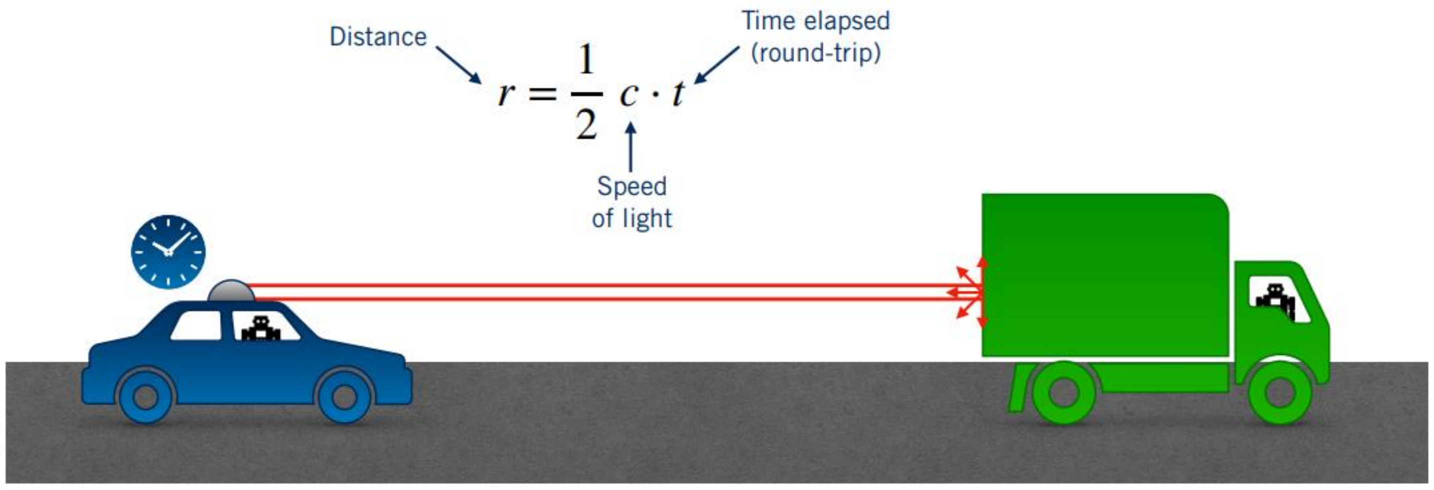
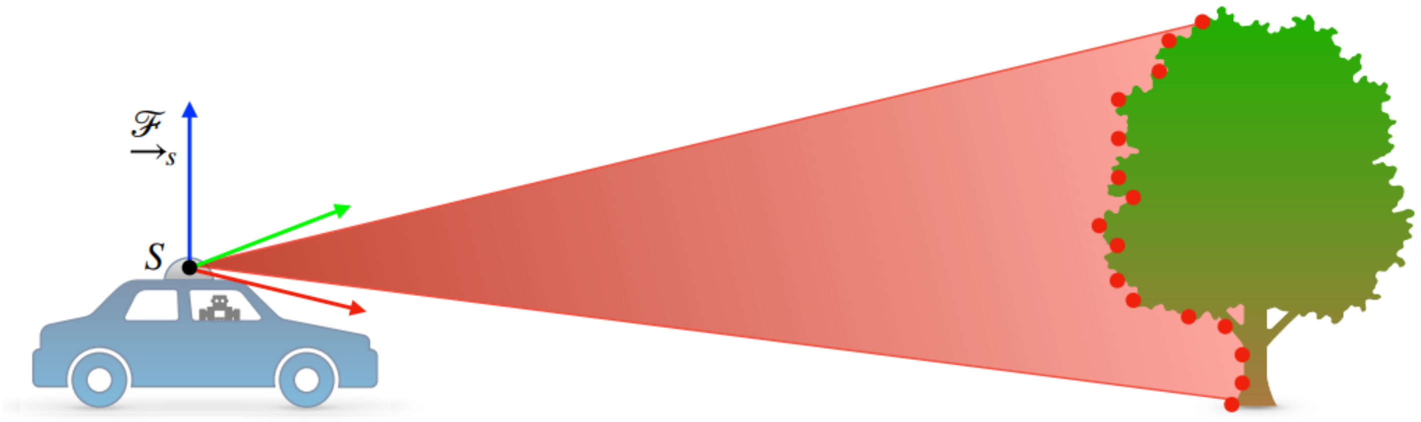
LIDAR sensor

Lidar (Light Detection And Ranging) sensing gives us high resolution data by sending out thousands of laser signals. These lasers bounce off objects, returning to the sensor where we can then determine how far away objects are by timing how long it takes for the signal to return. Also, we can tell a little bit about the object that was hit by measuring the intensity of the returned signal.

