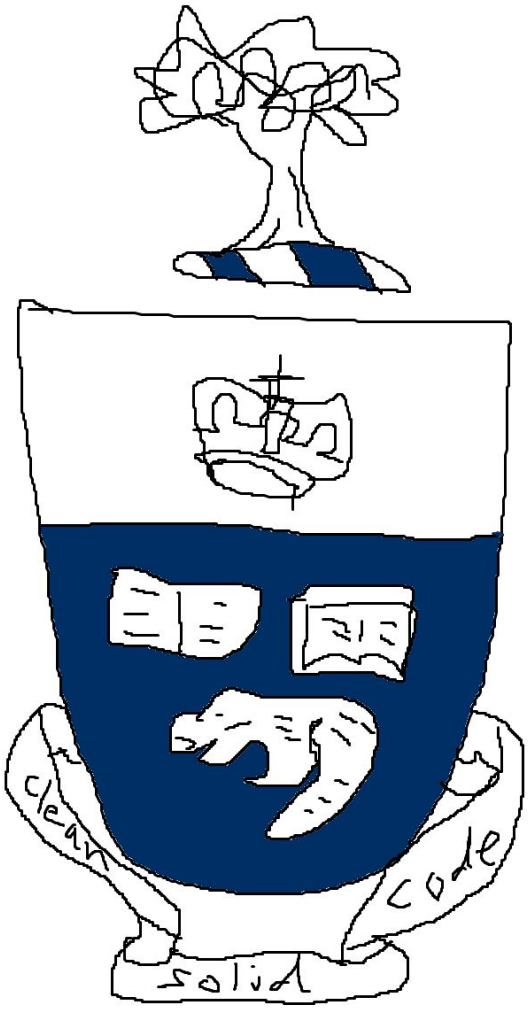


Deliverable 3: Selecting Features

Deliverable #3: Selecting Features
March 22th 2019
Team Members: Minqi Wang, Shuang Wu,
James Nicol, Xinrui Tong, Zixing Gong

Table of Contents

Feature 1 Outline

Issue Name: Option to place legend labels near to the data
Issue #: 12939
URL: https://github.com/matplotlib/matplotlib/issues/12939
Description
This feature request is to create a new API called label_line() that allows the user to place line labels at the end of the line, as opposed to restricting line labels to the legend. It was originally requested as an extension to the legends API but was prefered to be a separate feature similar to annotate or CLabel. This feature requires extensive work to ensure that there is no overlap in line labels and that line labels persist on zoom in/out.
UML Diagrams
The following UML sequence diagram and related class diagram shows the potential new classes and new methods' behaviours to existing stable classes.

Potential New Classes:

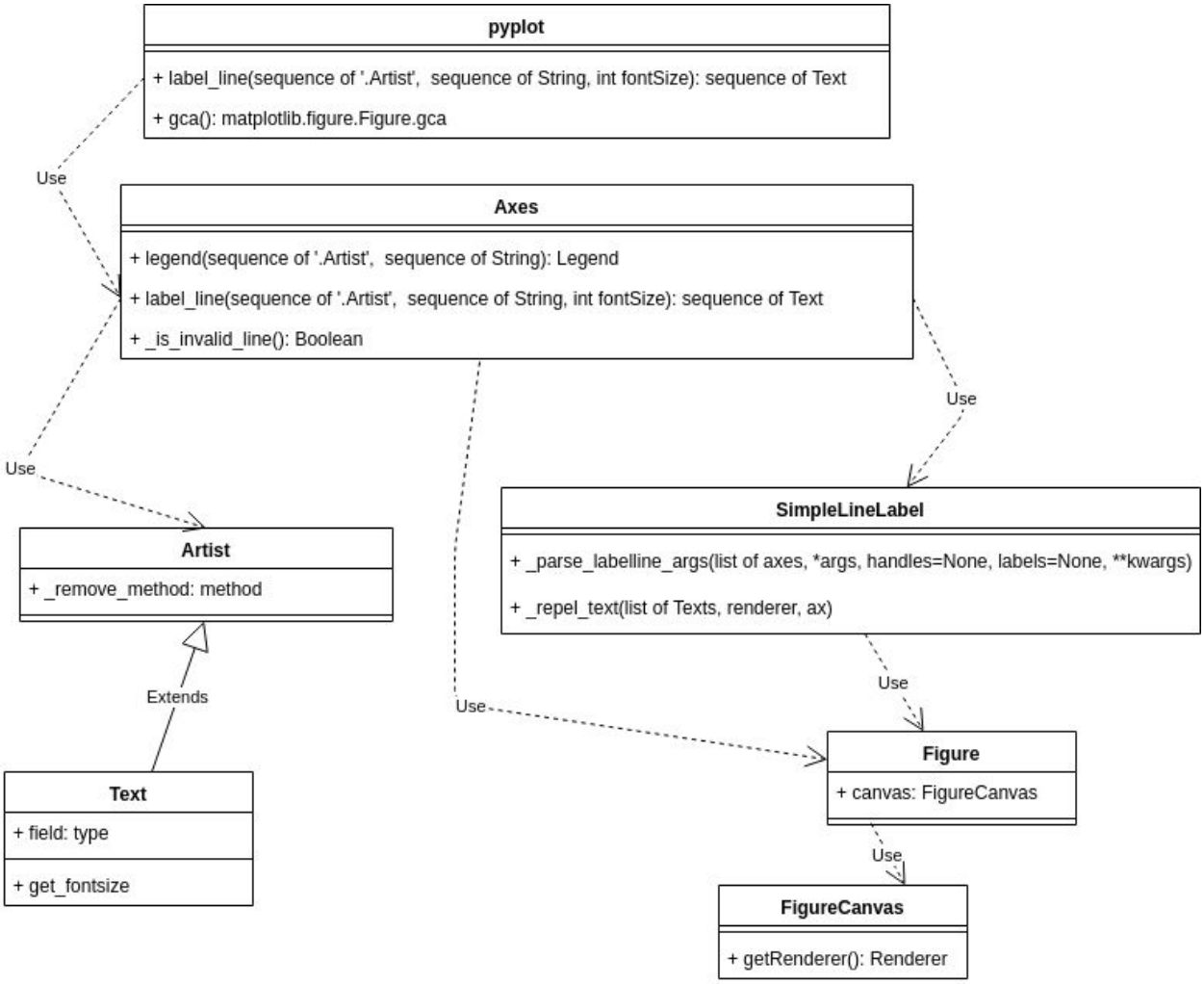
- SimpleLineLabel with methods: _parse_labelline_args(), _repel_text()

