

geoplotlib: a Python Toolbox for Visualizing Geographical Data

Andrea Cuttome^{1*}, Sune Lehmann^{1,2}, Jakob Eg Larsen¹

1 DTU Compute, Technical University of Denmark, Kgs. Lyngby, Denmark

2 The Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark

* ancu@dtu.dk

Abstract

We introduce geoplotlib, an open-source python toolbox for visualizing geographical data. geoplotlib supports the development of hardware-accelerated interactive visualizations in pure python, and provides implementations of dot maps, kernel density estimation, spatial graphs, Voronoi tesselation, shapefiles and many more common spatial visualizations. We describe geoplotlib design, functionalities and use cases.

Introduction

Geographical data visualization is a fundamental tool for communicating results related to geospatial analyses, and for generating hypotheses during exploratory data analysis [1, 2]. The constantly increasing availability of geolocated data from social media, mobile devices and spatial databases implies that we need new tools for exploring, mining and visualizing large-scale spatial datasets.

The python programming language [3] has been gaining attention as a data analysis tool in the scientific community [4,5] thanks to the clarity and simplicity of its syntax, and due to an abundance of third-parties libraries e.g. within many disciplines including scientific computing [6, 7], machine learning [8], bayesian modeling [9], neuroscience [10], and bioinformatics [11]. Currently, however, there is limited support for geographical visualization.

Here, we introduce geoplotlib, a python toolbox for visualizing geographical data. geoplotlib provides a simple yet powerful API to generate geographical visualizations on OpenStreetMap [12] tiles. We release geoplotlib as open-source software [13], accompanied by a rich set of examples and documentation.

In the remainder of this paper, we discuss existing tools for geographical visualization and document the geoplotlib functionalities in detail, and finally we evaluate the computational performance on a large-scale dataset.

Related work

In this section we compare existing tools for visualizing geographical data using python. We divide the related work into three categories: pure python packages, HTML-based packages and Geographical Information System plug-ins.

Pure-python packages

The matplotlib [14] library has become the de-facto standard for data visualization in python and provides a large array of visualization tools including scatter and line plots, surface views, 3D plots, barcharts, and boxplots, but it does not provide any support for visualization on a geographical map by default.

The Basemap [15] and Cartopy [16] packages support multiple geographical projections, and provide several visualizations including point plots, heatmaps, contour plots, and shapefiles. PySAL [17] is an open-source library of spatial analysis functions written in Python and provides a number of basic plotting tools, mainly for shapefiles. These libraries however do not allow a user to draw on map tiles, and have limited support for custom visualizations, interactivity, and animation.

HTML-based packages

There is a very rich ecosystem for data visualization for the web. A number of frameworks allow users to generate plots and charts: we cite as representative Protoviz [18], d3 [19], Google Charts [20], sigma.js [21]. There is also a large number of libraries for displaying online tile maps, including Google Maps [22], Bing Maps [23], Leaflet [24], OpenLayers [25], ModestMaps [26], PolyMaps [27].

In order to generate a HTML visualization from python code, it is needed to generate the HTML and JavaScript code that maps the data to the graphical elements. A number of libraries attempt to automate the conversion, such as Folium [28], Vincent [29] and mplleaflet [30]. This process however is often complex, error-prone and time consuming. The complexity can be even greater if some support for animation or interaction is needed. Finally, the JavaScript rendering performance may not be adequate for plotting very large datasets.

Geographical Information System plugins

Geographic Information Systems (GIS) such as QGIS [31], GrassGIS [32], ARCGIS [33], MapInfo [34] provide very powerful tools for spatial data analysis and visualization. GIS tools usually provide some support for python scripting, although the availability varies from one to another. The main limitation of GIS products is their complexity, requiring a significant amount of training to be used effectively, and as discussed before, the need to export the data from python.

Overview

An overview of the geoplotlib architecture is given in Fig. 1. geoplotlib builds on top of numpy [6] and scipy [7] for numerical computations, and OpenGL/pyglet [35] for graphical rendering. geoplotlib implements the map rendering, the geographical projection, the user interface interaction and a number of common geographical visualizations.

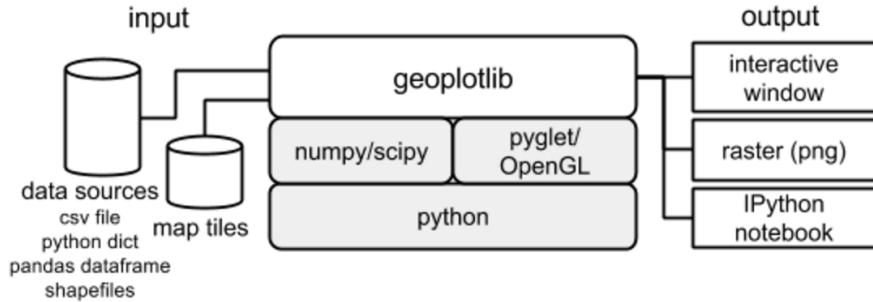


Fig 1: Conceptual overview of the geoplotlib architecture. geoplotlib builds on top of numpy, scipy, and OpenGL/pyglet. It allows to generate geographical visualization as raster, in an interactive window or inside IPython notebooks.

Design principles

geoplotlib is designed according to three key principles:

- *simplicity*: geoplotlib tries to minimize the complexity of designing visualizations by providing a set of built-in tools for the most common tasks such as density visualization, spatial graphs, and shapefiles. The geoplotlib API is inspired by the matplotlib [14] programming model and syntax, the de-facto standard for data visualization in python; this makes it easier for matplotlib users to get started.
- *integration*: geoplotlib visualizations are standard python scripts, and may contain any arbitrary python code and use any other package. There is no need to export to other formats (e.g. shapefiles, HTML) or use external programs. This supports a complete integration with the rich python data analysis ecosystem such as scientific computing, machine learning and numerical analysis packages. The visualization can even run within an IPython [36] session, supporting interactive data analysis and facilitating the iterative design for visualizations.
- *performance*: under the hood, geoplotlib uses numpy/scipy for fast numerical computations, and pyglet/OpenGL for hardware-accelerated graphical

rendering. This allows the visualizations to scale to millions of datapoints in realtime.

