Detection of Lines in 3-D Point Clouds using K-Lines Clustering

Frank Yifeng Lai [1] **Nick Shaju** [*1]

Abstract

Point clouds are heavily used in robotics and autonomous vehicles for object representations in 3D space. Line extraction from 3D space is important especially in self-driving car applications to detect poles and street signs. This paper goes into detail about the use of K-Lines Clustering, an algorithm inspired by K-Means Clustering, to classify lines from 3D point clouds generated from LiDARs. This paper compares the line detection performance to another popular non-deep-learning-based line detection algorithm and finds significant improvements in fidelity and accuracy when presented with noisy data.

1. Introduction

A 3D point cloud consists of a large number of 3D points represented simply as x, y, z euclidean coordinates. Point clouds are a common method to represent the 3D space surrounding a device and can be generated using common sensors such as LiDAR and stereo-depth cameras. However, like any sensor these point clouds generated are susceptible to interference and are often noisy, resulting in random unwanted artifacts or loss of information.

The use of point clouds for object representations in 3D space is quite popular in the fields of robotics and autonomous vehicles. Point cloud information gathered from sensors then, can be used to perceive objects in the car's surroundings. A pivotal step in this process then is the extraction of various features from the 3D space provided. This is particularly useful in self-driving car applications, where abstract landmarks are produced from 3D LiDARs, in specific pole-like objects in urban settings as shown in Figure 1 and (Kampker et al., 2019).

[*]Equal contribution [1] Mechanical and Mechatronics Engineering, University of Waterloo, Waterloo Ontario, Canada. Correspondence to: Frank Yifeng Lai <fylai@uwaterloo.ca>, Nick Shaju <nshaju@uwaterloo.ca>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

Figure 1. Kampker et al., 2019b. *Illustration of the landmark detection*

The ideal way is to define a line such that the mean square error is minimized for the distance of all points that make up the line. However, it is difficult to do so for three main reasons. First, the optimization function is computationally heavy; second, this method requires prior knowledge of the number of lines present in the space; third, the method has a difficult time dealing with outliers as they can skew the predicted line in unpredictable ways. In many applications, an algorithm called RANSAC is employed. RANSAC makes random guesses about the position of the line and returns the best estimate after a certain stopping criteria. Many algorithms involve trying to solve the problem with brute force.

1.1. Related Work

Hough Transform: A commonly used method to extract line information in 2D and 3D space is by using Hough transformations. One such paper demonstrates an algorithm that is able to efficiently extract 3D lines using the aforementioned transformations (Dalitz et al., 2017). On a high level, Hough transformations quantize all possible parameters that describe a line in multi-dimensional space into discrete bins and generate a histogram to determine which subset of parameters has the highest occupancy. The results then undergo thresholding in order to determine the most successful parameters that encompass the most points in the space and lines are extracted using those parameters. The benefits to using this method are that the number of lines does have to be initially known, multiple line seg-

ments are able to be distinguished within a single line, and also it is able to reject clusters that do not represent lines. However, downsides include large computational resource requirements for higher dimensions such as 3D spaces, and feature lines must be extracted for every successive frame in a dataset with many continuous frames. Furthermore, Hough Transform is not able to detect line segments. It is only able to detect infinite lines represented by a direction and a positional offset. This is problematic since the ability to detect the length of the lines proves important to identifying what type of object is being seen.

1.2. Proposed Solution

The proposed method is to extract line features from 3D line data using a method similar to K-means clustering. Using Loyd's algorithm, clusters are determined using the line of best fit, which would be updated in every subsequent iteration until convergence. Using clustering has a few benefits. First, clustering is a $O(k * n * i)$ algorithm where k is the number of clusters, n is the number of points in the space and i is the max number of iterations allowed. In comparison, Hough transformations are $O(h^d * n)$ where h is the number of bins of each parameter spanning the entire space, d is the number of dimensions and n is the number of points in the point cloud. Hence, clustering is an improvement in cases where the number of dimensions can be very high in comparison to Hough transforms which have substantial memory requirements in order to generate the large matrices required for multidimensional histogram computation. However, using simple clustering comes with a few drawbacks. K-means clustering requires a known number of clusters to compute. It also considers all the points in one cluster or another which could cause errors due to outliers from noise and might lead to false detections. Catering K-means to purely detect lines might cause overfitting, where it is unable to reject clusters that do not represent lines in 3D space.