Potential New Method of Stable Classes:

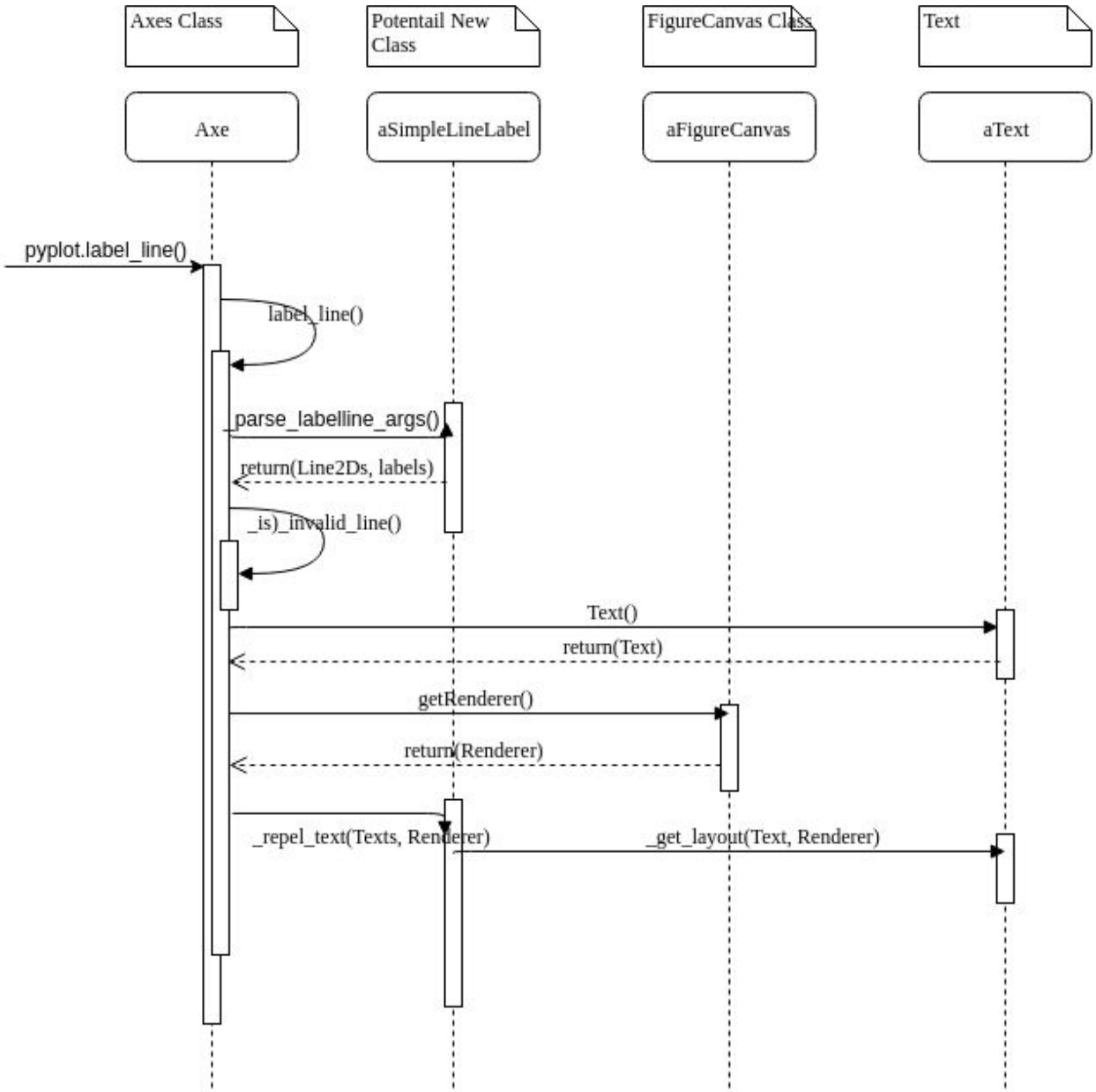
- label_line() in pyplot
- label_line(), _is_invalid_line() in Axes

Potential Method of Stable Classes in Use:

- get_fontsize() in Text
- getRenderer() in FigureCanvas


pyplot
+ label_line(sequence of '.Artist', sequence of String, int fontSize): sequence of Text
+ gca(): matplotlib.figure.Figure.gca
Use
Axes
+ legend(sequence of '.Artist', sequence of String): Legend
+ label_line(sequence of '.Artist', sequence of String, int fontSize): sequence of Text
+ _is_invalid_line(): Boolean
Use
Use
Artist
+ _remove_method: method
SimpleLineLabel
+ _parse_labelline_args(list of axes, *args, handles=None, labels=None, **kwargs)
+ _repel_text(list of Texts, renderer, ax)
Extends
Use
Use
Text
+ field: type
+ get_fontsize
Figure
+ canvas: FigureCanvas
Use
FigureCanvas
+ getRenderer(): Renderer

Code Traces(Sequence Diagram)



Implementation Plans

The basic solution is to find the lines end's x and y position as well as the color of the lines, and place the text labels in appropriate positions using this information. Some additional work will need to be done to ensure no overlap of line labels and that line labels persist in desired position on zoom in/out.