- How does LIDAR work?
  The laser sends out a very short pulse, few nano seconds, and then measures the time it takes for the pulse to go to the object and back to the LIDAR where it's detected, this time gives us the distance to that object.



You scan this laser beams across the field of view, which means you get for each point in the field of view.

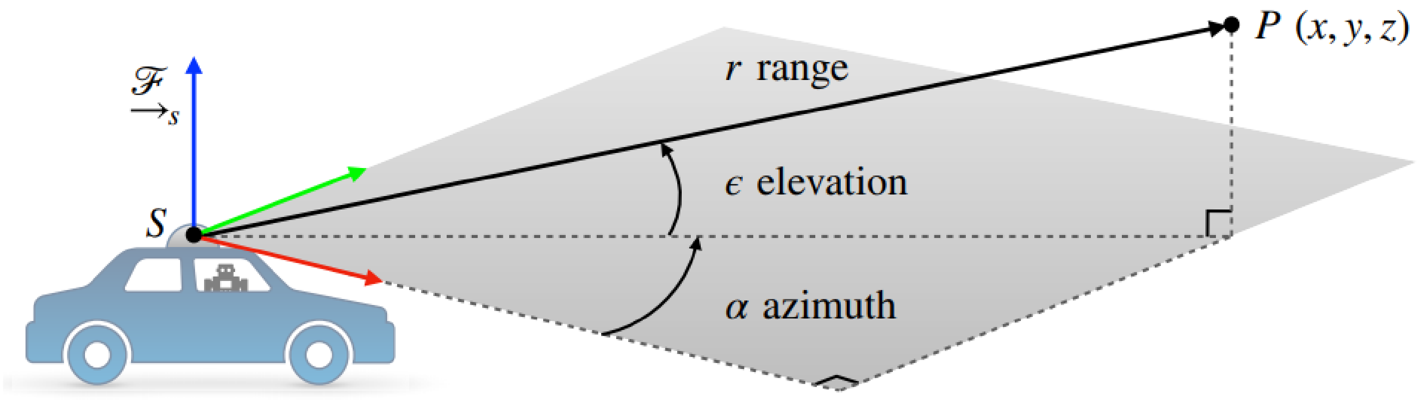


Let's dive into how lidar data is stored. Lidar data is stored in a format called Point Cloud Data (PCD for short). A .pcd file is a list of (x,y,z) cartesian coordinates along with intensity values (i), it's a single snapshot of the environment, so after a single scan.

LIDAR has been an enabling technology for self-driving cars because it can see in all directions and is able to provide very accurate range information. In fact, with few exceptions, most self-driving cars on the road today are equipped with some type of LIDAR sensor.

1 LIDAR coordinate system

The coordinate system for point cloud data is the same as the car's local coordinate system. In this coordinate system the x axis is pointing towards the front of the car, and the y axis is pointing to the left of the car. Also since this coordinate system is right-handed the z axis points up above the car.



$$range(r) = \frac{t}{2} * c$$

$$t: \ time\ a\ laser\ signal\ takes\ to\ be\ emitted\ and\ received\ again$$

$$c: the\ speed\ of\ light$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r\cos(\alpha)\cos(\epsilon) \\ r\sin(\alpha)\cos(\epsilon) \\ r\ sin(\epsilon) \end{bmatrix}$$

QUESTION

While scanning with a VLP 64, a laser signal from the top layer takes 66.7 ns to be emitted and received again. The laser is emitted at a -24.8 degree incline from the X axis and horizontally travels along the X axis. Knowing that the speed of light is 299792458 m/s, what would be the coordinates of this laser point (X,Y,Z) in meters?

$$r = \frac{t}{2}c = \frac{66.7 * 10^{-9}}{2} * 299792458 = 10\ m$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 * \cos(0)\cos(-24.8) \\ 10 * \sin(0)\cos(-24.8) \\ 10 * \ sin(-24.8) \end{bmatrix} = \begin{bmatrix} 9.08 \\ 0 \\ -4.19 \end{bmatrix} = (9.08, 0, -4.19)m$$

