Forward Perception Using a 2D LiDAR on the Highway for Intelligent Transportation

by

Eric N Willcox III

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

by

April 2016

APPROVED:

Professor Xinming Huang, Major Thesis Advisor

Professor John McNeill, Associate Department Head

Abstract

For a little over the past decade since the DARPA Grand Challenge in 2004 and the more successful Urban Challenge in 2007 autonomous vehicles have seen a surge in popularity with car manufacturers, and companies such as Google and Uber. Light Detection And Ranging (LiDAR) has been one of the major sensors in use to sense for acting on the surrounding environment instead of the classic radar which has a much narrower field of vision. However the cost of the higher end 3D LiDAR systems which started seeing use during the DARPA challenges still have the high cost of $70,000 a piece which is an issue when trying to design a consumer friendly system on a family car.

This work aims to investigate alternate 2D LiDAR systems to the costly systems currently in use in many prototypes to find a cost efficient alternative that can detect and track obstacles in front of a vehicle.

The introduction begins by summarizing some related prior works, particularly papers from after the Grand Challenge as well as some about the competition itself. Detection and tracking methods for point clouds generated by the LiDAR are explored including ways to search through the data in an efficient manner to meet real-time constraints. Some of the trade-offs in going from a 3D system to a 2D system and examined along with how some of the drawbacks can be mitigated.

Acknowledgments

I'd like to thank Professor Huang for allowing me to join the intelligent transportation team and giving me a large amount of freedom to work on something of my choosing. During the course of the project he also suggested a lot of good ideas to support my research. Many thanks to the others in the intelligent transportation group for suggesting ideas during our meetings and to Roy Sun for attaching all the sensors to his car so we can collect data.

Finally, thanks to my friends and family for the support over the last couple of years while I finished up my education. Including Thanacha Choopojcharoen for helping with ideas and radar data collection as well as Joe Rodriguez for operating my laptop while I drove my car during the early stages of data collection.

Contents

1 Background **1**

1.1 LiDAR . . . 1

1.1.1 Continuous waveform . . . 3

2 Introduction **5**

2.1 Motivation . . . 5

2.2 LiDAR vs radar . . . 7

2.3 Prior work on autonomous navigation . . . 9

2.4 LiDAR chosen . . . 11

3 Mapping with LiDAR **12**

3.1 Filtering . . . 13

3.1.1 Pass through filter . . . 13

3.1.2 Radius outlier removal . . . 14

3.2 Segmentation . . . 14

3.2.1 k-d tree . . . 16

3.2.2 Calculating normals . . . 19

3.2.3 Region growing segmentation algorithm . . . 20

3.2.3.1 Results . . . 23

3.3 Creating an occupancy grid . . . 24
3.3.1 Results . . . 26

4 Searching for obstacles **29**

4.1 Searching collision zones . . . 30
4.2 Tracking obstacles . . . 30
4.2.1 Detecting obstacles . . . 30
4.2.1.1 Top level function . . . 33
4.2.1.2 Detection and tracking of clusters . . . 35
4.3 Results . . . 39
4.3.1 Detection . . . 42
4.3.2 Tracking . . . 42
4.4 Run time . . . 46

5 Conclusions **48**

List of Figures

1.1 LiDAR scanning a room with an obstacle 3

1.2 Multi-echo technology 4

2.1 Example cloud from a 3D LiDAR 6

2.2 Radar system and tracking a single object 8

2.3 Self-driving car from Google and DARPA Challenge 9

2.4 Chosen LiDAR system 11

3.1 Pass through filter results 15

3.2 Filtering out lone points 16

3.3 k-d tree construction 18

3.4 Calculated normal of KNN points 20

3.5 Segmenting different faces 21

3.6 Camera view of data to be segmented by LiDAR 23

3.7 Cloud being segmented based off of region growing segmentation. 25

3.8 Example occupancy grid 26

3.9 Occupancy grid of entire scan 27

3.10 Closeup of search grid 28

4.1 Lane searching configuration 31

4.2 Search pattern 32

4.3 Same car taking occupying a different number of cells 33
4.4 Camera view of obstacles being tracked 39
4.5 Point cloud for tracking test 40
4.6 Occupancy grid for tracking test 41
4.7 Searching results 43
4.8 Speed of cars in two lanes 44
4.9 Moving average of the data 46

Chapter 1

Background

Contents

1.1 LiDAR **1**

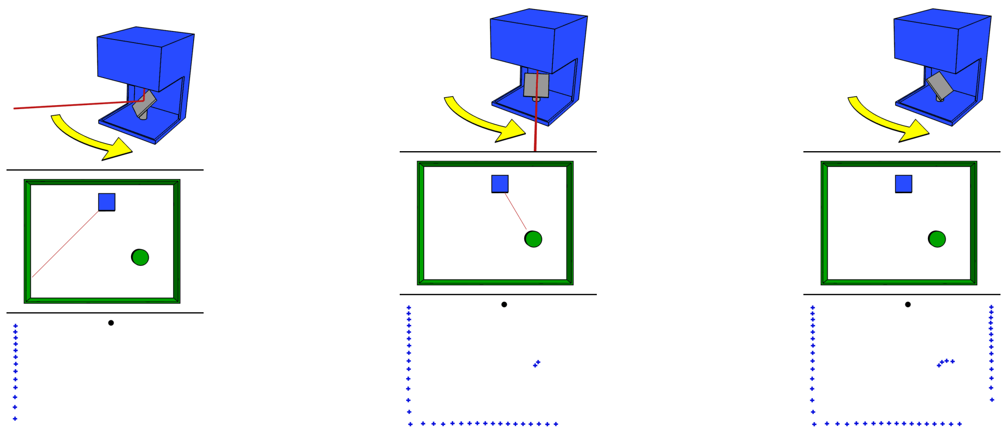
1.1.1 Continuous waveform 3

This chapter provides an introduction to LiDAR systems including an overview of how they work.

1.1 LiDAR

Light Detection and Ranging (or LiDAR / LADAR) uses a pulsed laser beam that it measures the return time of to accurately detect the distance to the object the beam hits as well as the intensity of the returned beam which indicates how reflective the surface is. They typically operate in the near-infrared spectrum and some can work outdoors at ranges varying from a few meters out to over a hundred meters.

A LiDAR can measure a single point (one dimensional), a plane of points (two dimensional), or measure multiple planes to scan an entire area of points (three dimensional). While a single point would be useful for backing up a car to alert the driver if there is any obstacles in the way, it is not sufficient for navigational purposes as it does not give enough information. A two dimensional LiDAR gives out all all points along a plane every scan which begins to be sufficient for navigational purposes but does not give a full picture of a scene which is useful but comes with many more points that have to be analyzed. The LiDAR works by having a rapidly rotating mirror (around 20Hz) with the laser aimed at it spin around the full scan area and reading back the return beam. Each point is given back as a polar coordinate where the angle is given of the return as well as the distance however it trivial to convert it to cartesian. Figure 1.1 shows how a 2D LiDAR works where the top image shows the LiDAR, the middle image shows the LiDAR inside of a room with an obstacle, and the bottom image shows what the system "sees" which is called a *point cloud*. Three dimensional LiDAR systems can give over one hundred thousand points per scan (also 20 Hz) but the points are in multiple parallel planes to create a three dimensional space. Most work similar to the 2D systems just with more lasers going at the same time however there are some that also have the mirror scan in the vertical direction as well as horizontal at the same time.



(a) Scan starting, laser on (b) Cylinder hit, blocks laser from seeing wall (c) Scan finished, laser off until mirror rotates back to start

Figure 1.1: LiDAR scanning a room with an obstacle

1.1.1 Continuous waveform

Typically LiDAR systems will only get one return for each point but there are 2D systems that can get multiple returns per point. These systems are *continuous waveform* meaning that if the laser hit an object that can be penetrated such as a window it will continue to travel and will return the distance of the next return and so on. [1] Many commercially available 2D systems have this available to a degree where they will get a few of these bounce backs, which they call multi-echo meaning it will get N returns. An example of this works can be seen in Figure 1.2 which shows a SICK LiDAR detecting five returns and penetrating glass, fog, rain, and dust. On a car, this is a very useful thing as it allows the LiDAR to work during inclement weather where other systems such as cameras can fail.

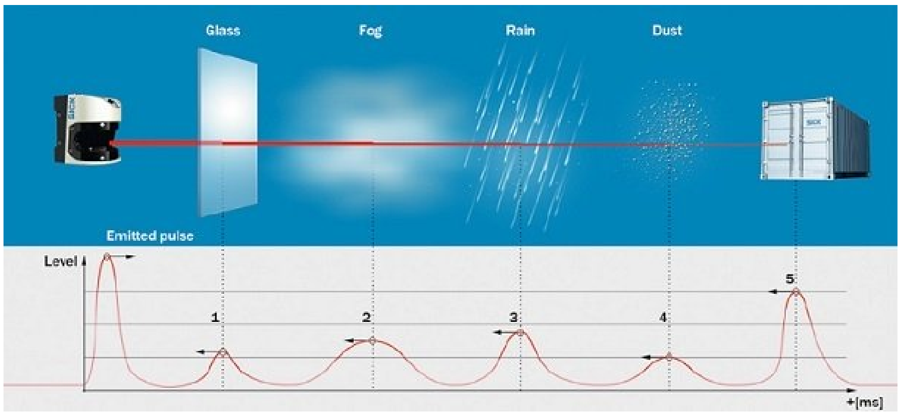


Figure 1.2: Multi-echo technology. Image courtesy of SICK

Chapter 2

Introduction

Contents

2.1 Motivation **5**

2.2 LiDAR vs radar **7**

2.3 Prior work on autonomous navigation **9**

2.4 LiDAR chosen **11**

This chapter introduces the motivation of this thesis as well as a brief look into some of the previous work done with LiDAR on vehicles. It also briefly looks at a simple radar system which was looked at early on and decided not to be pursued in favor of LiDAR.

2.1 Motivation

Many years ago during the DARPA Grand Challenge and Urban Challenge many cars were designed to be able to navigate in many different environments. Many different levels of success were encountered but none better than the winner, Boss, by Tartan Racing Team [2]. However the sensors used are infeasible to put into anything other than a research platform, the 64 line LiDAR from Velodyne (64E) costs $70,000 alone and gets 2.2 million points per second. An example point cloud from a 3D sensor is shown in Figure 2.1 which contains over 7 million points which can be achieved in a little over three seconds with the Velodyne 64E. Every one of these points has an X, Y, Z, and intensity component which are all floating point numbers meaning the cloud also takes a large amount of memory.

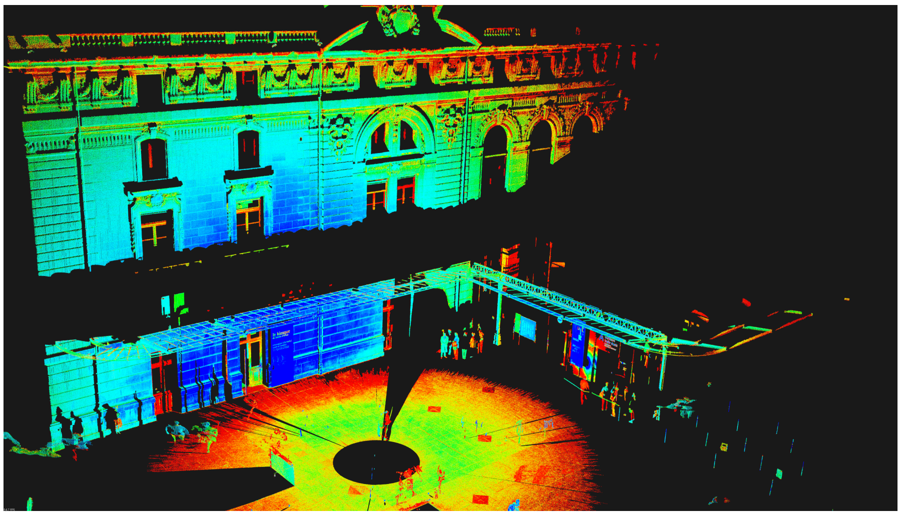


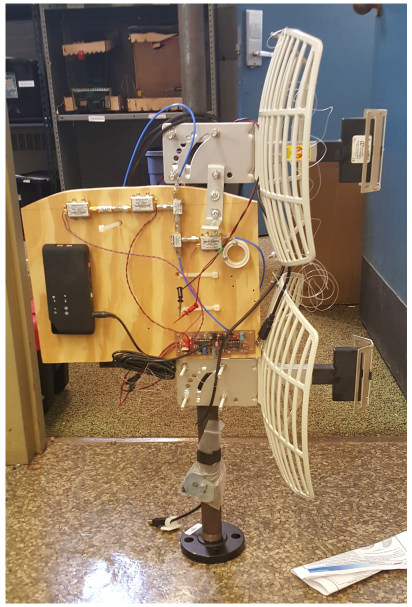
Figure 2.1: Example cloud from a 3D LiDAR