To circumvent these issues, the following 3 assumptions are made about the point cloud data:

- The data should come from a LiDAR with known parameters and intrinsics. The variance of such data is given or can be experimentally determined
- Data in the point cloud represents real objects so they can only form a line or a surface. Each line segment has a continuous line of points in them.
- The LiDAR used has high enough resolution that any potential lines of interest should have a high density of points.

Using the first assumption, it is possible to reject outliers if a probability threshold *p* is exceeded, and this threshold can be heuristically determined. This has the effect of creating an inclusion cylinder around each line of best fit that only considers particular points close enough to be in the cluster. Then using the second assumption, it should be possible to determine the length of each line segment and split up clusters into multiple line segments if the points within it do not fit the distribution of a line, which can be determined using the variance of points in the cluster. The third assumption provides the possibility of destroying clusters that cannot gather enough points as via the assumption they most likely do not represent a line. An interesting by-product of these assumptions is the ability to expand and grow the number of clusters/lines. Since it is possible for a point to be considered part of no clusters, after a set amount of iterations, if there is a significant number of points that are not part of any cluster, they can all be considered a part of one new cluster, by which a new line segment can be generated and can be broken down and filtered using the previously mentioned processes.

2. Line Detection Using K-Lines

The general procedure to detect lines follows the typical Loyd's algorithm method of fitting cluster means to the dataset. Much like Loyd's algorithm, the algorithm first attempts to segment the point clouds based on distance to each estimated line. Then it recomputes the line by generating a line of best fit using the new point cloud. The line of best fit generation is the step that utilizes the concept of cost to make improvements. The algorithm then deviates from Loyd's algorithm by introducing 3 new steps, namely segmentation, pruning and unused point recycling. Much like K-means clustering, this algorithm is iterative and the final result improves as the number of iterations increases. The overarching algorithm can be found in Algorithm 1.

Algorithm 1 Line Detection through K-Lines

Input: point cloud p, training iterations $iter$, number of lines num_lines, variance of LiDAR points v, probability of inclusion $prob$
repeat
  classifier fit
  for $i = 0$ **to** $iter - 1$ **do**
    Classify all points to closest cluster
    Update line of best fit
    Segment clusters that meet criteria
    if Lines are settled and unused points $> \%$ **then**
      Create line from unused points
      Classify all points to closest cluster
      Segment clusters that meet criteria
    end if
    Prune similar lines
  end for
until Iterations $iter$ have been completed

2.1. Cluster Generation

Point clusters are defined by a prototype which is modeled as a line segment for this algorithm. The line segment is simply a line of best fit with start and end points determined

by the spread of the point cloud. The algorithm maintains multiple lines and associates a point cloud with each line.

Lines are initialized using the given initial number of lines and 2 random points chosen within the bounds of the point cloud. Similar to K-means clustering, all points that are deemed close enough to these initial lines are added to its cluster. The metric for how close a point is to a cluster would be determined by the inclusion criteria.

2.1.1. Distance Computation

This line detection algorithm is built to detect line segments, thus a distance metric is needed to determine the distance from a point to a line segment. Figure 2 shows the 3 possibilities of determining the distance from a point to a line segment. In the case of p_1 or p_3, when the point is outside of the line, the distance to the line segment is to the closest endpoint. In the case of p_2 when the point is within the bounds of the line, the closest distance is the orthogonal line to the segment. An algorithm is incorporated into the distance calculation function to determine if the point lies on the inside or outside of the line segment. The distance is calculated as orthogonal distance if it is inside the line segment. However, in the remaining cases, it is calculated as half of the minimum distance to the endpoint. This is to promote exploration by allowing the cluster to expand towards points outside the line's bounds.