2 The Point Cloud Library (PCL)

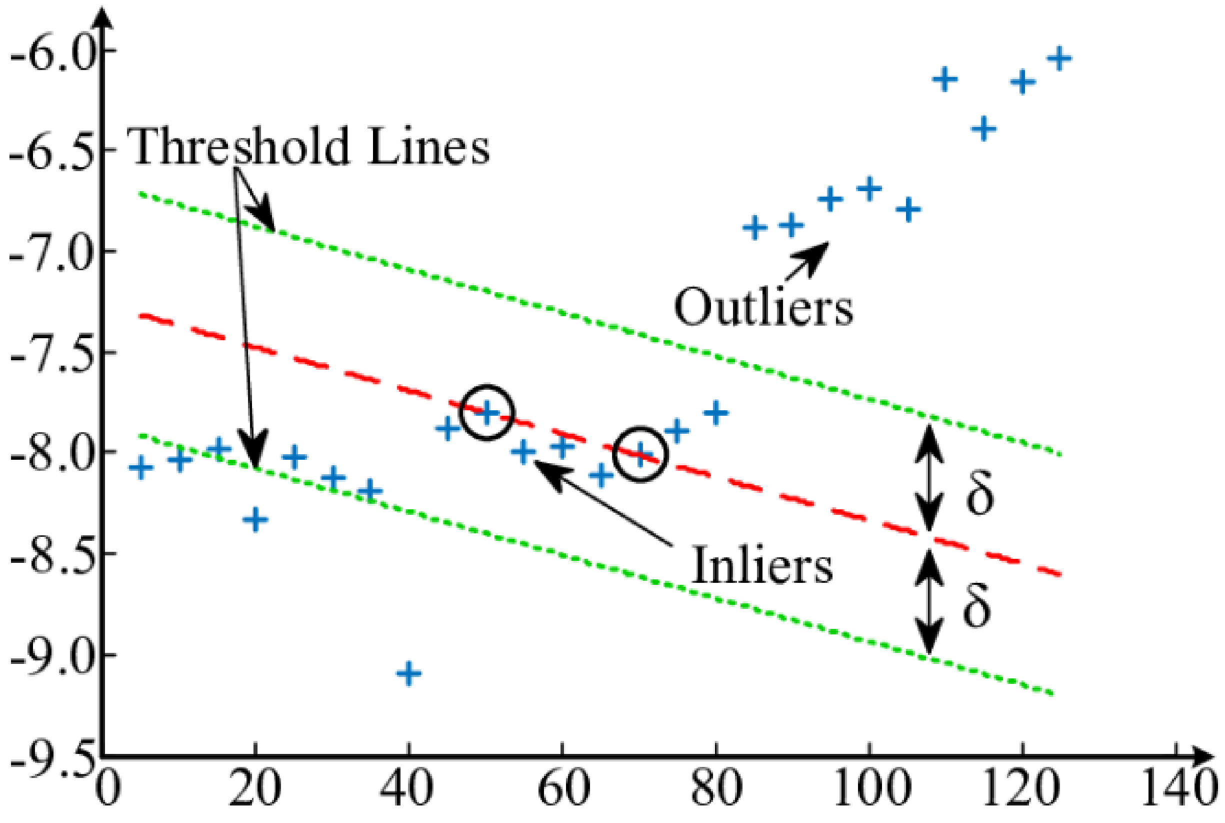
- Open-source Point Cloud Library (PCL) has many useful functions for doing basic and advanced operations on point clouds in C++.
- Widely used in industry.
- Unofficial Python bindings exist.

3 Point Cloud Segmentation

We want to be able to locate obstacles in the scene. However, some objects in our scene are not obstacles. What would be objects that appear in the pcd but are not obstacles? For the most part, any free space on the road is not an obstacle, and if the road is flat it's fairly straightforward to pick out road points from non-road points. To do this we will use a method called Planar Segmentation which uses the RANSAC (random sample consensus) algorithm.

3.1 RANSAC

RANSAC stands for Random Sample Consensus, and is a method for detecting outliers in data. RANSAC runs for a max number of iterations, and returns the model with the best fit. Each iteration randomly picks a subsample of the data and fits a model through it, such as a line or a plane. Then the iteration with the highest number of inliers or the lowest noise is used as the best model.



One type of RANSAC version selects the smallest possible subset of points to fit. For a line, that would be two points, and for a plane three points. Then the number of inliers are counted, by iterating through every remaining point and calculating its distance to the model. The points that are within a certain distance to the model are counted as inliers. The iteration that has the highest number of inliers is then the best model.

3.1.1 Implementing RANSAC for Lines

You want to be able to identify which points belong to the line that was originally generated and which points are outliers. To do this you will randomly sample two points from the cloud and fit a line between the points.

