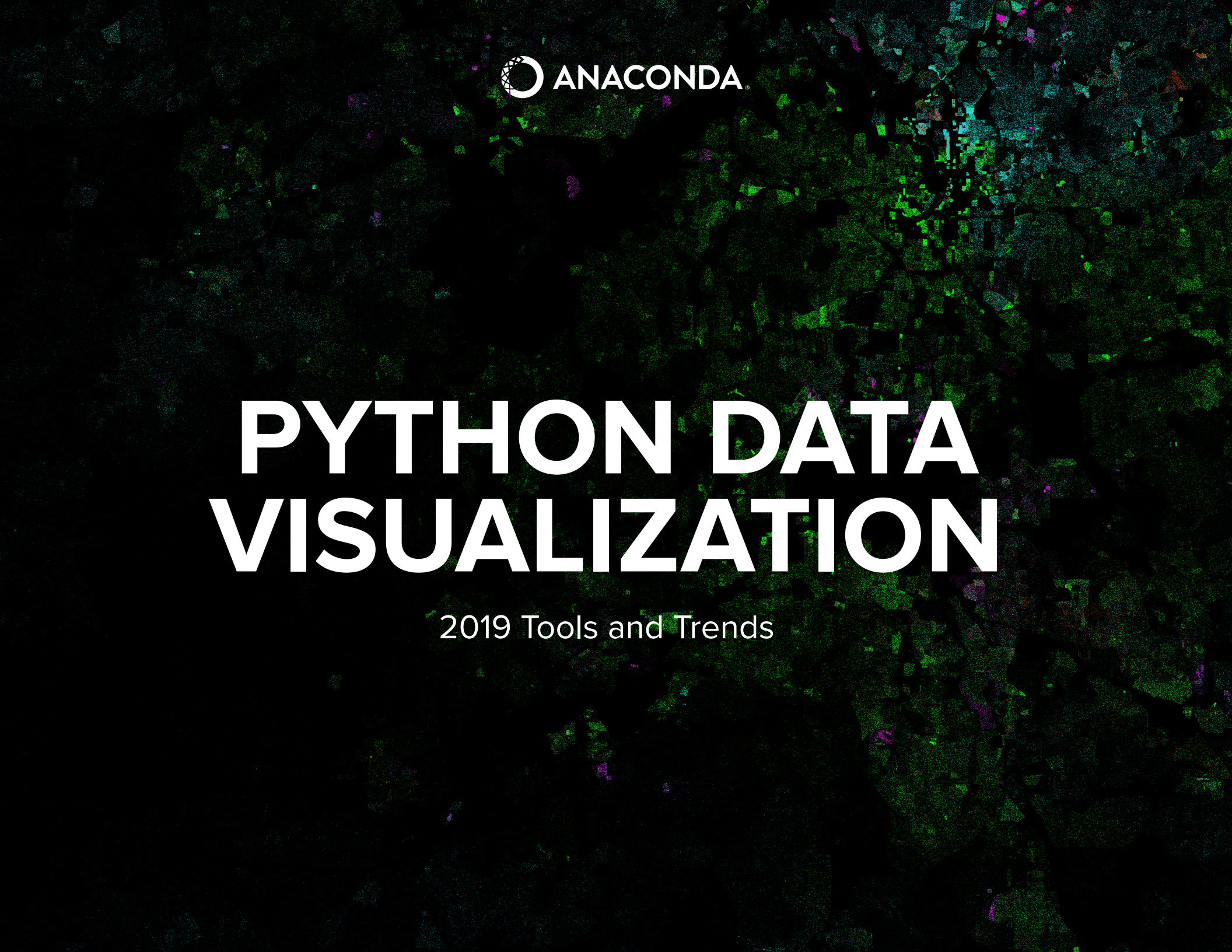

ANACONDA.
PYTHON DATA VISUALIZATION
2019 Tools and Trends

Introduction

Having a myriad of separate Python visualization libraries to choose from is confusing and likely to lead new users down suboptimal paths.

After learning one library, it is difficult to re-learn others that may be more suitable for later tasks. Is there hope that Python could tell a simpler story? Can users be steered toward a smaller number of starting points without getting cut off from important functionality?

This eBook is designed to help you navigate the Python visualization landscape. I'll discuss the packages currently available, how they are linked, evolution of these tools in recent years, and where to go from here.

