Matplotlib: Revisiting Text/Font Handling

Proposal: Google Summer of Code 2021
Organisation: NumFOCUS
Sub-Organisation: Matplotlib

(Mentors: Thomas A Caswell & Antony Lee)

OVERVIEW

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations, which has become a *de-facto* Python plotting library. It allows user text to be rendered on the canvas, includes extensive support for mathematical expressions, raster and vector outputs, arbitrary rotations, and supports unicode.
Much of the inspiration behind its font manager is inspired from W3C compliant algorithms, allowing users to interact with font properties like *font-size*, *font-weight*, *font-family*, etc.

However, the current way Matplotlib handles fonts and general text layout is not ideal, which is what this proposal aims to tackle.

It is divided into three subgoals, such that by the end of the project completion, TeX exporting mechanisms would use the same structural layout for most backends, and every exported PS/PDF would contain embedded glyphs which are subsetted from the whole font. This would be done with an implementation of a redesigned text-first font interface, essentially enabling a font-fallback mechanism, such that all font-family members will be parsed before rendering a “tofu”.

GOALS

1. Font-Subsetting overhaul:
    a. Identifying the cases where subsetting fails (Type 42 Fonts)
    b. Consider offloading subsetting to fontTools' pyftsubset
        i. Build a proof-of-concept before considering this option
2. Revisit the way text layout works:
    a. Re-architect crucial components of FT2Text
        i. Design inspiration from Raqm
            1. Range-based approach
            2. Setters/Getters as __init__ or constructor attributes
        ii. Implement font-fallback in FT2Text.set_text
            1. Parse font-family while constructing the object
            2. Warn about the fallback
    b. Inspect ways to load font-family
        i. As a stack or an unordered_set or an unordered_map
        ii. Global loading and passing it down to FT2Font
3. Different TeX export mechanisms (PS vs PDF/SVG), same structural layout:
    a. Studying the differences between different backends
        i. PS vs PDF/SVG
        ii. Core Matplotlib vs mplcairo
    b. Rendering with usetex with inspiration from mplcairo
4. Documentation and tests

ABOUT ME

Contact Details

Name	Aitik Gupta
Email	aitikgupta@gmail.com
Personal Website	aitikgupta.github.io
GitHub	/aitikgupta
LinkedIn	/aitik-gupta
Time Zone	Indian Standard Time (GMT + 5:30)
University	Indian Institute of Information Technology, Gwalior
Major	Information Technology
Contact No.	+91 9821633928

Personal Background

I am a third-year undergraduate student currently pursuing a Dual Degree (B.Tech + M.Tech) in Information Technology at Indian Institute of Information Technology, Gwalior. With Pop OS as my primary operating system, I use fish shell for most of my tasks and VS Code as my text editor.

I find thinking along the lines of clean and object-oriented code enjoyable, mostly using Python and C++. My interests lie in the field of Software Engineering with Machine Learning, but I'm always up for new and exciting challenges! My personal projects apply technologies including Machine Learning, Computer Vision etc. to realtime utilities, and are open-sourced at GitHub.

During my sophomore year, my interests began expanding in the domain of Machine Learning, where I learnt about various amazing open-source libraries like NumPy, SciPy, pandas, and Matplotlib. Gradually, in my third year, I explored the field of Computer Vision during my internship at a startup from May to November 2020. A big chunk of my work was to integrate their native C++ codebase to Android via JNI calls.

To actuate my learnings from the internship, I worked upon my own research along with a friend from my university. The paper [arXiv] was accepted in CoDS-COMAD'21 [ACM Digital Library].

During this period, I picked up the knack for open-source and started contributing to libraries like Matplotlib, NumPy, CuPy, and such likes. I quickly got involved in Matplotlib's community, since it is very welcoming and beginner-friendly; its dev call was the very first I attended with people from all around the world. After discussing with the mentors I decided to take up the task of rethinking the way Matplotlib handles texts/fonts during the upcoming Summer 2021 as a GSoC project.

