Probe Data Analysis for Road Slope

Feiyu Chen, Zunan Guo

Introduction

What are probe data?

Probe data are a set of positions of a moving object over space and time. It’s usually acquired by sensors such as GPS, Bluetooth readers, and toll tag readers, etc. The targeted moving objects include cars , people with smartphones, and other dedicated probe vehicles.

Application:

The probe data can be used to

1. Measure traffic speeds and delay of intersections to assist the traffic design.

2. Help urban planner to do better transportation planning.

3. For map refinement/creation, such as **measuring road slope**, as required in this homework.

Overview: How to estimate road slope from probe data

The data given in the homework include **probes** and **links (roads)**:

Probes have attributes of **position** and **altitude**.

Links have attributes of **positions of its two ends.**

We could first use some algorithm to match each probe to certain link. Then, for each link, use its probes to fit a line, and the slope of the fitted line would be a good estimate of the link's slope.

Overview: How to estimate road slope from probe data

Workflow

1. **Preprocessing**

Preprocess raw data to acquire the position of each probe and link.

2. **Road matching**

Match each probe to link.

3. **Slope estimation**

Estimate the slope of each link.

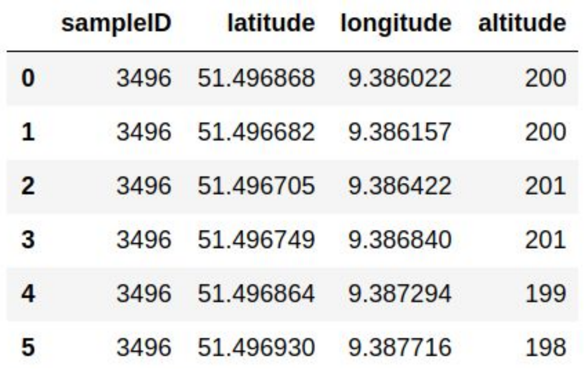
4. **Result**

Evaluate the estimated slope by comparing it to the collected slope value.

Data Preprocessing

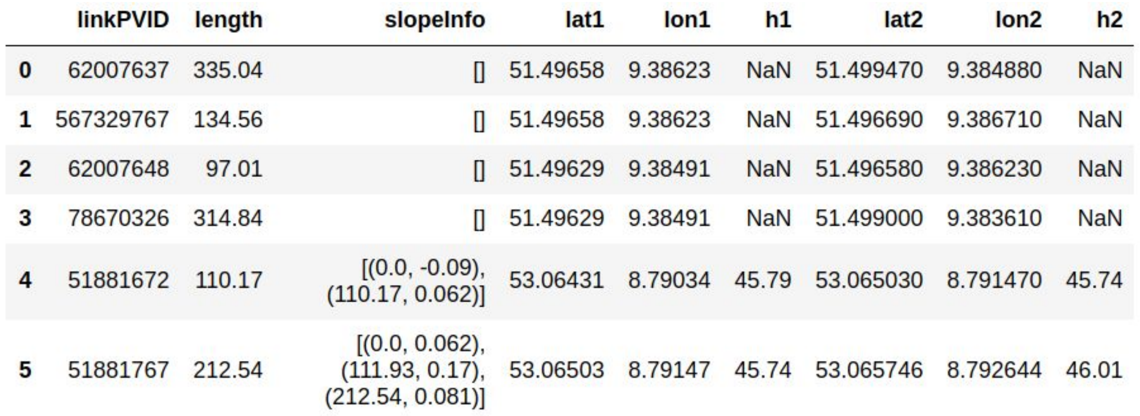
Read and filter data

There are a variety of attributes of probe and link in the given csv files, such as *dateTime*, *sourceCode*, *speed*, etc. However, what we need only is the ID and position information of each probe and link. We used *python* and *pandas* library to read the csv files, and then filtered out the non-important data. Part of the data are shown in the figures below:



	sampleID	latitude	longitude	altitude
0	3496	51.496868	9.386022	200
1	3496	51.496682	9.386157	200
2	3496	51.496705	9.386422	201
3	3496	51.496749	9.386840	201
4	3496	51.496864	9.387294	199
5	3496	51.496930	9.387716	198