Key Part No.1: Place the label to the end of the Line2D
SimpleLineLabel._parse_labelline_args() should provide us the handlers(which are lines) and the labels(strings)

line.get_date() provides x-data, y-data of a line, then check the x-data
Use labels to create texts objects and place them (parallel unless special circumstances such as collision) according to the y-data of the last point of the lines. If there are multiple lines with same endpoints, we should order them according to the second last endpoints, do this recursively until either the order is set, or run out of points' data. This may be more difficult than expect as Matplotlib tend to have a lot of issues related to proper order rendering.

Key Part No.2: Avoid the overlapping
To avoid the overlapping after having an order of the text label, we may get the layout of the text label after we have the effective renderer to work with and use this information to adjust the position of the texts. There are effective algorithms to solve this problem on the internet, though not directly applicable, we should be able to implement a working model.

Key Part No.3: Preserve the position when zoom in/out
This need more works and thoughts since it may involve add codes to some event triggered method, even backend logic, since the labels needed to recalculate their position when events are triggered.

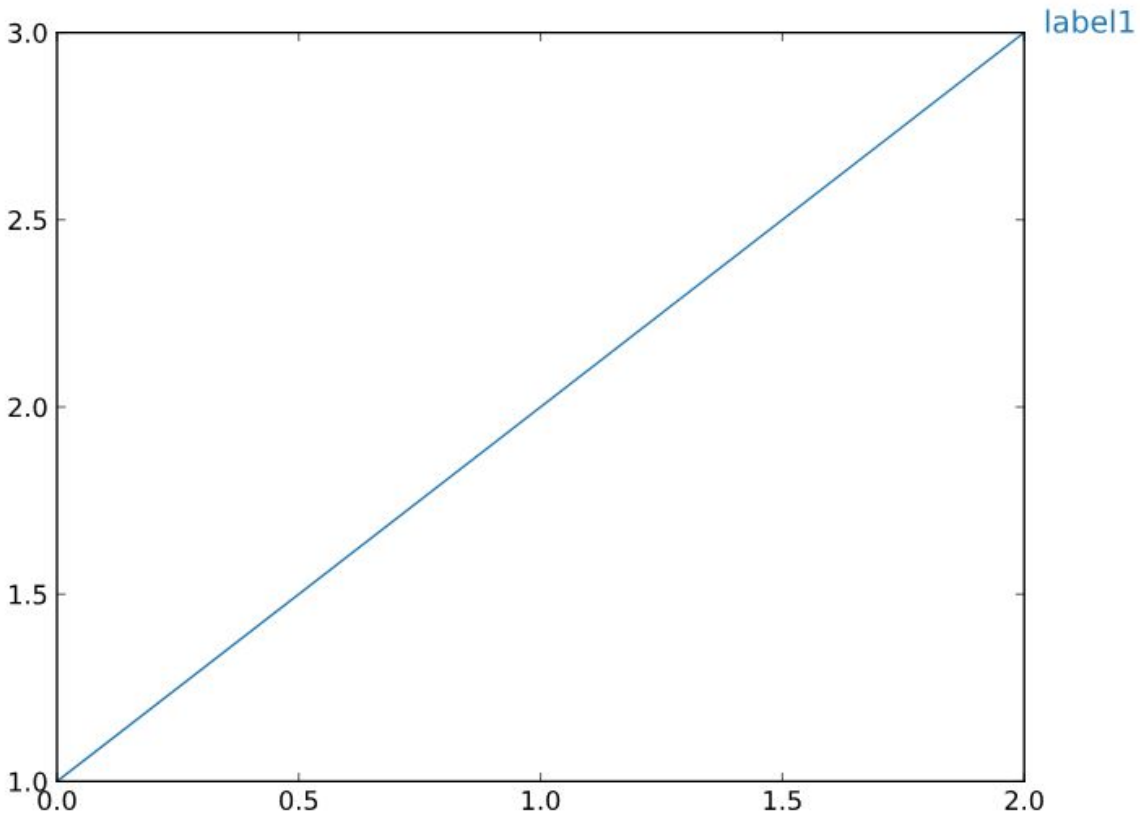
Reason For Issue Choice:

Adding labels to lines offers interesting automation choices and is challenging yet doable. The main crux of the issue would be not only resolving overlaps but ensuring the labels look clean and natural. The feature seems like it would be a natural addition to the matplotlib vast array of features. Thus, Team Plagiarism decided on this issue due to it being both challenge and rewarding to implement. The best features are often,those you would already expect to be there and we believe Issue 12939 will be a natural fit to Matplotlib existing features.