A first script

A simple geoplotlib script looks like this:

```
data = read_csv('data/bus.csv')
geoplotlib.dot(data)
geoplotlib.show()
```

This script launches the geoplotlib window and shows a dot map of the data points, in this example the location of bus stops in Denmark (Fig. 2). geoplotlib automatically determines the map bounding box, downloads the map tiles, perform the geographical projection, draws the base map and the visualization layers (the dots in this example). The map is interactive and allows a user to zoom and pan with mouse and keyboard.

As discussed above, the usage of the geoplotlib API is very similar to matplotlib. The visualization canvas is initially empty, and each command adds a new layer of graphics. The geoplotlib window is displayed when `show()` is called. Alternatively, the map can be rendered to image file using `savefig('filename')`, or displayed inline in an IPython notebook using `inline()`.

Layers

The geoplotlib package provides several common geographical visualizations in form of layers. The API provides convenient methods for quickly adding a new visualization layer. In this section we provide a summary of the built-in visualizations. The data for all examples is available on the project website [13].

Dot Map

An elementary operation in geographical visualization is to display “what is where”, that is to place a graphic element on the map for each of the objects in consideration. This provides an immediate idea of the absolute and relative locations of objects. Moreover, the density of points directly maps to the density of objects on geographical surface, identifying zones of higher and lower density. An example of dot map is shown in Fig. 2. The dot map shows the spatial distribution of bus stops in Denmark at a glance. The zones of higher density – corresponding to the Copenhagen metropolitan area and to the other major cities are immediately recognizable. The `dot` method allows users to configure points size, color and transparency, and optionally to attach a dynamic tooltip to each point.

2D Histogram

One limitation of dot maps is that it is hard to distinguish between areas of high density, as the number of point is so high that they uniformly cover the

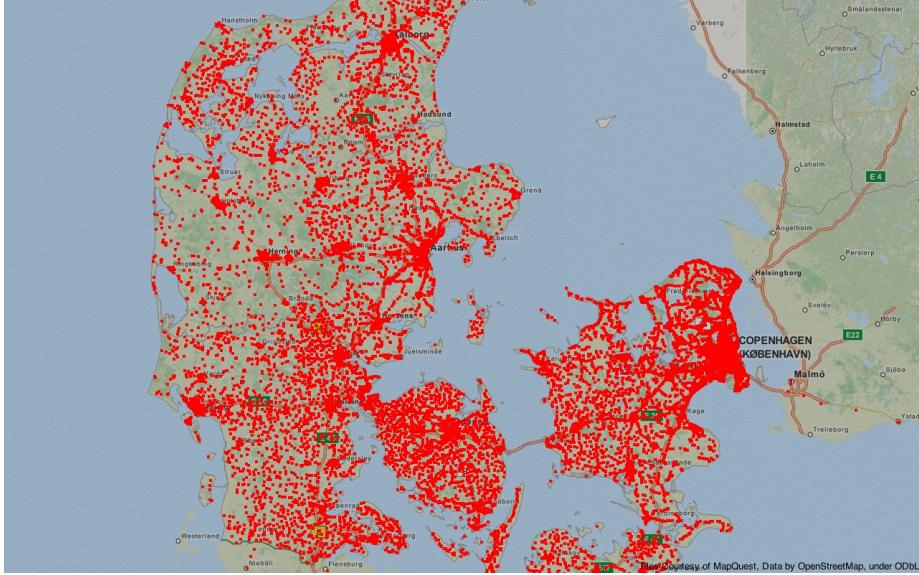


Fig 2: A dot map of bus stops in Denmark, where each sample is represented by a point.

visualization canvas. A more direct visualization of density is to compute a 2D histogram of point coordinates. A uniformly spaced grid is placed on the map, and the number of samples within each cell is counted. This value is an approximation of the density, and can be visualized using a color scale. In geoplotlib we can generate the 2D histogram of the data using `hist`:

```
data = read_csv('data/opencellid_dk.csv')
geoplotlib.hist(data, colorscale='sqrt', binsize=8)
geoplotlib.show()
```

Here `binsize` refers to the size in pixels of the histogram bins.

The example above loads some data related to cell tower positions in Denmark, and then generates a histogram with a specific colorscale and bin size (Fig. 3). Compared to the dot map example, the histogram provides a clearer depiction of the density distribution.

Heatmap

The main deficiency of histogram visualizations is that they are discrete approximations of a (effectively continuous) density function. This creates a dependence on the bin size and offset, rendering histograms sensitive to noise and outliers. To generate a smoother approximation, a kernel density estimator approximates the true density function applying kernel functions in a window around each point [37]. The size of this window depends on the bandwidth parameter: a smaller bandwidth will produce more detailed but also noisier estimation, while