James A. Bednar
Manager, Technical Services
at Anaconda, Inc.
Contributor to Datashader, GeoViews, HoloViews, Panel, hvPlot, and Bokeh

TABLE OF CONTENTS

CHAPTER 1

NAVIGATING THE MANY LIBRARIES

The Current Landscape

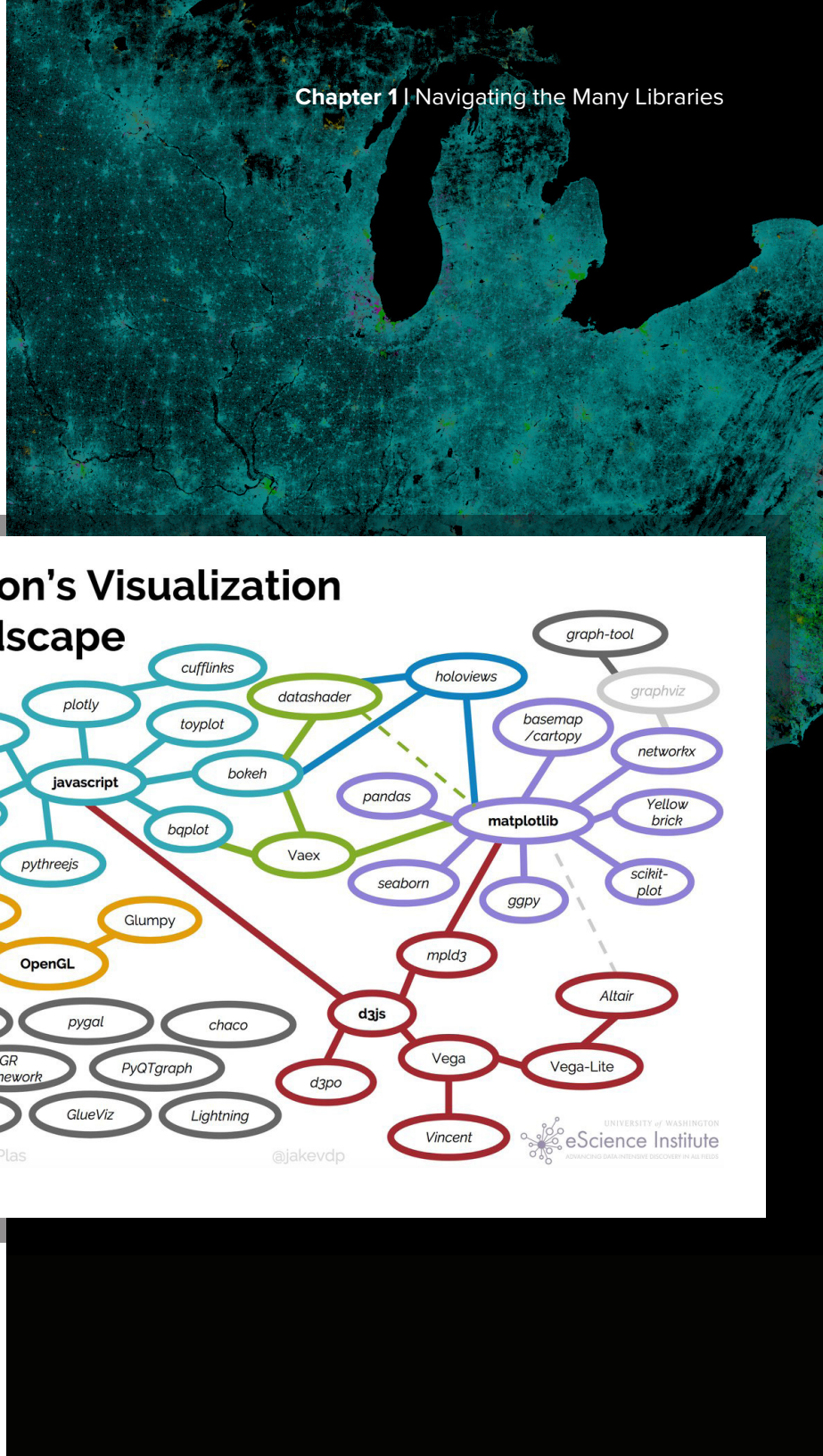
To set the stage, this is Jake VanderPlas's 2017 overview of how the many different visualization libraries in Python relate to each other.

Here you can see several main groups of libraries, each with a different origin, history, and focus.

SciVis Libraries

The clearly separable group in orange towards the middle-left of the figure is the SciVis libraries, for visualizing physically situated data. These tools (VisPy, glumpy, GR, Mayavi, ParaView, VTK, and yt) primarily build on the 1992 OpenGL graphics standard, delivering graphics-intensive visualizations of physical processes in three or four dimensions (3D over time), for regular or irregularly gridded data.

These libraries predate HTML5's support for rich web applications, generally focusing on high-performance desktop-GUI applications in engineering or scientific contexts.



Differentiating Factors Between InfoVis Tools

The other libraries nearly all fall into the InfoVis group, focusing on visualizations of information in arbitrary spaces, not necessarily the 3D physical world. InfoVis libraries use the two dimensions of the printed page or computer screen to make abstract spaces interpretable, typically with axes and labels. The InfoVis libraries can be further broken down into numerous subgroups.

The choice of library is more than a matter of personal preference or convenience.

The following breakdown by history and technology helps explain how we got to the current profusion of Python viz packages. It also helps explain why there are such major differences in user-level functionality between the various packages—specifically in the supported plot types, data sizes, user interfaces, and API types.

These differences make the choice of library more than a matter of personal preference or convenience and very important to understand.

InfoVis Libraries Breakdown

MATPLOTLIB

Released in 2003, one of the oldest and by far the most popular of the InfoVis libraries with a very extensive range of 2D plot types and output formats.