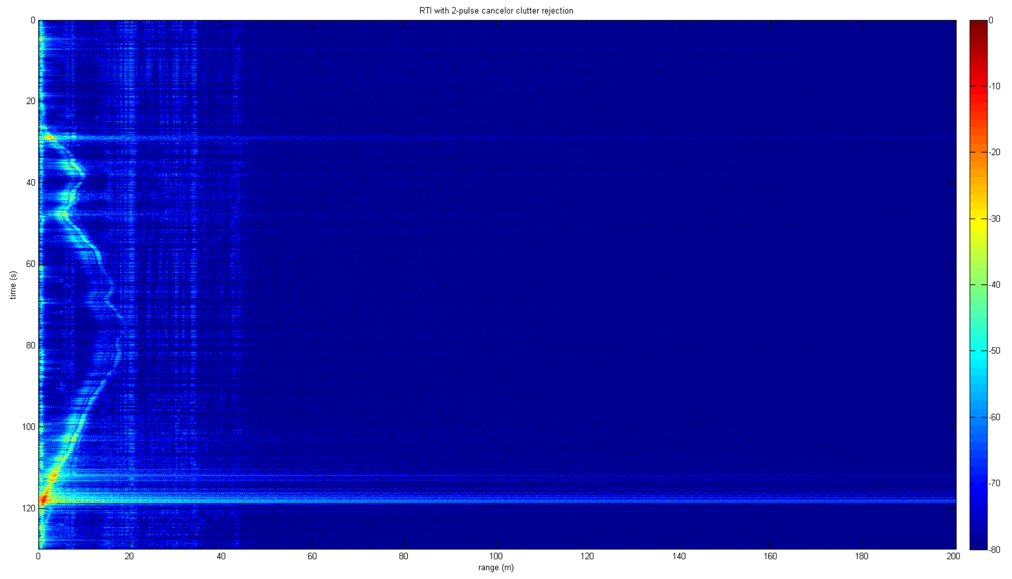
Many of the major car manufacturers as well as Google, Uber, and many others have all been working on their own versions of autonomous cars since the competition but many still have the issues of being infeasible. Either they are too expensive, need large battery banks to run computers, or can't work with the current infrastructure in place already on the road systems.

Autonomous vehicles are going projected to be 75% of all vehicles on the road by the year 2040 but before this can happen many of the issues need to be addressed [3]. One of the issues looked at in this paper is using a 2D LiDAR instead of a 3D LiDAR to see if it is possible to still get workable data from.

2.2 LiDAR vs radar

Radar can do some similar things to LiDAR such as penetrating through weather (depending on the wavelength used). However radar cannot get the type of resolution that a LiDAR can nor can it get the same field of view without making a much larger radar. Early in the project a radar was tested seen in Figure 2.2 along with a person running in front of it and the results can be seen where they run approximately 20 meters away from the system and eventually back to it. However as can readily be seen this data has a large amount of noise, even after filtering and it is only tracking a single object which would work if following a car but not for looking into other lanes. Radar will not be covered in this paper, this is just meant to give a quick comparison between what will be shown later with LiDAR.


(a) Radar used in testing

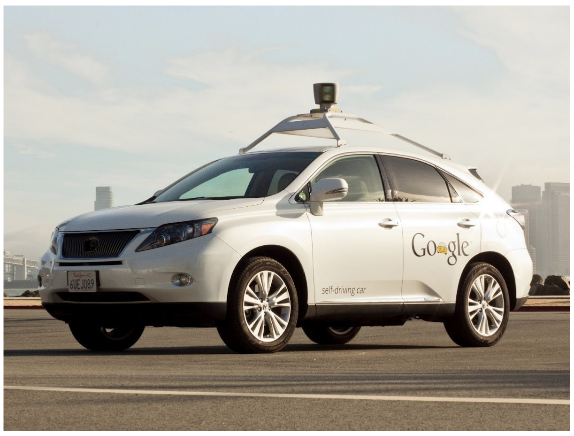


(b) Data of a person running out 20m and back

Figure 2.2: Radar system and tracking a single object

2.3 Prior work on autonomous navigation

In a little over the past decade since the DARPA Grand Challenge autonomous vehicles have been seeing a surge in popularity with most major vehicle manufacturers joining in as well as companies such as Google [4] and Uber. [5] [6] [7] Figure 2.3a shows one of the Google self-driving cars which has a Velodyne 64E 3D LiDAR system on the roof and Figure 2.3 shows Boss, the winner of the DARPA Urban Challenge with a large array of sensors on the roof including the same Velodyne sensor. [8] One interesting topic is driving with tentacles which is presented in [9] and came about from the DARPA challenge talks about one method to narrow down the large amount of data from the 3D LiDARs and what paths can be taken as 3D simply produces too much data to deal with and it accounts for how the vehicle is traveling and tries to narrow down to a few possible paths at the current speed.


(a) Google's self-driving car

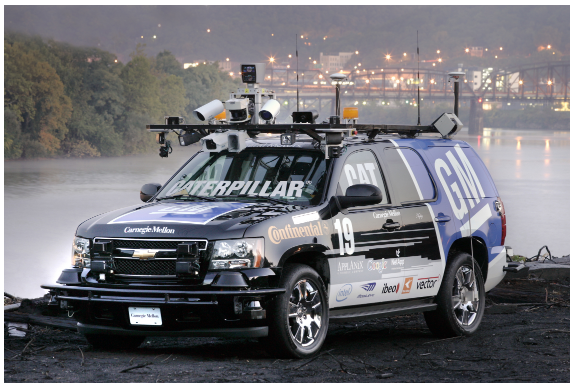

(b) Boss - winner of the DARPA Urban Challenge

Figure 2.3: Self-driving car from Google and DARPA Challenge

Another application of 3D LiDARs as well as a 2D LiDAR pointed at the ground is detecting the shape of the road as shown in [10], [11], [12], [13] (also uses stereo vision for mapping), [14], [15]. Cameras have typically been used for lane detection and finding the edges of the road if no marking appear however LiDAR systems are capable of doing this by using the returned intensity value to detect the difference between the asphalt and lane markers or even the height difference between the road surface and the edges. The advantage of LiDAR is that unlike camera systems that fail at night if there is not adequate lighting the LiDAR solution can continue to work without issue.

There has been some work in people detection in [16] and [17] which is typically a vision system problem as with LiDAR systems, especially 2D ones have a hard time classifying a person but will do very well detecting that there is an obstacle. While not a focus of this paper as people do not walk on the highways it is an important feature to consider when moving to an urban setting.

Cars such as the Tesla have adaptive cruise control using radar which is similar to what the system in this paper can achieve with a LiDAR if a state machine added. Some research has been done in prior in [18] and [19] which both use LiDAR while the second also uses a radar system in addition.

Due to the prohibitive cost of the 3D LiDAR systems, there has been some work trying to create a more cost effective system such as [20] which proposed a cheap sonar and LiDAR method aimed at poorer countries for $225 USD and also implemented peer-to-peer communication with other cars. Some other interesting work has been done with cheap warning systems using a 1D LiDAR on a bike in [21] which could be useful for blind spots on the car in place of more expensive solutions.

The last large area of research with is localization methods such as [22], [23], and [24] all of which use 2D LiDAR systems as a ”push broom” meaning it is aimed at the road, some along with a secondary LiDAR looking for landmarks to try to localize the location of the car. However one major disadvantage with all of these methods is they rely on having 3D priors of the area already which means they are not useful until a map is already built.

2.4 LiDAR chosen

The LiDAR chosen for this thesis is the Hokuyo UXM-30LXH-EWA which has a field of view of 190° and can see out 80m. It was chosen due to its far range and ability to detect four returns, 1mm resolution, and ethernet interface. The LiDAR can be seen in Figure 2.4

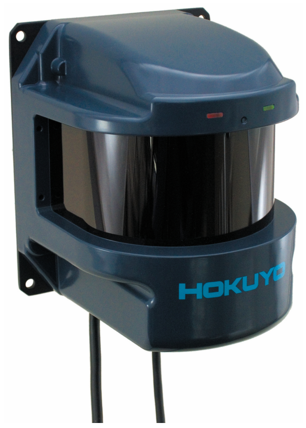


Figure 2.4: Chosen LiDAR system. Image courtesy of Hokuyo

Chapter 3

Mapping with LiDAR

Contents

3.1 Filtering 13

3.1.1 Pass through filter 13

3.1.2 Radius outlier removal 14

3.2 Segmentation 14

3.2.1 k-d tree 16

3.2.2 Calculating normals 19

3.2.3 Region growing segmentation algorithm 20

3.3 Creating an occupancy grid 24

3.3.1 Results 26

In this chapter, we look at how to take the raw LiDAR data and how to transform it into usable data in the form of a map. We use some simple filtering in PCL to remove points that timed out and reported being at maximum range as well as an optional filter to remove lone points that could indicate rain, dust, insects or other small objects by looking around for neighbors within a fixed distance and discarding points which do not meet a minimum set criteria. The data is then segmented into smaller point clusters that likely belong together and allows for recognizing different objects when searching for obstacles later otherwise there is no way to differentiate objects. After all the filtering and segmentation is done, the final step to create a map is to put the segmented cloud into an occupancy grid which will move the points into a searchable grid to look for obstacles.

3.1 Filtering

The filtering needed is very simple for both the pass through filter and the radius outlier removal method. The radius outlier removal is only truly necessary during inclement weather however it can also remove stray points that could happen to appear due to dirt being kicked up on the road and insects flying in front of the car which is not uncommon at night time as they fly towards the headlights. Each point cloud needs to be filtered before placing any points into the occupancy grid discussed in Section 3.3 as if a single point lies within a grid cell then it is marked as occupied and if each grid cell represented a large area (say a meter) it would indicate that there is a 1 meter wide obstacle in that grid cell. If this happened immediately in front of the car then the car would want to hit the brakes even though only a bug is in front of the sensor which on the highway can cause a rear-end collision due to abrupt braking.

3.1.1 Pass through filter

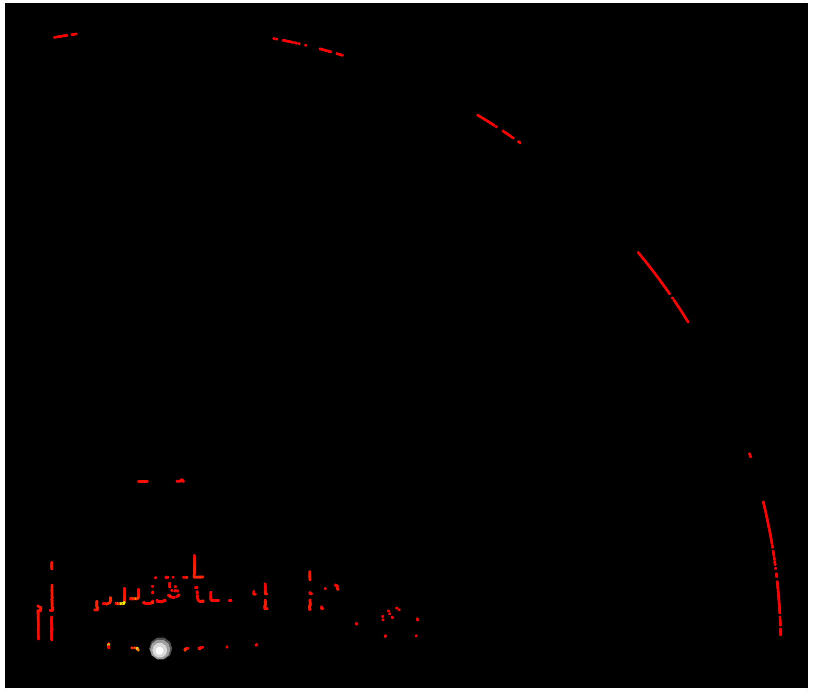
If one of the LiDAR beams goes out past the maximum range it will never report back a distance as it went of range and will make it seem like there is an obstacle at maximum range when in fact there is nothing there. This is done with the use of a very simple pass through filter which check every point in the cloud and sees if it falls within the specified bounds and tossing any that do not meet the criteria and the results of this can be seen in Figure 3.1.