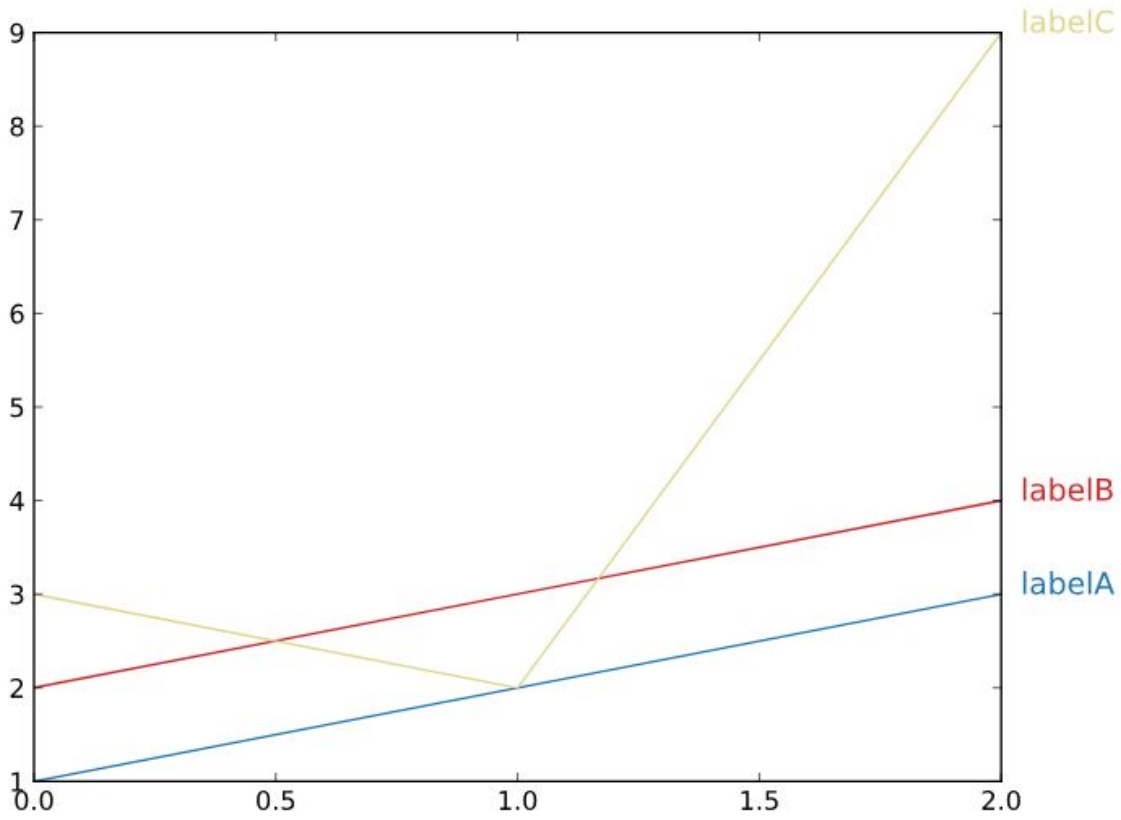
Acceptance Tests:
Navigate to the matplotlib/examples/line_labels/ folder which contains 3 examples demonstrating the feature.

Then run the python files and confirm that the labels in the plots created by the example code matches the positioning of the labels in the following images:

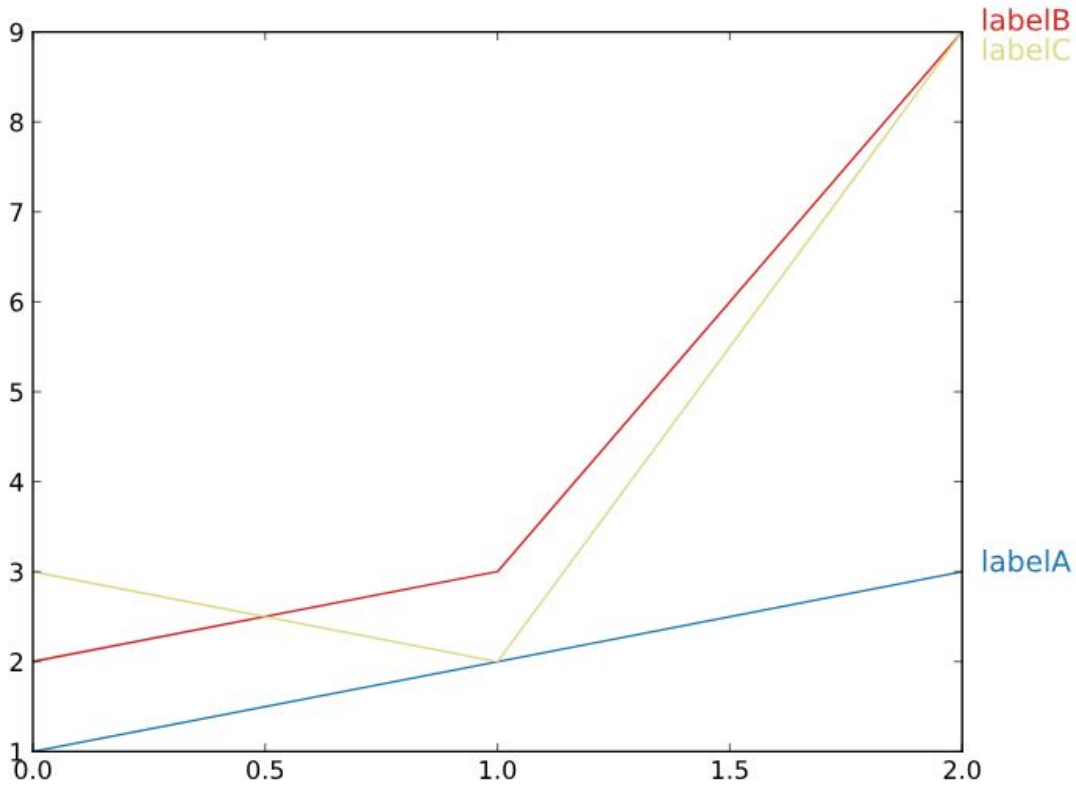
Test 1 - Simple Label:
Simple plot with one line should display after running Python on simple_label.py



