Matplotlib - Foundation of Visualization in Python

July 26, 2019

1 Goals

Matplotlib (http://matplotlib.org) is the fundamental data visualization library for the scientific Python stack of tools, providing a visual interface used by over a million[1] data scientists in conjunction with other foundational tools like Numpy and SciPy (https://www.scipy.org/about.html). It is used across a wide spectrum of fields, including bio-medical imaging, microscopy, and genomics [CJL$^+$06, WAT18, vdWSNI$^+$14] [SIW$^+$11, GHWB09, HWSY12, KR12, CRDZ$^+$14, LRA$^+$14, JSC$^+$15, APE$^+$14], and we expect the user base to continue to grow as Python is adopted by more scientists in the life sciences. It has a large array of downstream packages that implement domian-specific plotting solutions.

The source has been actively developed and maintained by a vibrant, primarily volunteer, community over the last 16 years. However, given the scale, scope, and importance of the project, we are at the limit of what can be developed and maintained with a primarily volunteer effort.

This proposal supports 2.5 person years of developer support for Matplotlib, support that is crucial to set the groundwork and shepherd the project through the next 16 years:

a) Maintenance of the library, including curating new and existing Issues and Pull Requests.
b) Developing a comprehensive plan to evolve the core architecture of Matplotlib.
c) Developing the tools, documentation, and community to foster a rich eco-system of domain-specific plotting tools built on Matplotlib.

1.1 Maintenance of the library

Matplotlib depends on a vibrant developer community, but is large enough that this requires organization and planning of lead developers with enough time to ensure timely decision making. Currently, New Issues and Pull Requests are being submitted faster than our volunteers can review them. Over the past few years we have merged PRs and closed issues at about about [X per month], but about [Y] new issues and PRs are opened monthly; currently we have about 1200 open issues and 300 open PRs. Among the latter are useful contributions and bug fixes that, possibly with additional attention and polish, could improve Matplotlib for direct users and downstream packages. The backlog is discouraging for new or occasional contributors, and distracting for core developers.

To maintain Matplotlib's health we need to do the following:

- Sort the current backlog of Issues and Pull Requests in terms of topic, difficultly, and urgency.
- Ensure newly opened Issues are sorted and Pull Requests are reviewed in a timely manner.

[1] Estimated from pypi download numbers, conda download numbers, and the number of unique monthly visitors to the documentation website

- Address critical bugs and regressions in a timely manner.
- On-board new contributors to the project team. This is critical to diversifying and sustaining our developer community.
- Maintain backward compatibility as much as practical. If we do break API, ensure it is intentional, justified, and well documented.
- Manage group decisions about proposed enhancements, features, and breaking API changes.

None of this is to demote the importance of the volunteer contributors, but instead to better co-ordinate and nurture their efforts, with the goal of growing and sustaining a diverse community of expert contributors.

1.2 Road-map and Architecture

Matplotlib needs sustained attention to map out the library's architecture for the next decade. The current architecture[2] of Matplotlib was developed 15 years ago [Hun07]. That it is still in use is a testament to its initial design; but that design does not reflect recent developments in data structures, software design, and visualization. Matplotlib does not natively know how to exploit structured (e.g. `pandas` or `xarray`) or streaming data. Some of the design choices about how to store user data internally and where the main plotting name space lives are starting to be constrains as Matplotlib scales to more domains. While there are many downstream domain-specific libraries that are built on top of Matplotlib, interoperability between them is problematic.

1.2.1 Homogenization of the Application Programming Interface (API)

The library has grown organically over time through the contributions of many people (approx. 900 individuals) and the code has accumulated many small inconsistencies in the API. Similar methods have different argument order, e.g., `ax.text(x, y, s)` vs `ax.annotation(s, (x, y))`, and some keyword arguments can be singular or plural, e.g., `color` vs `colors`. These subtle issues add friction for users, but are hard to fix without breaking existing code somewhere in our large user base. Our goal is to minimize breakage, but still unify the API. Taking into account **all** of the APIs, we will carefully consider which to leave as they are, which to deprecate, and which to replace.

1.2.2 API generalization

Currently Matplotlib has two main user-facing APIs: the `pyplot` API and the `Object Oriented` (`OO`) API. The `pyplot` API closely follows MATLAB and is built around the concept of a global "current Figure" and "current Axes". While convenient for quick interactive usage, this is problematic if used in down stream libraries and frequently leads to surprising and undesired coupling between different parts of the code. On the other hand the `OO` interface is more explicit and flexible but more verbose. However, the main name space for plotting methods is on the `Axes` object which leads to three issues with the API. First because the plotting methods provided by Matplotlib are methods on a class third party domain-specific packages can never feel "First Class" as they will not be implemented as `Axes` methods. Second, there are some visualizations that require putting `Artists` on multiple `Axes` which is not something that can be naturally expressed as an `Axes` method. Lastly, because all of the plotting methods (along with some additional `Axes` specific methods) are in a single name space, there are over 250 methods on the `Axes` class which makes it extremely hard to discover if the method you need exists.