Logistics

GSoC timeline is in sync with my summer break and would allow me enough time to work on this project. I'm fully aware of GSoC time commitments and I plan to devote the expected time (18 hours a week) at the very least. There's uncertainty as to when my college will reopen due to the ongoing pandemic, but even if some part of the timeline coincides, there will be no exams or assignments in the first half of the semester. Something worth mentioning is that during the application review period (April 13 - May 17), I would be away for around a week at my real sister's wedding. (April 26 - May 5)

Open Source Contributions

Here are some of my major open source contributions:

Contributions towards Matplotlib:

matplotlib/matplotlib	Add overset and underset support for mathtext	**Merged (+71 -0)**
matplotlib/matplotlib	Strictly increasing check with test coverage for streamplot grid	**Merged (+54 -2)**

matplotlib/matplotlib	Fix over/under mathtext symbols	**Merged (+13 -8)**
matplotlib/matplotlib	Add support to edit subplot configurations via textbox	**Draft (+51 -11)**
matplotlib/matplotlib	Expand ScalarMappable.set_array to accept array-like inputs	**Draft (+22 -4)**

Other major Open-Source Contributions:

numpy/numpy	DEP: Shift correlate mode parsing to C and deprecate inexact matches	**Merged (+129 -22)**
cupy/cupy	Add equal_nan toggle for NaN values in array_equal	**Merged (+34 -2)**
opencv/opencv	calib3d: check for minimum corresponding points to calculate homography	**Merged (+28 -1)**
GNOME/pitivi	Imitate placeholder for Title	**Merged (+230 -85)**
GNOME/pitivi	Disable mouse-scrolling in custom effect widgets	**Merged (+24 -15)**

Issues Opened:

matplotlib/matplotlib	sphinxsidebar text overflow
numpy/numpy	Core: Convolve and Correlate mode check in core/numeric.py
cupy/cupy	[Bug, Enhancement] array_equal returns False on identical arrays with NaN values
cupy/cupy	Unsupported f-string format
cupy/cupy	Dictionary keys cannot be CuPy element

The Proposal

Current Situation

Matplotlib's various backends deal with a lot of conversions independently, which, at the time of their origin had no unification, due to a lot of dependent factors. This creates a lot of inconsistency issues for exported files, along with embedding whole Type 42 fonts.
FT2Font, the font interface for Matplotlib internals which was written around 10 years ago, does not take care of things like font-fallback, which is a layout concept such that if glyph of a character is not found in one of the fonts, it should *fall-back* to the next available font. Due to this, a lot of FT2Font internals could be reasoned into a redesign.

Proposed Features and Changes

1. Font Subsetting

Embedding only the required glyphs when exporting text into a file is called Font Subsetting. Some recent work has been successful for subsetting Type3 fonts for all backends, yet Type 1 and 42 fonts remain non-subsetted for PostScript/PDF backends. This is essentially important for fonts like CJK (Chinese, Japanese, Korean) since they have a lot of glyphs embedded in a single font file (one font file size can range from a few KBs to a lot of MBs - for example, unifont), and if they're not properly subsetted - it leads to Matplotlib yielding large PDF/PS files.

Backend	Font Type	Subsetted?
PDF (Portable Document Format)	Type 1	✖
	Type 3	✔
	Type 42	✖[1]

[1] A proof-of-concept has been analysed for the PDF backend which uses fontTools to do the heavy (and a bit hacky) lifting.

PS (PostScript)	Type 3	✔
	Type 42	✖

A major chunk of this section would be experimenting with fontTools's PyFTSubset, which is an OpenType font subsetter and optimizer. It accepts any TT- or CFF-flavored OpenType (.otf or .ttf) or WOFF (.woff) font file. The subsetted glyph set is based on the specified glyphs or characters, and specified OpenType layout features.

The tool also performs some size-reducing optimizations, aimed for using subset fonts as webfonts. Individual optimizations can be enabled or disabled, and are enabled by default when they are safe.
The hacky way to execute this would be to:

1. Subset the glyphs from the font file (using fontTools)
2. Create a temporary font file with just the subsetted glyphs (using `tempfile.NamedTemporaryFile`)
3. Use that subsetted font file (with FT2Font) and embed it in the output file (PDF/PS)

Note that this works under an assumption that we already know which glyphs are required.
As an initial plan, this is a safer route since it does not touch the already-built font embedding machinery. Since Type 42 font type has a lot of bugs, the very first step would be to formulate a proof-of-concept using the principles mentioned above. (similar to the proof-of-concept for PDF backend)

This section contains some parts of the library which were not touched for a long time, it is safe to assume diving into it could lead into some more deeper problems. This is one of the reasons why this section of the proposal would take up a decent amount in the planned timeline.

2. Revisiting text layout

This section aims to redesign some crucial components of FT2Font, the font interface for Matplotlib. From this section of the proposal, implementing a font-fallback mechanism would be the primary work-product.