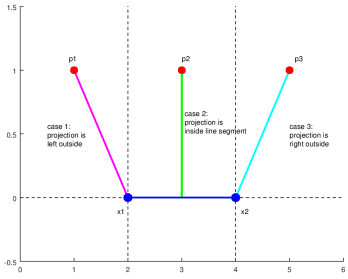


Figure 2. A Line and 3 possible cases for distance computation

2.1.2. Inclusion Criteria

To determine if a point is included in a line cluster, the variance and probability metrics are used in addition to distance. Using the methods described in section Section 2.1.1, the distance from each line can be computed for every point in the point cloud. The minimum distance is taken and that point would be considered a candidate for the line that it is closest to. However, the point is only added to the cluster if it fulfills the inclusion criteria. For each line, the algorithm assumes that LiDAR scans are generated in a Gaussian distribution around the line it detects. Then given the distance and expected variance, it is possible to compute the probability of a particular point belonging to a line and its cluster. A probability threshold is applied to the distance and the point is added to the cluster if the point has a high enough probability. Otherwise it is considered unused.

2.2. Update Step

The update step involves taking each updated cluster and their new point cloud, and generating the line of best fit for those points. The algorithm assumes any real line will have a sufficient number of points representing it. Further in this step, if any cluster is deemed to have an insufficient number of points, they are discarded as those lines could be attributed to random sensor noise. The threshold for the minimum number of points is set to an arbitrary value.

2.3. Segmentation

During the fitting process, especially when new lines are generated by random sampling, it is possible that the singular line estimate bridges multiple clusters. This would be undesirable since by using the naive K-means clustering approach, there would be no way to correct for this mistake. However, utilizing the assumption that the points represent continuous lines and that points are generated with Gaussian distribution, it is possible to break up a particular line into smaller segments that better occupy the individual subclusters. Each point in the cluster's point cloud is projected onto the line as one dimensional point along its direction. The algorithm then looks for gaps in the projected points that are much larger than the average distance of each point in the cluster and for points that display much higher variance than the initialised value. The algorithm then segments the line at these points since they likely represent separate clusters.

2.4. Pruning

During the clustering and segmentation, a side-effect could occur where clusters occupy points in close proximity, and hence form lines that are very similar in direction, starting, and ending points. Hence, an important step that needs to occur is pruning so that lines with overlapping properties do not continue to be generated and iterated upon.

The metrics for pruning are defined as earlier stated, to be the direction, starting, and ending points. If clusters have lines with these properties within a set tolerance of each other, a single cluster would be chosen and inherit the points of all clusters being deleted. Then a new line of best fit would be generated and updated using the accumulated point cloud for the chosen cluster.

2.5. Unused Point Recycling

Once an iteration has occurred using the functions laid out so far, the points that have so far not been assigned to any cluster, can now be processed using the assumption laid out in Section 1.2. A check is first performed to see if the lines have settled. This check is done by checking if the number of points in each line's point cloud has changed at all. If there is no change between the size of any of the point clouds, the algorithm assumes the clusters has converged. At this point, if the percentage of unused points exceeds a certain limit (to be determined through testing), then a new cluster and line can be generated based on the unused points and added to the cluster list using the initialization strategy as specified in Section 2.1.

3. Evaluation and Experimental Results

3.1. Evaluation Process

Testing: The proposed algorithm was tested using synthetic data generated with 3D ground truth lines representing various lines in different challenging settings. Gaussian noise was added to the position of these points in order to simulate the accuracy of a typical LiDAR while also incorporating erroneous values within the bounds of the point cloud, so as to simulate random noise/artifacts. Real LiDAR data was not used due to difficulties associated with acquiring and parsing such data.