Matplotlib also predated HTML5's support for rich web applications, focusing instead on static images for publication along with interactive figures using desktop-GUI toolkits like Qt and GTK. Matplotlib includes some 3D support, but much more limited than the SciVis libraries provide.

JSON

As JavaScript libraries have matured like D3, their functionality has been captured in declarative JSON specifications (Vega, Vega-Lite).

JSON specs make it simple to generate JavaScript plots from any language, now including Python (via Altair and previously via vincent). Having the full plot specification available as portable JSON allows integration across many types of tools.

MATPLOTLIB-BASED

A variety of tools have built on Matplotlib's 2D-plotting capability over the years.

These libraries either use it as a rendering engine for a certain type of data or in a certain domain (pandas, NetworkX, Cartopy, yt, etc.), or provide a higher-level API on top to simplify plot creation (ggplot, plotnine, HoloViews, GeoViews), or extend it with additional types of plots (seaborn, etc.).

WEBGL

Just as HTML5 did for 2D JavaScript plotting, the WebGL standard made 3D interactivity in the browser and Jupyter feasible, leading to 3D in-browser plotting built on three.js (pythreejs, ipyvolume), vtk.js (itk-jupyter-widgets), or regl (Plotly).

None of these newer web-based 3D approaches capture the breadth and depth of the desktop SciVis 3D libraries, but they do allow full integration with Jupyter notebooks and easy sharing and remote usage via the web. Even though WebGL tools have some applications in common with the SciVis tools, they are probably more closely tied with the other InfoVis tools.

JAVASCRIPT

Once HTML5 allowed rich interactivity in browsers, many libraries arose to provide interactive 2D plots for web pages and in Jupyter notebooks–either using custom JS (Bokeh, Toyplot) or primarily wrapping existing JS libraries like D3 (Plotly, bqplot).

Wrapping existing JS makes it easy to add new plots created for the large JS market (as for Plotly), while using custom JS allows defining lower level JS primitives that can be combined into completely new plot types from within Python (as for Bokeh).

OTHER

Many other libraries, beyond those listed in Jake's diagram, provide other complementary functionality (e.g., graphviz for visualizing networks, or veusz for GUI-based InfoVis plotting).

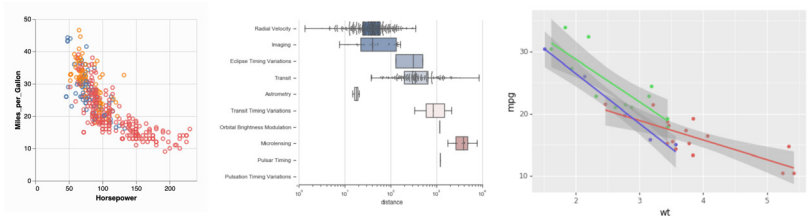
CHAPTER 2

WHAT IS SUPPORTED IN EACH LIBRARY

Plot Types

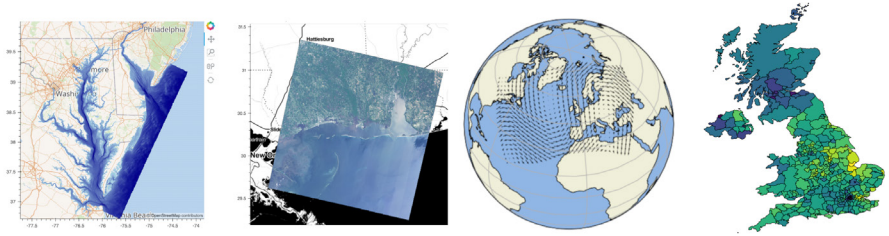
The most basic plot types are shared between multiple libraries, and others are only available in certain libraries.

Given the number of libraries, plot types, and their changes over time, it is very difficult to precisely characterize what's supported in each library. It is usually clear what the focus is if you look at the example galleries for each library.



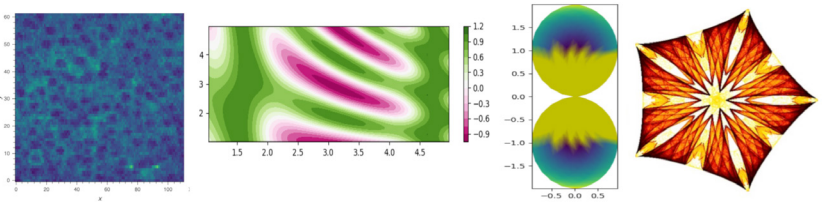
Statistical plots (scatter plots, lines, areas, bars, histograms)

Covered well by nearly all InfoVis libraries, but are the main focus for Seaborn, bqplot, Altair, ggplot2, and plotnine.