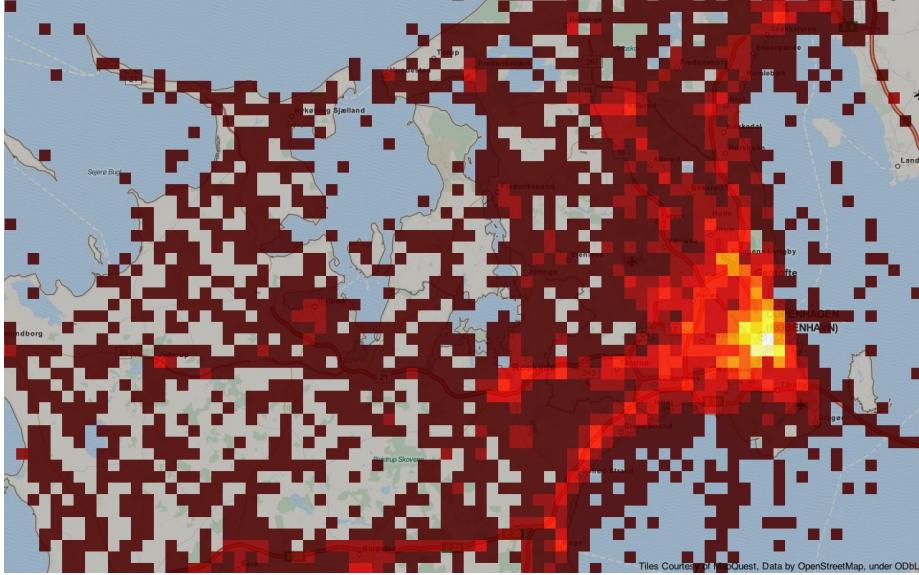


Fig 3: A 2D histogram of cell tower locations in Denmark, using the ‘hot’ colormap (dark red is lower, yellow-white is higher).

a larger bandwidth will produce a less detailed but smoother estimation. A kernel estimation function can then be visualized by a surface where the color encodes the density value (this visualization is often called a “heatmap”). In geoplotlib, the `kde` method generates a kernel density estimation visualization:

```
data = read_csv('data/opencellid_dk.csv')
geoplotlib.kde(data, bw=[5,5])
geoplotlib.show()
```

Fig. 4 shows the kernel density estimation applied to the cell tower data. Comparing the histogram from Fig. 3 with the kernel density estimation in Fig. 4, it is evident how the latter produces a smoother and consequently clearer visualization of density. The kernel bandwidth (in screen coordinates) can be configured to regulate the smoothness. The density upper bound can be set to clip density values over a threshold. Also the density lower bound can be set, to avoid rendering areas of very low density:

```
# lowering clip_above changes
# the max value in the color scale
geoplotlib.kde(data, bw=[5,5], cut_below=1e-6, clip_above=1)

# different bandwidths
geoplotlib.kde(data, bw=[20,20], cmap='coolwarm', cut_below=1e-6)
geoplotlib.kde(data, bw=[2,2], cmap='coolwarm', cut_below=1e-6)

# linear colorscale
geoplotlib.kde(data, bw=[5,5], cmap='jet',
               cut_below=1e-6, scaling='lin')
```

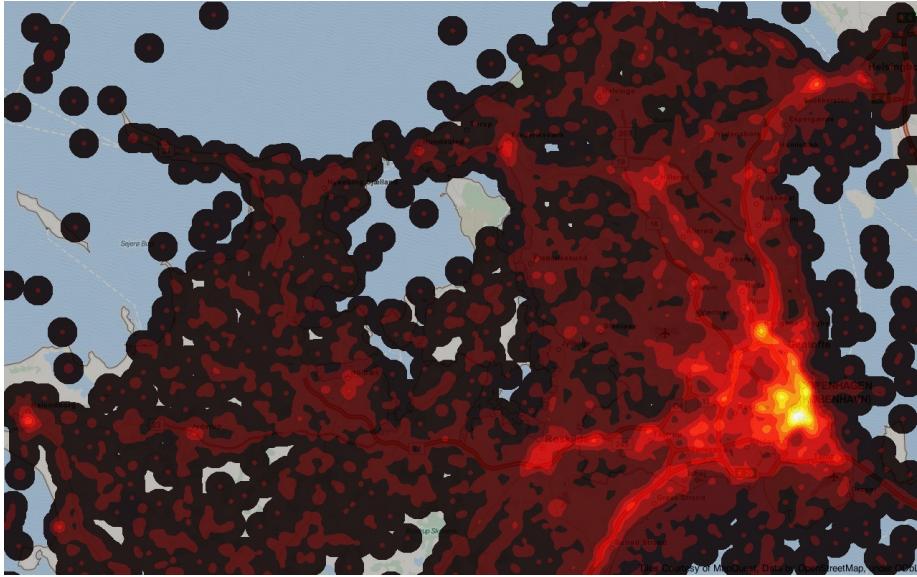


Fig 4: A heatmap (kernel density estimation) of the cell tower locations in Denmark, using a hot colormap (dark red is lower, yellow-white is higher). The kernel density produces a much smoother estimation and therefore a clearer visual representation of the density, if compared with a histogram (Fig. 3).

Markers

In some cases it is useful to represent objects on the map using custom symbols with specific meaning. The `markers` method allows a user to place customs markers on the map:

```

metro = read_csv('./data/metro.csv')
s_tog = read_csv('./data/s-tog.csv')

geoplotlib.markers(metro, 'data/m.png',
                   f_tooltip=lambda r: r['name'])
geoplotlib.markers(s_tog, 'data/s-tog.png',
                   f_tooltip=lambda r: r['name'])
geoplotlib.show()

```

Fig. 5 shows an example of custom markers for metro and train stops in Copenhagen. Markers graphics can be any common raster format (png, jpeg, tiff), and can be rescaled to a custom size. Optionally a dynamic tooltip can be attached to each marker.