Test 2 - Complete Label:
Multiple lines that criss-cross should display after running complete_label.py

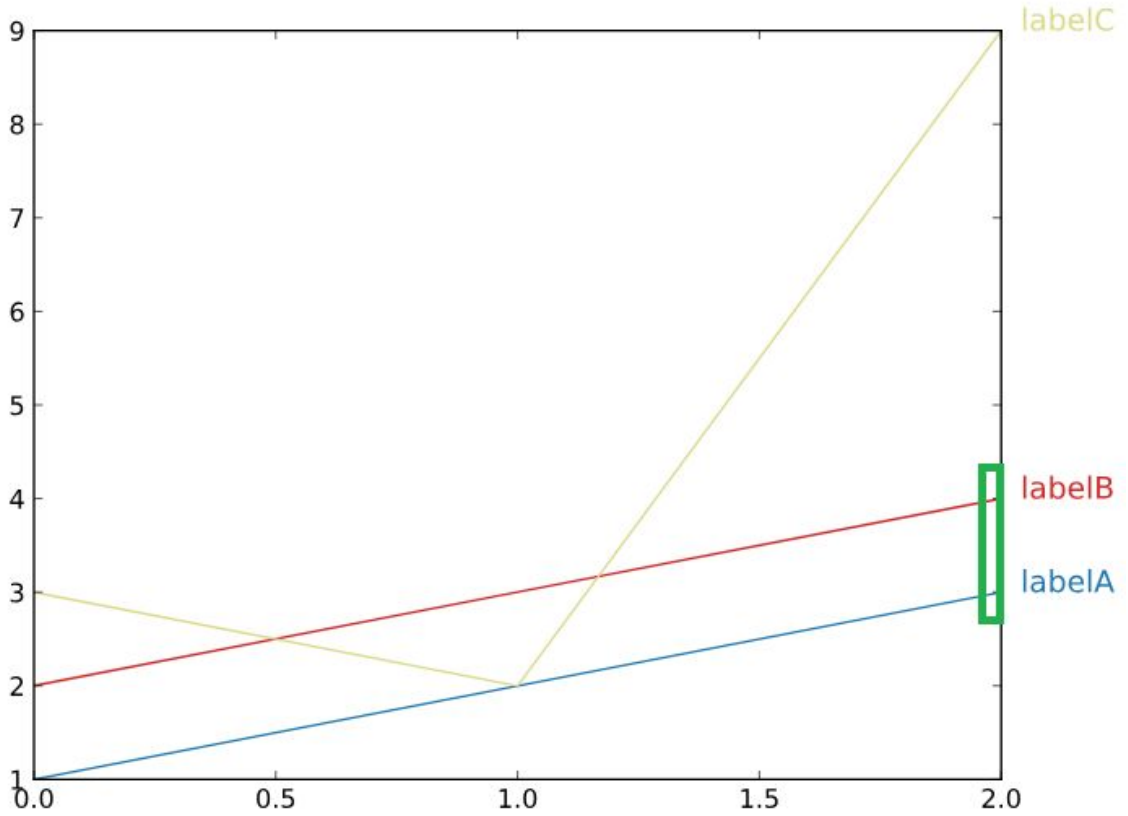


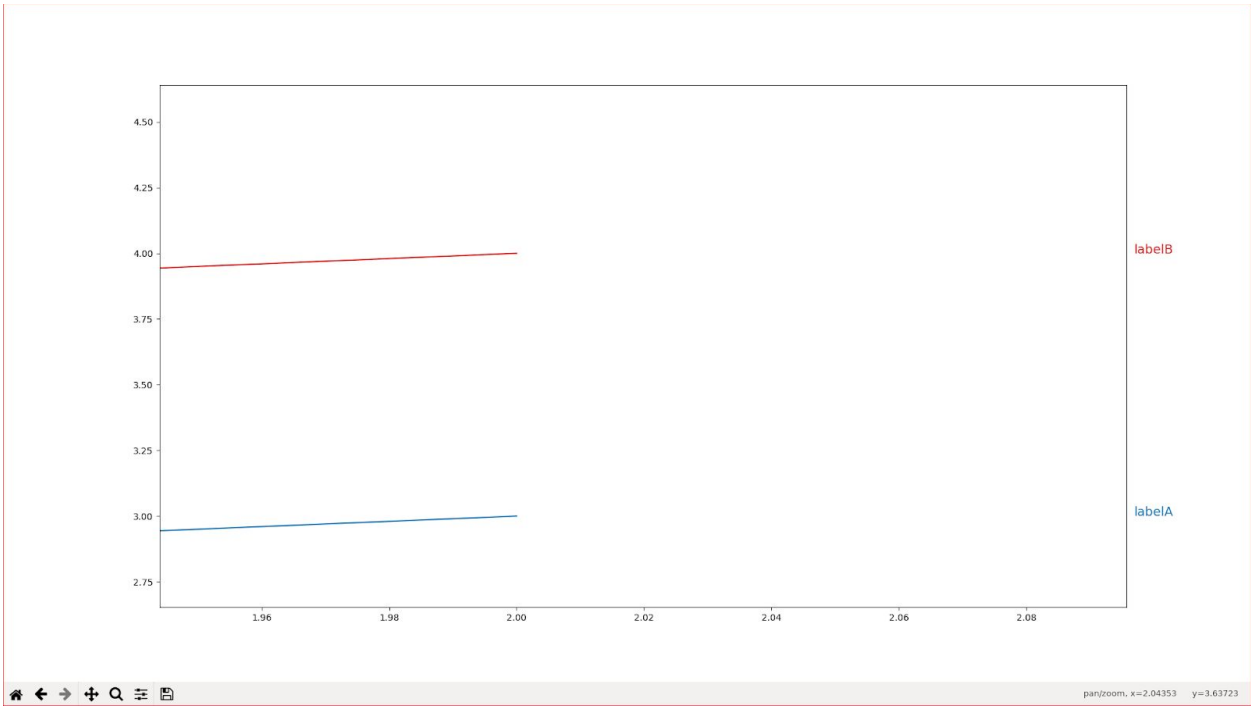
Test 3 - Collision Label:
Plot where lines collide at the end should display after running collision_label.py