3.1.2 Radius outlier removal

Radius outlier removal is another simple filter that goes through every point and checks to see if there are a chosen number of points within a set distance and if not that point is deleted. This is important as mentioned because if a single point lies within the occupancy grid then it will be marked as occupied, and obstacles such as rain drops and bugs are nonfactors in our planning and those points need to be pruned. This process can be seen in Figure 3.2 where each circle around the blue, green, and purple have a specified distance "D" which is the distance to look for neighbors within. Using this example if a minimum of two points are required the point in circle 1 would be deleted while the rest kept and if two were required then the points in circles 1 and 2 would be deleted while only the points within 3 would be kept.

3.2 Segmentation

Many new segmentation methods for point clouds have been developed over the past two decades to help deal with the rise in popularity of LiDAR over radar, the biggest focus of which has been on 3D LiDAR which can get a large number of points per scan which makes the clouds hard to process without additional information. Many of the algorithms however can also be applied to 2D systems and the one in use in this paper is called *region growing segmentation* which tries to look at the K-nearest neighbors (KNN) to first estimate normals of every single point in the cloud and then to segment it based off of the the difference in the angles of the neighboring normals. To be able to find the nearest neighbors, the first step is to create a search tree to be able to quickly


(a) No returns along the arc

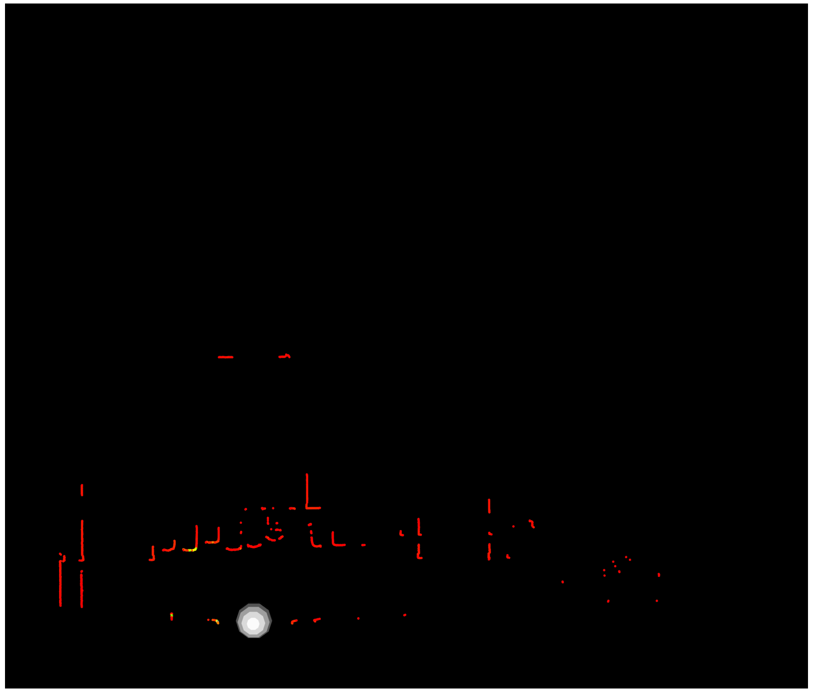

(b) No returns filtered out

Figure 3.1: Pass through filter results

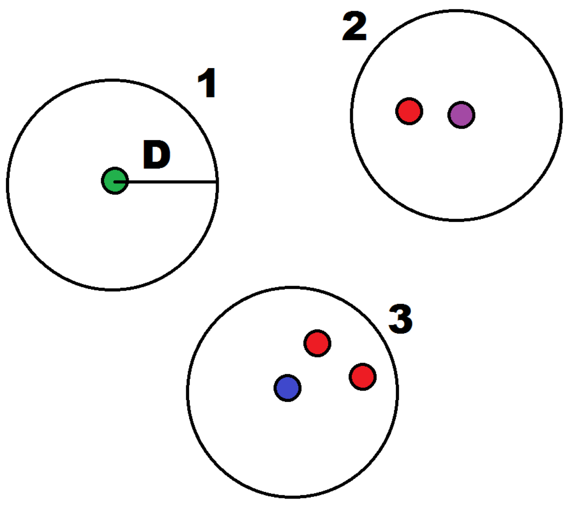


Figure 3.2: Filtering out lone points

find the neighbor for any point as it is needed in both the normal calculation as well as during the segmentation process.

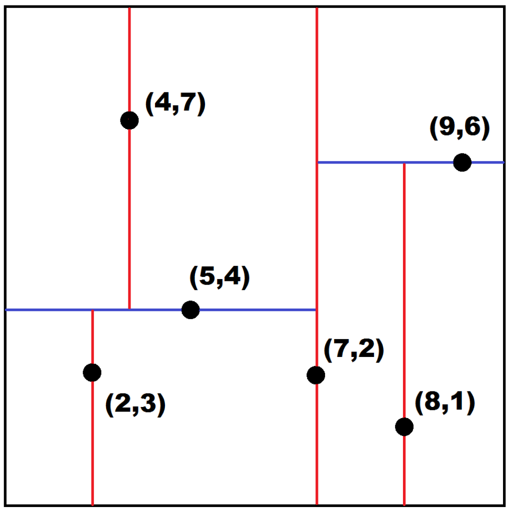
3.2.1 k-d tree

A k-d tree turns the point cloud into a balanced tree that can be searched to find the K-nearest neighbors of any point and is used in normal calculation as well as later in the segmenting the cloud. Being a 2D cloud it is a relatively simple process to split the cloud using the process as described in [25] that takes $O(nlogn)$ time and $O(n)$ space and is shown in the recursive function of Algorithm 1. The recursive function works by starting with a tree and breaking it into a left and right tree by splitting along the X or Y axis in an alternating fashion to leave half of the remaining points on one side of the line and the rest on the other until a lone point remains.

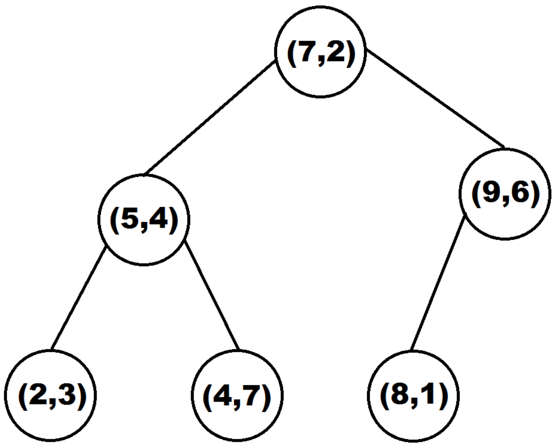
Algorithm 1: BUILDKDTREE Builds a k-d tree out of a 2D point cloud

```
Input: Point cloud = $P$, depth = $d$
Output: Balanced k-d tree $v$
1  if $P$ contains only one point then
2      return a leaf storing this point
3  else if depth is even then
4      Split $P$ with a vertical line through the median x-coordinate into $P_1$ (left of or
        on $l$) and $P_2$ (right of $l$)
5  else
6      Split $P$ with a horizontal line through the median y-coordinate into $P_1$ (below
        or on $l$) and $P_2$ (above $l$)
7  $v_{left} \leftarrow BuildKdTree(P_1, depth + 1)$
8  $v_{right} \leftarrow BuildKdTree(P_2, depth + 1)$
9  Create a node $v$ storing $l$, make $v_{left}$ the left child of $v$, and make $v_{right}$ the right
    child of $v$
10 return $v$
```

This process is shown in Figure 3.3 where a 2D cloud with six points is converted into a tree using the process described by Algorithm 1 by first splitting the X axis at 7 which leaves three points to the left and two points to the right. From there the Y axis is split at 4 and 6 which leaves two and one point respectively on either side of the initial split and from there we are left with out last points which we can use to generate the tree that we saw in Figure 3.3.



(a) A 2D point cloud along with X and Y splits shown to generate a k-d tree



(b) Tree representation

Figure 3.3: k-d tree construction

With the tree constructed, it is straight forward to search through it as we now have all the points that lie at either side of the point of interest so it's a matter of breaking the tree apart to find the KNN that can be used in normal calculation as well as segmentation. This process is described in Algorithm 2 from [25] which is a recursive function which continually breaks apart a tree in the following process which would search for the nearest neighbor.

1. Recursively traverse down the tree splitting the search on whichever side is closest to the given point until a leaf is hit and save that as the best match
2. Go back up the tree and check for better matches on the other side of the split, if so check it otherwise keep going up
3. Return the best matching point that was found

Algorithm 2: SEARCHKDTREE Builds a k-d tree out of a 2D point cloud

```
Input: The root of a k-d tree $v$, a range $R$
Output: All points at leaves below $v$ that lie within the range $R$
1  if $v$ is a leaf then
2      return Whether $v$ lies within $R$
3  if $region(lc(v))$ is fully contained in $R$ then
4      return $SUBTREE(lc(v))$
5  else if $region(lc(v))$ intersects $R$ then
6      return $SEARCHKDTREE(lc(v), R)$
7  if $region(rc(v))$ is fully contained in $R$ then
8      return $SUBTREE(rc(v))$
9  else if $region(rc(v))$ intersects $R$ then
10     return $SEARCHKDTREE(rc(v), R)$
```

3.2.2 Calculating normals

Once a k-d tree is generated the next step is to use it to take the k points to fit a plane to it, but because the point cloud is 2D this comes down to fitting a line to a set of points and taking the inverse slope. The simplest way to do this is with least squares fitting as explained in [26] and it can be done with a special case known as linear least squares fit which is given in Equation 3.1 where the normal is represented by a line of $y = \beta_1 + \beta_2 x$.

$$y = \beta_1 + \sum_{i=2}^{n+1} \beta_i x_{i-1} \tag{3.1}$$

Going through all points using the KNN means that any points that are separated by a large enough distance will not be considered in calculating the normal which stops different objects from affecting the calculations provided there are enough nearby points. One of the obvious drawbacks is when a small object is detected with not enough neighbors points that are a considerable distance away will affect normal calculations however this is dealt with in the region growing segmentation.

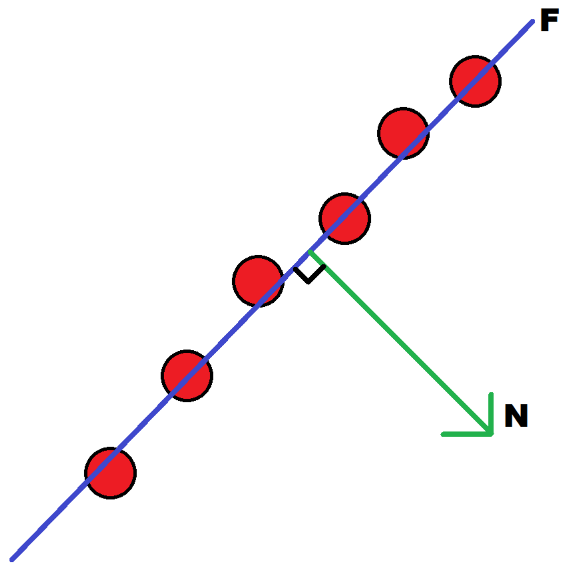


Figure 3.4: Calculated normal of KNN points

3.2.3 Region growing segmentation algorithm

Region growing segmentation is a method that takes the normals of a point cloud and tries to separate data based off of *smoothness* meaning that it tries to group similar normal calculations together. The smoothness helps differentiate different faces of objects as it can be helpful when iterating over the different segments which is shown in Figure 3.5 where three different connected walls are all fitted with lines matching the detected points and each line if a different color representing a different segment. Segmenting the data in this way is not very important while on the highway however and therefore a lot of curvature is allowed for as if a connected object if found while driving, especially on the highway we need to track the entire object moving and not the separate faces which could be an issue as we would see multiple object moving when there is only one.

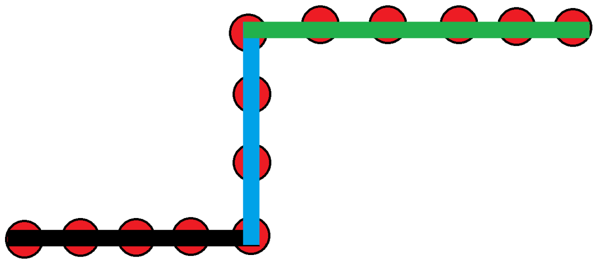


Figure 3.5: Segmenting different faces

The modifiable parameters for the algorithm are the number of neighbors to look for, the minimum and maximum cluster size, and the allowable curve detected from the changing normals between points. The issue mentioned previously of not having enough points for a small object and getting bad normals is dealt with by having a minimum cluster size which will just group them in with the nearest object. This is mostly seen in scans that has bushes and other shrubs where the laser does not get a continuous object back and it ends up grouping all of the small clusters together This process is described in [27] and is is presented in Algorithm 3 and works as follows.