Spatial Graph

Spatial graphs are a special type of graphs where nodes have a well-defined spatial configuration. Examples includes transport networks (bus routes, train tracks, flight paths), supply chain networks, phone call networks and commute

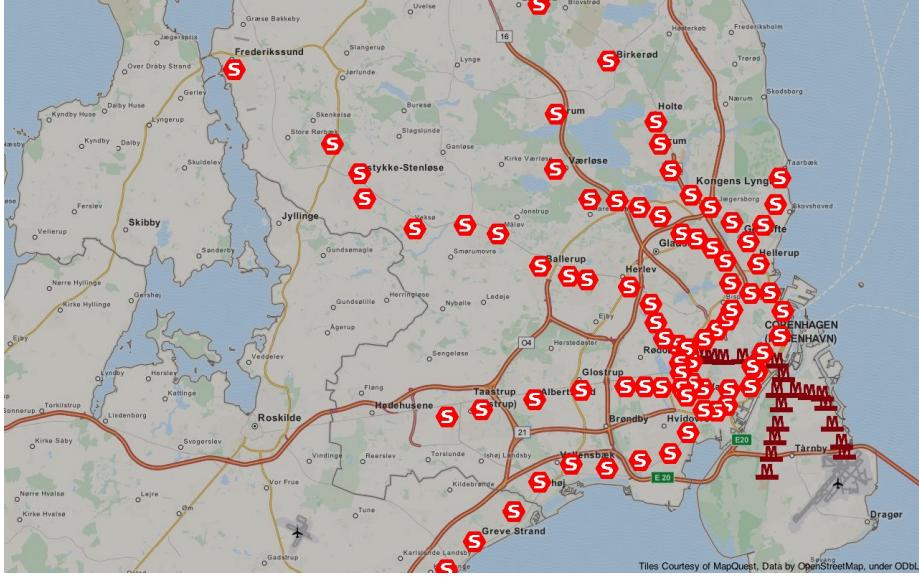


Fig 5: Markers showing the location of train and metro stations in the Copenhagen area. geoplotlib can use any raster image (png, jpg) as markers.

networks. In geoplotlib `graph` renders a spatial graph:

```
data = read_csv('./data/flights.csv')
geoplotlib.graph(data,
                  src_lat='lat_departure',
                  src_lon='lon_departure',
                  dest_lat='lat_arrival',
                  dest_lon='lon_arrival',
                  color='hot_r',
                  alpha=16,
                  linewidth=2)
geoplotlib.show()
```

Fig. 6 shows the resulting spatial graph of airport locations, where each node represents an airport and each edge represents a flight connection. Edges are colored using a colormap encoding the edge length.

Voronoi Tessellation

A Voronoi tessellation [38] is a partition of space into regions induced by some seed points, so that each region (called a Voronoi cell) consists of all points closer to a specific seed than to any others. The analysis of Voronoi tessellation is used in numerous fields including ecology, hydrology, epidemiology, mining and mobility studies.

In geoplotlib `voronoi` can be used to generate a Voronoi tessellation visualization. Voronoi cell fill, shading and colors can be configured.

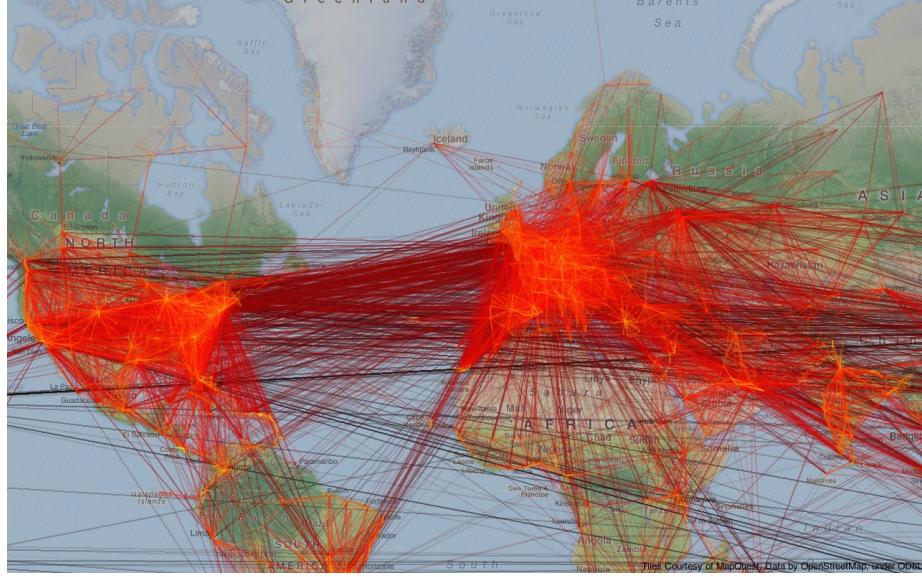


Fig 6: Spatial graph of airport locations, where each node represents an airport and each edge represent a flight connection. Edges are colored using a colormap encoding the edge length.

```
data = read_csv('data/bus.csv')
geoplotlib.voronoi(data, line_color='b')
geoplotlib.show()
```

Fig. 7 provides an example of Voronoi tessellation of bus stops in Denmark. Voronoi cells provide a measure of the space closer to one stop than any others. The density of points is also captured by the size of Voronoi cells, as smaller cells indicate more densely covered areas.

Delaunay triangulation

A Delaunay triangulation [39] is a convenient method for generating triangles meshes from a set of points. In geoplotlib the `delaunay` method can be used for this purpose. The edge color can be configured to a fixed value, or to encode the length of the edges.

```
data = read_csv('data/bus.csv')
geoplotlib.delaunay(data, cmap='hot_r')
geoplotlib.show()
```

Fig. 8 shows the Delaunay triangulation of bus stops, with edges colored according to length.

Convex Hull

A convex hull [39] of a set of finite points is the smallest convex polygon that contains all the points. Convex hulls can be used for example to visualize the approximate area corresponding to a set of points. In geoplotlib:

```
geoplotlib.convexhull(data, color, fill=True)
```

Fig. 9 shows the bus stops points split into 6 groups, and each group is represented by a differently colored convex hull.