Test 4 - Zoom Label (Test 2 but zoomed in):
Plot that has been zoomed in on

Green rectangle is the zoomed in region





Unit Tests:

The Unit Test suite has been implemented on our repos version of matplotlib and can be run using the following command:

“pytest lib/matplotlib/tests/test_label.py”

All 5 tests are expected to pass.

Note: The test suite has been included under Deliverable_3/Unit Tests/ for quick viewing.

Unit tests 1-3:

The first three test uses image comparison testing for png, pdf and svg images to ensure the generated graph matches the reference image for automatic label placement with one line.

Unit tests 4-6:

The next three test uses image comparison testing for png, pdf and svg images to ensure the generated graph matches the reference image for label placement on 3 lines.

Unit tests 7-9:

The next three test uses image comparison testing for png, pdf and svg images to ensure the generated graph matches the reference image for label collision on placement.

Unit test 10:

The 10th test tests for an error that occurs when there are unexpected amount of argument inputted into label line method and that an exception will be raised.

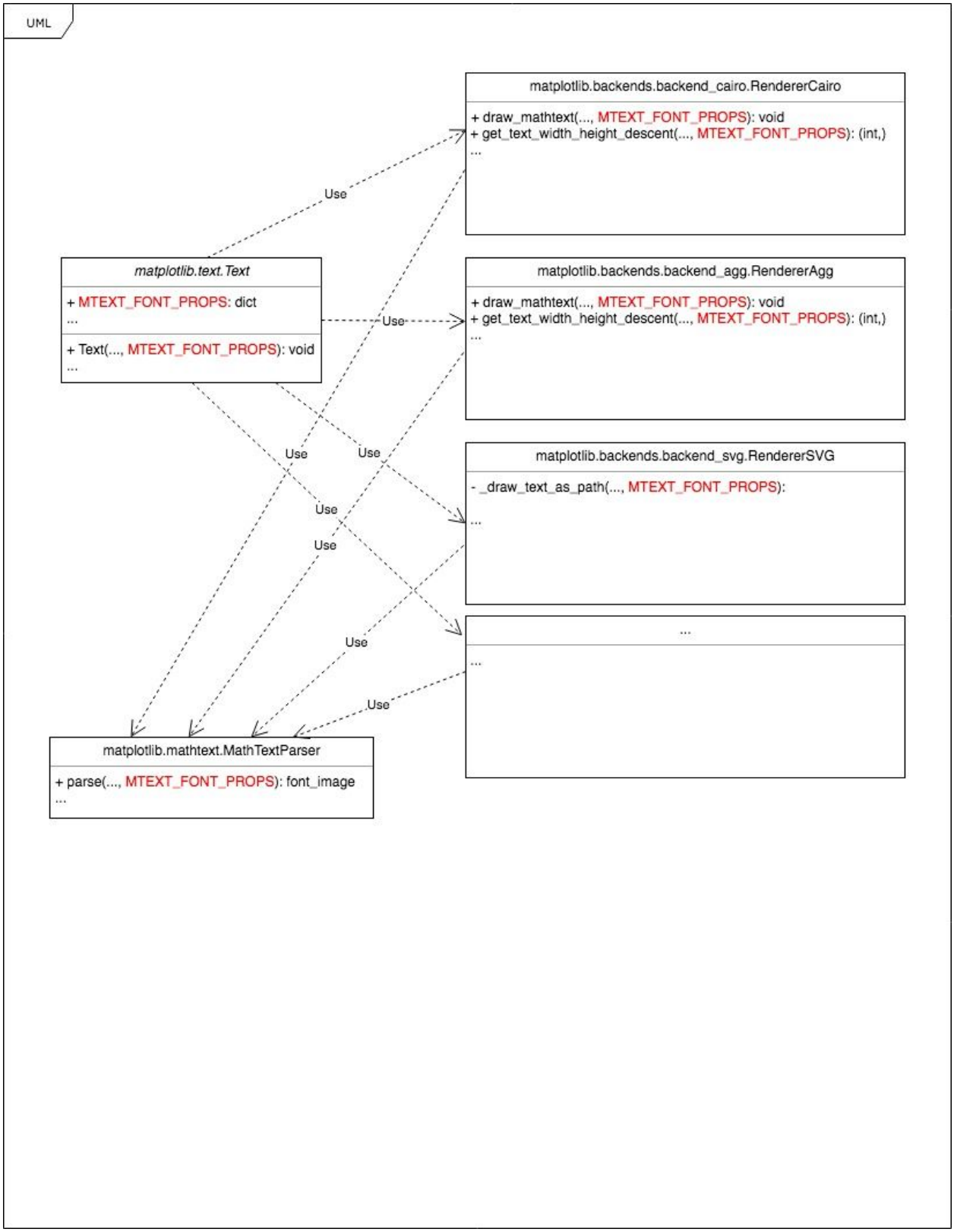
Unit test 11:

The 11th test tests that an exception is raised when handles is not iterable.