Probe data



	linkPVID	length	slopeInfo	lat1	lon1	h1	lat2	lon2	h2
0	62007637	335.04	[]	51.49658	9.38623	NaN	51.499470	9.384880	NaN
1	567329767	134.56	[]	51.49658	9.38623	NaN	51.496690	9.386710	NaN
2	62007648	97.01	[]	51.49629	9.38491	NaN	51.496580	9.386230	NaN
3	78670326	314.84	[]	51.49629	9.38491	NaN	51.499000	9.383610	NaN
4	51881672	110.17	[(0.0, -0.09), (110.17, 0.062)]	53.06431	8.79034	45.79	53.065030	8.791470	45.74
5	51881767	212.54	[(0.0, 0.062), (111.93, 0.17), (212.54, 0.081)]	53.06503	8.79147	45.74	53.065746	8.792644	46.01

Link data

Data Preprocessing

Convert coordinate from Spherical to Cartesian

Positions in the raw data are represented in a Spherical coordinate, with a format of (latitude, longitude). However, it’s difficult to do visualization or measuring distances in Spherical coordinate, so we converted the position into the Cartesian coordinate.

How to convert (latitude, longitude) into (x, y) ?

We can choose a point on the sphere as the origin, and establish a Cartesian coordinate on it. The origin is chose at the average position of all the probes (or links), in order to reduce the error from coordinate conversion. (An example of this error is: On a world map, the regions near North and South poles look larger than normal.)

See next slide for the formula.

Data Preprocessing

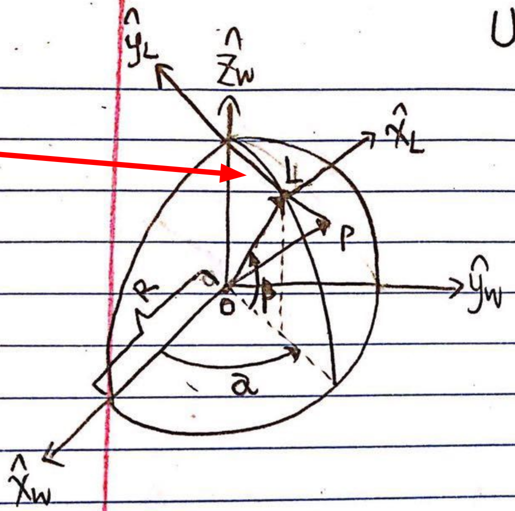
Convert coordinate from Spherical to Cartesian

Suppose the average position of all probes is **L**.

The coordinate on it is {L}.

The coordinate on the center of Earth is {W}.

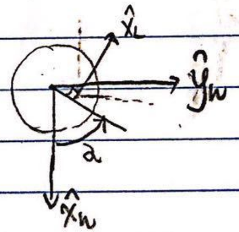
For a point in {W} represented by (latitude, longitude), we can compute its position in the {L} coordinate by using the formats on the right.



Under $\{\hat{x}_W, \hat{y}_W, \hat{z}_W\}$ Coordinate:

$$\vec{OL} = \begin{bmatrix} R\cos\beta_L \cos\alpha_L \\ R\cos\beta_L \sin\alpha_L \\ R\sin\beta_L \end{bmatrix}$$