Shapefiles

Shapefile [40] is a popular file format for describing vector graphics for geographical information systems. geoplotlib uses pyshp [41] to parse the shapefiles. The line color can be configured and an optional tooltip can be attached to each shape. In the following example we display the *kommuner* administrative regions in Denmark (Fig. 10):

```
geoplotlib.shapefiles('data/dk_kommune/dk_kommune',
                      f_tooltip=lambda attr: attr['STEDNAVN'],
                      color=[0,0,255])
geoplotlib.show()
```

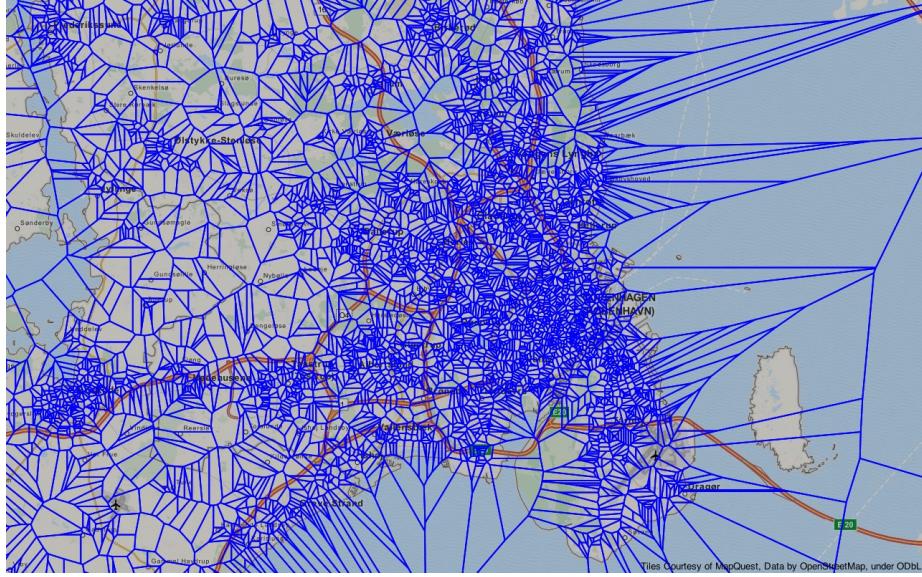


Fig 7: Voronoi tessellation of bus stops in Denmark. Voronoi cells provide an estimation of the space closer to one stop than any others. The density of points is also captured by the size of Voronoi cells, as smaller cells indicate more densely covered areas.

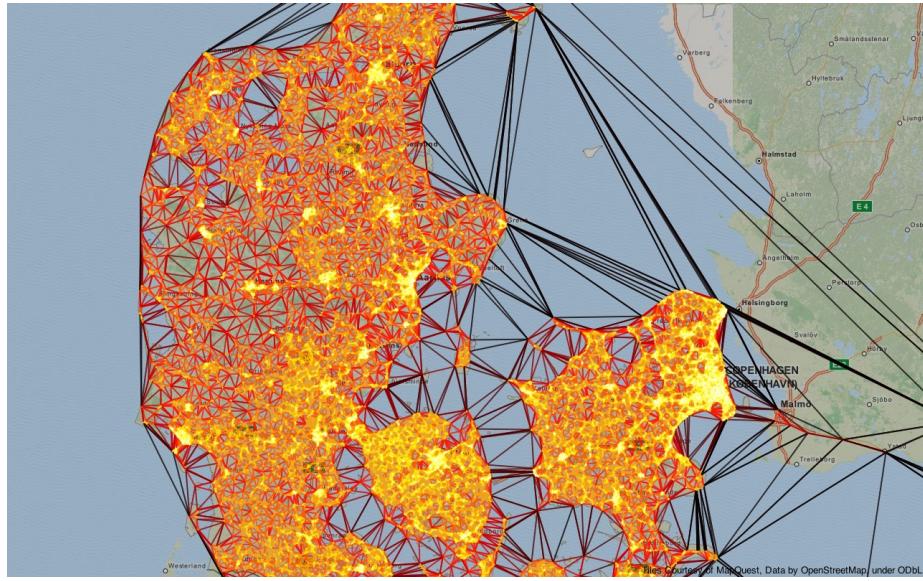


Fig 8: A Delaunay triangulation of the bus stops in Denmark, with edges colored according to length

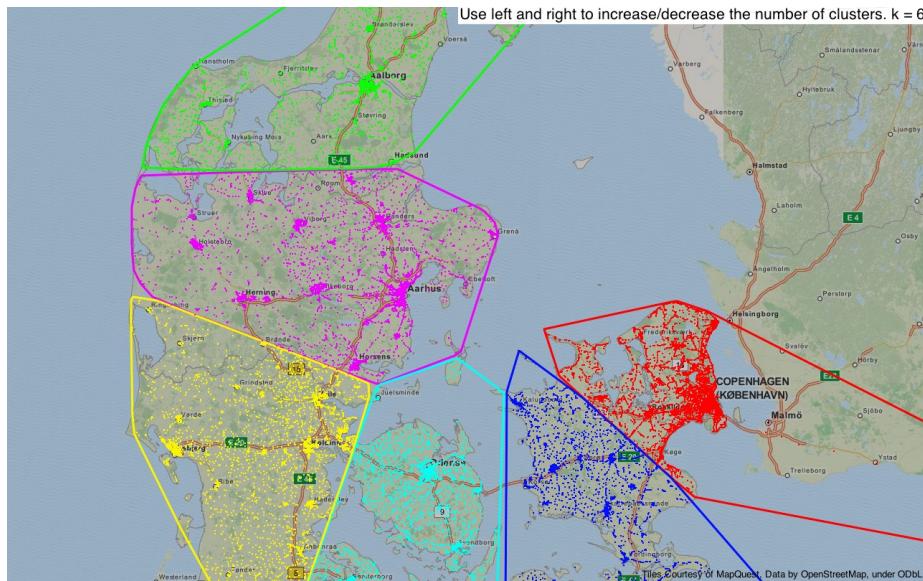


Fig 9: The bus stops points are split into 6 groups, and each group is represented by a different colored convex hull.