Equation of a Line Through Two Points in 2D

For variables x and y and coefficients A, B, and C, the general equation of a line is:

$$Ax + By + C = 0$$

Given two points: point1 (x1, y1) and point2 (x2, y2), the line through point1 and point2 has the specific form:

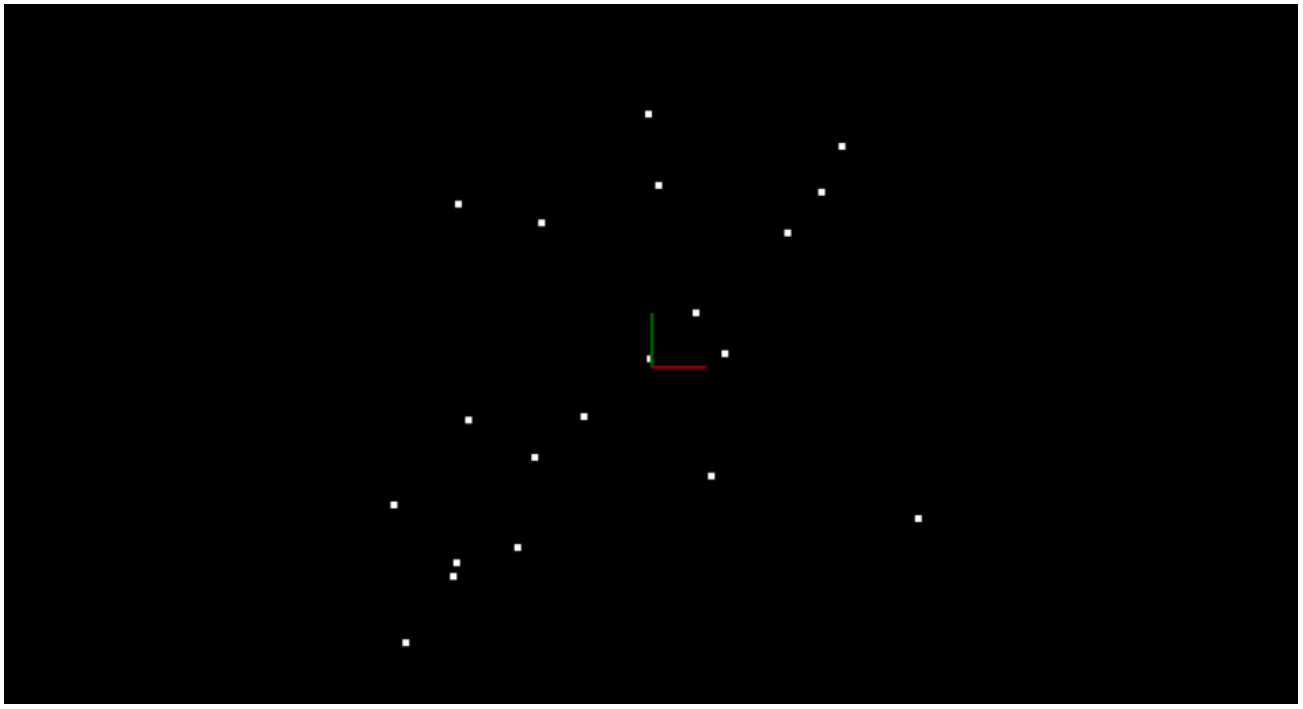
$$(y_1 - y_2)x + (x_2 - x_1)y + (x_1 * y_2 - x_2 * y_1) = 0$$