1. Start with a cloud sorted by normals to have the smooth (flat) parts first
2. Pull the first point adding it to the seeds and start to looking at the neighbors
3. If the neighbor has an acceptable difference in angles with respect to the normal, add it to the seed
4. Remove the current seed and report it as one region
5. Repeat until everything is classified

Algorithm 3: REGIONGROWINGSEGMENTATION Segments a given point cloud using smoothness constraint

```
   Input: Point cloud = $\{P\}$, point normals $\{N\}$, residuals $\{r\}$, neighbor finding
          function $\mathbf{\Omega}(.)$, residual threshold $r_{th}$, angle threshold $\theta_{th}$
   Output: A segmented point cloud $\{R_c\}$
1  Initialize Region List $\{R\} \leftarrow \Phi$, Available points list $\{A\} \leftarrow \{1...P_{count}\}$
2  while $\{A\}$ is not empty do
3      Current region $\{R_c\} \leftarrow \emptyset$, Current seeds $\{S_c\} \leftarrow \Phi$
4      Point with minimum residual in $\{A\} \rightarrow P_{min}$
5      $P_{min} \xrightarrow{insert} \{S_c\}\&\{R_c\}$
6      $P_{min} \xrightarrow{remove} \{A\}$
7      for $i = 0$ to size($\{S_c\}$) do
8          Find nearest neighbors of current seed point $\{B_c\} \leftarrow \mathbf{\Omega}(S_c\{i\})$
9          for $j = 0$ to size($\{B_c\}$) do
10             Current neighbor point $P_j \leftarrow B_c\{j\}$
11             if $\{A\}$ contains $P_j$ and $\cos^{-1}(|\langle N\{S_c\{i\}\}, N\{P_j\}\rangle|) < \theta_{th}$ then
12                 $P_j \xrightarrow{insert} \{R_c\}$
13                 $P_j \xrightarrow{remove} \{A\}$
14                 if $r\{P_j\} < r_{th}$ then
15                     $P_j \xrightarrow{insert} \{S_c\}$
16     Add current region to global segment list $\{R_c\} \xrightarrow{insert} \{R\}$
17 Sort $\{R_c\}$ according to the size of the region. return $\{R_c\}$
```

3.2.3.1 Results

Results are very good from the LiDAR and cars almost always segmented as one continuous object which is the main desired outcome. A scan was taken from the data set can be seen in Figure 3.7 where in the raw cloud it is colored by intensity value and in the segmented cloud the points are colored by which segment they belong to. A corresponding image from a camera can be seen in Figure 3.6 taken from the stereo camera system attached to the roof of the car. Looking at the data from the camera the main features that can be seen is the car in the left lane as well as trees on either side of the road.

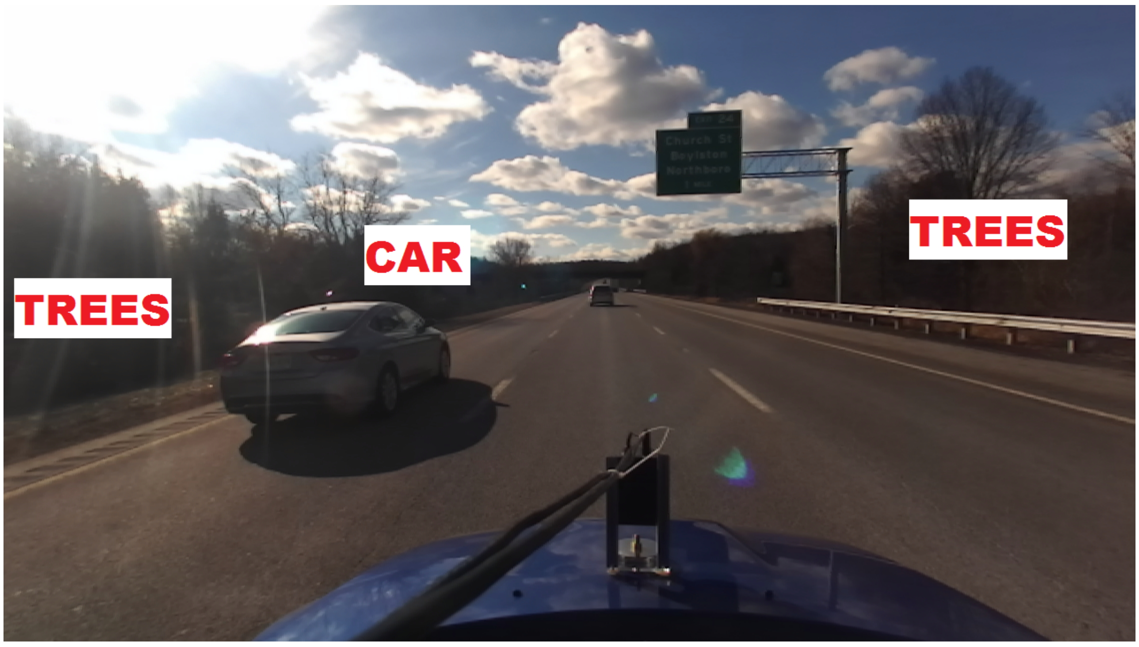


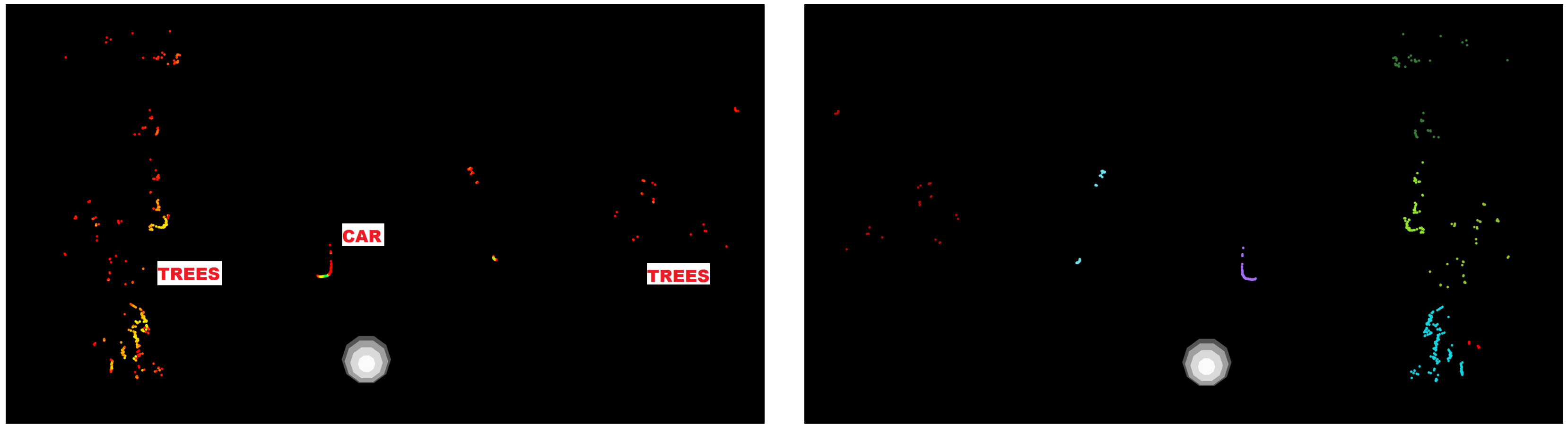
Figure 3.6: Camera view of data to be segmented by LiDAR

Looking at the LiDAR data all of these features are very easy to detect and will be used when creating and filling an occupancy grid with the detected features. The segmented cloud is saved as a second point cloud where each point has a number associated with it corresponding to which cluster it belongs to as well as a link to the original cloud so the original point can be examined again if desired.

3.3 Creating an occupancy grid

One of the issues with having a point cloud is the resolution of the LiDAR has too high of a resolution (1mm) and having a 160m by 80m searching distance would results in 12,800,000 possible spots for a point to appear which is too many spots to check for an obstacle in real-time. The way this issue is addressed is by creating an occupancy grid which subdivides the space into are more manageable form where any points that fall within a grid cell means the cell is marked as occupied by an object. This can be seen easily with the example shown in Figure 3.8 where some simple shapes are drawn on the grid and any cell that contains any bit of the shapes is shaded gray marking it as occupied meaning nothing can pass through it. The advantage of this method being that the cloud will be reduced as all of the points that are tightly packed can be mapped to the same grid cell and if a good good resolution is picked this will make the data easily searchable while still retaining enough data about the detected objects.

The grid was created by using a 2D fixed sized array with a resolution of 20cm x 20cm based off of [28] as well as empirically testing scans to find a good trade off between the overall size of the grid as well as retaining the shape of obstacles and results in a total of 320,000 cells. As the LiDAR is in the center of the bottom of the cloud as was seen in Figure 3.7 the cloud needs to be shifted to be stored into an array as half the data is in the negative X space which cannot be indexed in an array so everything is shifted by 40m or half of the total X distance. From here both the X and Y location of the points



(a) Raw, unsegmented cloud

(b) Segmented cloud

Figure 3.7: Cloud being segmented based off of region growing segmentation. Note: the white sphere represents the position of the LiDAR.

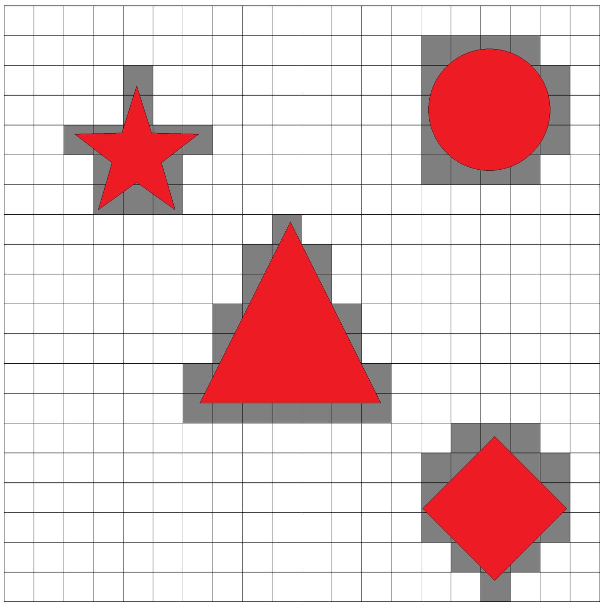


Figure 3.8: Example occupancy grid

is divided by the chosen resolution of 20cm to place each point into the appropriate grid, and the point's segmented cluster number is used to mark each cell as occupied so each cluster can be differentiated from another.

3.3.1 Results

Converting the same scan from Figure 3.7 into an occupancy grid results in Figure 3.9 where the dark blue cells represent free space. As is expected much of the grid is unoccupied as the point cloud indicates and all of the obstacles are colored according to their cluster similar to the segmented point cloud.