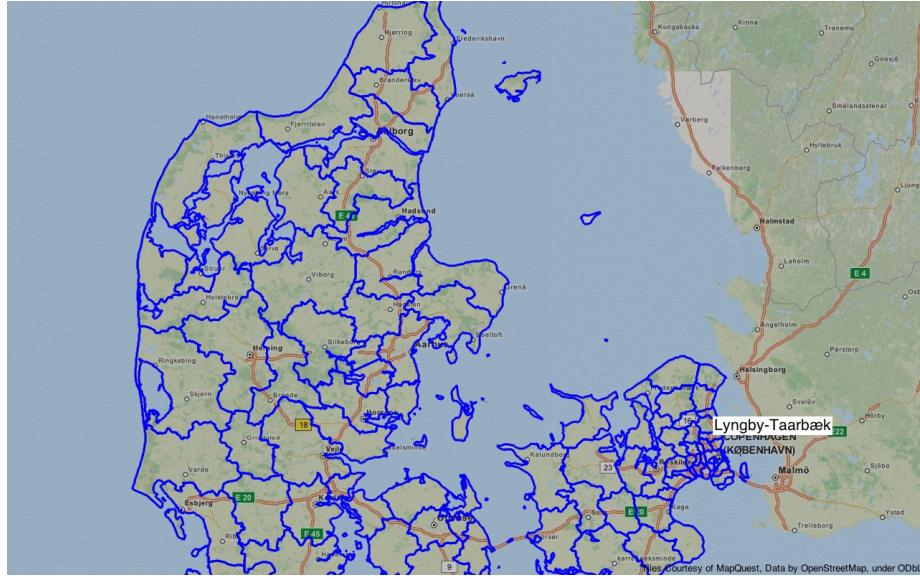


Fig 10: Rendering the shapefiles for the *kommuner* administrative regions in Denmark.

GeoJSON

GeoJSON [42] is a human-readable format for encoding geographical data, such as polygons and lines. geoplotlib can render shapes from the GeoJSON format, and shape color and tooltip can be dynamically altered to encode data. For instance GeoJSON shapes can be used to generate a choropleth where each geographic unit is colored to encode a continuous variable. In the following example (Fig. 11) we generate a choropleth of unemployment in USA [43]:

```

def get_color(properties):
    key = str(int(properties['STATE']))
    key += properties['COUNTY']
    if key in unemployment:
        return cmap.to_color(unemployment.get(key),
                             .15, 'lin')
    else:
        return [0, 0, 0, 0]

with open('data/unemployment.json') as fin:
    unemployment = json.load(fin)

cmap = ColorMap('Blues', alpha=255, levels=10)
geoplotlib.geojson('data/gz_2010_us_050_00_20m.json',
                   fill=True, color=get_color,
                   f_tooltip=lambda properties: properties['NAME'])
geoplotlib.geojson('data/gz_2010_us_050_00_20m.json',
                   fill=False, color=[255, 255, 255, 64])
geoplotlib.show()

```

Advanced Functionalities

Data access

The `DataAccessObject` class is the fundamental interface between the raw data and all the geoplotlib visualizations. A `DataAccessObject` is conceptually similar to a table with one column for each field and one row for each sample. This paradigm is very common in data analysis terminology, and is equivalent to ndarrays in numpy, and dataframes in pandas and R. A `DataAccessObject` can be initialized by reading a comma-separated values (CSV) file with the built-in `read_csv` method, or can be constructed from a python dict, or from a pandas [44] dataframe:

```
dao1 = DataAccessObject({ 'field1': somevalues ,  
                         'field2': othervalues })  
dao2 = DataAccessObject(mydataframe)  
dao3 = read_csv('somefile.csv')
```

The only two fields required are `lat` and `lon`, which represent to the geographic coordinates. Most of the built-in visualization implicitly refer to these two fields to locate entities in space. `DataAccessObject` also provides a few method for basic data wrangling, such as filtering, grouping, renaming and deleting rows and columns.

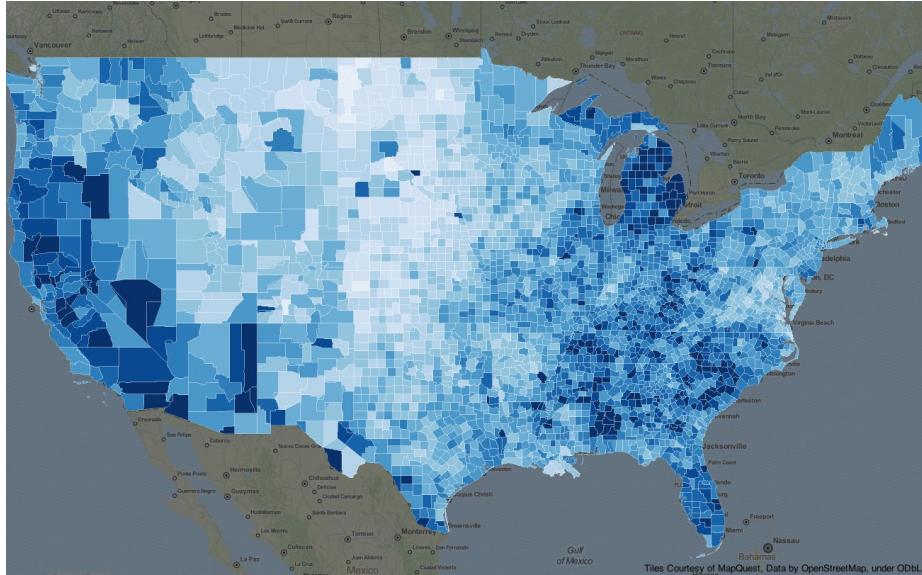


Fig 11: Choropleth of unemployment in USA using GeoJSON shapefiles

Tile providers

Any OpenStreetMap tile server can be configured using the `tile_provider` method (users are kindly asked to check the tile usage policy for the selected server, and make sure to provide attribution as needed). A number of common free tiles providers are supported, including Stamen Watercolor and Toner [45], CartoDB Positron and DarkMatter [46].