Using Distance to Find the Best Line

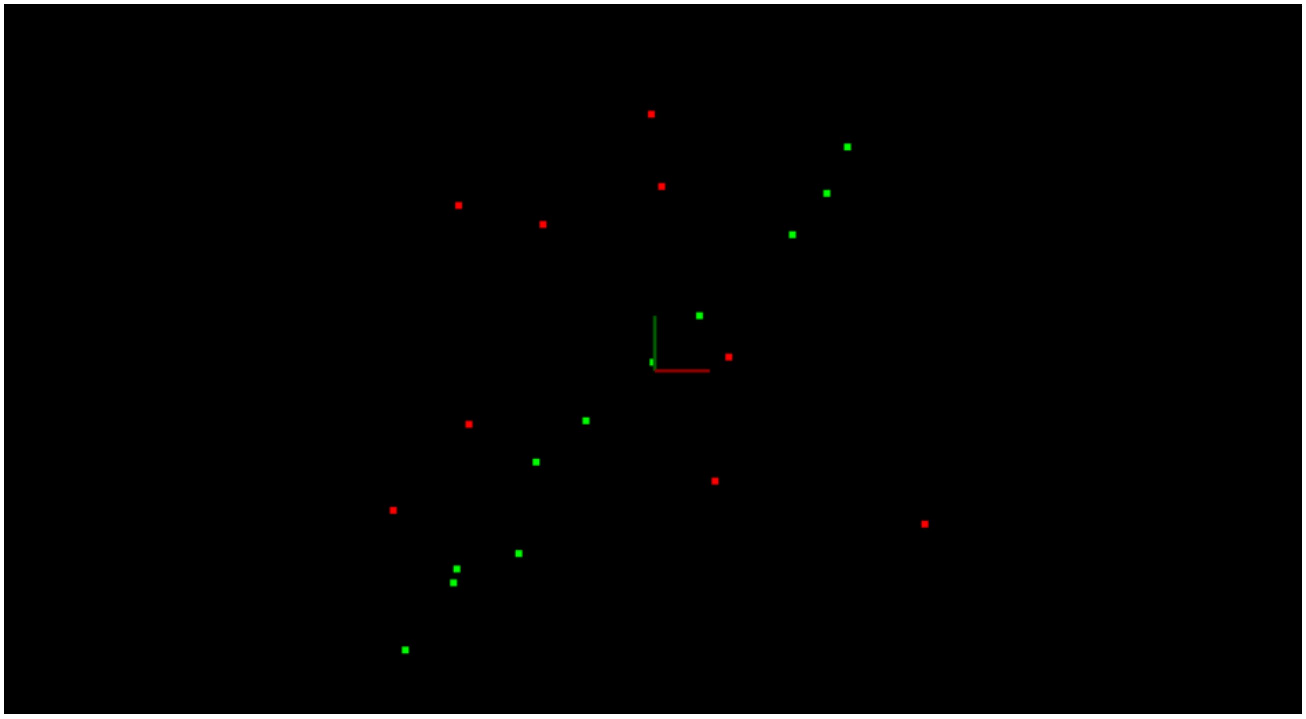
After fitting the line, you can then iterate through all the points and determine if they are inliers by measuring how far away each point is from the line. You can do this for each iteration keeping track of which fitted line had the highest number of inliers. The line with the most inliers will be the best model. The equation for calculating distance between a point and line is shown below.

$$d = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}}$$

This is simple 2D point cloud data



Below are the results of doing RANSAC to fit a line from the data above. Inliers are green while outliers are red. The function had a max iteration count of 50 and a distance tolerance of 0.5. The max iteration size to run depends on the ratio of inliers to the total number of points. The more inliers our data contains the higher the probability of selecting inliers to fit the line to, and the fewer iterations you need to get a high probability of selecting a good model.



3.1.2 Extending RANSAC to Planes

Now that you are getting the hang of RANSACing it, and understanding RANSAC for fitting a line, you can do the same thing for fitting a plane in a 3D point cloud.

Equation of a Plane through Three Points

$$Ax + By + Cz + D = 0$$

For

- point1 = (x1, y1, z1)
- point2 = (x2, y2, z2)
- point3 = (x3, y3, z3)

Use point1 as a reference and define two vectors on the plane v1 and v2 as follows:

- Vector v1 travels from point1 to point2.
- Vector v2 travels from point1 to point3.

$$v1 = < x2 - x1, y2 - y1, z2 - z1 >$$
$$v2 = < x3 - x1, y3 - y1, z3 - z1 >$$

Find normal vector to the plane by taking cross product of $v1 \times v2$:

$$v1 \times v2 = < (y2-y1)(z3-z1)-(z2-z1)(y3-y1),$$

$$(z2-z1)(x3-x1)-(x2-x1)(z3-z1),$$

$$(x2-x1)(y3-y1)-(y2-y1)(x3-x1) >$$

To simplify notation, we can write it in the form:

$$v1 \times v2 = < i, j, k >$$

then,

$$i(x - x1) + j(y - y1) + k(z - z1) = 0,$$
$$ix + jy + kz - (ix1 + jy1 + kz1) = 0$$