Three experiments were conducted to verify the robustness and performance.

3.1.1. Test 1

The first test set consists of 4 simplistic vertical lines, positioned equidistant from each other, as shown in Figure 3. This was chosen as such to provide a straightforward case for detection while being able to test its performance with noisy data. The noise metric used to generate the test was 1, meaning for each true LiDAR data that represents a line there is one point of false noisy data generated. In practice, LiDARs will never generate data that is this noisy. However, it serves as a good indicator that this algorithm is able to filter out noise in the point cloud.

3.1.2. Test 2

The second test consists of various lines of different orientations and lengths, in close proximity. This test also increases the variance of the LiDAR points generated to 0.5. There are 2 lines that intersect each other, as shown in Figure 4. The purpose of this test is to evaluate performance against vastly different line types, while also assessing its ability to classify intersecting lines correctly when the LiDAR points themselves are highly variable.

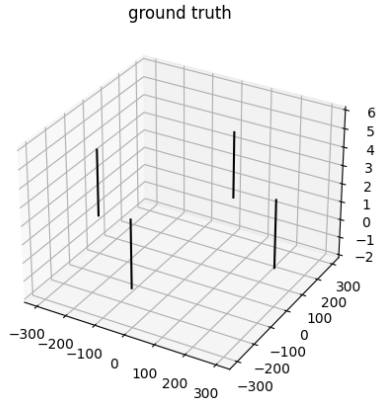


Figure 3. Ground Truth. Test 2

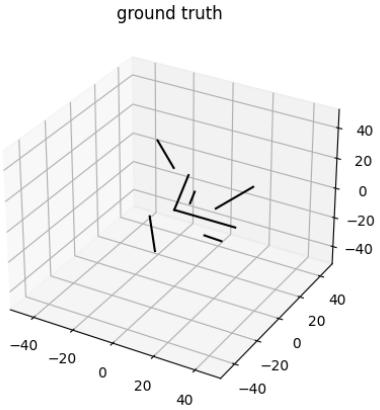


Figure 4. Ground Truth. Test 2

3.1.3. Test 3

Test 3 simulates the types of lines that an autonomous vehicle might see when it is performing self-driving tasks on the road. Here, the lines are generated by simulating poles-like objects on a typical north American street. The 9 rows of vertical lines represent light poles. they are approximately 12ft tall and 30ft apart. This adheres to the recommendations of the Global Designing Cities Initiative specified by (Initiative). smaller poles representing shorter pole-like objects such as fire hydrants were also included in addition to taller objects such as billboards. Overall this test generates data that represents pole-like data relatively well. The performance on this test can reflect how the algorithm will perform on real LiDAR data gathered from driving.

3.2. Results

These situations were then tested on the algorithm, implemented in python. For the purposes of evaluating the perfor-

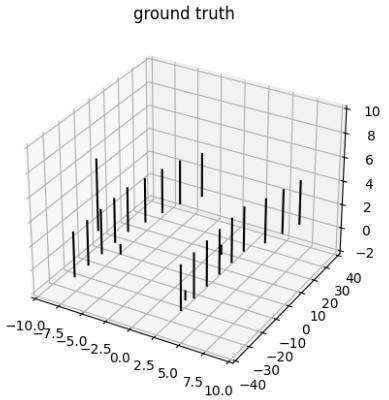


Figure 5. Ground Truth. Test 3

mance, the same tests were applied to a Hough transform platform available online (Dalitz et al., 2017). Results for each test can be found in Figure 6, Figure 7 and Figure 8. Note that the generated lines with point cloud images color code the point cloud using blue points for points that are included as part of a line cluster and black points for points that are unused. From this, it is easy to see which lines were not detected by the algorithm.

