Geospatial density time-series with Matplotlib

This post is going to explain how to create an animations from heatmaps of dynamic commute data. I've written this article to be a more conceptual explanation that is complementary to the Jupyter notebook containing the full code. If you have downloaded this article as a PDF from Github, then you can find the original blog (containing GIFs) at http://quorumetrix.blogspot.ca/

The analysis broadly involves two steps. First we must make a list of the commute trajectories that satisfy our criteria. Second, we must extract all the points of the trajectories and organize them in a way to be visualized dynamically, in this case using a 3D array.

I obtained all of the necessary data from the Montreal Open Data Portal, specifically the MTL Trajet data 'trip_final.json' and it is important to also have the boundaries of the city boroughs, (in this case LIMADMIN.json).

In the interest of making this widely applicable, I'll try to explain the steps in a manner that is agnostic to what city we're dealing with - so long as one's home city has had a similar project to collect transit data. The json format is used as a standard, but I can't guarantee things will be stored in quite the same way between cities, and discovering exactly how to properly extract each data point is half the battle. In my case, I had to load the json into a dictionary data type in Python, and experiment through trial and error to find the right way of indexing the contents to extract the positions for each record.

After importing the necessary packages into the current kernel, we run the **load_trip_data.py** script which loads the relevant variables into memory. Briefly, this script loads the json data into

Python dictionaries, then iterates sorts through the data to make it more accessible and easier to understand. For example, this script creates a 2D array called **coords_if** that stores the coordinates of the initial and final point of each trajectory. This script also cleans up some of the erroneous text. For example: "Baie-d'UrfÃ©" is corrected to be “Baie-d'Urfé"

So as a first pass of the dictionary, while loading the values into Python, for simplicity’s sake I created an array to contain the initial and final point of each displacement. Knowing how many entries there were, I could preallocate an empty array in numpy, where the number of rows reflects the number of data entries (293,330), and the 4 columns represent the latitude and longitude of the initial and final point of each trajectory. This is of particular interest when we want to localize the start and end-points to specific boroughs of the city, or decide which begin on or off the island. Importantly, the script also creates a vector of the same length (293,330) containing the unique IDs for each trajectory, which is critical for comparing lists with different selection criteria.

To visualize the traffic flowing onto the island, I had to isolate the trajectories that began outside of all of the borough shapefiles, and that ended inside. I’m not yet a master of Python, but the best way I found to do this was to use the Path package for storing vertices of the borough shapefiles, and to create a Path object containing all the coordinates of all the burroughs. I then used the built-in **contains_points** function to return a boolean mask of the trajectories that start in the selection of points (on the island). It’s worth noting that by changing the set of shapes we compare against, we can find points that start in any specific borough (see a few examples at the end of the tutorial).

We use what I’ll refer to as the ‘*masking*’ method, which you can follow along with the table below. We first want to find the initial coordinates (first two columns) starting on the island, and then invert the boolean vector (using **numpy.invert**) to return a boolean vector referring to the trajectories starting off the island. To compare between lists, I need to use the IDs, so I apply the mask to the ID vector to create a list of IDs for all trajectories beginning off the island, **Mask1**.

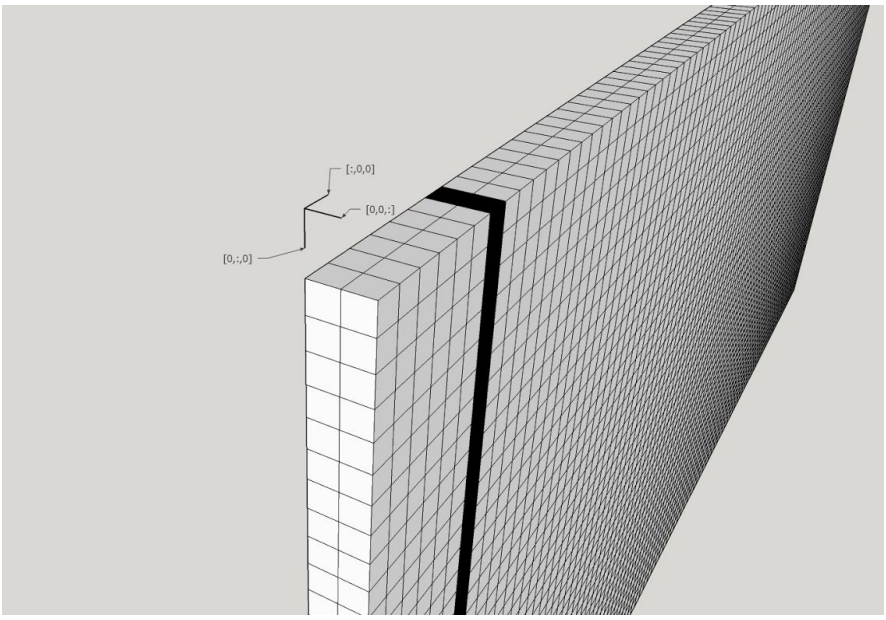
The step for **Mask2** is much simpler, I wanted to find the IDs for all trajectories ending on the island, so created a boolean list using **Path.contains_points**, this time using the 3rd and 4th column [:,2:]. Again I apply this boolean mask to the ID list **ids** to return the list ending on the island. Finally, Since it’s the overlap of these two groups I’m interested in, I determined list of overlapping ID numbers (using **numpy.isin**) - which gives me the trajectories of interest.