$$A = i,$$
$$B = j,$$
$$C = k,$$
$$D = -(ix1 + jy1 + kz1)$$

Distance between point and plane

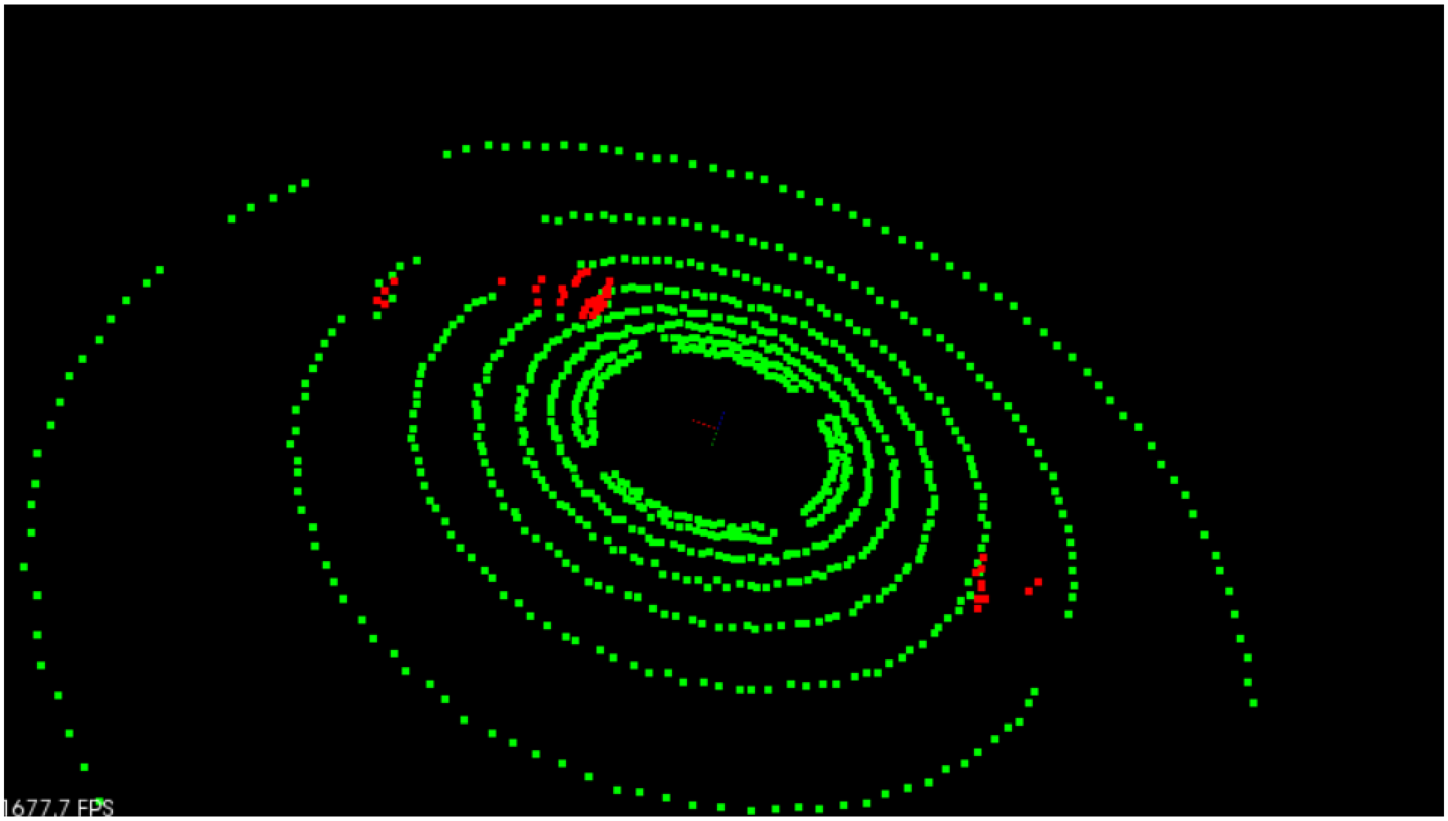
If the plane is

$$\mathrm{Ax + By + Cz + D = 0}$$

then given a point (x, y, z), the distance from the point to the plane is:

$$d = \frac{|Ax + By + Cz + D|}{\sqrt{A^2 + B^2 + C^2}}$$

This image shows: Segment and separating point clouds(road points in green, and other obstacle points in red).



4 Clustering Obstacles

If you want to do multiple object tracking with cars, pedestrians, and bicyclists, for instance. One way to do that grouping and cluster point cloud data is called Euclidean clustering.