Defining custom layers

The built-in visualizations provide various commonly used tools for geographical data visualization. Multiple layers can be combined into a single visualization for richer display. For even more complex visualizations, geoplotlib allows users to define custom layers. In order to generate a new visualization, a new class extending `BaseLayer` must be defined. The custom layer must at least define an `invalidate` and a `draw` method. The `invalidate` method is called each time the map projection must be recalculated, which typically happens each time that the map zoom-level changes. The `invalidate` method receives a `Projection` object, which provides methods for transforming the data points from the geographic coordinates to screen coordinates. The screen coordinates can then be passed to a `BatchPainter` object for the rendering. A `BatchPainter` can efficiently draw OpenGL primitives such as points, lines and polygons. The `draw` method is called at each frame, and typically calls the `batch_draw` method of the painter prepared during `invalidate`. The following is a complete example of a custom layer, which simply draws samples as points:

```
class CustomLayer(BaseLayer):
    def __init__(self, data):
        self.data = data

    def invalidate(self, proj):
        x, y = proj.lonlat_to_screen(self.data['lon'],
                                      self.data['lat'])
        self.painter = BatchPainter()
        self.painter.points(x, y)

    def draw(self, proj, mouse_x, mouse_y, ui_manager):
        self.painter.batch_draw()
```

The final step needed is to add the layer to the visualization using `add_layer`, then call `show`:

```
geoplotlib.add_layer(CustomLayer(mydata))
geoplotlib.show()
```

Animation

A custom layer can be also used for creating animated visualizations. Each time the `draw` method is called, the custom layer can update its state to the next frame. As an example, let us imagine having data containing the position of

an object over time. A simple animation can use a frame counter, and at each frame render only the datapoint at the current instant:

```
class AnimatedLayer(BaseLayer):

    def __init__(self, data):
        self.data = data
        self.frame_counter = 0

    def invalidate(self, proj):
        self.x, self.y = proj.lonlat_to_screen(
            self.data['lon'], self.data['lat'])

    def draw(self, proj, mouse_x, mouse_y, ui_manager):
        self.painter = BatchPainter()
        self.painter.points(self.x[self.frame_counter],
                            self.y[self.frame_counter])
        self.painter.batch_draw()
        self.frame_counter += 1
```

Notice that in this case we do not initialize the `BatchPainter` inside `invalidate`, but we create a new one at each frame. We also keep track of the current frame with the `frame_counter` variable. Even this very simple code is able to visualize a non-trivial animation of an object moving over time. To produce a movie from the animation, individual frames can be captured using the `screenshot` method, and then combined together.

Colormaps

Colors can be used as additional mapping for encoding information into a visualization. Continuous variables (for example points density or the edges distances) can be mapped to a continuous color scale. The `ColorMap` class allows a user to perform this conversion. A `ColorMap` object is constructed by passing any of the matplotlib colorscales, and optionally an alpha value and a number of discretization levels. The `to_color` method performs the conversion from real value to color:

```
# hot colormap
cmap = ColorMap('hot')

# Reds colormap with transparency
cmap = ColorMap('Reds', 128)

# coolwarm colormap with 4 levels
cmap = ColorMap('coolwarm', levels=4)

# linear scaling
cmap.to_color(10, 100, 'lin')

# logarithmic scaling
cmap.to_color(10, 100, 'log')

# square-root scaling
cmap.to_color(10, 100, 'sqrt')
```

Discrete variables such as categories can be represented using categorical colormaps. The `colorbrewer` method provides access to the ColorBrewer [47] colors. Categorical colormaps can be also generated from regular colormaps using using `create_set_cmap`:

```
cmap1 = colorbrewer([1,2,3])
cmap2 = create_set_cmap('hsv', [1,2,3])
```

Controlling the map view

The map view is determined by the projection parameters: the latitude offset, the longitude offset and the zoom level. By default, the projection is chosen so to fit all selected points, with the maximum zoom level possible. The view can be changed to a specific portion of the map by passing a `BoundingBox` object to the `set_bbox` method. A `BoundingBox` object defines the map view boundaries, and can be constructed in multiple ways. The most direct way is to specify two ranges of latitudes and longitudes. Alternatively, a `BoundingBox` can be constructed to fit a subset of points using the `from_points` methods.

```
bbox1 = BoundingBox(north=51.3, west=-124.3,
                    south=14.8, east=-56.8)
bbox2 = BoundingBox.from_points(lons, lats)
```

Interactivity

Finally, geoplotlib allows users to create interactive visualizations by providing support for rendering a user interface, and dynamically changing the visualization on user input:

- on-screen text such as information or status can be added using the `UiManager` class.
- mouseover tooltips can be configured on arbitrary graphical elements or screen regions using the `HotspotManager` class.
- layers can be configured to react to specific key presses by defining a `on_key_release` method

Performance

We test the performance of geoplotlib by generating some of the described visualization on a dataset consisting of one million samples, using the default visualization parameters. All tests consider only the time needed for the actual rendering of the visualization, excluding the time for loading the data. The measurements are repeated 10 times for each visualization type. The experiments were performed on a MacBook Pro 2012 with an Intel 2.3 GHz i7 CPU, 8 GB RAM and nVidia GeForce GT 650M GPU. Table 1 shows that in all cases the visualizations require only a few seconds, thus demonstrating that geoplotlib is suitable even for large-scale datasets.

Table 1: Execution time for one million samples

visualization type	mean time [s]	SD [s]
dot	1.57	0.08
graph	1.99	0.09
hist	8.12	0.55
kde	5.37	0.37
voronoi	3.08	0.66

Conclusion

We have presented geoplotlib, a python toolbox for generating geographical visualizations. We demonstrated how geoplotlib provides a simple yet powerful API to visualize geographical data, greatly facilitating exploratory data analysis of geographical information. We believe that geoplotlib can become a powerful tool in the data analyst toolbox, both for analyzing complex spatial patterns and for communicating results in forms of geographical visualizations. Future work includes the addition of more visualization tools, and the integration of spatial analysis methods.