Mask1 (Starting off-island)	**Mask2** (Ending on-island)	**Mask3** (Overlap)	**ids**	**sharedIDs**
0	1	0	XXXXXXX1	XXXXXXX3

1	0	0	XXXXXXX2		XXXXXXX5
1	1	1	XXXXXXX3	XXXXXXX3	XXXXXXX8
0	1	0	XXXXXXX4		
1	1	1	XXXXXXX5	XXXXXXX5	
0	0	0	XXXXXXX6		
1	0	0	XXXXXXX7		
1	1	1	XXXXXXX8	XXXXXXX8	
0	1	0	XXXXXXX9		
0	0	0	XXXXXX10		
0	1	0	XXXXXX11		
1	0	0	XXXXXX12		

The second section is more computationally intensive, and is helped by a certain level of comfort thinking about multidimensional arrays. I've created the image below using sketchup as a guide.

`coordMat = np.empty((nIds,length_cutoff,2))`



Since I'm interested in the behaviour of people through time, the trick is to set up an array with the correct number of dimensions to make it easy to slice for plotting. I knew from an earlier analysis that the vast majority (almost all) of recorded trajectories had fewer than 600 points, so I used this as an upper limit to the length (length_cutoff). I also know the number of trajectories (**nIds**) we'll be considering from the length of **sharedIDs**. Finally, for each time point and each trajectory, I want two coordinates (latitude and longitude to represent the spatial location). Therefore I created a 3D array in numpy, and preallocated all elements as empty:

```
coordMat = np.empty((nIds,length_cutoff,2)))
```

In this array, each row represents a different trajectory [:,0,0], each of the 2 columns represents latitude and longitude respectively [0,0,:], and the depth of the array reflects the time [0,:,0]. In order to create a dynamic visualization with respect to time, we will be slicing the array along the time-dimension [0,:,0], indicated by the black shading in the figure above.
Note: the normal convention would be to represent the third dimension [0,0,:] as depth in the image, but I preferred this orientation to better see the shape of the array.

We then iterate through each of the recorded trajectories, query the ID number, and compare it to the list of IDs I've generated. For each of the items that is included on my list, I create a temporary array to store all of the coordinates associated with this trajectory.

```
coordMat[int(i_id), temp_i:, :] = tempArray
```

Since not all of the entries have the same number of coordinates, I had to decide where to store them in the array. Since I want the animation to have all coordinates ending at the same time, I decided to store the coordinates from the current trajectory at the end of the array, so I do so using it's length (size / 2 for an array of 2 columns).

Once this has been done for every trajectory, the coordinate array (**coordMat**) is finished, it is then time to visualize it. To make sure everything looks as it should, a scatter plot for each timepoint is a good place to start. Iterate through each of the time values:

[:, **t**, :] where 0 < **t** < 600.

and plot a scatter plot of the array sliced through that value of **t**. I should note that in this case **t** is representing a relative time, since we've not done anything to line the trajectories up with respect to the absolute timestamp (which I have done in a previous visualization).

To make a dynamic video or gif of the data through time, we ensure that the plot keeps the same axes, is replotted and exported at every value of **t**. By numbering the exported images (.png's) sequentially, we can load them into imageJ or Fiji to create a movie or gif.