$$\hat{x}_L = \begin{bmatrix} -\sin\alpha \\ \cos\alpha \\ 0 \end{bmatrix}$$



$$\hat{y}_L = \frac{\vec{OL}}{\|\vec{OL}\|} \times \vec{x}_L$$

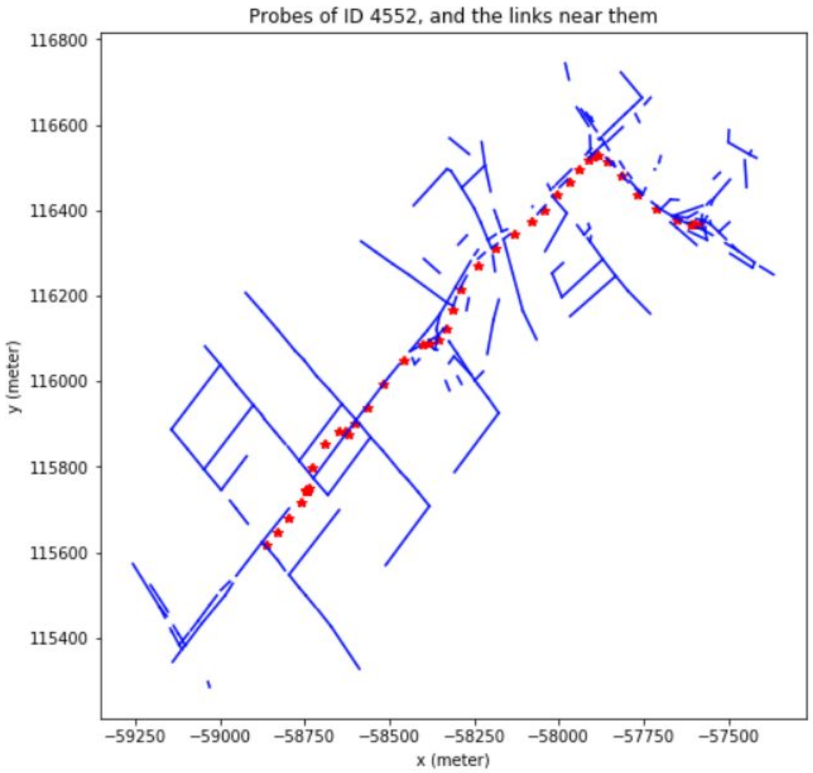
Any point P in $\{\hat{x}_W, \hat{y}_W, \hat{z}_W\}$ coordinate $= \begin{bmatrix} R\cdot\cos\beta_P \cdot \cos\alpha_P \\ R\cdot\cos\beta_P \cdot \sin\alpha_P \\ R\cdot\sin\beta_P \end{bmatrix} = \vec{OP}$

Its position in $\{\hat{x}_L, \hat{y}_L\}$ coordinate $= \begin{bmatrix} (\vec{OP} - \vec{OL})\cdot\hat{x}_L \\ (\vec{OP} - \vec{OL})\cdot\hat{y}_L \end{bmatrix}$

See the code: *def convert_lat_lon_to_x_y(*

Data Preprocessing

Example probes and links in new coordinate



See the code: *def plot_probes_and_links(*

Data Preprocessing

Remove links that are far away from probes

Before doing road matching, we remove the links that are approximately 1000 meters far away from all of the probes.

The purpose is to speed up the data processing by focusing on links that really matter.
(For example, during debugging the program, we only used about 200 probes. In this case, we don't want to do road matching with all 300k links. So we removed the remove links, and retained only about 1000 links for processing.)

However, when processing the whole dataset, this step is not essential, because all links are within 1000 meters of certain probe.

See the code: *def filt_data_by_range(*

Data Preprocessing

Build K-d tree for fast searching a probe’s nearest neighbor links

K-d Tree (k-dimensional tree) is a Binary Search Tree for querying a point’s nearest neighbor links in O(log(N)) time, where N is the number of points inside the tree. The time complexity of building the tree is O(N*log(N)).

We use *scipy.spatial.KDTree* to build a tree of all the links.
The center position of a link is used as the “key” for indexing inside the K-d Tree.

For each probe, we will first find some neighbor links. The number of neighbors is set as 10, in order to ensure that they contain the true nearest link to the probe.

See the code: *class KDTreeLinks(*

Road Matching - v1

Purpose:

The purpose of road matching is to find:

For a link, what are this link’s matched probes?

So that we can use these probes to estimate the slope of the link.

Road Matching - v1

Pseudo-code:

```
For each probe:
    Find Q=10 neighbor links by K-d tree.
    Measure the distances to them
    If 1st_min_dist < Th2=50, continue
    If (1st_min_dist / 2nd_min_dist) < Th1=0.4, continue
    Match success! Store result.
```

If we only query 1 link, it might not be the real nearest link.

We use distance of “point-to-line-segment”. See next slide.

Link too far away is not a good match.

If a probe is nearest to two links, with a similar distance, then this matching might be wrong. Thus we abandon it.

See the code: *# Do road matching*

Road Matching - v2

There is a problem of “Road matching v1”: **For probes with the same ID, they should be matched to a set of links that are connected to each other.** However, “Road matching v1” cannot guarantee this.

Thus, we implemented the algorithm introduced in the class, which solves the “link connection” problem.