Zooming in on one of the features on the grid shows Figure 3.10 where the white grid represents the 20cm resolution and the larger red grid represents 200cm cells. The directly connected object see in this figure is the pole that was seen in the camera view on the right side of the image near the tree line and the few detected returns from the trees behind it can also be seen to belong to the same cluster due to there not being

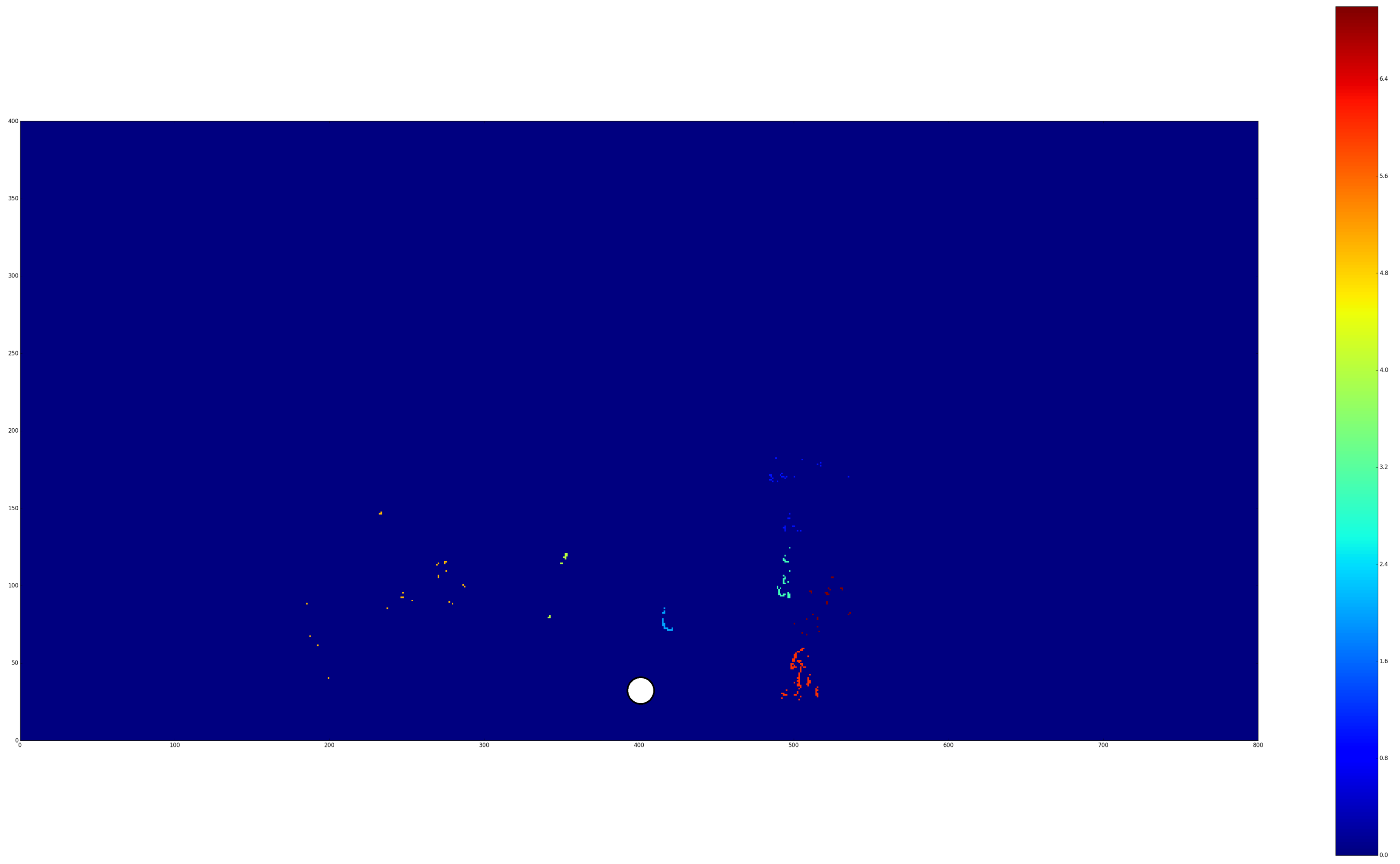


Figure 3.9: Occupancy grid of entire scan. Note: the white circle represents the position of the LiDAR and the the scan is flipped horizontally.

enough points in the cluster for either by themselves.

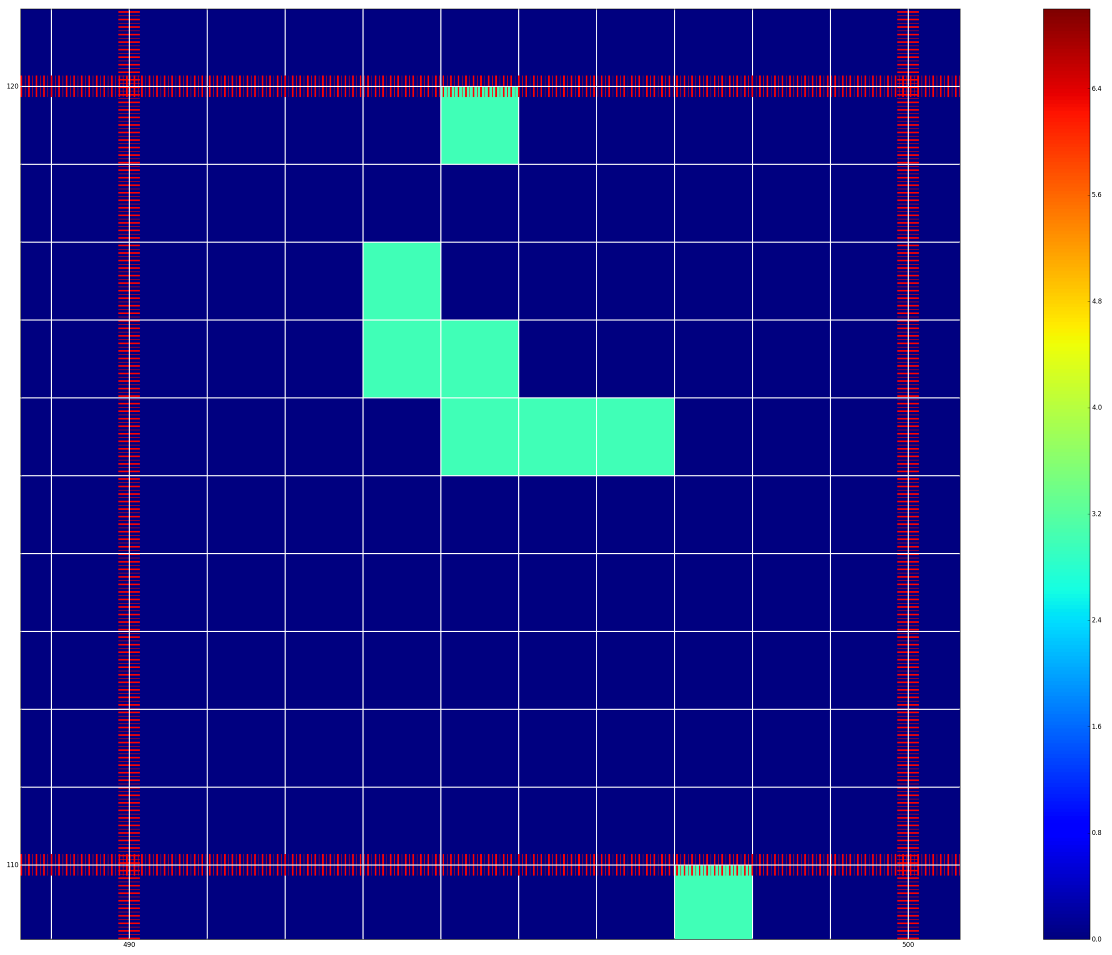


Figure 3.10: Minor ticks = 20 cm, major = 200 cm

Chapter 4

Searching for obstacles

Contents

4.1 Searching collision zones 30

4.2 Tracking obstacles 30

4.2.1 Detecting obstacles 30

4.3 Results 39

4.3.1 Detection 42

4.3.2 Tracking 42

4.4 Run time 46

Once the occupancy grid is constructed the next step is to search through the grid for obstacles so in the next step we can track them at each time step. In this chapter we look at how we accomplish object detection and tracking within the occupancy grid and how the grid was searched with real-time constraints.

4.1 Searching collision zones

The first 20m in front of the car in both the side lanes are deemed the *collision zones* as traveling at 65mph a car can travel that distance in about 0.5 seconds so if anything is closer than that we do not want to consider merging due to safety reasons. All that the LiDAR will look for is if any sized obstacle is within this first 20m of the lane and if so it will report it as unsafe to merge. The breakdowns of the lanes can be seen in Figure 4.1 where each rectangular zone is 20m long and 3m wide which is the average width of a lane on the highway, the lanes marked "T" are tracking areas and the ones marked "C" are the collision zones. How this looks on the actual grid can be seen in Figure 4.2.

4.2 Tracking obstacles

4.2.1 Detecting obstacles

At highway speeds a car can only go straight or change lanes so searching for obstacles can be vastly simplified to look at our current lane as well as to either side side of the car to check on cars or other obstacles within the lanes. The first step is to check the center lane, where we are traveling, for any obstacles and if we see any unoccupied cells we need to see if it has been seen previously or if it is a brand new obstacle, or conversely if we see nothing but last time step had an obstacle we need to prune it as we are no longer tracking it. The way that a detected object is matched with any previously seen ones is based off of the width of the obstacle which is allowed to be one grid cell wider or thinner, and this is because of Figure 4.3 which shows a car occupying three cells in one scan and two cells in another. This is because the grid is fixed in reference to the

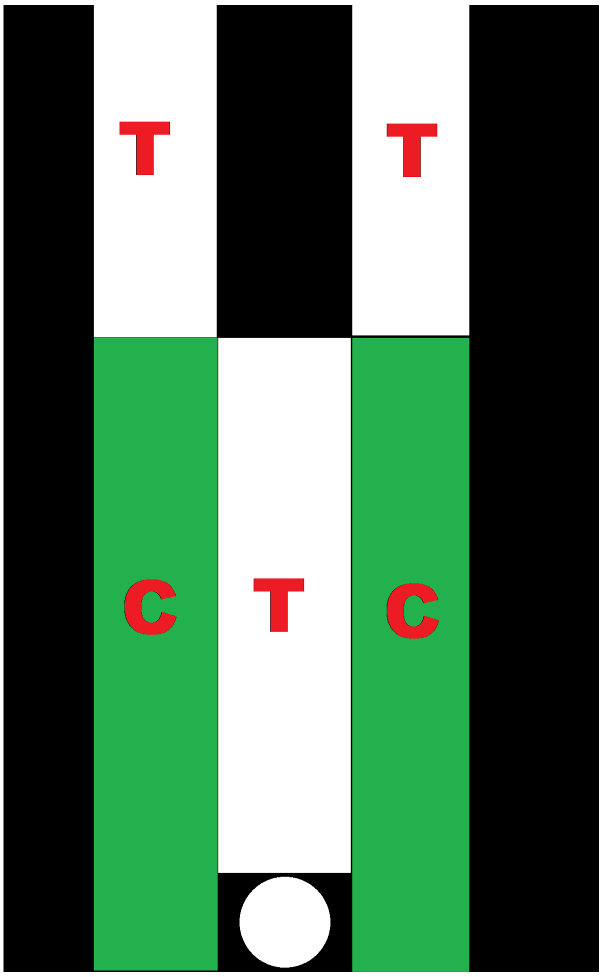


Figure 4.1: Lane searching configuration. Note: the white sphere represents the position of the LiDAR

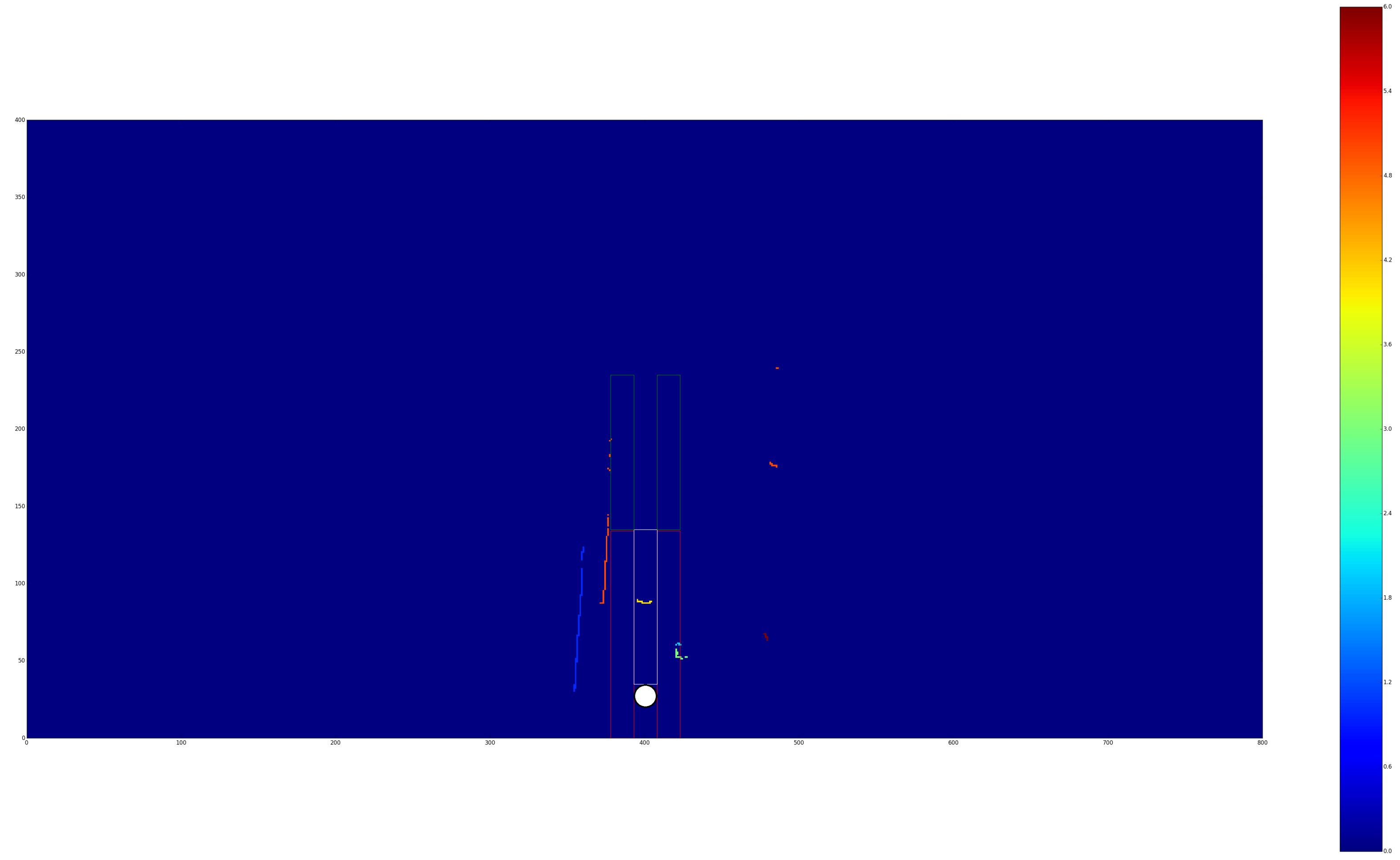


Figure 4.2: Search pattern. Note: the white sphere represents the position of the LiDAR and the scan is flipped horizontally

LiDAR and any shift of obstacles to the left or the right can mean that they can sit just barely inside of one cell to mark it as occupied and this is a drawback of the resolution of the grid however it is not difficult to account for by allowing the shape to grow or shrink by one between scans.