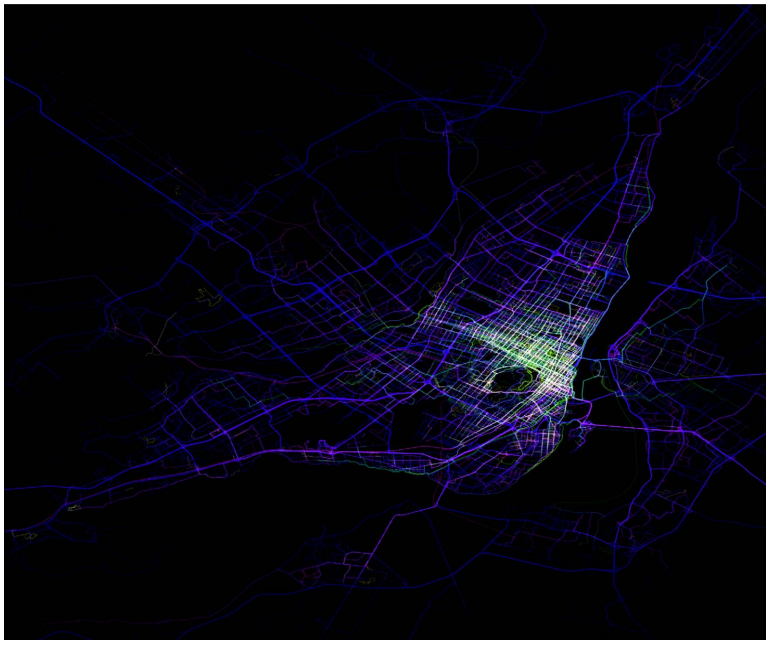
Now that this procedure is working, such that a new scatterplot is made for each different time-slice of the array, we can make any kind of plot we want. I settled on the hexbin plot from Matplotlib as my favorite, which is how I made the animation at the start of this article. You can also choose from a variety of colormaps, while my preference was for 'inferno' or 'bone'.

So that's it for the basic script to plot all trajectories heading between two user-defined locations, plotted as a function of relative time. You can play around with it and change the code at key locations in order to experiment with other effects. Below I show a few examples variants I've made with this dataset:

Lowering the number of bins from 500 to 100 gives a cool retro-nintendo style appearance:

By inverting the selection criteria, I found all the trajectories which started on the island and ended off the island, and added these frames to the initial gif. I found that it was a little too quick in the middle, so I created another version where the trajectories which begin and end on the island are displayed in between the first two.

I had also previously made a static version where I created a grayscale hexbin plot for each mode of transit, by adding if blocks into the main loop. After having a grayscale image for each mode of transit, I merged the channels in Fiji for a multicolor image (shown below).



If each individual data point is of little importance, but the overall population trends are, then a contour plot (matplotlib contourf) can show the most travelled paths:

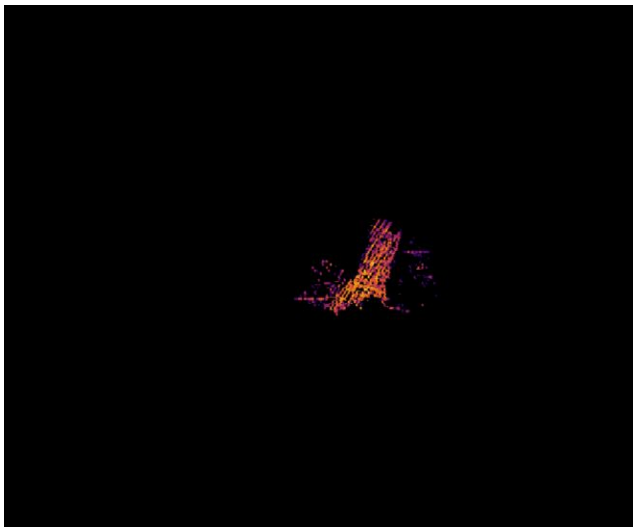
We can also look specifically for traffic leaving a single borough by changing what shapefiles are loaded into the Path using in the ‘masking’ method above. For example, the code:

```
mask1 = allPaths.contains_points(coords_if[:,:2])
```

Can be replaced with:

```
mask1 = currBurrPath.contains_points(coords_if[:,:2])
```

Where **currBurrPath** contains only the shape coordinates of a single borough of interest. The gif below shows and example of exclusively traffic leaving the downtown area.



Well that’s all for now, I hope that this explanation and the source code is useful and can help people make similar visualizations of data from their home city. Please feel free to give me feedback on this or other articles, and follow me on **@Quorumetrix** on Twitter, Facebook or Instagram to be notified of new content.