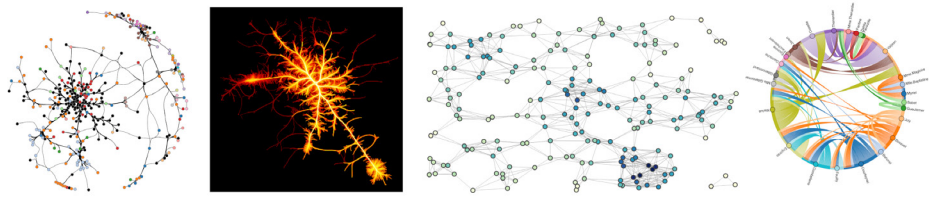
Geographical data

Matplotlib (with Cartopy), GeoViews, ipyleaflet, and Plotly.



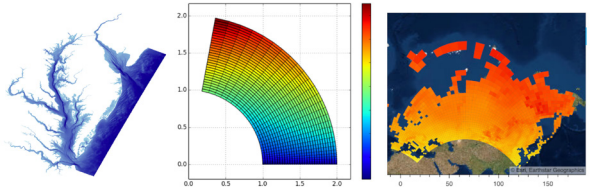
Multidimensional arrays (regular grids, rectangular meshes)

Well supported by Bokeh, Datashader, HoloViews, Matplotlib, Plotly, plus most of the SciVis libraries.



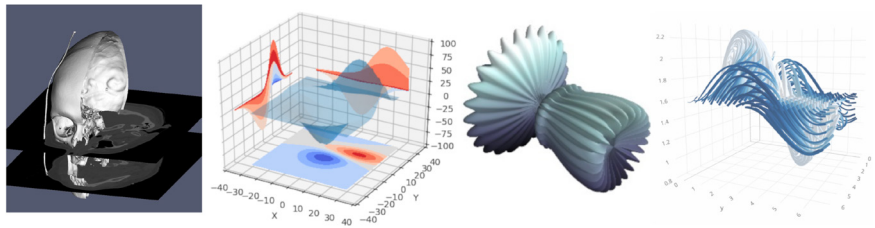
Networks/graphs

NetworkX, Plotly, Bokeh, HoloViews, and Datashader.



Irregular 2D meshes (triangular grids)

Well supported by the SciVis libraries plus Matplotlib, Bokeh, Datashader, and HoloViews.



3D (meshes, scatter, etc.)

Fully supported by the SciVis libraries, plus some support in Plotly, Matplotlib, HoloViews, and ipyvolume.

Data Size

The architecture and underlying technology for each library determine the data sizes supported, and thus whether the library is appropriate for large images, movies, multidimensional arrays, long time series, meshes, or other sizeable datasets.

SciVis

Can generally handle very large gridded datasets (gigabytes or larger) using compiled data libraries and native GUI apps.

Matplotlib-based

Can typically handle hundreds of thousands of points with reasonable performance, or more in some special cases (depending on backend).

JSON

Without special handling, JSON's text-based encoding of data limits JSON-based specifications to a few thousand points up to a few hundred thousand points, due to the file sizes and text processing required.

JavaScript

ipywidgets, Bokeh, and Plotly all use JSON but augment it with additional binary-data transport mechanisms so that they can handle hundreds of thousands to millions of data points in some cases.

WebGL

JavaScript libraries using an HTML Canvas are limited to hundreds of thousands of points for good performance, but WebGL allows up to millions (via ipyvolume, Plotly, and in some cases Bokeh).

Server-side rendering

External InfoVis server-side rendering from Datashader or Vaex allows billions, trillions, or more data points in web browsers. This is done by converting arbitrarily large distributed or out-of-core datasets into fixed-sized images that embed in the client browser.

User Interfaces and Publishing

The various libraries differ dramatically in the ways that plots can be used.

Static Images

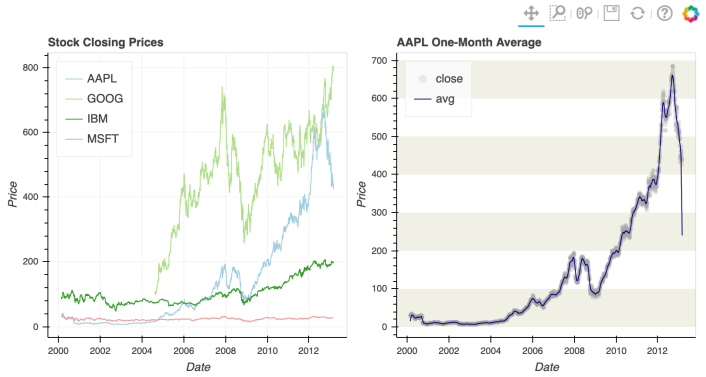
Most libraries can now operate headlessly to create static images, at least in PNG and typically in smooth vector formats like SVG or PDF.

Native GUI App

The SciVis libraries plus Matplotlib, Veusz, and Vaex can create OS-specific GUI windows, which provide high performance support for large data sets and integration with other desktop applications. However, they are tied to a specific OS and usually need to run locally rather than over the web. In some cases, JavaScript-based tools can also be embedded in native apps by embedding a web browser.

Export to Interactive HTML

Most of the JavaScript and JSON libraries can be operated in a serverless mode that generates interactive plots (zooming, panning, etc.) that can be emailed or posted on web servers without Python available.



