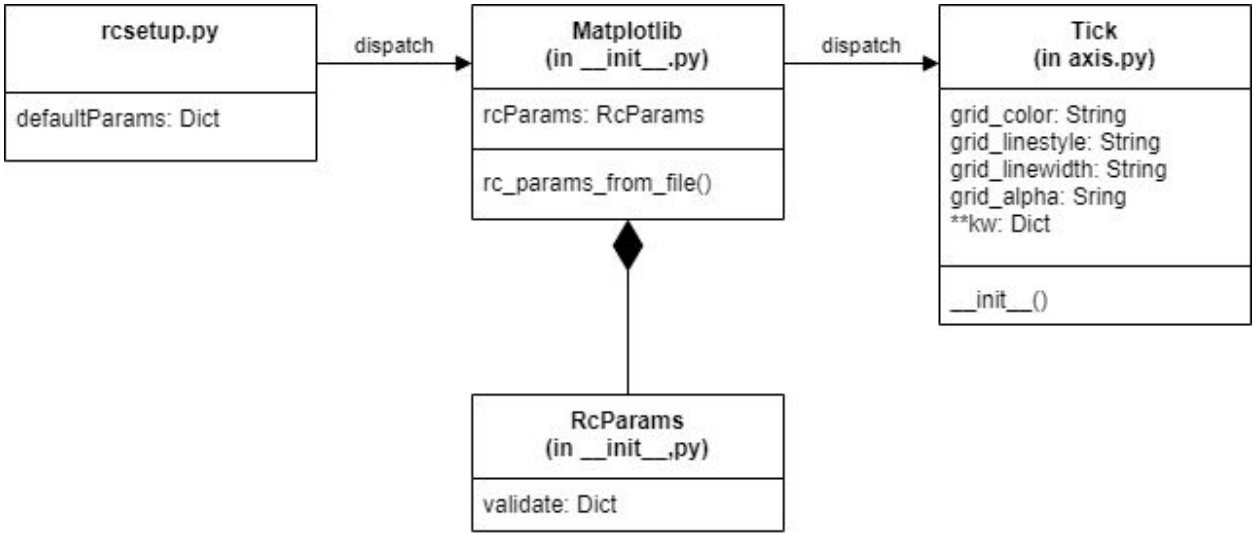
```
grid_minor_color
grid_minor_linestyle
grid_minor_linewidth
Grid_minor_alpha
```

The `__init__` of `Tick` and the `grid_kw` line here: https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/axis.py#L143

will also need to be modified to reflect the additional styling properties of minor grid lines.

As well, we anticipate being able to use code related to `xtick` and `ytick` as a guide and reference on how to properly implement minor and major style customization for grid lines, as the following line leads us to show that minor and major styling of ticks already exists in some locations of Matplotlib: https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/rcsetup.py#L1347

Below is a diagram that shows the interactions of the relevant and modified files for this feature implementation.



Issue 2

[Feature request] Use Enums for enumerated types #14642

https://github.com/matplotlib/matplotlib/issues/14642

Description:

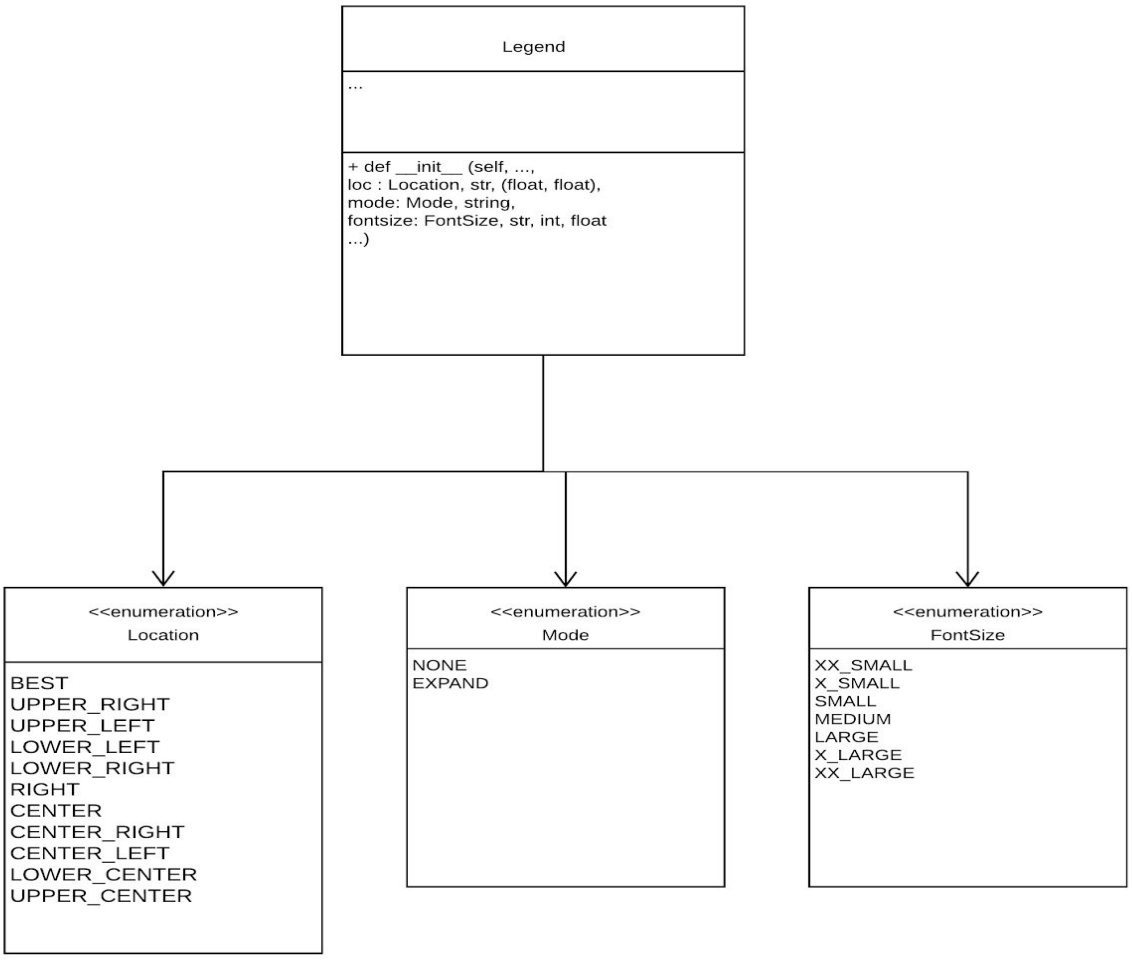
Currently, when using Matplotlib features such as legend, users will have to type out the string that corresponds to the value associated with it. If the user is not familiar with Matplotlib, they will have to search up the docs every time to find out which axis on which they want the legend. There are also other problems with typing out the strings which include (by clbarnes on GitHub)

1. Values do not autocomplete and cannot be checked for correctness by most IDEs.
2. Remembering the canonical dashes, underscores and capitals is unnecessarily complicated.

Using enums would solve these issues as users could, instead, let their IDE do the work and make Matplotlib to be more user friendly. This is beneficial to the overall longevity of Matplotlib as making it user friendly would attract and retain new users.

Relevant Code

Going off of the example from clbarnes on GitHub, we would need to create a new enumeration that would be used by `legend.py`. The two would be associated and would more or less resemble this as UML.



Calling the function would look more like this `plt.legend(loc=Location.LEFT, mode=Mode.EXPAND, fontsize=Fontsize.SMALL) [Taken from github issue link]`

So our plan would be to create the new enums inside of the file and using Legend would require these imports. Then after, we would need to map these values to the ones being used currently inside of the function.