[2] https://www.aosabook.org/en/matplotlib.html

To address these issues we will investigate moving plotting name space from the `Axes` method to top-level functions that take in data, style, and `Axes`s. This will allow "first-party" and "third-party" plotting tools to be on a level playing field and makes it possible to cleanly write plotting tools that produce multiple `Artist`s across multiple `Axes`s. We will ensure that all new functions have consistent naming and call conventions. These functions will return the rich composites discussed in the next section. Having all of the plotting functions, both first- and third-party, as top level functions allows for curated domain-specific name spaces, which will aid in the discoverabilty of plotting functions by users. This is a big change in the API that will require time to carefully consider the consequences.

1.2.3 Rich Composite `Artists`

`Artist`s are the "middle layer" of Matplotlib that encode user-intent, style, and data. An `Artist` can "draw" its self when `Figure` is rendered to screen or disk. To update the style or data users interact with the `Artists`. Currently, Matplotlib has a mix of "primitive" `Artist`s (for things like lines, images, and patches) and "composite" artists for things like the whole `Figure`. In some cases the mapping between the user API and the resulting `Artists` is one-to-one, but in other cases one user call may generate many decoupled `Artist`s, for example, `errorbar` creates up to 5 independent `Artist`s (the line connecting the points, the error bars, the error bar caps, etc) or `hist` which returns many independent `Artist`s. In either case, if the user wants to update the plot they must touch all of the `Artists` in a carefully coordinated way. We should be providing the user with a single object on which they can interact with to elide the underlying details of how the `Artist` is rendered. This will simplify user code for interactive data exploration, handling streaming data, and generating animations.

1.2.4 Data Model

"Structured data" combines multiple pieces of, possibly heterogeneous, data along with labels, dimensions, units, and metadata into single data structure encoding information about the relationship, implicitly or explicitly, between the components. Currently Matplotlib requires users or downstream libraries to split this information into its components and pass them individually to plotting methods, destroying the structure and relationships. Internally, each plotting method and `Artist`s handle sanitizing and storing user input independently and slightly differently. This means that some common functionality, such as handling data with attached units (e.g., degrees Celsius, dates), is scattered throughout the code base. This leads to inconsistencies in behavior across the library and makes it difficult to write code that updates the user data, as is required for interactive exploration, streaming, and animation use cases.

We will re-organize the internal data representation in Matplotlib. The goal is to have a simple model appropriate for the base Matplotlib library and, more importantly, to have the technical underpinning to allow domain-specific downstream libraries to handle and update structured data in a coherent fashion. By removing the direct data storage from the `Artist`s and defining an API for data objects we will make it easy to:

- natively consume structured data;
- implement smart down sampling of plotted data based on view limits;
- seamlessly update the underlying data, either streaming or interactively;
- and back the plot with non-numpy arrays, including queries to a database.

By defining a clear API to access data we will be able to decouple the development of the data storage from the `Artists`. This will enable implementations of the data layer that have exotic dependencies to be used with `Artists` from the core library, and third-party `Artists` to rely on the data layer from the core library.

1.2.5 Additional Export Methods

The primary way to export a Matplotlib figure is to render it to either a raster or vector file format. From there it can be displayed to an interactive window or saved to disk and be used like any other image file. However, there is currently no good way to "reopen" a Matplotlib `Figure` or export it to another plotting library, such as `bokeh`, `d3` or `QtCharts`. Further, due to the way Matplotlib internals are implemented, it is difficult to take advantage GPUs to accelerate drawing.

To address these problems we will investigate adding two additional export paths. One at a high-level, suitable for a Matplotlib-specific file format and interoperability with other high-level plotting libraries, and one at a low level scene-graph level, suitable to pass to a GPU.

1.3 Coordination with downstream projects

This means there will always be a need for domain specific visualization libraries. Much of the domain-specific specialization is carried in the semantics of the structured data and the specific visualization needs of the domain. The most common visualizations in a domain need to be one or two simple lines of code for the end-practitioners with the "obvious" customization options surfaced. Our goal is to make these libraries as thin and easy to write as possible.

To this end we will identify and engage with downstream libraries in the life sciences that are currently using Matplotlib for their visualization to identify their pain points and ensure that we are actually solving their problems. In particular we plan to engage with `scikit-learn`[3], `CellProfiler`[4], `scanpy`[4], `starfish`[4], `nipy`, and `scikit-image`[4]

2 Expected outcomes, success evaluation and metrics

2.1 Issue and PR curation

Quantitatively evaluating maintenance work can be tricky, some Issues or PRs can take minutes to review where as others can take days to months of effort, however we believe that there is value at looking at the net number of new Issues and Pull requests. We will reduce this number, ideally closing Issues and Pull Requests faster than they are opened. NumPy has had success in reversing the ever increasing trends in the number of Issues / Pull Requests with paid developers [5].

We will evaluate and label every open Issue and Pull Request determining: assigning an action, a priority, and an estimated difficulty. Once that is done, we will aim to have all new Issues and Pull Requests labeled within 7 days of being opened.