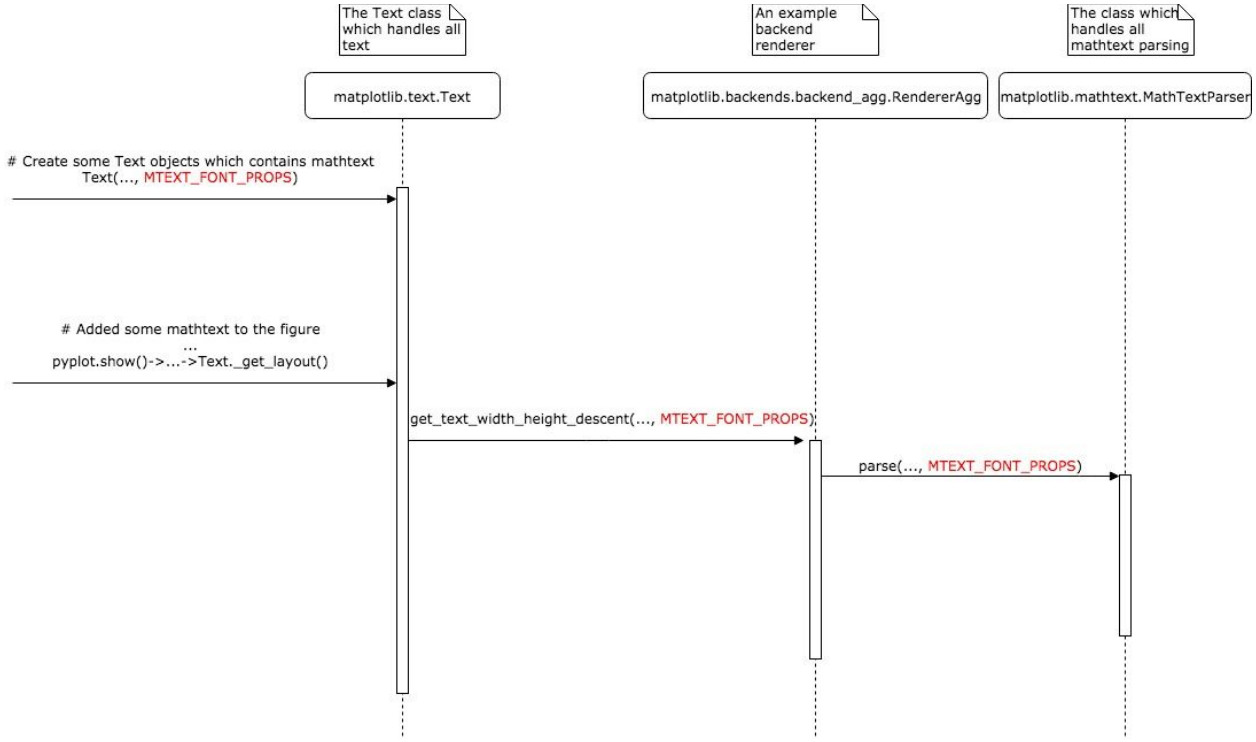
Feature 2 Outline

Aside: Since we have decided on implementing Sean's feature, we will explore this one but not do it since we need to do some pretty heavy backend work (which might be C++).

Issue Name: 'mathtext.fontset' is only available as an rcParam, should be carried along with text object
Issue #: 7107
URL:
https://github.com/matplotlib/matplotlib/issues/7107?fbclid=IwAR1-HHsGAWxN-l0YAmhr1ociqlrhahJPj0Iztwl7tV6gJLRoJcQul2H_oBw
Description:
This issue requests for the capability of setting unique mathtext font properties for each text object. Currently the font properties of mathtext can only be set through some values in "rcParams", which is a global configuration object, so once a font property is set there all of the mathtext are affected. Due to the extensive interaction between the mathtext parsing code and the rendering code for each supported backend, adding this capability probably requires changes to be made for all of the backends.
Implementation Strategy:
Basically what we need to do is to add a new parameter of type dictionary to the constructor of the Text class which will hold all possible font properties of mathtext, then let it be propagated to the mathtext parsing code -- matplotlib.mathtext.MathTextParser.parse(), while updating all the backend code to properly use the new parse() function.
UML Diagrams:
Below is the structural UML diagram which indicates where the code changes have to be made: (The newly added parameter is marked as red)


UML
matplotlib.backends.backend_cairo.RendererCairo
+ draw_mathtext(..., MTEXT_FONT_PROPS): void
+ get_text_width_height_descent(..., MTEXT_FONT_PROPS): (int,)
...
matplotlib.text.Text
+ MTEXT_FONT_PROPS: dict
...
+ Text(..., MTEXT_FONT_PROPS): void
...
matplotlib.backends.backend_agg.RendererAgg
+ draw_mathtext(..., MTEXT_FONT_PROPS): void
+ get_text_width_height_descent(..., MTEXT_FONT_PROPS): (int,)
...
matplotlib.backends.backend_svg.RendererSVG
- _draw_text_as_path(..., MTEXT_FONT_PROPS):
...
...
...
Use
Use
Use
Use
Use
Use
Use
Use
Use
matplotlib.mathtext.MathTextParser
+ parse(..., MTEXT_FONT_PROPS): font_image
...