See next slide

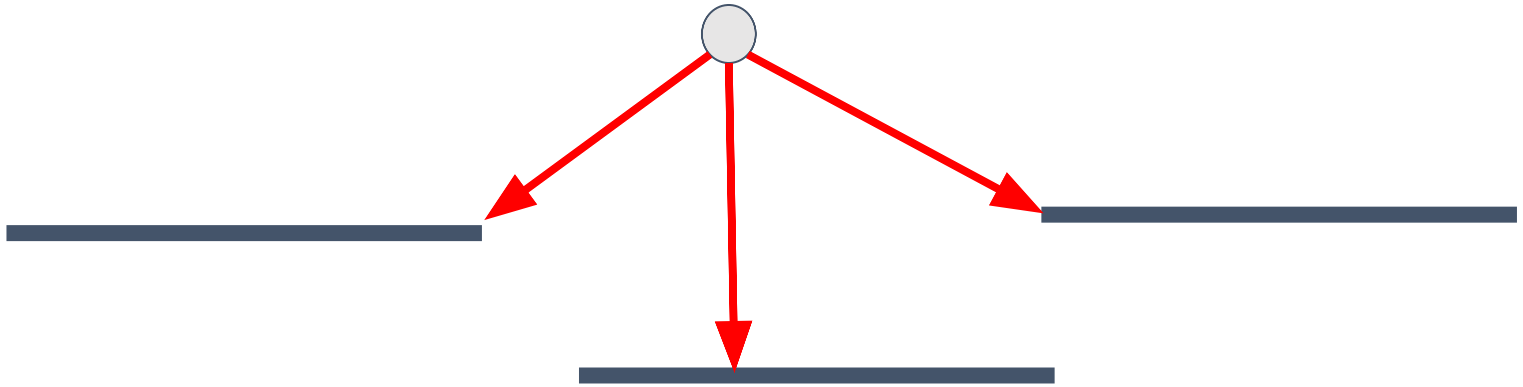
See the folder: *road_matching_on_self-created_grid_map*

Point-to-line-segment distance

How to measure the distance between a probe and a link?

We use the "Point-to-line-segment distance" as described in the image below.

The red arrow represents the distance from a point to the line segment.



See the code: *def calc_dist_to_line_segment(*

Road Matching - v2

There is a problem of “Road matching v1”: **For probes with the same ID, they should be matched to a set of links that are connected to each other.** However, “Road matching v1” cannot guarantee this.

Thus, we implemented an algorithm introduced during the class, which solves the “link connection” problem. We also created a random toy dataset (a grid map), and tested the algorithm.

See followings slides:

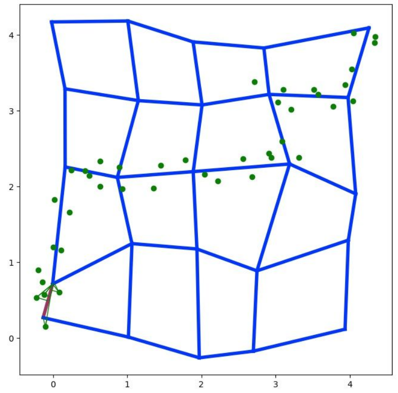
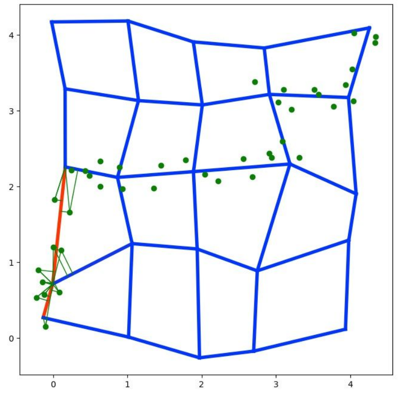
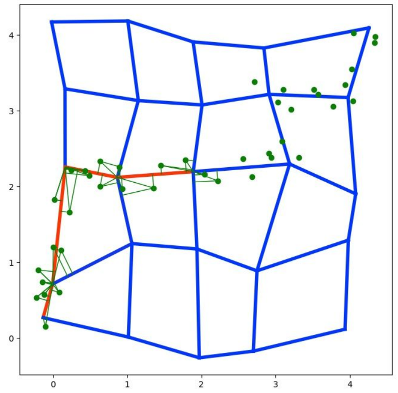
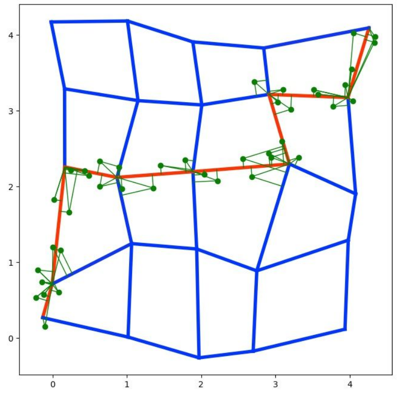
See the folder: *road_matching_on_self-created_grid_map*

Road Matching - v2

First, set each link's weight to 0.