Feature Chosen for Implementation

The feature our group chose to implement is issue 1 (#13919: Impossible to configure minor/major grid line style independently in rcParams). The link to the issue is here. The feature we want to implement is being able to customize major and minor grid line style using rcParams keys. We decided to select this feature for implementation because it would make a common task (setting major and minor grid line styles) a lot more efficient, consistent, and simple.

Implementing this feature also will not affect previous code, as the second, less efficient option (of programmatically customizing major and minor grid line styles using functions) will still exist and will not be affected.

The reason as to why our team ultimately rejected issue 2 was that it is too verbose and unclean as the code base grows. Matplotlib is currently very large as is and the introduction of enums for every mapping used would be an enormous overhaul, as well as making it very difficult to read. As it is currently, users of mpl (Matplotlib) only need to look at the documentation briefly to understand, whereas implementing this change would result in a much longer time investment in having to learn and understand arguments since looking at it can be confusing with all the different enums.

Tests

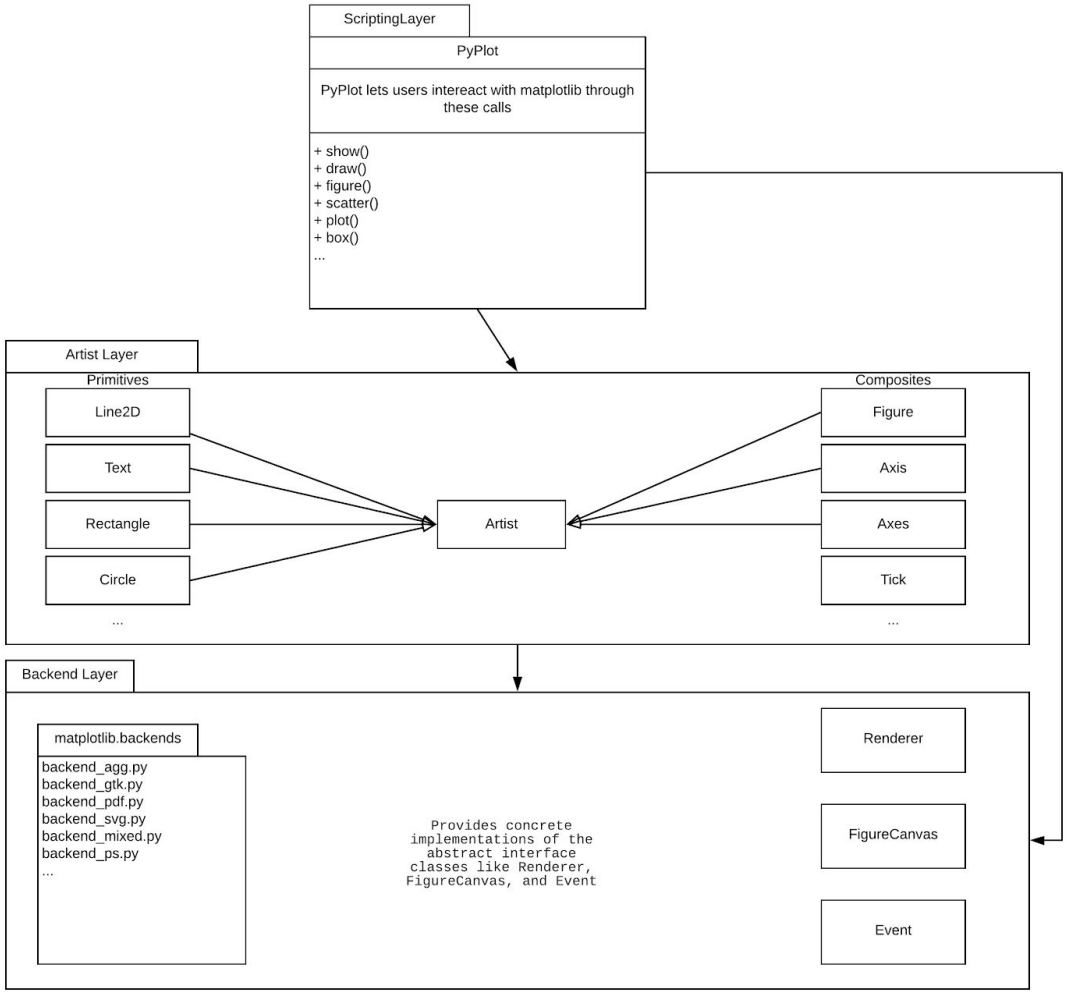
Acceptance Tests

```
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['grid.minor.color'] = '#ff5d00'
rcParams['grid.minor.linestyle'] = '--'
rcParams['grid.minor.alpha'] = '0.2'
rcParams['grid.minor.linewidth'] = '1'
rcParams['grid.major.color'] = '#666666'
rcParams['grid.major.linestyle'] = '-'
rcParams['grid.major.alpha'] = '1'
rcParams['grid.major.linewidth'] = '2'

# The Data
x = [1, 2, 3, 4]
y = [234, 124,368, 343]

# Create the figure and axes objects
fig, ax = plt.subplots(1, figsize=(8, 6))
fig.suptitle('Example Of Plot With Grid Lines')

# Plot the data
ax.plot(x,y)

# Show the grid lines as dark grey lines
plt.grid(b=True)

plt.show()
```