Below is the sequence diagram which demonstrates how the newly added parameter can be pushed throughout a particular code path (which is just the code path shown in the code trace screenshot below):



Code Traces:

To do the code trace, I manually put a error-prone print statement in the mathtext parsing code (i.e. the matplotlib.mathtext.MathTextParser.parse() function). Here's the script I ran to get the code trace:

```
1  import matplotlib
2  from matplotlib import pyplot as plt
3
4  plt.figure(figsize=(8,8))
5  ax = plt.gca()
6  matplotlib.rcParams['mathtext.fontset'] = 'stixsans'
7  mathtext1 = matplotlib.text.Text(10, 10, r"mathtext1 $\mathrm{abc123}^{123}$")
8  #matplotlib.rcParams['mathtext.fontset'] = 'dejavuserif'
9  #mathtext2 = matplotlib.text.Text(15, 15, r"mathtext2 $a=5$")
10 ax.add_artist(mathtext1)
11 ax.add_artist(mathtext2)
12 ax.set_xlim(0,20)
13 ax.set_ylim(0,20)
14 plt.show()
15
```

Here is the code trace:

```
Traceback (most recent call last):
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/backends/backend_macosx.py", line 80, in _draw
    self.figure.draw(renderer)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/artist.py", line 38, in draw_wrapper
    return draw(artist, renderer, *args, **kwargs)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/figure.py", line 1707, in draw
    renderer, self, artists, self.suppressComposite)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/image.py", line 135, in _draw_list_compositing_images
    a.draw(renderer)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/artist.py", line 38, in draw_wrapper
    return draw(artist, renderer, *args, **kwargs)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/axes/_base.py", line 2638, in draw
    mimage._draw_list_compositing_images(renderer, self, artists)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/image.py", line 135, in _draw_list_compositing_images
    a.draw(renderer)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/artist.py", line 38, in draw_wrapper
    return draw(artist, renderer, *args, **kwargs)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/text.py", line 670, in draw
    bbox, info, descent = textobj._get_layout(renderer)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/text.py", line 298, in _get_layout
    clean_line, self._fontproperties, ismath=ismath)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/backends/backend_agg.py", line 206, in get_text_width_height_descent
    self.mathtext_parser.parse(s, self.dpi, prop)
  File "/Users/ken/anaconda/lib/python3.6/site-packages/matplotlib-0+unknown-py3.6-macosx-10.7-x86_64.egg/matplotlib/mathtext.py", line 3359, in parse
    print("Text: " + s + "; Backend Class: " + str(type(backend)) + "; Fontset Class: " + str(type(font_output)) + 1);
TypeError: must be str, not int
```

As we can see, when plotting a figure with some mathtext, we will eventually call MathTextParser.parse() to parse the mathtext so that they can be properly rendered later. From the screenshot below of this function, we can see how the ‘fontset’ property is retrieved just from ‘rcParams’. We want setting such properties through arguments to each text object to be possible. Note that, also shown in the screenshot below, ‘backend’ indicates a particular backend being used in a run, which can be any of the supported backends so this is why all backends need to be fixed accordingly.

```
3335      def parse(self, s, dpi = 72, prop = None):
3336          """
3337          Parse the given math expression *s* at the given *dpi*.  If
3338          *prop* is provided, it is a
3339          :class:`~matplotlib.font_manager.FontProperties` object
3340          specifying the "default" font to use in the math expression,
3341          used for all non-math text.
3342
3343          The results are cached, so multiple calls to :meth:`parse`
3344          with the same expression should be fast.
3345          """
3346
3347          if prop is None:
3348              prop = FontProperties()
3349
3350          if self._output == 'ps' and rcParams['ps.useafm']:
3351              font_output = StandardPsFonts(prop)
3352          else:
3353              backend = self._backend_mapping[self._output]()
3354              fontset = rcParams['mathtext.fontset'].lower()
3355              cbook._check_in_list(self._font_type_mapping, fontset=fontset)
3356              fontset_class = self._font_type_mapping[fontset]
3357              font_output = fontset_class(prop, backend)
3358
3359          fontsize = prop.get_size_in_points()
3360
3361          # This is a class variable so we don't rebuild the parser
3362          # with each request.
3363          if self._parser is None:
3364              self.__class__._parser = Parser()
3365
3366          box = self._parser.parse(s, font_output, fontsize, dpi)
3367          font_output.set_canvas_size(box.width, box.height, box.depth)
3368          return font_output.get_results(box)
```

Implementation Plan:

Since fixing all the backends might be very difficult, we might want to start with the one that gives us the most “return of investment” -- The AGG backend matplotlib.backends.backend_agg.py which seems to have a lot of common functions shared by different backends. Starting with fixing a single backend also helps us quickly verify the effectiveness of our implementation strategy (i.e. the new parameter). Once we fix the AGG backend, each of us can take another 1 or 2 backends to fix and eventually we will have a nice set of fixed backends.