4.1 Euclidean Clustering

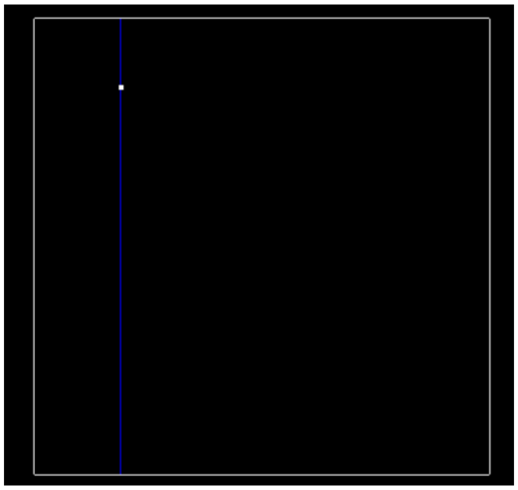
The idea is you associate groups of points by how close together they are. To do a nearest neighbor search efficiently, you use a KD-Tree data structure which, on average, speeds up your look up time from O(n) to O(log(n)). This is because the tree allows you to better break up your search space. By grouping points into regions in a KD-Tree, you can avoid calculating distance for possibly thousands of points just because you know they are not even considered in a close enough region.

4.1.1 Implementing KD-Tree

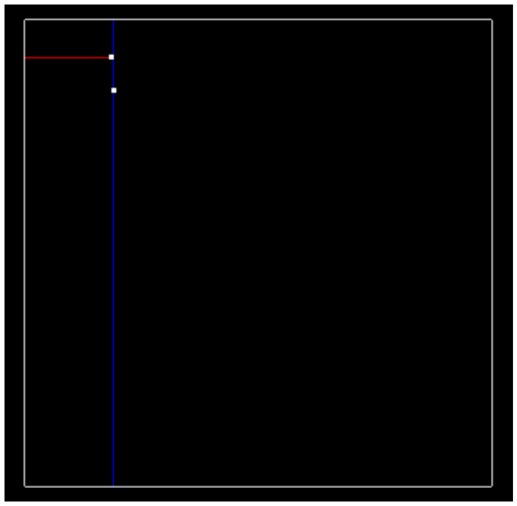
A KD-Tree is a binary tree that splits points between alternating axes. By separating space by splitting regions, nearest neighbor search can be made much faster when using an algorithm like Euclidean clustering.

4.1.1.1 Inserting Points into KD-Tree

Now let's talk about how exactly the tree is created. At the very beginning when the tree is empty, root is NULL. The point inserted becomes the root, and splits the x region. Here is what this visually looks like, after inserting the first point (-6.2, 7).

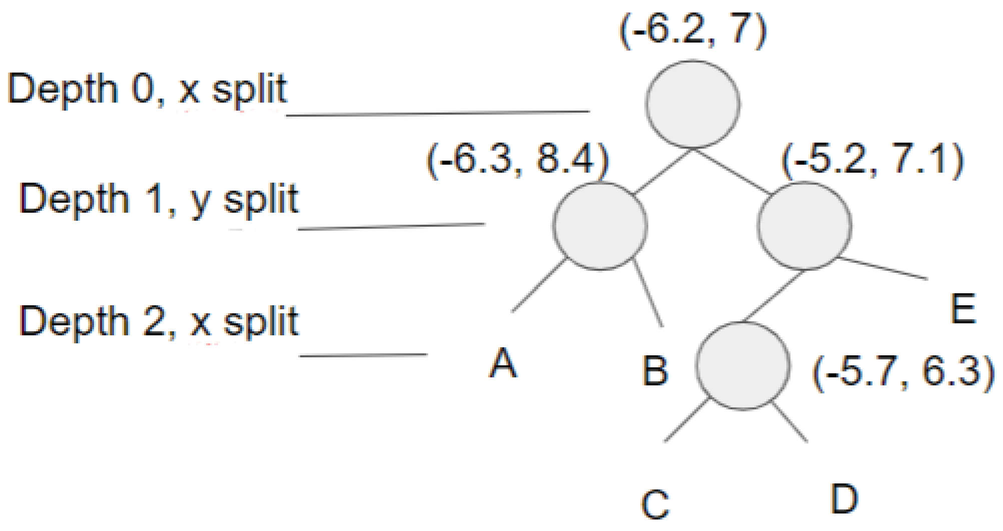


The next point is (-6.3, 8.4). Since we previously split in the x-dimension, and -6.3 is less than -6.2. This Node will be created and be a part of root's left node. The point (-6.3, 8.4) will split the region in the y dimension.



To recap, the root was at depth 0, and split the x region. The next point became the left child of root and had a depth of 1, and split the y region.

Then here is what the tree looks like after inserting two more points (-5.2, 7.1) and (-5.7, 6.3), and having another x split division from point (-5.7, 6.3). The tree is now at depth 2.



QUESTION

Which node should the point (7.2, 6.1) be inserted to?

Depth 0: we look at x => 7.2 > -6.2   then, go to right

Depth 1: we look at y => 6.1 < 7.1   then, go to left

Depth 2: we look at x => 7.2 > -5.7   then, go to right

Then, the node D (7.2, 6.1).