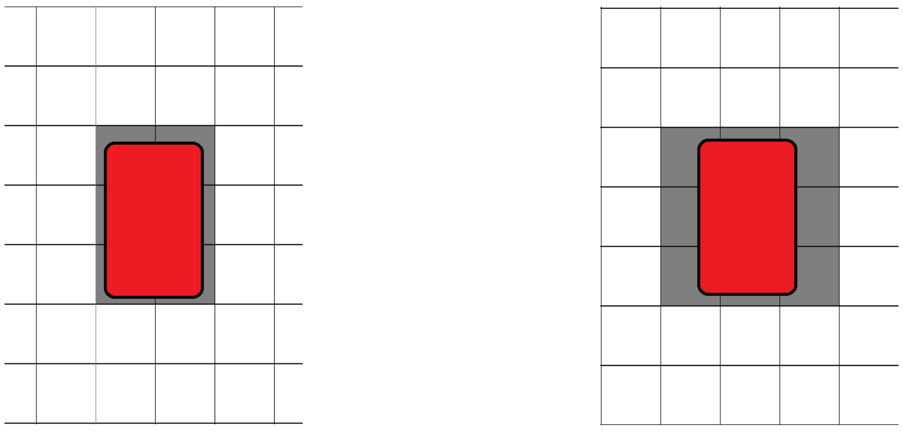

(a) 2 cells wide car (b) 3 cells wide car

Figure 4.3: Same car taking occupying a different number of cells

4.2.1.1 Top level function

The side lanes work in the exact same fashion but they instead start tracking *after* 20m as before that is the collision zone discussed in the previous section. The first step of the tracking process can be seen in Listing 4.1 which is the *top level function* that takes the following steps which uses the two functions from the Section 4.2.1.2 as well as some other similar functions omitted for brevity.

1. Define the lane search bounds based off of the desired lane to check and set *LaneClustersPtr* to be equal to any previously seen clusters
2. Record any found clusters into *clustersSeen*

3. If no clusters were found clear any previous tracking data and exit
4. If any clusters were found, try to see if we can find any matches to our previous data and store it into *matchingHits*
5. If there are no matches input everything as clusters that we'll begin to track
6. If there are matches update the speeds of them as appropriate

,

```
void laserScansMaps::calcSpeed(char lane){
  //will store the speeds of anything we see
  speedList* laneClustersPtr;

  //select appropriate list for our pointer
  int leftCellX, leftCellY;
  switch(lane){
    //pick proper start/end cells
  }

  //grab the clusters that we see
  clusterList clustersSeen = searchForward(laneClustersPtr, leftCellX,
    leftCellY);
  //if we got no clusters, we have no speeds
  //just empty the list as we aren't tracking anything
  if(!clustersSeen.size()){
    laneClusters.clear();
    //all done, nothing was seen
    return;
  }

  //otherwise let's look for a matching cluster if we have any previous
    data
  clusterMatches matchingHits;
  if(laneClusters.size()){
    findMatches(clustersSeen, matchingHits, laneClustersPtr);
  }

  //if we have no previous speeds (hence no match) it's new detection
  if(!laneClusters.size()){
    fillNoPrior(clustersSeen, laneClustersPtr);
    //we're done if we didn't have anything already
    return;
  }

  //if we had previous speeds and at least some matches
  if(matchingHits.size()){
    fillWithMatches(clustersSeen, matchingHits, laneClustersPtr);
  }
  //if we have anything left that's new, we need to add it
  fillRemaining(clustersSeen, laneClustersPtr);

  //print out the speeds of what we're tracking
}
```

Listing 4.1: Top level for detection and tracking

4.2.1.2 Detection and tracking of clusters

The next step in the process is to look for obstacles for the tracking zones as shown in Listing 4.2 and perform the following actions

1. Create a list that will store
    - Cluster number (from segmentation)
    - Minimum and maximum X location hits (to calculate the width)
    - Minimum Y distance detected per cluster
2. Search through the grid cells for our defined range (lane location)
3. When we first see the cluster (not in the list) add it with the corresponding data filled in representing that grid cell
4. When we iterate through the other cells, modify each cluster as appropriate until finished
5. Return the final list back to the top level function

,

```
clusterList laserScansMaps::searchForward(int leftCellX, int leftCellY,
    int width, int dist){
  //will store all of our clusters that we saw and
  //cluster#,minHit,maxHit,dist
  clusterList clustersSeen;

  //iterate through every cell in the lane we're searching
  for(int i = leftCellX; i < leftCellX + width; i++){
    for(int j = 0; j < dist; j++){
      int cell = grid[i][j+leftCellY];
      if(cell > 0){
        //if the cluster exists in the list, modify if needed
        int foundCluster = 0;
        for(int k = 0; k < clustersSeen.size(); k++){
          //if the i element matches the cluster we saw, we modify the values if needed
          if(clustersSeen[k][CLUSTERIDX] == cell){
            //add 1 incase it's a 0, we'll subtract it later (0 matches free space)
            foundCluster = k+1;
            break;
          }
        }
        //if we found the cluster
        if(foundCluster){
          //shift index back (undo earlier shift up 1)
          int idx = foundCluster - 1;
          //find the edges of the width, modify previously seen max/min as appropriate
          if(i < clustersSeen[idx][MINHITIDX]) clustersSeen[idx][MINHITIDX] = i;
          if(i > clustersSeen[idx][MAXHITIDX]) clustersSeen[idx][MAXHITIDX] = i;
          //distance (take cell closest to us)
          if( (j+leftCellY) < clustersSeen[idx][DISTIDX] ) clustersSeen[idx][DISTIDX] = j+leftCellY;
        }
        //otherwise we need to add the cluster to our list
        else{
          //create a new cluster and set the values
          scanArr newHit;
          newHit[CLUSTERIDX] = cell;
          newHit[MINHITIDX] = i;
          newHit[MAXHITIDX] = i;
          newHit[DISTIDX] = j + leftCellY;
          clustersSeen.push_back(newHit);
        }

      }
    }
  }
  //if we found clusters report the speeds
  //otherwise report we saw nothing

  //return the list of clusters
  return clustersSeen;
}
```

Listing 4.2: Search for obstacles

The next step is to go through every found cluster and fill in *matchingHits* in the top level which is a list that stores the index of where a match lies in the previous data and the current data. This comparison is done off of width as it is the only data that is available from the 2D point cloud for comparison along with last seen distance. The *matchingHits* list is then returned back to the top level function which now has a link between the past to the current objects that it can operate on to calculate new speeds as appropriate.

The tracking function has a few variations for adding clusters to the previously seen list however they all work in a similar manner. Listing 4.3 shows how all of the clusters that were seen are taken in and how they are updated to calculate the speed of a previously seen object which we know based off of the detector. The steps are as follows for every matching cluster we found previously

1. For every match we have from *findMatches()* we pull out the data from its past and current width, speed, and distance
2. If the obstacle is still at the same distance, increment the *INSTIDX* variable so that we know it was seen at the same distance an additional time
3. If the obstacle is at a new distance, take the *INSTIDX* variable and multiply it by the resolution of the grid to update the speed then change the *INSTIDX* variable to 1 as it's the first time we've seen it at this distance
4. Update the width of the obstacle if necessary
5. Erase the previously seen cluster from our currently seen list so we don't add it in again when filling in the rest of the data

```
void laserScansMaps::fillWithMatches(clusterList& clustersSeen,
    clusterMatches& matchingHits, speedList* laneClustersPtr){
  //de-reference the pointer
  speedList& laneClusters = *laneClustersPtr;

  //report how many different clusters we found and need to calculate the
   speed of

  //for every cluster we found
  for(int i = 0; i < matchingHits.size(); i++){
    //we need to update the matching index of the cluster to speed
    //to change the width, distance, speed, instances (all as appropriate)
    int hitsIdx = matchingHits[i][0];
    int speedsIdx = matchingHits[i][1];

    //get the new and previous distances so we can find the speed
    float newDist = clustersSeen[hitsIdx][DISTIDX];
    float newWidth = clustersSeen[hitsIdx][MAXHITIDX] - clustersSeen[
   hitsIdx][MINHITIDX];
    float prevDist = laneClusters[speedsIdx][DISTIDX];
    float prevSpeed = laneClusters[speedsIdx][SPEEDIDX];

    //if the distance hasn't changed
    if(newDist == prevDist) {
      //if it's been the same for a while, set the speed to 0
      laneClusters[speedsIdx][INSTIDX] += 1;
      //update width if needed
      laneClusters[speedsIdx][WIDTHIDX] = newWidth;
      if(laneClusters[speedsIdx][INSTIDX] > SPEEDDECAY) laneClusters[
   speedsIdx][SPEEDIDX] = 0.0;
    }
    //otherwise we need to find the speed of the object
    else{
      //distance in cm changed
      float changedDist = (newDist - prevDist) * CELLINV;
      //report cells changed in how many steps
      float changedStep = laneClusters[speedsIdx][INSTIDX];
      float changedSpeed = changedDist / (changedStep * FREQ);
      //save speed as mph
      laneClusters[speedsIdx][SPEEDIDX] = changedSpeed * CMSTOMPH;
      //first time we've seen this
      laneClusters[speedsIdx][INSTIDX] = 1.0;
      laneClusters[speedsIdx][DISTIDX] = newDist;
      laneClusters[speedsIdx][WIDTHIDX] = newWidth;
    }
    //now we need to erase what we just did from hits
    clustersSeen.erase(clustersSeen.begin() + hitsIdx - i);
  }
}
```

Listing 4.3: Speed calculations

4.3 Results

One scene tested for tracking results can be seen in Figure 4.4 where one car is in the center lane and is tracked for velocity and a second car starting in the collision zone and eventually entering the tracking zone for the side lane as they are going much faster than our car. The corresponding point cloud can be seen in Figure 4.5 along with the occupancy grid in Figure 4.6 both of which shows both cars in their starting positions and this scene is used both in the following sections.