1. Setup Matplotlib.
2. Create a .py file, copy paste the program above.
3. Run the file (for example, by opening terminal and typing `python gridExample.py`)
4. Verify that the minor grid lines are dashed lnes (e.g. - - - -), are of orange colour, with low opacity and thin width.
5. Verify that the major grid lines are solid lines (.e.g. -----), are of dark gray color, with high opacity and a width of 2.

The reason for choosing acceptance testing -- as opposed to unit or any other kind of testing -- is because these changes are made more specifically as a feature that users can use. Therefore it makes more sense that a user uses it in a case rather than a unit test that makes sure it works. It is more of a coding preference and making unit tests to make sure it “works” is not really testing the usefulness of this feature.

System Architecture



The architecture of Matplotlib is divided into three main sections: backend layer, artist layer, and scripting layer. These are stacked on top of one another.

The bottom-most layer is called the backend layer and provides the concrete implementations of the abstract interface classes, FigureCanvas, Renderer, and Event. FigureCanvas is essentially the canvas of Matplotlib on which graphs and pictures are drawn. The Renderer is basically the paintbrush that draws on the canvas by rendering its commands on FigureCanvas. Event handles inputs from the user such as keyboard input and mouse events.

The middle layer is called the artist layer and does the actual work. The artist layer takes the Renderer and puts the figures onto the canvas. It contains all the features such as title, lines, tick labels, images etc.

The top-most layer is called the scripting layer and is basically the UI of the program. It makes it easier for the user to interact and use all the APIs that are built into the backend of the program.

Design Patterns

Factory Pattern

The Matplotlib base class for all scales, defined in `scale.py`, has a factory that other classes would call (for example in the `axis.py`) to get the correct scale. There are many scales that other classes can get access to, for example, the following are included but not exclusive:

`LogitScale`
`SymmetricalLogScale`
`LogScale`
`FuncScale`
...
Dictionary of `scales`:
https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/scale.py#L624

`Scale_factory`:
https://github.com/matplotlib/matplotlib/blob/3fe3d1a33608977d7b7c09e37eb8a966c58e781a/lib/matplotlib/scale.py#L639

Bridge Pattern

Bridge Pattern is used to decouple abstraction from the implementation by providing a bridge structure between them. This design pattern can be observed by looking at the `Locator` class and `Axis` class. The `Locator` class illustrates the scales that are used when axises are created on a graph, where it is defined in the `Axis` class. The way the bridge design pattern is implemented is that each `Axis` object uses a `Locator` object which is interchangeable with other `Locator` objects. The `Locator` object sets the ticks on the axis, while the `Axis` object defines the axes of the graph.

`Axis`:
matplotlib/axis.py at master · matplotlib/matplotlib

`Locator`:
matplotlib/ticker.py at master · matplotlib/matplotlib