2.2 Road-map and Architecture

We will write a white paper and road map documenting the proposed design, critical use-cases, and requirements for the data model and API overhaul.

To validate the design, we will develop end-to-end prototypes targeting one or more of the life-science libraries discussed above.

[3] Also applying for Essential Open Source Software for Science

[4] Currently funded by CZI

[5] https://github.com/seberg/numpy_talk_plots/blob/master/plots_used_in_talk/issues_prs_delta.pdf

3 Work Plan

The funds will be paid to:

- Fund Thomas Caswell's position for 6 months. Caswell is currently the lead developer of Matplotlib and an Associate Computational Scientist at Brookhaven National Laboratory. His long-term experience, API design expertise, and project leadership are critical to the success of the work in this proposal. He will work on all aspects of the proposal.
- Fund Hannah Aizenman's position for 12 months. Aizenman has been a core-contributor Matplotlib for three years and has previously contributed support for string-categorical values. She is a PhD candidate in computer science studying visualization at The City College of New York. Her work on the architecture of Matplotlib will be the basis of her PhD thesis. Aizenman will take the lead on the data model design and new-contributor on-boarding.
- Fund 12 months of a yet-to-be identified software engineer to support all aspects of the proposal but focusing on maintenance, prototyping, and engaging down-stream libraries.
- Travel to key Scientific and Python conferences (such as SciPy or PyCon) and for in-person meetings if required.

We want to use this dedicated effort to leverage and empower the Matplotlib developer community. In terms of direct work on the code base an equal amount of time will be spent mentoring and reviewing code from community members rather than directly implementing features or fixing bugs. All of the design work will be done in public with input from the community.

Part of this work is to develop the project road-map.

4 Existing Support

Thomas Caswell has 4hrs/wk from Brookhaven National Lab to work on Matplotlib.

References

[APE$^+$14] Alexandre Abraham, Fabian Pedregosa, Michael Eickenberg, Philippe Gervais, Andreas Mueller, Jean Kossaifi, Alexandre Gramfort, Bertrand Thirion, and Gael Varoquaux. Machine learning for neuroimaging with scikit-learn. *Frontiers in Neuroinformatics*, 8:14, 2014.

[CJL$^+$06] Anne E. Carpenter, Thouis R. Jones, Michael R. Lamprecht, Colin Clarke, In Han Kang, Ola Friman, David A. Guertin, Joo Han Chang, Robert A. Lindquist, Jason Moffat, Polina Golland, and David M. Sabatini. Cellprofiler: image analysis software for identifying and quantifying cell phenotypes. *Genome Biology*, 7(10):R100, Oct 2006.

[CRDZ$^+$14] Thomas M. Carlile, Maria F. Rojas-Duran, Boris Zinshteyn, Hakyung Shin, Kristen M. Bartoli, and Wendy V. Gilbert. Pseudouridine profiling reveals regulated mrna pseudouridylation in yeast and human cells. *Nature*, 515:143 EP –, Sep 2014.

[GHWB09] Ryan N. Gutenkunst, Ryan D. Hernandez, Scott H. Williamson, and Carlos D. Bustamante. Inferring the joint demographic history of multiple populations from multidimensional snp frequency data. *PLOS Genetics*, 5(10):1–11, 10 2009.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[HWSY12] Tamar Hashimshony, Florian Wagner, Noa Sher, and Itai Yanai. Cel-seq: Single-cell rna-seq by multiplexed linear amplification. *Cell Reports*, 2(3):666 – 673, 2012.

[JSC$^+$15] Xiaolong Jiang, Shan Shen, Cathryn R. Cadwell, Philipp Berens, Fabian Sinz, Alexander S. Ecker, Saumil Patel, and Andreas S. Tolias. Principles of connectivity among morphologically defined cell types in adult neocortex. *Science*, 350(6264), 2015.

[KR12] Johannes Kster and Sven Rahmann. Snakemakea scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 08 2012.

[LRA$^+$14] Arthur Laganowsky, Eamonn Reading, Timothy M. Allison, Martin B. Ulmschneider, Matteo T. Degiacomi, Andrew J. Baldwin, and Carol V. Robinson. Membrane proteins bind lipids selectively to modulate their structure and function. *Nature*, 510:172 EP –, Jun 2014.

[SIW$^+$11] Nicola Segata, Jacques Izard, Levi Waldron, Dirk Gevers, Larisa Miropolsky, Wendy S. Garrett, and Curtis Huttenhower. Metagenomic biomarker discovery and explanation. *Genome Biology*, 12(6):R60, Jun 2011.

[vdWSNI$^+$14] Stfan van der Walt, Johannes L. Schnberger, Juan Nunez-Iglesias, Franois Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, and Tony and Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, June 2014.

[WAT18] F. Alexander Wolf, Philipp Angerer, and Fabian J. Theis. Scanpy: large-scale single-cell gene expression data analysis. *Genome Biology*, 19(1):15, Feb 2018.