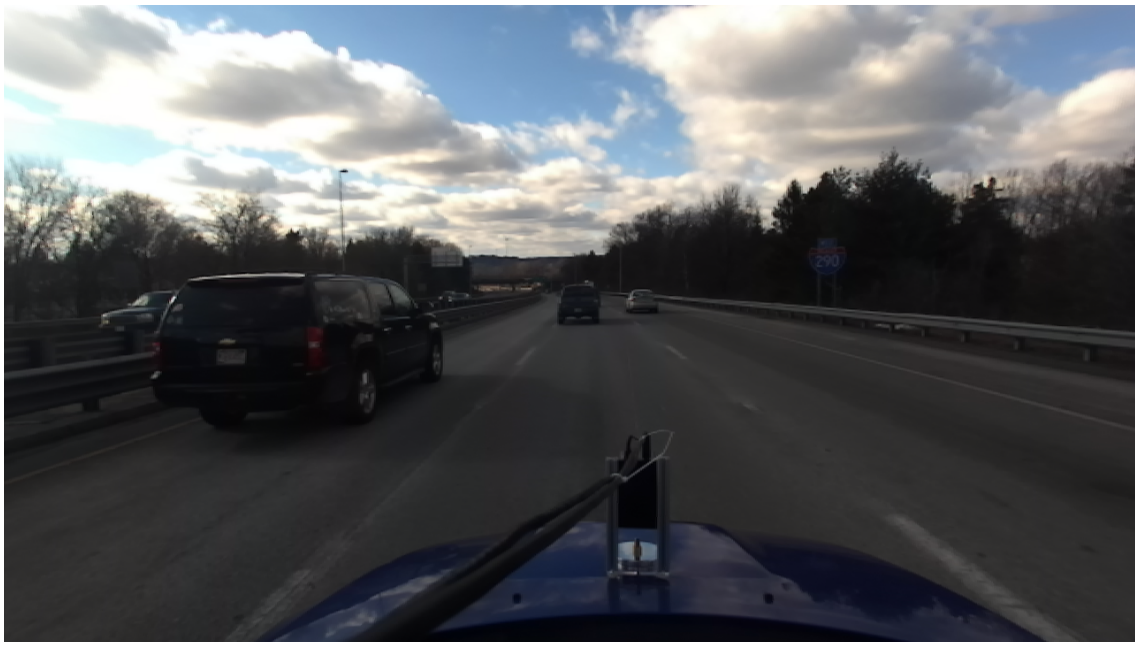


Figure 4.4: Camera view of obstacles being tracked

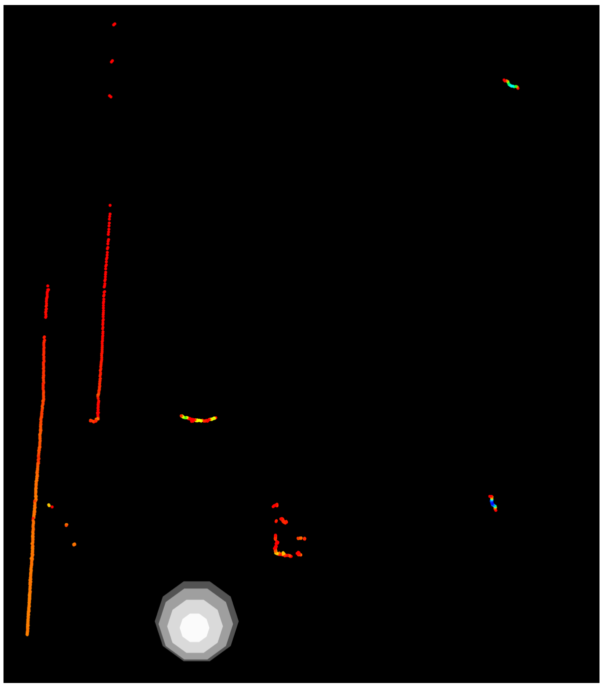


Figure 4.5: Point cloud for tracking test. Note: the white sphere represents the position of the LiDAR and the scan is flipped horizontally

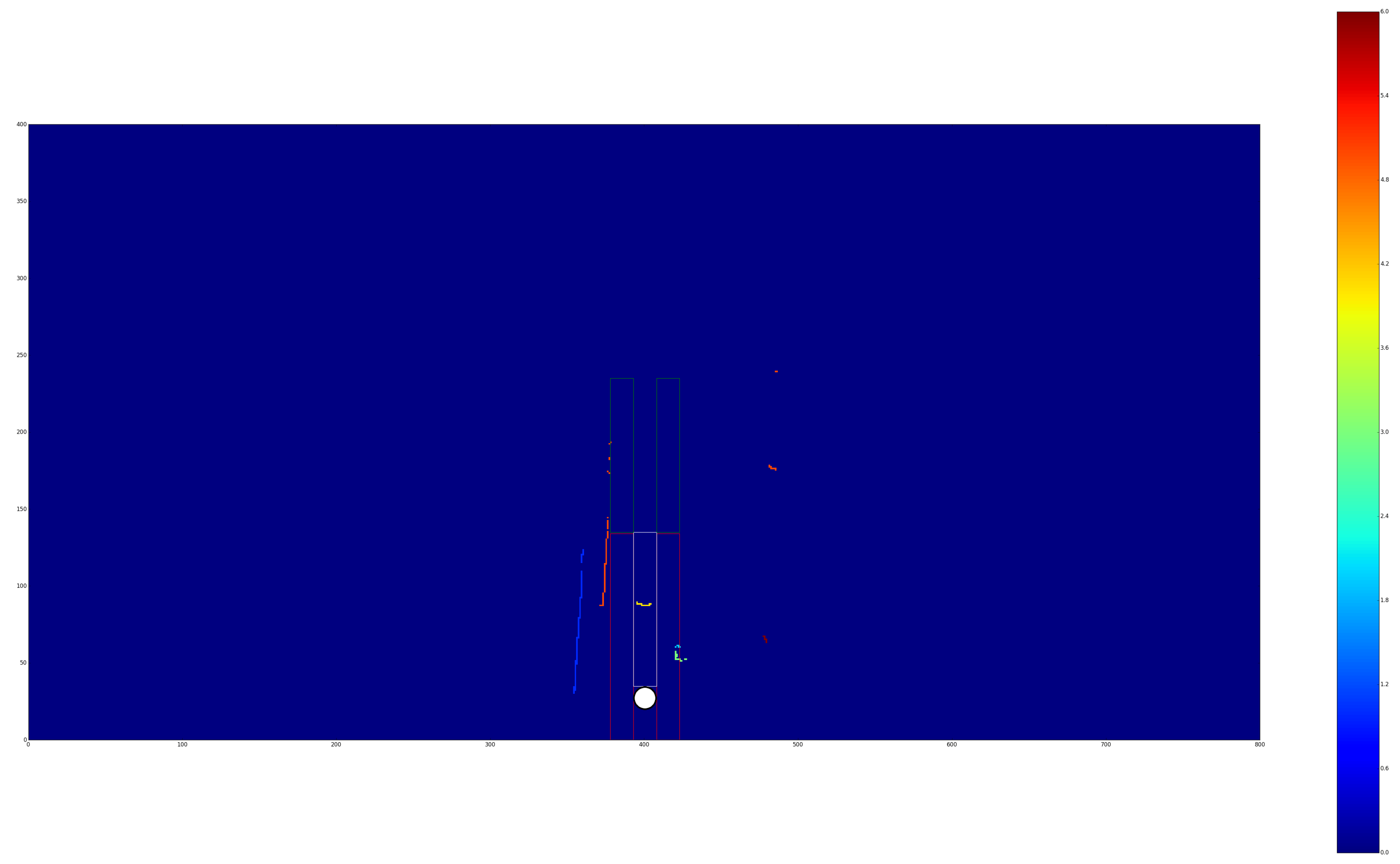


Figure 4.6: Occupancy grid for tracking test. Note: the white sphere represents the position of the LiDAR and the scan is flipped horizontally

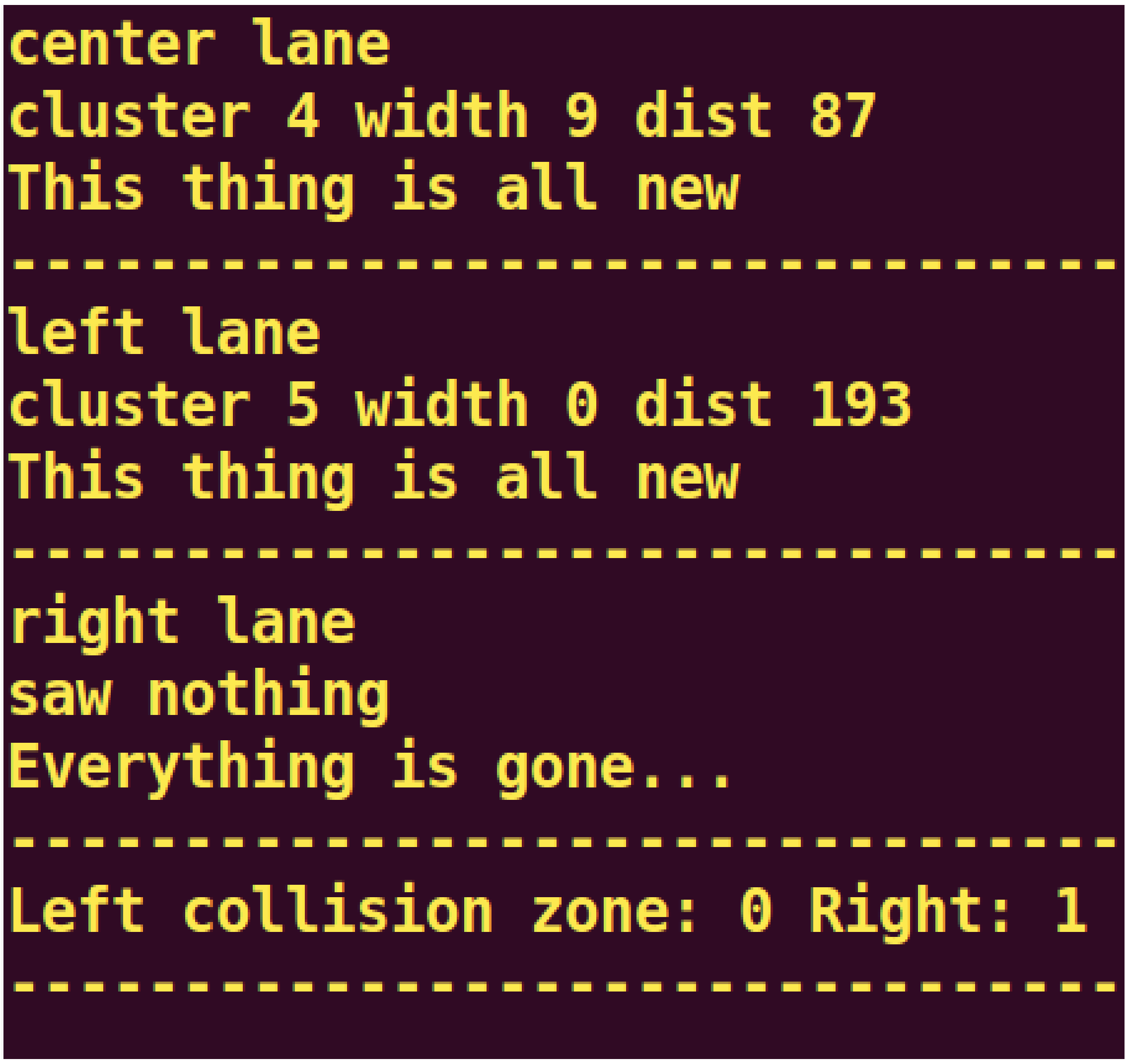
4.3.1 Detection

Figure 4.7 shows the output of the program which sees the center car 87 grid cells away (17.4m), detects an obstacle which happens to be a wall in the left lane and finally the car next to us within the collision zone. The detection of the wall within the tracking zone is undesirable however it is not really an issue as normally we would have an additional sensor such as a camera or another LiDAR detecting lanes telling us that is not a valid path and additionally as we track it we will find that it moves at impossible speeds however this data is still somewhat useful. In the case of an emergency where it is needed to change lanes due to a danger within the center lane we've already found that the right line is not safe however if the side lane were a breakdown lane we would still be able to detect that it is empty and can be entered if an obstacle needed to be immediately avoided within the current lane.