Iterate through a set of probes with the same ID: for each probe, find its neighbors, measure distances, and increase the weight of these links based on the inverse of the distance.

Figures below visualize the process of finding neighbor links and assigning weights: Green dots are probes. Green lines are the matches to links. red link has high weight, blue link has low weight.

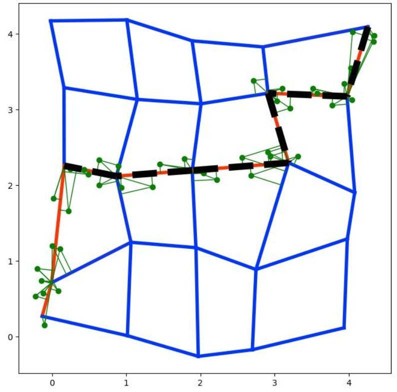





Road Matching - v2

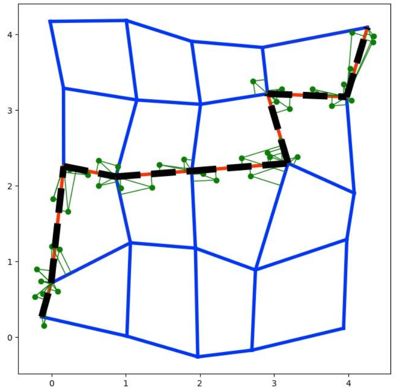
Then, use 2 depth-first search (DFS) to find the trajectory (connected links).

1: Starting from any link with a weight larger than threshold. Use DFS to find one end of the trajectory, by trying to walk through a largest sum of weight.

2: Starting from the end, do the 2nd DFS to find another end of the trajectory that maximizes the sum of weight. In this way, we can find a connected trajectory that probes are matched to.



1st DFS

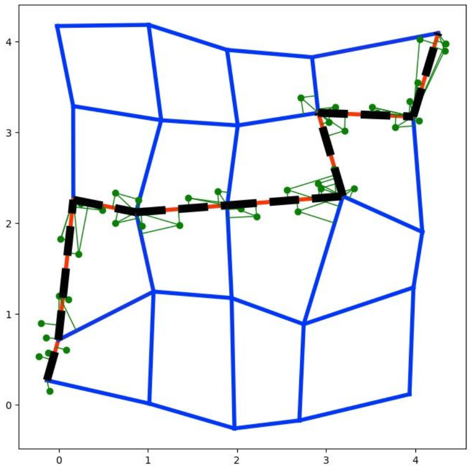


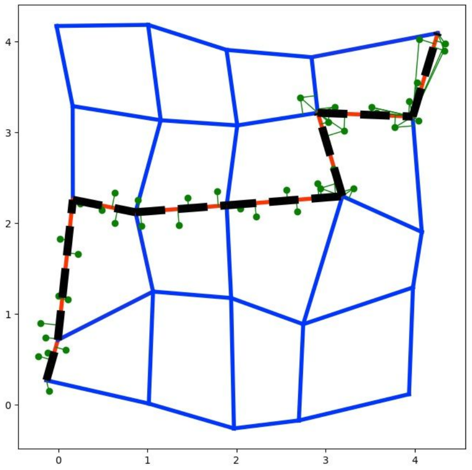
2nd DFS

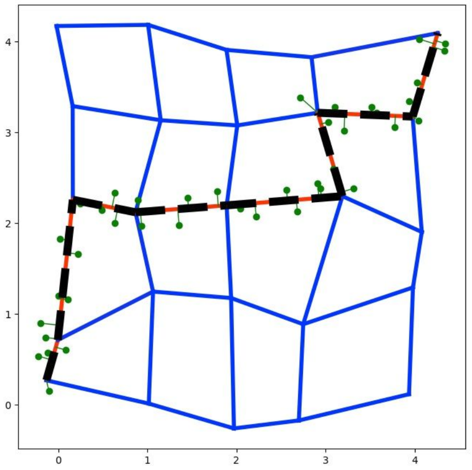
Road Matching - v2

Finally, we match each probe to a nearest link that is part of the detected connected trajectory.