Jupyter Notebooks

Most InfoVis libraries now support interactive use in Jupyter Notebooks, with JavaScript-based plots backed by Python. The ipywidgets-based projects provide tighter integration with Jupyter, while some other approaches give only limited interactivity in Jupyter (e.g., HoloViews when used with Matplotlib rather than Bokeh).

Standalone Web-Based Dashboards and Apps

Plotly graphs can be used in separate deployable apps with Dash, while Bokeh, HoloViews, and GeoViews can be deployed using Bokeh Server. Most of the other InfoVis libraries can be deployed as dashboards using the new Panel library, including Matplotlib, Altair, Plotly, Datashader, hvPlot, Seaborn, plotnine, and yt, with varying levels of interactivity.

Despite their web-based interactivity, the ipywidgets-based libraries (ipyleaflet, pythreejs, ipyvolume, bqplot) are difficult to deploy as public-facing apps because the Jupyter server allows arbitrary code execution. See the defunct Jupyter dashboards project, flask-ipywidgets, and voila for potential solutions.

InfoVis API types

The various InfoVis libraries offer a huge range of programming interfaces suitable for different types of users and ways of creating visualizations. These APIs differ by orders of magnitude in how much code is needed to do common tasks, compared to how much control they provide to the user for handling uncommon tasks. The lowest-level APIs allow very detailed control for composing primitives into new types of plots:

Declarative Graphics APIs

The Grammar of Graphics-based (GoG) libraries like ggplot, plotnine, as well as those more loosely based on GoG like Altair, bqplot, and Bokeh, all provide a natural way to compose low-level graphical primitives like axes and glyphs to create a full plot. Composing such elements is straightforward for simple plots, but highly verbose for complex plots that contain many such glyphs in different combinations.

Object-oriented Matplotlib API

Matplotlib's main API allows full control and compositionality, but it is complex and verbose even for some common tasks that are simple with other APIs, such as creating subfigures.

JS APIs

Because many web-based Python libraries are built on an existing JavaScript framework like D3, it is always possible to drop down to the JS API for unusual tasks, but doing so is not usually straightforward.

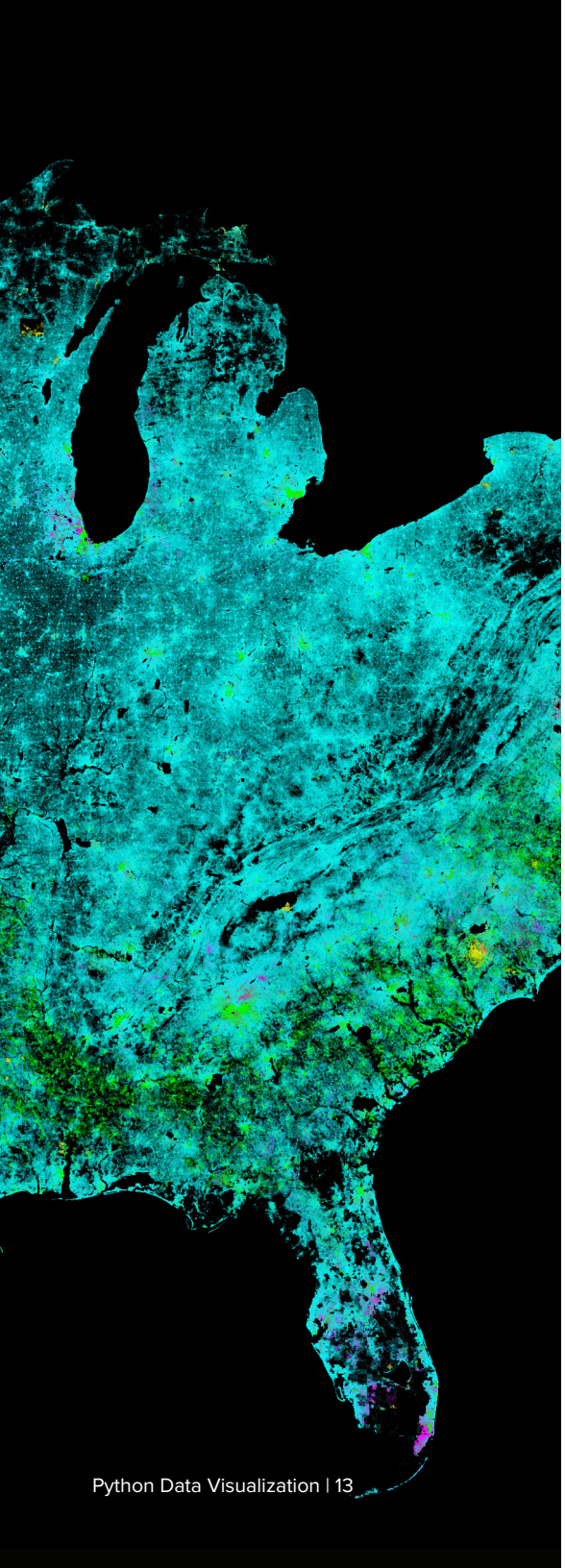
Because using these low-level APIs can be tedious and error prone for day-to-day analysis and data exploration, alternative APIs are also available that capture common patterns and tasks at a higher level. Libraries generally offer a single higher-level API, except that there are now several alternative APIs for Matplotlib and Bokeh. For most users, the following higher-level APIs may make more suitable starting points.