Matplotlib's font handling had seen quite a lot of improvements during its initial days, and has been a core part of the library since then. It includes support for raster and vector outputs, newline separated text with arbitrary rotations, and also supports unicode. As a result, one can export PDF/PS/PNG/SVG with fonts directly embedded into them, such that "what you see is what you get in the hardcopy".

The whole idea behind FT2Font is to provide users (and other parts of the library) an interface to interact with a *single* font object. Its set_text method accepts a series of characters, such that their font-faces are loaded serially from the font file (on which the FT2Font object was initialized). The loaded font-faces are then converted from their Glyphs (building blocks which represent a single readable character) object to a bitmap. After we extract the bitmap from the FT2Font interface, all that's left is for a backend to render it on canvas.

Since this workflow revolves around a single font object, it fails to capture the whole essence behind font-fallback (which requires multiple fonts of the font-family to exist within the same object). Matplotlib Enhancement Proposal (MEP) - 14 discusses the introduction of a new concept of "Text Engine", such that a single proposed TextFont class would represent text for a given set of font properties, which wouldn't be limited to a single font file. MEP-14 also brings up important issues with the current Font Selection algorithm, and proposes to define a core set of font parameters that would work across all text engines, where the 'text engines' mean one of the following three:

Built-In	Built-In	Subprocess LaTeX
Normal Text	MathText	UseTeX

Among these text engines, Mathtext is one of the most unique contributions from Matplotlib. It aims to implement a subset of TeX markup using Python, and is more lightweight and faster than the full LaTeX installation. It ships alongside Matplotlib with its own fonts to provide a math expression parser and a layout engine.

Once multiple font objects start existing in FT2Font after implementing this section of the proposal, it would be wrong to mention it as a *single* font-interface, rather it could be better referenced simply as *'fonts-interface'*, indicating it as a general FreeType wrapper for dealing with fonts and their families.

The interface has not seen a lot of architectural improvements for a long period of time, which is why some of its internals could be redesigned using design principles from a text layout library like Raqm:

```
// A sliding window approach to set glyphs by setting face ranges
void FT2Font::fill_glyphs(std::vector<FT_Face> faces, uint32_t *codepoints)
{
    unsigned int start = 0, end = 0;
    FT_UInt glyph_index;
    // initialize current face to the initial face
    FT_Face current_face = faces[0];
    // temporary vector to hold indexes
    std::vector<FT_Face> glyphs_indexes;
    unsigned int start = 0, end = 0;

    while (end < codepoints.size()) {
        unsigned int f = 0;

        // visiting every available face to find glyph (Font-Fallback)
        while (f < faces.size()) {
            glyph_index = FT_Get_Char_Index(faces[f], codepoints[end]);
            if (glyph_index != 0) f++; // FT_Get_Char_Index returns 0 if not found
            else break;
        }


        // if none of the faces has the glyph, raise a warning/error
        if (f == faces.size()) {
            throw_ft_error("Glyph not found in any fallback font");
        }

        // if the new face isn't the face previous character(s) matched to
        else if (faces[f] != current_face) {
            // set glyphs of current face from start to end
            // also handle other logic
            ft_set_face_range(start, end, glyphs_indexes, current_face);
            glyphs_indexes.clear();
            current_face = faces[f];
            start = end;
        }

        // iterate to next character
        end++;
        glyphs_indexes.append(glyph_index);
    }
}
```

The function ft_set_face_range will access the overall glyphs vector, adding the actual glyphs to it by calling FT_Load_Glyph on the face and glyph index given as input. It would also handle logic like Kerning and Glyph Transformation.

Currently, all this logic is implemented in FT2Font.set_text. By incorporating the above snippet into FT2Font, the implementation of set_text would change accordingly. Similarly, the implementation of ft_get_char_index_or_warn, FT2Font.load_char, FT2Font.loaf_glyph, etc. would also change.