In both test 1 and test 2, K-Lines Clustering was able to detect all the lines while rejecting the noise. In test 1, the lines are not as accurate in length due to the noise. When compared to Hough Transform, K-Lines Clustering was able to detect lines better than Hough transform in a shorter amount of time. This is because Hough Transform does not do well when the search space is larger and is likely more responsive to the high amounts of noise in this dataset. In test 2, it is clear that Hough transform is also able to perform well. When running K-Lines Clustering, it is sometimes possible to miss detection on smaller lines since the number of points that represent them is too few to trigger the unused points recycling algorithm.

The most interesting test is test 3. While Hough transform was unable to detect most of the lines accurately, K-Lines Clustering was able to detect most of the lines. The longer the lines, the more accurate the detection was. However, due to the large number of points existing in the point cloud, K-Lines clustering was very slow. This shows that although K-Lines Clustering is able to classify viable lines for autonomous car driving data, it struggles to do so in real-time.

3.3. Discussion

From the three tests performed it is clear that K-Lines clustering performs well given noisy point cloud data. It is able to effectively reject random noise and classify lines to a high degree of accuracy given LiDAR points with Gaussian noise.

Table 1. K-Line results results for all tests, for 30 iterations

DATA SET	RUN TIME (S)	ALL LINES DETECTED
TEST 1	7.99	√
TEST 2	15.64	√
TEST 3	162.34	×

One thing to note is that when compared to the Hough transform, K-Lines clustering scales with the number of points present in the point cloud as well as the number of lines it needs to maintain while Hough Transforms scales with the dimensions and size of the search space. Thus, K-Lines clustering will perform more efficiently if lines are far spread apart and relatively short. This is one of the downsides as typically for applications with LiDAR data, the range of the LiDAR is within 100 meters, At this range, lines are expected to be dense. Although it has high accuracy for dense point cloud data, it is unable to do so with comparable speed to other line detection methods.

Another note is the variance. From testing, since the clustering takes variance into account, increasing variance does not significantly reduce the accuracy of the line detections except at high values where line point clouds start merging into one another. However, this phenomenon is unlikely to happen given the accuracy of modern LiDARs.

The three main hyper-parameters that the user needs to input are the probability threshold, initial lines, and the number of iterations.

One of the hyper-parameters that the user needs to input is the probability threshold. The probability threshold across all three tests referenced in Section 3 was set to 0.975%. This means a point is only discarded if the likelihood of it being a part of the line is less than 0.025%. Through experimentation, it is determined that a high percentage generally allowed the algorithm to perform better. However, this empirical data was gathered assuming there were no objects other than lines in the 3D space. This high threshold may perform worse if surfaces were also included.

When performing the experiments, the number of initial lines was also modulated to see how the algorithm would behave based on the number of random initial lines. This was proven to not be a significant factor, due to the fact that the algorithm can spawn and destroy lines each iteration. However, when the number of initial guesses are high, the computational expense is increased but will have a higher chance at detecting all the lines in the search space. When the number of initial lines is lower, K-Lines clustering will be significantly faster at the cost of missing line detections. Overall, although not crucial, it is important to make good initial line guesses for performance reasons.

Finally, the iterations also vary depending on the type of test. For the test in Figure 6, the algorithm is able to converge to accurate lines with 1-2 iterations and with growing complexity and number of lines, more iterations are required to find all possible lines as in Figure 7 and Figure 8.

4. Conclusions and Next Steps

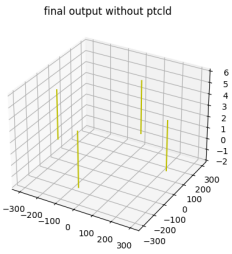
Although the proposed algorithm presents more accurate results when compared to Hough transforms, it performs much slower compared to it and other available neural-net based 3D object detection. However, performance with consecutive frames at the moment is untested, and could be an avenue to explore. Further, due to the earlier made assumptions, detection in situations with multiple surfaces in addition to lines is unknown. Another possibility to explore is implementing the algorithm in a compiled language such as C ++ to see larger computational benefits.