The process is visualized as below, where we sequentially remove the redundant matches (green line) of each probe. The 3rd figure is the final matching result.







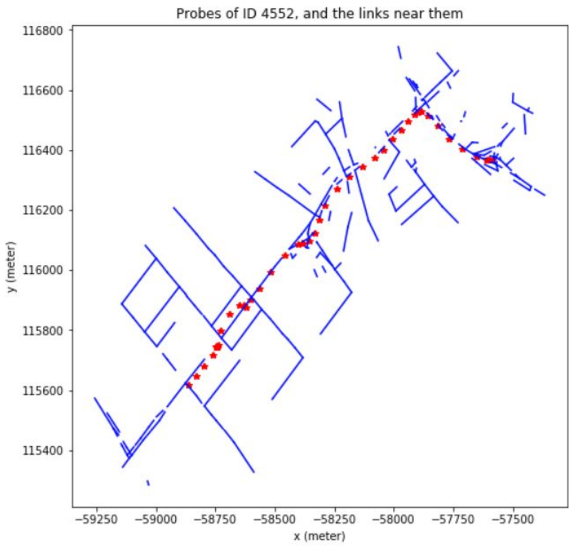
Road Matching - v2

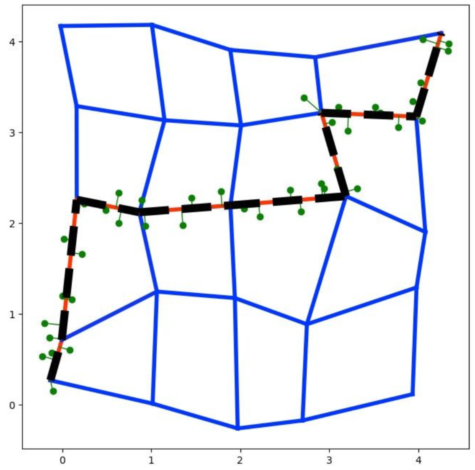
Analysis:

This method could guarantee that matched links are connected.

However, it only works well when raw links are all connected to each other. For the left figure below, which is drawn from the real dataset, some links are not disconnected, and the algorithm will fail.

Thus, in our final implementation of the homework, we chose the “Road Matching - v1”, which can work on the dirty data.





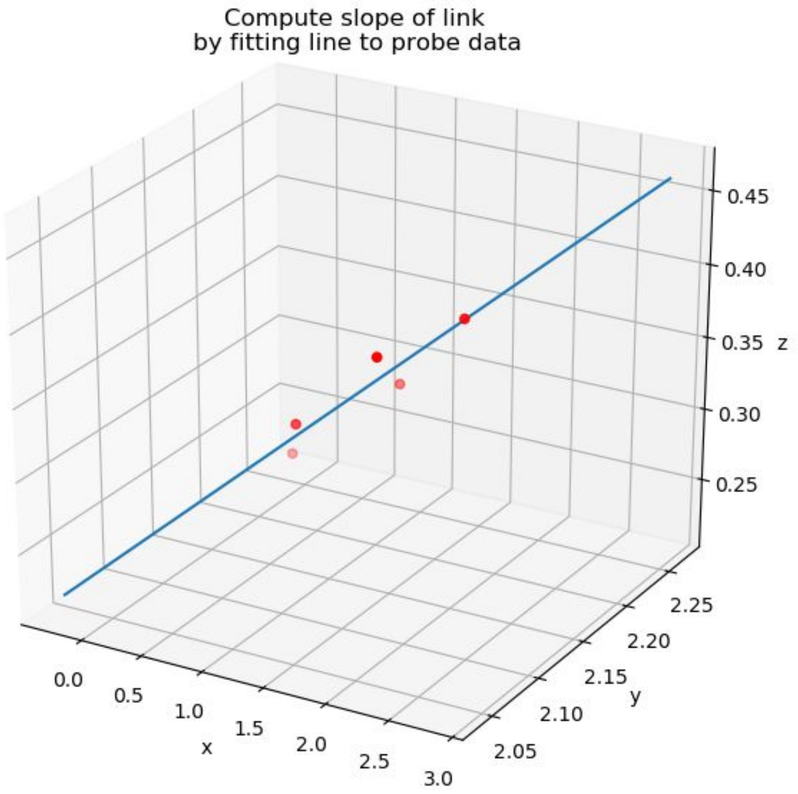
Estimate Slope

After acquiring the probes that are matched to a certain link, we fit a line to these probes' (x, y, h) positions.
The slope of the fitted line is the estimated slope of the link.