Since the above implementation needs all possible faces for fallback to work, font-family needs to exist in the FT2Font interface. There are two possible ways to execute this:

1. Font-family can be added as a parameter passed to its constructor along with a set_family method. This font-family should be parsed and stored in an std::unordered_set<FT_Face> or an std::unordered_map<key, value> where value would be an FT_Face and the key would be a unique identifier.
2. Since FT2Font itself is an interface for a single font, we can load every font in the font-family globally, and pass it down as an argument to FT2Font.set_text

The first method would require a lightweight wrapper around loading the font face. (which is already implemented in FT2Font's constructor)
The second method, however, would involve a Python middleware to load all fonts in the family and pass it down to the set_text method. This loading can be cached for faster access, in fact, Matplotlib already caches a list of fonts with their FT2Font interface - a quick search in the cache would be efficient for most cases using the existing font_manager.get_font method.

```
from matplotlib import font_manager as fm

def get_family_faces(font_family: list) -> list:
    """
    Parse font_family to get FT2Font interfaces.

    Parameters
    ----------
    font_family: List of absolute paths of each font in family.

    Returns
    -------
    A list of FT2Font interfaces for the given fonts.
    """
```

```
fontfamily_faces = []
for file_path in font_family:
    fontfamily_faces.append(fm.get_font(file_path))
return fontfamily_faces
```

This processing will happen to the font-family set by an end-user or the existing families in matplotlib.rcParams. As a result, some Python/C++ wrappers in ft2font_wrapper would also need some requisite changes.

3. Different export backends, same structural layout

Matplotlib can generate high-quality output in a number of formats, including EPS, JPG, JPEG, PDF, PGF, PNG, PS, RAW, RGBA, SVG, SVGZ, TIF and TIFF. Certain backends handle more than one output format:

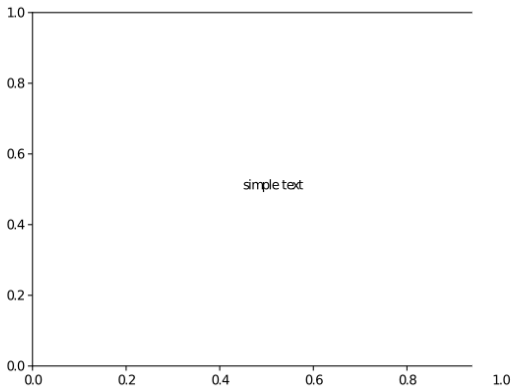
Backend	Output Format (filtered by > 1)
Agg	JPG/JPEG, PNG, RAW/RGBA, TIF/TIFF
PS	EPS/PS
SVG	SVG/SVGZ

Mathtext, the default TeX layout engine which is shipped along with Matplotlib supports only a subset of LaTeX operations. To tackle this, a user can set usetex=True, which calls the system's LaTeX installation to render a DVI which is extended by Matplotlib's parser to include fonts. From here, different backends take different routes, as also mentioned in this comment:

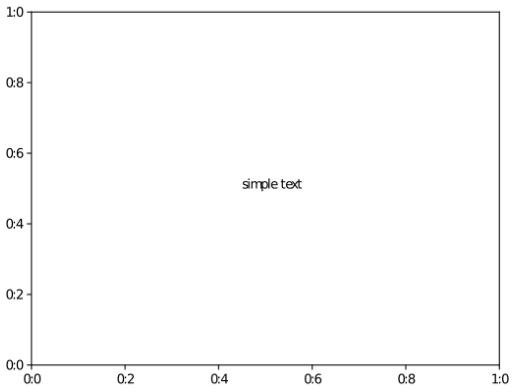
- Agg: Text->LaTeX->DVI->PNG->Image
- PDF/SVG: Text->LaTeX->DVI->Glyphs->PDF/SVG
- PS (special case): Text->psfrag->LaTeX->DVI->PS

PostScript-backend does things a bit differently than PDF/SVG backends because of a reason stated here. PSfrag was introduced as a dependency which works as a middleware between the raw Matplotlib's text and LaTeX's input. As a result, it introduced some bug(s) which are still not solved with Matplotlib's latest release.

mplcairo on the other hand is a complete implementation of a Cairo backend for Matplotlib. Such advancement is a proof-of-concept that a unification mentioned in this comment is possible. It also fixes bugs like following:



EPS generated by Core Matplotlib



EPS generated by mplcairo

This PR renders an interactive diff between these images via GitHub.

Using inspiration from mplcairo, this section of the proposal aims to unify the TeX exporting mechanism for PS backend in Matplotlib.

- The first steps to achieving this would be to figure out the differences between Matplotlib's PS and PDF/SVG TeX exporting layout.
- After that figuring out the same differences between Matplotlib and mplcairo's PS backends would be relevant.
- The dependency to PSfrag would be lost. This also means that passing `usetex=True` to `Text` class will now work for PS backend.

Studying the PS spec would be helpful in solving some salient problems (like #131). The time and efforts given to this section would be slightly less than compared to the other sections since porting is straightforward, but could bring out some rudimentary issues.