References

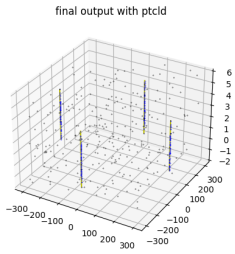
Dalitz, C., Schramke, T., and Jeltsch, M. Iterative Hough Transform for Line Detection in 3D Point Clouds. *Image Processing On Line*, 7:184–196, 2017. https://doi.org/10.5201/ipol.2017.208.

Initiative, G. D. C. Lighting Design Guidance. URL https://globaldesigningcities.org/publication/global-street-design-guide/utilities-and-infrastructure/lighting-and-technology/lighting-design-guidance/. publisher:.

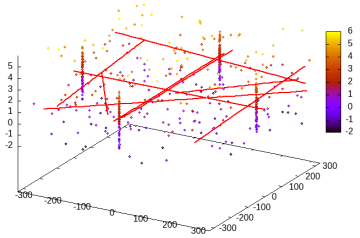
Kampker, A., Hatzenbuehler, J., Klein, L., Sefati, M., Kreiskoether, K. D., and Gert, D. Concept study for vehicle self-localization using neural networks for detection of pole-like landmarks. In Strand, M., Dillmann, R., Menegatti, E., and Ghidoni, S. (eds.), *Intelligent Autonomous Systems 15*, pp. 689–705, Cham, 2019. Springer International Publishing. ISBN 978-3-030-01370-7.



(a) K-Lines. Test 1. Generated Lines

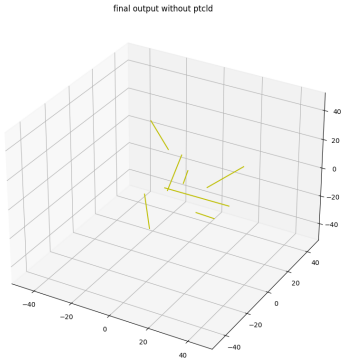


(b) K-Lines. Test 1. Generated Lines with point cloud

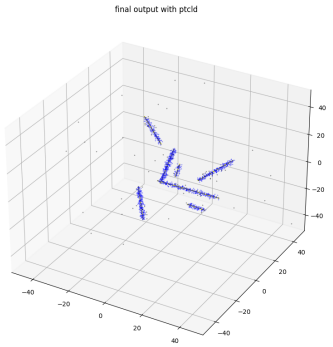


(c) Hough Transform. Test 1

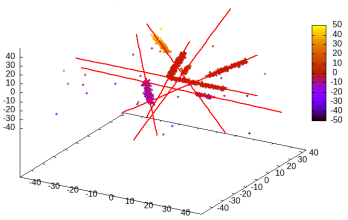
Figure 6. Results and comparison for Test 1



(a) K-Lines. Test 2. Generated Lines

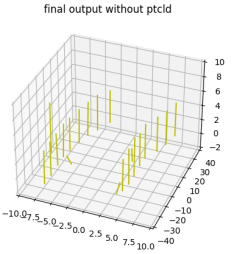


(b) K-Lines. Test 2. Generated Lines with point cloud

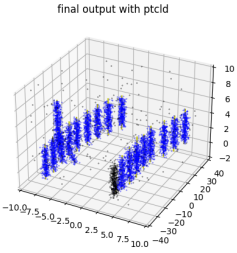


(c) Hough Transform. Test 2

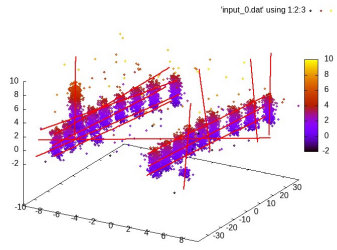
Figure 7. Results and comparison for Test 2



(a) K-Lines. Test 3. Generated Lines



(b) K-Lines. Test 3. Generated Lines with point cloud



(c) Hough Transform. Test 3

Figure 8. Results and comparison for Test 3