Acknowledgments

This work is funded in part by the High Resolution Networks project (The Villum Foundation), as well as Social Fabric (University of Copenhagen).

References

- [1] Tukey JW. Exploratory Data Analysis. Addison-Wesley; 1977.
- [2] Andrienko N, Andrienko G. Exploratory Analysis of Spatial and Temporal Data: A Systematic Approach. Springer Science & Business Media; 2006.
- [3] Van Rossum G, Drake Jr FL. Python Reference Manual. Centrum voor Wiskunde en Informatica Amsterdam; 1995.
- [4] Millman KJ, Aivazis M. Python for Scientists and Engineers. Computing in Science & Engineering. 2011;13(2):9–12. doi:<http://dx.doi.org/10.1109/MCSE.2011.36>.
- [5] Oliphant TE. Python For Scientific Computing. Computing in Science and Engineering. 2007;9(3):10–20.
- [6] Van Der Walt S, Colbert SC, Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering. 2011;13(2):22–30.

- [7] Jones E, Oliphant T, Peterson P, et al.. SciPy: Open Source Scientific Tools for Python; 2001-. Available from: <http://www.scipy.org/>.
- [8] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*. 2011;12:2825–2830.
- [9] Patil A, Huard D, Fonnesbeck CJ. PyMC: Bayesian Stochastic Modelling in Python. *Journal of statistical software*. 2010;35(4):1.
- [10] Peirce JW. PsychoPy — Psychophysics Software in Python. *Journal of neuroscience methods*. 2007;162(1):8–13.
- [11] Cock PJ, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, et al. BioPython: Freely Available Python Tools for Computational Molecular Biology and Bioinformatics. *Bioinformatics*. 2009;25(11):1422–1423.
- [12] Haklay M, Weber P. OpenStreetMap: User-Generated Street Maps. *Pervasive Computing, IEEE*. 2008;7(4):12–18.
- [13] Cuttone A. geoplotlib; 2015-2016. Available from: <https://github.com/andrea-cuttone/geoplotlib>.
- [14] Hunter JD. Matplotlib: A 2D Graphics Environment. *Computing in Science and Engineering*. 2007;9(3):90–95.
- [15] Whitaker J. Basemap;. Available from: <http://matplotlib.org/basemap/>.
- [16] Met Office. Cartopy: A Cartographic Python Library with a Matplotlib Interface; 2010 - 2015. Available from: <http://scitools.org.uk/cartopy>.
- [17] Rey SJ, Anselin L. In: Fischer MM, Getis A, editors. PySAL a Python Library for Spatial Analytical Methods. Berlin: Springer-Verlag; 2010. p. 175–193.
- [18] Bostock M, Heer J. Protovis: A Graphical Toolkit for Visualization. *Visualization and Computer Graphics, IEEE Transactions on*. 2009;15(6):1121–1128.
- [19] Bostock M, Ogievetsky V, Heer J. D³ Data-Driven Documents. *Visualization and Computer Graphics, IEEE Transactions on*. 2011;17(12):2301–2309.
- [20] Google Charts;. Available from: <https://developers.google.com/chart/>.
- [21] Alexis Jacomy GP. sigmajs;. Available from: <http://sigmajs.org/>.
- [22] Google Maps;. Available from: <https://developers.google.com/maps/>.

- [23] Bing Maps;. Available from: <https://www.bingmapsportal.com/>.
- [24] leafletjs;. Available from: <http://leafletjs.com/>.
- [25] openlayers;. Available from: <http://openlayers.org/>.
- [26] modestmaps;. Available from: <http://modestmaps.com/>.
- [27] polymaps;. Available from: <http://polymaps.org/>.
- [28] folium;. Available from: <https://github.com/wrobstory/folium>.
- [29] vincent;. Available from: <https://github.com/wrobstory/vincent>.
- [30] mplleaflet;. Available from: <https://github.com/jwass/mplleaflet>.
- [31] QGis. Quantum GIS Geographic Information System. Open Source Geospatial Foundation Project. 2011;.
- [32] GrassGIS;. Available from: <http://grass.osgeo.org/>.
- [33] ArcGIS;. Available from: <http://www.arcgis.com/>.
- [34] MapInfo;. Available from: <http://www.mapinfo.com/>.
- [35] pyglet;. Available from: <https://bitbucket.org/pyglet/pyglet/wiki/Home>.
- [36] Pérez F, Granger BE. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*. 2007;9(3):21–29.
- [37] Parzen E. On Estimation Of a Probability Density Function and Mode. *The annals of mathematical statistics*. 1962; p. 1065–1076.
- [38] Aurenhammer F. Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys (CSUR)*. 1991;23(3):345–405.
- [39] De Berg M, Van Kreveld M, Overmars M, Schwarzkopf OC. Computational Geometry. Springer; 2000.
- [40] ESRI. ESRI Shapefile Technical Description; 1998.
- [41] pyshp;. Available from: <https://github.com/GeospatialPython/pyshp>.
- [42] Butler H. geoJSON;. Available from: <http://geojson.org/geojson-spec.html>.
- [43] Bostock M. Unemployment in USA;. Available from: <http://bl.ocks.org/mbostock/4060606>.
- [44] McKinney W. Data Structures for Statistical Computing in Python. In: van der Walt S, Millman J, editors. Proceedings of the 9th Python in Science Conference; 2010. p. 51 – 56.

- [45] Stamen;. Available from: <http://maps.stamen.com/>.
- [46] CartoDB;. Available from: <http://cartodb.com/basemaps/>.
- [47] Harrower M, Brewer CA. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal*. 2003;40(1):27–37.