How to fit line? We use PCA to obtain the largest component of the data, which is the direction of the line. Then, compute slope from the line direction.

Finally, we set a slope to "invalid" if this estimated slope is larger than 5 degrees. The reasons is that, for some links, its neighbor probes have a large difference in height h, but small different in (x, y) location, causing a slope of 5~80 degrees, which is apparently wrong. So we manually set a threshold and filtered out these data.

Another way to deal with noisy data is using RANSAC. We've implemented it, but it took 0.2s for each link, and 3 hours for all 200k links. The time cost is too large, so we didn't adopt it at last.



See the code: *def fit_line(* and *# Compute slope*

Result

Data:

There are a total of **3375745** probes, and **200089** links.

Time cost:

10 minutes to construct probe/link python object from raw data.
10 seconds to build K-d Tree indexing all the links.
25 minutes to do the road matching.
10 seconds to compute the slope for each link.
Evaluated on a laptop with 2.20GHz Intel i7 CPU of 12 threads.

Memory consumption:

Approximately 2.5 GB

Result

Road matching:

As described above, we set 2 criteria to check whether a match between probe and link is good.

The result is: the number of matched probes is 1581004, out of the total 3375745 probes. The ratio is 46.8%.

Estimate slope:

Total links = 200089

Number of valid ground truth slope data = 53517

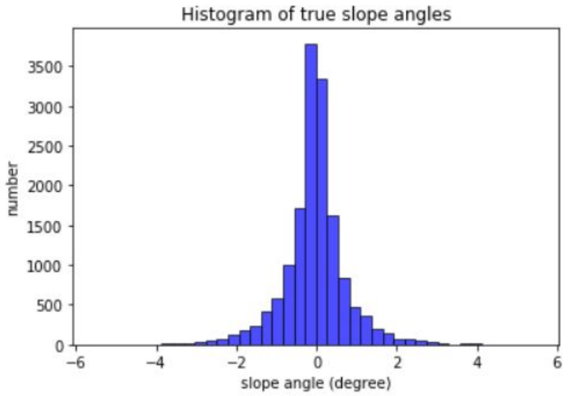
Number of valid estimated slope data = 47065

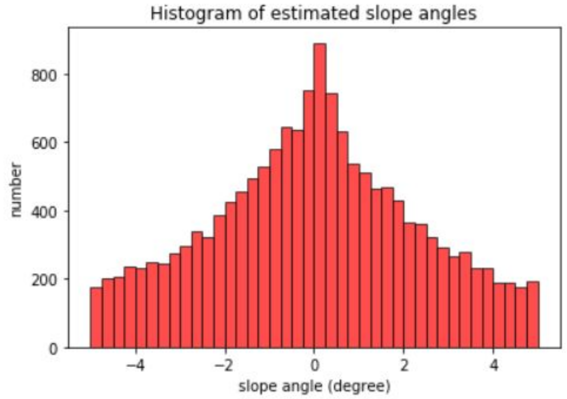
The intersection of the above two data = 15462

In next slide, we will compare and evaluate the estimated and ground truth values of that 15462 pairs of slopes.

Result of estimated slope

The histogram of the true and estimated slope angles are shown on the right. Here the true angles are from the homework dataset, by taking the average of all the slope angles of a certain link.





Result of estimated slope

The error between estimated angles and true angles are shown on the right.