Imperative .plot() APIs

When working with high-level data structures like Pandas dataframes or Xarray DataArrays, it is convenient to be able to request a plot directly from the data structure using .plot(). By default, the resulting Matplotlib-rendered plots are static (non-interactive) and difficult to compose into larger figures, but add-on packages are also now available using the same API to provide interactive and in some cases fully compositional plots (via hvPlot) for a wide range of data structures and underlying plotting libraries. Thus users who learn the basic Pandas/Xarray .plot() API can make use of a broad range of libraries and capability with a relatively small investment.

Other high-level APIs

Seaborn, Chartify, Plotly Express, and similar libraries provide a concise interface for constructing complex plots that fit certain stereotyped patterns, automating the selection of graphical elements for common situations like faceting and statistical aggregation.



Declarative Data-Centric APIs

With each of the above types of APIs, generating a visualization is a one-way process of adding low-level graphical elements (as in declarative or object-oriented graphics APIs) or specifying additional options to .plot(), culminating in a completed visualization. With a data-centric API like HoloViews or GeoViews, creating a particular visualization consists of annotating data with additional semantic information, such as declaring dimensions and units. Subsequent views and aggregations or slices of the data will then make use of this information, making it possible to make all the data visualizable in any combination, rather than specifically building individual plots as needed.

Stateful Pyplot API

Matplotlib's basic interface allows Matlab-style imperative commands manipulating global state, which is concise for some simple cases. However, it is difficult for users to reason about how the state changes, and the lack of compositionality in the API makes it largely limited to a specific set of supported configurations, so Pyplot and similar approaches for other libraries are not recommended for new users; use one of the other high-level approaches above instead!

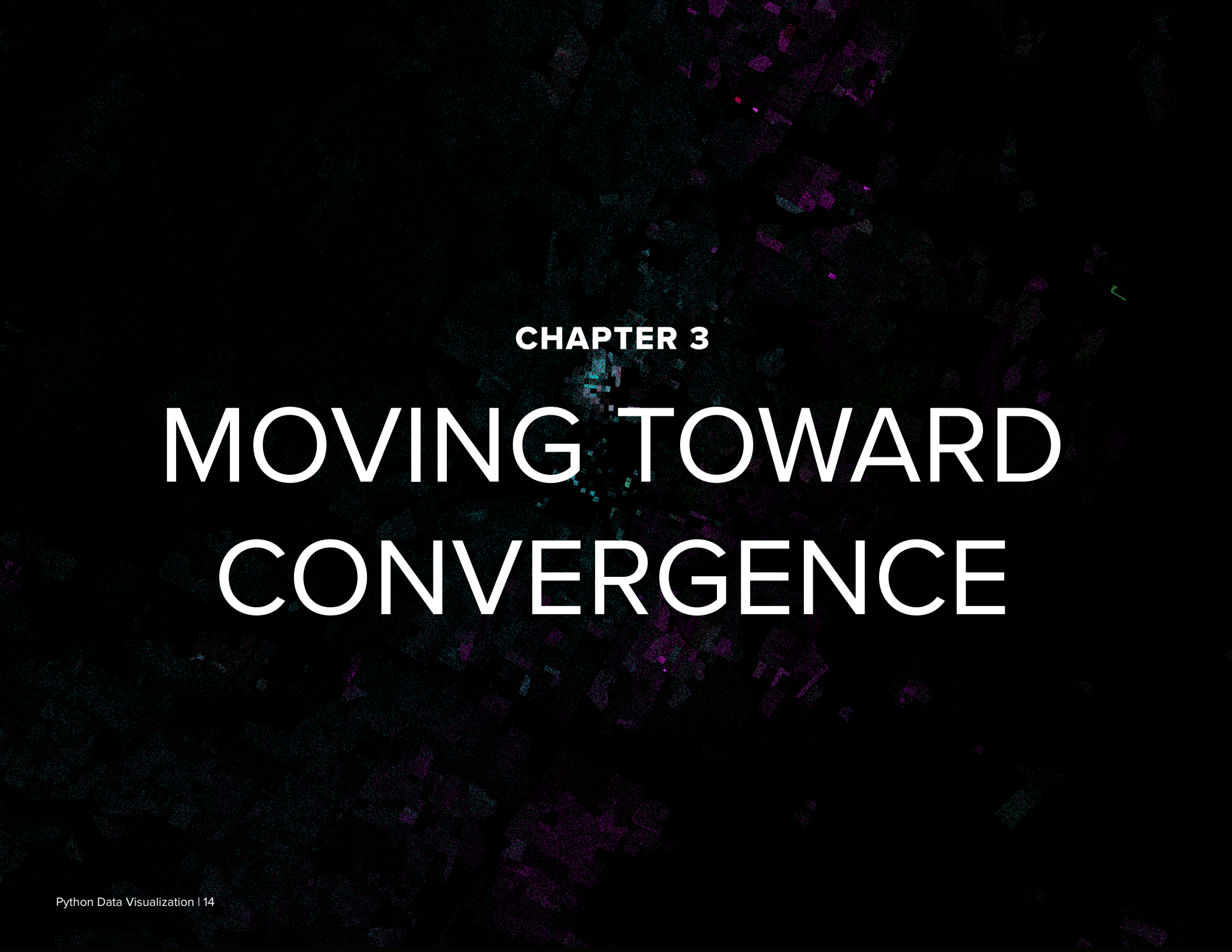
In practice, the various different APIs are suitable for different technical backgrounds, preferred workflows, and desire for customization compared to the ease of obtaining basic results. With any of the APIs, some tasks will become easier, and others more difficult, so it is vitally important to choose an appropriate API.