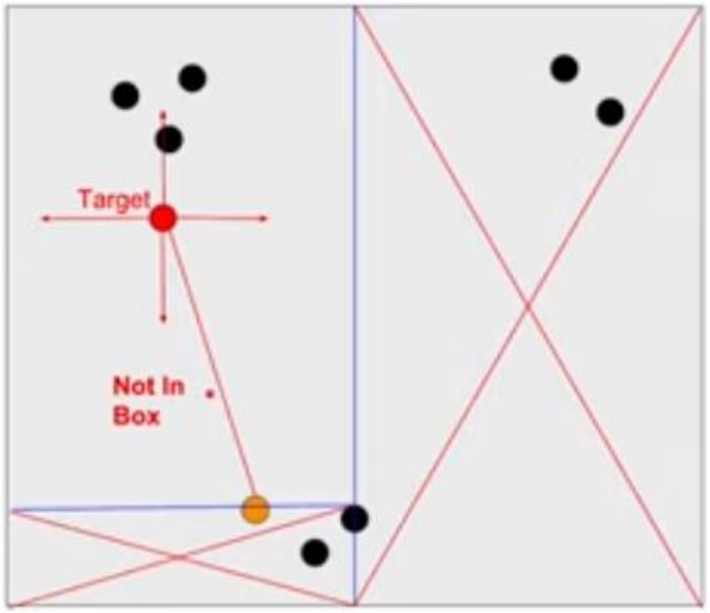
Having a balanced tree that evenly splits regions improves the search time for finding points later. To improve the tree, insert points that alternate between splitting the x region and the y region evenly. To do this pick the median of sorted x and y points. For instance, if you are inserting the first four points that we used above (-6.3, 8.4), (-6.2, 7), (-5.2, 7.1), (-5.7, 6.3) we would first insert (-5.2,7.1) since it is the median along the x axis. If there is an even number of elements the lower median is chosen. The next point to be inserted would be (-6.2, 7), the median of the three points for y. This would be followed by (-5.7,6.3) the lower median between the two for x, and then finally (-6.3,8.4). This ordering will allow the tree to more evenly split the region space and improve search time later.

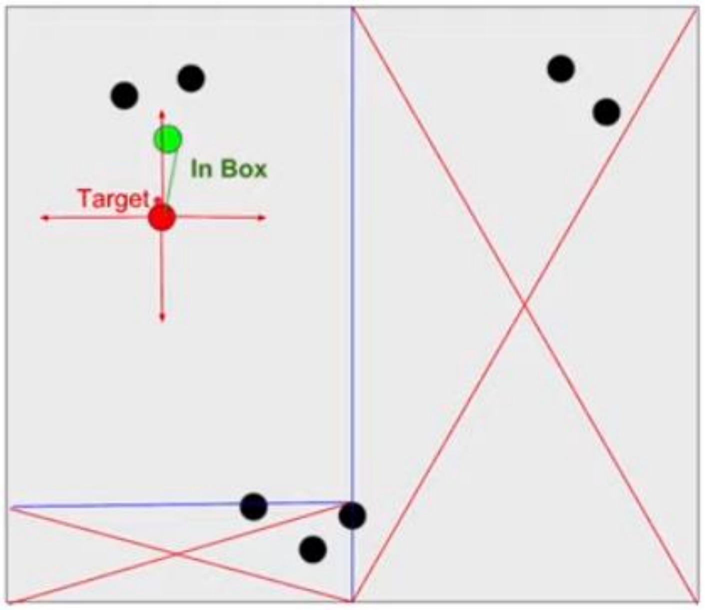
4.1.1.2 Searching Points in a KD-Tree

Once points are able to be inserted into the tree, the next step is being able to search for nearby points inside the tree compared to a given target point. Points within a distance of distance Tolerance are considered to be nearby. The KD-Tree is able to split regions and allows certain regions to be completely ruled out, speeding up the process of finding nearby neighbors.

The naive approach of finding nearby neighbors is to go through every single point in the tree and compare their distances with the target, selecting point indices that fall within the distance tolerance of the target.

If the current node point is within this box then you can directly calculate the distance and see if the point id should be added to the list of nearby ids. Then you see if your box crosses over the node division region and if it does compare that next node. You do this recursively, with the advantage being that if the box region is not inside some division region you completely skip that branch.





4.1.2 Implementing Euclidean Clustering

Once the KD-Tree method for searching for nearby points is implemented, it's not difficult to implement a Euclidean clustering method that groups individual cluster indices based on their proximity.

Clusters shown in different colors, red, green, and blue.