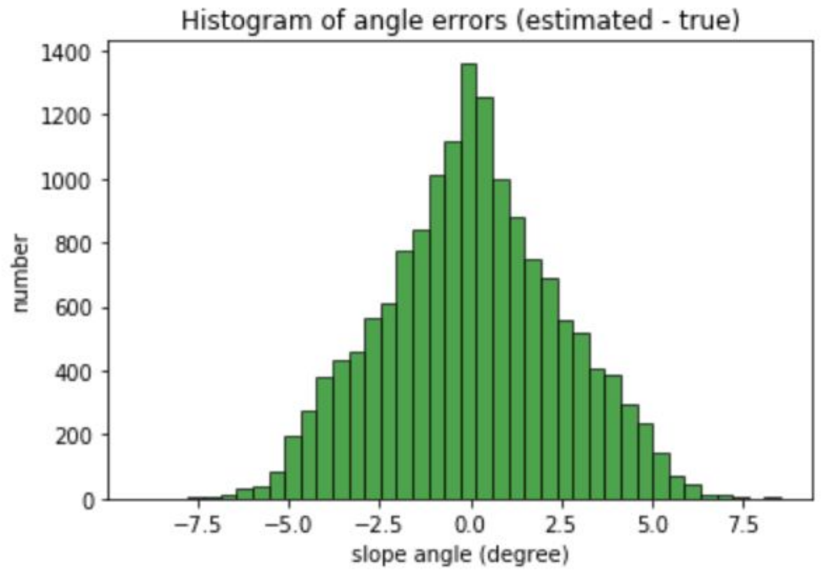
We can observe that errors mainly range from -5 to 5 degrees. This error is too large for real applications.

Why the error is so large?

We examined the data and found that it might due to the noisy value of probe's height:

For some links, their matched probes have a **small (x, y) distance** difference but a **large height difference**, causing an abnormal slope angle.

In future work, we should try to filter the data and obtain good probes' height before doing sloping estimation.



Conclusion

In conclusion, we built a simple pipeline for slope estimation from raw data.

Our work relies heavily on Python’s powerful pandas, numpy/scipy, and matplotlib libraries (for tabular visualization, numeric/scientific computing, and plotting).

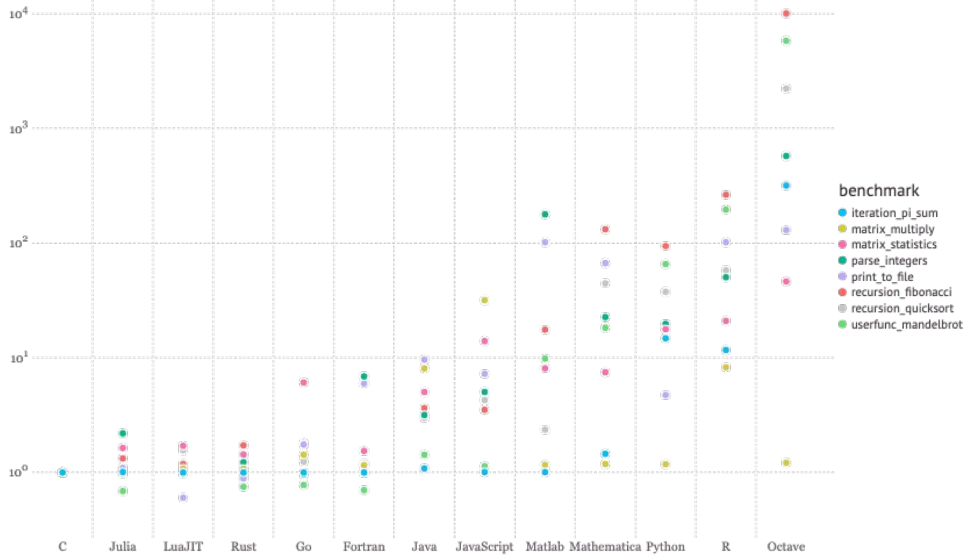
We spent quite some time to construct a toy model (mentioned in road-matching v2 section) to test different theories and hypotheses before scaling a reliable solution to a larger scale. We observe that a complex algorithm might have better accuracy, but its scalability is not as good as simple algorithms when applying to dirty and noisy data.

One key assumption we made is that the selected region for probe and link data can be relatively well fitted on a plane (in our X_L, Y_L coordinates). If the chosen region is too large as the curve of the earth’s sphere becomes obvious, our estimation may not be accurate.

The resultant estimated slope angles have large error comparing to the ground truth slopes. The possible causes are: 1) Large error in some probes’ height. 2) Potential logic error in the implemented road matching algorithm. Future work should focus on these two aspects to improve the performance of road matching and slope estimation.

Future work

One focus for future work is to optimize our prototype and speed up the entire pipeline. Potential trials can be made with Julia or C++. Although C++ is known for its industrial strength, Julia may be a better choice for its **high level expression** and flexibility in research (a wide range of packages support for numeric/scientific computing, plotting, etc.) while maintaining **very high performance**.



source: *https://julialang.org/benchmarks/*

Thank you!