Emerging Trends

As you can see, there is a huge range of visualization functionality available for Python, with a diversity in approach and focus that is reflected in the large number of libraries available.

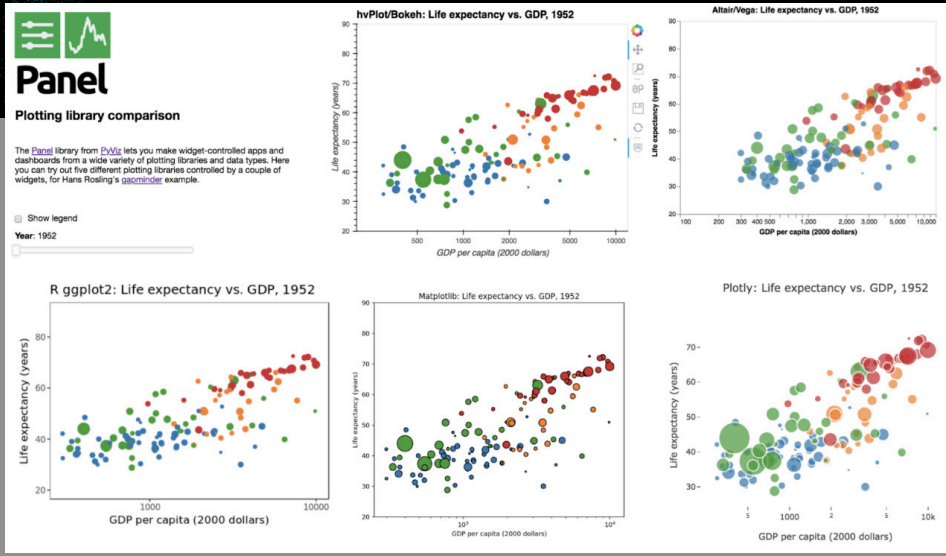
Differences between approaches remain important and have far-reaching implications, meaning that users need to take these differences into consideration before investing deeply into any particular approach.

Luckily, trends toward convergence are helping make it less crucial which libraries users select.



CHAPTER 3

MOVING TOWARD CONVERGENCE



Libraries Becoming More Similar

An important theme that emerged from SciPy 2018 was convergence—Python libraries becoming more similar in capability as they mature over time and share ideas and approaches.

These trends of convergence have started to erase previous clear distinctions between each library. This is great for users, though it makes blanket recommendations more difficult.

As before, we will separate the SciVis projects (typically 3D plotting situated in real-world space) from InfoVis projects (typically 2D plotting situated on the page or screen surface with arbitrary coordinate axes).

Image Output

JavaScript InfoVis libraries, like Bokeh and Plotly, have traditionally focused on interactive use in a web browser, and provided static output mainly as pixelated screenshots (and only with an internet connection in the case of Plotly).

Bokeh now supports PNG and SVG output, and Plotly graphs can be exported to PNG, SVG, or PDF via orca. Both libraries can now be used for publication-quality plots like the Matplotlib derivatives produce.

Users no longer have to decide at the outset of a project whether they might scalable-resolution static outputs eventually.

Big Data

Extremely large InfoVis data (more than 100,000 or a million points) previously required external Python or C data-rendering programs like Datashader and Vaex.

Server-side data rendering has now been integrated into several JavaScript-based libraries so that they can be used interactively (using Vaex in bqplot and Datashader in HoloViews, GeoViews, hvPlot, and now Plotly).

Today, there are many alternatives for working with very large InfoVis datasets in Python.

3D in Notebooks

OpenGL-based 3D libraries previously worked only in native GUI contexts, but Mayavi now supports limited use inside of Jupyter notebooks, making it possible to capture and disseminate workflows more readily, and complementing the browser-only 3D from ipyvolume and Plotly.

Widget/app support

Previous mechanisms for providing widgets and support for apps and dashboards were often specific to Python plotting libraries, such as Dash for Plotly and Bokeh Server/Bokeh Widgets for Bokeh. A wide variety of plotting libraries now support usage with ipywidgets, making it feasible to switch between them or combine them as needed for particular notebook-related tasks relatively easily. This broad base of support makes the particular choice of ipywidgets-based library less crucial at the outset of a project. Many different plotting libraries can also be used with the new Panel app/widget library, either using the ipywidgets-style “interact” interface or as separate objects, either in a Jupyter notebook or in a separate server (see example app in the image above, which combines plots from four Python libraries along with R’s ggplot2).