4. Documentation and Tests

Adding tests and documentation for this proposal to be successful would be important due to the changes in implementations.
Since font/text handling is one of the core features and is embedded in a lot of places inside Matplotlib, every milestone would need extensive testing such that it does not break other parts of the library.

Overview of what needs testing:

- Fallback Mechanism
- PostScript backend TeX Exporting
- Subsetting Type 42 fonts using PostScript and PDF Backend

Q. Who will benefit from this?

After the successful completion of this project, people who use CJK (Chinese, Japanese, Korean) fonts along with latin fonts will benefit from the font-fallback mechanism. Carrying out subsetting would let them export much smaller-size files, with only the required glyphs embedded.
Another group that will benefit are people who wish to use commercial fonts with Matplotlib. Licenses for many fonts *require* subsetting such that they can't be trivially copied from the output files generated from Matplotlib. Most people in the community use open-source fonts, but since the library has a very large user base, subsetting will be beneficial to people who want to use commercially licensed fonts in their plots.

A lot of Matplotlib text exporting bugs (even ones dated back to 2011) will be closed with the unification of a lot of logic behind different backends.

With these starting few steps, Matplotlib can come closer to a better text handler for almost all its users - how good a user's text renders on screen is one of the key components which makes a plotting library useful in day-to-day life.

Q. Why choose me for this project?

I was introduced to open-source contributions a few months ago, having made my first PR to NumPy and OpenCV was quite fulfilling. I feel excited about the code which explores the interface between Python and C++. With this strong motivation to write clean, object-oriented and efficient code I was introduced to Matplotlib, and I got involved quickly by making very small contributions and joining the Gitter channel. Almost every issue or pull request I have ever gone through in Matplotlib's GitHub repository has given me some insight.
Having attended some developer calls, I got inspired by how much effort it takes to maintain a legacy library, which almost everyone who has ever wanted to make a plot would have heard about, and being able to be a part of this organisation though this summer would be an amazing experience in its learning outcome.
I believe I can spend this time productively and efficiently, while getting mentored by some awesome folks at Matplotlib.

Timeline

This is a tentative list of objectives I plan to achieve in the mentioned time frame. There could well be situations where the project may get delayed or even lead to an early completion. I plan to handle such cases by having buffer weeks in the timeline.

Pre GSoC Period [... - May 17]

In this time period, I plan on developing some concrete test cases which would only pass if that section of the proposal is successfully completed, ultimately improving my understanding of the library. These would require a lot of revisits during the community bonding and the actual coding periods, but a basic structure of those tests would be set up beforehand.

Community Bonding Period [May 17 - June 7]

This time period will be dedicated to understanding some probable challenges that might become a complication during the coding period, with some help from the mentors. Figuring out some planning blunders beforehand would also be important if there are any. Isolating the related issues and pull requests (that might be solved after completion of the project) will also be a task during this timeline.

Week 1-2 [June 7 - June 21]

- ❖ Font Subsetting using the proposed way
- ❖ [Buffer Week]
    - ➢ Using a better way (if any)
    - ➢ Solving related bugs
    - ➢ Documentation/Tests

Week 3-6 [June 21 - July 12]

- ❖ Revisiting Text Layout [Python Interface]
- ❖ Revisiting Text Layout [C++ Interface]

- ❖ Visiting the ft2font_wrapper middleware
  - ➢ Documentation/Tests
- ❖ [Buffer Week]
  - ➢ Revamp the font-matching algorithm from W3C
  - ➢ Solving related bugs

----------PHASE 1 EVALUATION----------

Week 7-9 [July 19 - August 9]

- ❖ Different export backends, same structural layout
  - ➢ Study PS specifications
- ❖ Solving bugs without porting/with porting
- ❖ Documentations/Tests

Week 10 [August 9 - August 16]

- ❖ [Buffer Week]
  - ➢ Overall Documentation/Tests
  - ➢ Wrapping Up

----------FINAL EVALUATION----------

Stretch Goals

- Simplify/Speedup Type 1 Font Embedding [#19584]
- Breaking down PGF baseline tests into 2 parts [#19682]

Post GSoC

Continue working on stretch goals, increasing code coverage to completed goals by adding extensive tests, improving docs.

References

1. MEP-14
2. PDF-Reference
3. CSS font-style-matching
4. Matplotlib Documentation
5. FreeType Documentation
6. fontTools Documentation