4.3.2 Tracking

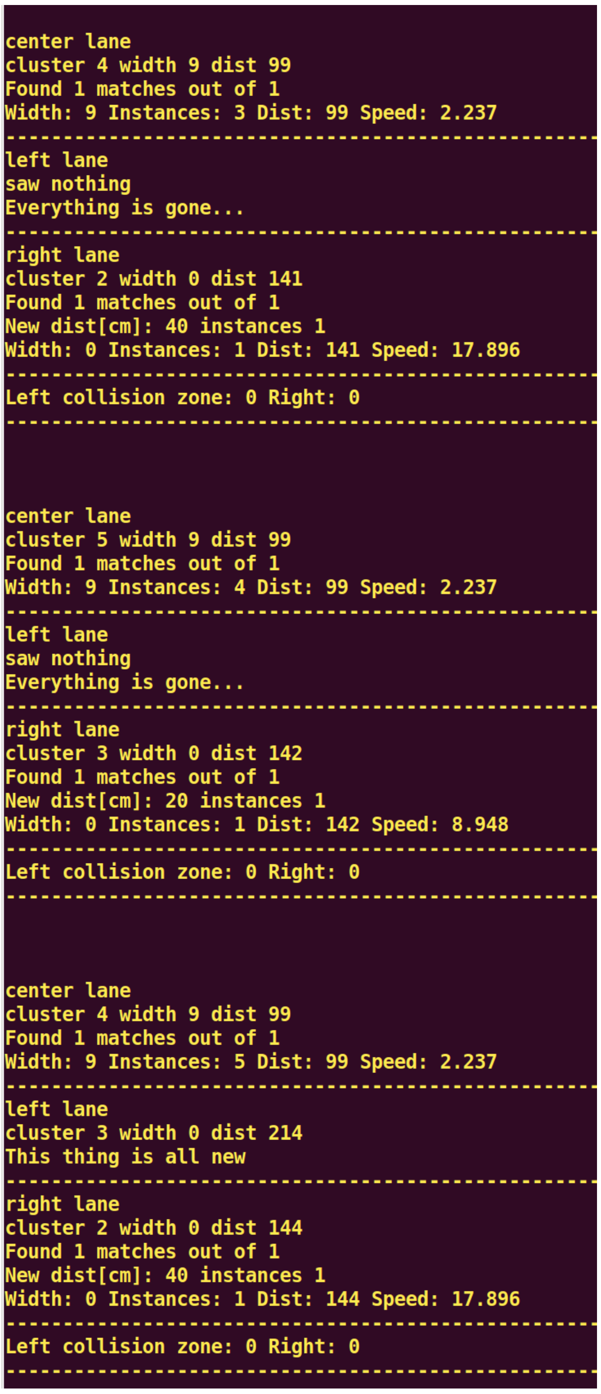
Taking the previous results of the detector and continuing to step through the data results in Figure 4.8 which shows the car in the center lane going 2.2 MPH faster than us which is 0.1 MPH off from his actual speed of 2.3 MPH. Once the car in the side lane entered the tracking zone and is stepped through three scans it can be seen flipping between 8.9 MPH and 17.9 MPH faster than us, a range of 9 MPH total in 0.05 sec intervals!

Measuring the actual speed of the car shows that it is traveling at 15 MPH which is in between the two value and this a a drawback of the occupancy grid method which can only detect changes in increments of one grid cell at a time and what is happening



```
center lane
cluster 4 width 9 dist 87
This thing is all new
--------------------------------
left lane
cluster 5 width 0 dist 193
This thing is all new
--------------------------------
right lane
saw nothing
Everything is gone...
--------------------------------
Left collision zone: 0 Right: 1
--------------------------------
```

Figure 4.7: Searching results. Note: the scan is flipped horizontally



```
center lane
cluster 4 width 9 dist 99
Found 1 matches out of 1
Width: 9 Instances: 3 Dist: 99 Speed: 2.237
---------------------------------------------------
left lane
saw nothing
Everything is gone...
---------------------------------------------------
right lane
cluster 2 width 0 dist 141
Found 1 matches out of 1
New dist[cm]: 40 instances 1
Width: 0 Instances: 1 Dist: 141 Speed: 17.896
---------------------------------------------------
Left collision zone: 0 Right: 0
---------------------------------------------------



center lane
cluster 5 width 9 dist 99
Found 1 matches out of 1
Width: 9 Instances: 4 Dist: 99 Speed: 2.237
---------------------------------------------------
left lane
saw nothing
Everything is gone...
---------------------------------------------------
right lane
cluster 3 width 0 dist 142
Found 1 matches out of 1
New dist[cm]: 20 instances 1
Width: 0 Instances: 1 Dist: 142 Speed: 8.948
---------------------------------------------------
Left collision zone: 0 Right: 0
---------------------------------------------------



center lane
cluster 4 width 9 dist 99
Found 1 matches out of 1
Width: 9 Instances: 5 Dist: 99 Speed: 2.237
---------------------------------------------------
left lane
cluster 3 width 0 dist 214
This thing is all new
---------------------------------------------------
right lane
cluster 2 width 0 dist 144
Found 1 matches out of 1
New dist[cm]: 40 instances 1
Width: 0 Instances: 1 Dist: 144 Speed: 17.896
---------------------------------------------------
Left collision zone: 0 Right: 0
---------------------------------------------------
```

Figure 4.8: Speed of cars in two lanes

in this particular case is he is alternating between traveling one cell in the 8 MPH scans and 2 cells in the 17 MPH scans. Tracking this further begins to reveal a sort of pattern shown in Table 4.1.

Time	0.05	0.10	0.15	0.20	0.25	0.30	0.35	.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80
Speed	8.948	17.896	8.948	17.896	8.948	17.896	17.896	8.948	17.896	17.896	8.948	8.948	17.896	17.896	8.948	17.896

Table 4.1: Recorded speeds over 16 steps

As can be seen there are some steps where it does not alternate between the two speeds and instead has multiple occurrences of a speed due to the nature of the occupancy grid. To more correctly track the speed there are two options each which are rather simple that can get a lot closer if it is required, the first of which is to take the moving average filter which just convolves the data data when a pattern such as this is detected. Using a window size of two and feeding in our data to a filter results in Figure 4.9 which can be seen to be at 15 MPH much of the time and with smoothing can allow for additional filtering to not allow the speed to change abruptly which happens at a couple of the time steps after reaching the 15 MPH estimation.

The second way to deal with this is to create a link from the point cloud to the occupancy grid cells where we need *at least* one point per occupied cell. We then can then perform all of the searching normally in the occupancy grid and then pull out a raw point to calculate the speed as the change in distance from the car which has 1mm resolution instead of 20cm. Doing this we also get the speed of 15 MPH however smoothing out the data is still desirable as it will allow for eliminating some of the noise of small changes in speed so there is not oscillation on a much smaller scale however

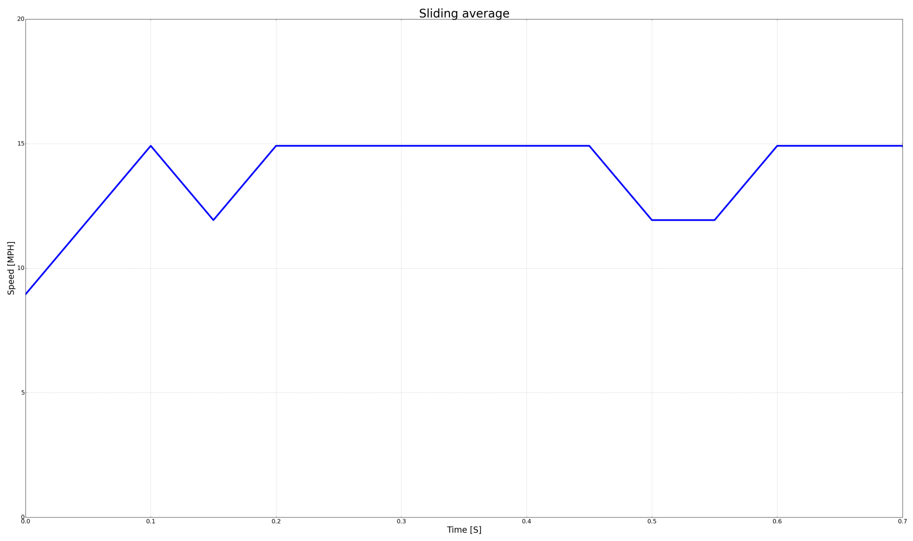


Figure 4.9: Moving average of the data

both solutions will give a more accurate speed.

4.4 Run time

As real-time performance is desired 100 scans were taken and every step was run on every scan and the average time was taken to get a total time of

$$\begin{array}{rcc} & 0.05\ [\mathrm{S}] & \text{to scan and segment the data} \\ + & 0.013\ [\mathrm{S}] & \text{to segment and store the data} \\ + & 7.393\mathrm{E}\text{-}5\ [\mathrm{S}] & \text{to search the grid and track data} \\ \hline & 0.063\ [\mathrm{S}] & \text{on a 4GHz i7} \end{array}$$

which means there is still 0.037 seconds left until the next scan that are free for additional work if desired in the future or if it is put onto a slower processor it has a higher probability of working as we only consume about 25% of our available time for everything.

Chapter 5

Conclusions

In this thesis, we present a way to track vehicles in front of of a car while traveling on the highway in real-time. We propose to use a data segmentation method to convert a point cloud into a segmented cloud representing different objects and then taking it and putting it into an occupancy grid that can be search efficiently.

For future work we would like to add additional sensors to be able to detect vehicles in the rear as well as the sides of the car and to detect the lanes in the road. Furthermore some simple filtering needs to be investigated to be able to get some better speed estimations if high accuracy tracking is desired.

Bibliography

[1] L. Utz, "Lidar laser beam returns," 2009.

[2] U. Anhalt and et al., "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[3] T. Litman, "Autonomous vehicle implementation predictions," *Victoria Transport Policy Institute*, vol. 28, 2014.

[4] E. Guizzo, "How googles self-driving car works," *IEEE Spectrum Online, October*, vol. 18, 2011.

[5] R. Hudda and et al., "Self driving cars," *College of Engineering University of California, Berkeley, Berkeley: College of Engineering University of California*, 2013.

[6] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[7] R. Rasshofer and K. Gresser, "Automotive radar and lidar systems for next generation driver assistance functions," *Advances in Radio Science*, vol. 3, no. B. 4, pp. 205–209, 2005.

[8] C. Urmson and et al., "Tartan racing: A multi-modal approach to the darpa urban challenge," 2007.

[9] V. Hundelshausen and et al., "Driving with tentacles: Integral structures for sensing and motion," *Journal of Field Robotics*, vol. 25, no. 9, pp. 640–673, 2008.

[10] K. Takagi and et al., "Road environment recognition using on-vehicle lidar," in *Intelligent Vehicles Symposium, 2006 IEEE*, pp. 120–125, IEEE, 2006.

[11] W. Zhang, "Lidar-based road and road-edge detection," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pp. 845–848, IEEE, 2010.

[12] K. Peterson, J. Ziglar, and P. E. Rybski, "Fast feature detection and stochastic parameter estimation of road shape using multiple lidar," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 612–619, IEEE, 2008.

[13] S. Kammel and B. Pitzer, "Lidar-based lane marker detection and mapping," in *Intelligent Vehicles Symposium, 2008 IEEE*, pp. 1137–1142, IEEE, 2008.

[14] A. Sayar and et al., "Approaches of road boundary and obstacle detection using lidar," in *Control and Automation Theory for Transportation Applications*, vol. 1, pp. 211–215, 2013.

[15] J. Moras, V. Cherfaoui, and P. Bonnifait, "A lidar perception scheme for intelligent vehicle navigation," in *Control Automation Robotics & Vision (ICARCV), 2010 11th International Conference on*, pp. 1809–1814, IEEE, 2010.

[16] Z. Liu, D. Liu, and T. Chen, "Vehicle detection and tracking with 2d laser range finders," in *Image and Signal Processing (CISP), 2013 6th International Congress on*, vol. 2, pp. 1006–1013, IEEE, 2013.

[17] M. Kocamaz and F. Porikli, "Unconstrained 1d range and 2d image based human detection," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 645–652, IEEE, 2013.

[18] C. W. Hsu and et al., "A path planning achievement of car following in motion control via lidar sensing," in *Industrial Electronics and Applications (ICIEA), 2010 the 5th IEEE Conference on*, pp. 1411–1416, IEEE, 2010.

[19] D. Göhring and et al., "Radar/lidar sensor fusion for car-following on highways," in *Automation, Robotics and Applications (ICARA), 2011 5th International Conference on*, pp. 407–412, IEEE, 2011.

[20] "A low-cost, installable intelligent helper module for automobiles," *I.J. Intelligent Systems and Applications*.

[21] P. Wallich, "An early-warning system for your bike," *Spectrum, IEEE*, vol. 52, no. 7, pp. 22–23, 2015.

[22] I. Baldwin and P. Newman, "Road vehicle localization with 2d push-broom lidar and 3d priors," in *Robotics and automation (ICRA), 2012 IEEE international conference on*, pp. 2611–2617, IEEE, 2012.

[23] I. Baldwin and P. Newman, "Laser-only road-vehicle localization with dual 2d push-broom lidars and 3d priors," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 2490–2497, IEEE, 2012.

[24] Z. Chong and et al., "Synthetic 2d lidar for precise vehicle localization in 3d urban environment," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 1554–1559, IEEE, 2013.

[25] A. Üngör, "Kd-trees and range trees." `https://www.cise.ufl.edu/class/cot5520fa09/CG_RangeKDtrees.pdf`, 2009.

[26] E. W. Weisstein, "Least squares fitting," 2002.

[27] T. Rabbani, F. Van Den Heuvel, and G. Vosselmann, "Segmentation of point clouds using smoothness constraint," *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, no. 5, pp. 248–253, 2006.

[28] Z. Liu, D. Liu, and T. Chen, "Vehicle detection and tracking with 2d laser range finders," in *Image and Signal Processing (CISP), 2013 6th International Congress on*, vol. 2, pp. 1006–1013, IEEE, 2013.