De-facto .plot() API standard

The pandas plotting API has emerged as a de-facto standard for 2D charts, with a similar set of calls on Pandas dataframes now able to generate plots using Matplotlib (natively in Pandas), Vega-lite (via pdvega), Plotly (via cufflinks), or Bokeh (via hvPlot). hvPlot also provides the same plotting API for many other data libraries (xarray, GeoPandas, Dask, Intake, Streamz), making it possible for users in many cases to learn one set of plotting commands using Pandas and then apply them to a wide range of libraries to get either static or interactive plots.

What Comes Next?

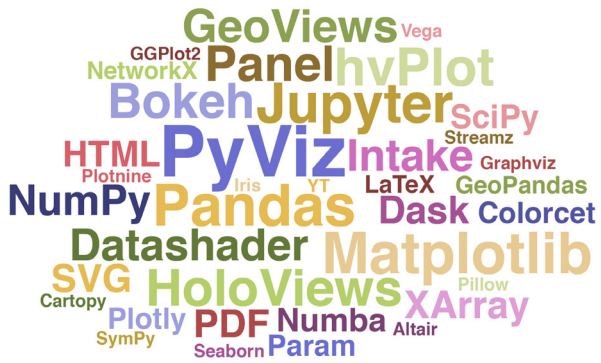
These trends towards convergence mean that users who commit to a particular Python viz library or type of library are no longer entirely cut off from other types of functionality.

Although the different histories and starting points outlined in chapter one remain important to understand, the implications are no longer quite as severe as in previous years.

Having so many separate Python visualization libraries to choose from can still be confusing to new users, which is why I and other representatives of the Python visualization community, spent time at SciPy 2018 discussing ways to simplify these libraries and steer users to a smaller number of starting points.

CHAPTER 4

WHERE TO GO FROM HERE



Visions for the Future

Anaconda's PyViz.org initiative takes the step to make data visualization in Python easier to use and learn. HoloViews and GeoViews now provide a single and concise high-level declarative data API for using multiple InfoVis libraries (currently including Bokeh, Matplotlib, Datashader, Cartopy, and Plotly), and Panel now provides a unified dashboarding approach across dozens of libraries and data formats.

If other InfoVis library authors support the high-level HoloViews API, then users could easily switch between backends depending on their immediate needs (e.g., for selecting different plot types), without having to learn a completely new library's API. Even without such support, Panel already allows plots to be combined from any of the above sources, into the same figure or dashboard.

Perhaps we do not need to achieve centralization of the libraries, but rather centralization of educational resources that can guide users to the appropriate libraries.

A representative of Matplotlib suggested that the large number of existing libraries was not necessarily an issue.

He believes it's entirely appropriate for Matplotlib to be the core workhorse for a large number of libraries building on it.

Not everything needs to be in Matplotlib itself, and Matplotlib makes an excellent basis on which to build other, higher-level 2D static-plotting functionality, due to its comprehensive support for low-level primitives and output formats.

Others argue that scientific professionals have indeed been completely overwhelmed by the sheer number of plotting possibilities in Python, yet the benefit of having so many different libraries available is recognized.

It seems unlikely that all the separate package authors would be able to coordinate closely, but perhaps the SciPy community could do better at educating users on the data models, assumptions, and outputs of each of the main visualization tools.

For example, perhaps we do not need to achieve centralization of the libraries, but rather centralization of educational resources that can guide users to the appropriate libraries. I agreed that if people were willing to work on this, PyViz.org would be a natural place to host such resources.

A representative of ipyvolume, bqplot, and ipywidgets argued that ipywidgets (aka Jupyter widgets) is already emerging as a de facto standard, supported by a wide range of libraries (ipyvolume, ipyleaflet, pythreejs, bqplot, and Plotly) that can now be mixed and matched as needed to provide interactive apps and plots in a Jupyter notebook.

A representative of Mayavi emphasized that mature SciVis tools like VTK, Mayavi, or ParaView cover important functionality not addressed by InfoVis-focused libraries, offering advanced and specialized visualization techniques for large and complex finite-element-method simulations.

These tools support visualization of a variety of data structures and the “long tail” of scientific research beyond just the initial visualization itself.

Conclusions and Outlook

Overall, it is clear that each of the main libraries represents a vibrant community of users and developers using different techniques to achieve different goals.

It is both unlikely and perhaps undesirable for the libraries to consolidate significantly because that would remove major differences in functionality. Any unification efforts would likely be distressing to some users of libraries not included in those efforts.

In any case, we can clearly do better at educating the public about how each library and initiative is most useful, steering users more efficiently into effective solutions for their various goals.

In particular, users need to consider the type of plots they want to use, the data sizes they work with, how they want to interact with and publish their plots, and what type of API they want (focusing on high-level capabilities or low-level control).

Library authors can help make these differences clear for each project, steering users towards appropriate solutions for their needs. Hopefully this eBook helps clarify the situation a bit!