Chau M. Nguyen

CONSTRUCTING DRIVABILITY MAPS FROM 3D LASER RANGE DATA FOR AUTONOMOUS VEHICLES

Abstract

The field of robotics is making successful progress towards endowing robots with human-like capabilities. One example is the well-known DARPA Urban Challenge, which demonstrated an autonomous ground vehicle with sophisticated software that could drive itself while preparing for interaction with other vehicles, human beings, and obstacles on the road. In the recent DARPA Urban Challenge, road blockage locations were unknown to contestants but pre-specified routes were provided that included all accessible road segments and other information such as waypoints, stop sign locations, and lane width. When driving a car in real life, human beings may not have knowledge of drivable road segments in advance. Instead, they make decisions on how to drive based on what they see at the moment. Hence, it is necessary for the autonomous vehicles to have the same capability to recognize the drivable roads at their current positions. Murarka and Kuipers presented a real-time stereo vision based mapping algorithm for identifying and modeling various hazards in urban environments by constructing a 3D model and segmenting it into distinct traversable ground regions [1]. However, their algorithm works in real-time for the robot's immediate vicinity of 10 x 10 meters. This is a relatively small space compared to our vehicle's immediate vicinity of 120 x 120 meters. The goal of this thesis is to present a revision of their algorithm that will allow the vehicle to discover accessible road segments along its path without relying on pre-specified routes. This will be particularly important in off-road situations where accessible road segments may not be pre-defined. This algorithm is expected to significantly improve the vehicle's performance in detecting and avoiding obstacles as well as navigating safely to the destination.

I. Introduction

This research is closely related to the Simultaneous Localization and Mapping (SLAM) problem in which an autonomous robot is programmed to build a map of an unknown environment and update this map during navigation. SLAM consists of two parts: localization and mapping. Most robots are equipped with a method of determining odometry, which describes the robot's relative position after every time step. However, odometry is rarely able to determine the exact location of the robot. Instead, it provides locations with uncertainties. Localization algorithms take into account odometry, sensor readings, and an environment overview to estimate the true location of the robot. Thus, the localization algorithm depends on the environment that is built by a mapping algorithm.

In order to construct a map of an unknown environment, the robot acquires data of the environment from its sensors. Sensors send out sensor signals and collect the returned signal information that represents locations of the reflected objects in the environment. Then, the occupancy grid mapping algorithm represents the map of the environment with reflected objects' positions in a two dimensional space (Figure 1).

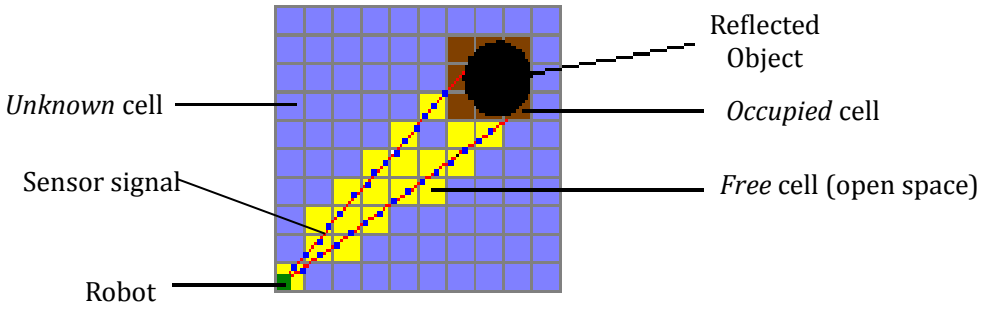

Figure 1

Each cell of the occupancy grid is *free, occupied,* or *unknown.* Initially, all cells in the occupancy grid are

marked as *unknown*. When the sensor signals return the coordinates of the reflected objects, the *ray-tracing* method is invoked to find the *open space* represented by the *free* cell. First, the r*ay-tracing* method sets the cells that contain locations of the reflected objects as *occupied*. Then, this method iterates through points (marked as blue dots in the Figure 1) along the rays that connect the coordinates of the reflected objects and the current position of the robot and marks these corresponding cells as*free*.

The occupancy grid mapping algorithm is implemented with a probabilistic method since sensor signals are also subject to noise and uncertainties. A cell can be marked as *occupied* by some previous sensor signals then marked as *free* later, and vice versa. To accommodate the uncertainties of sensor signals, each cell in the grid contains a value: 0 is *unknown,* a positive value is *occupied*, and a negative value is *free*. The *ray-tracing* method will update these values by adding a positive number to cells classified as *occupied* and a negative number to cells classified as *free*. At the end, we have the map represented by *free, occupied*, and *unknown* cells.

However, implementing a 2D occupancy grid on our autonomous vehicle demonstrates some weaknesses. First, 2D occupancy grids do not provide any information about the height of the reflected objects despite the fact that some sensors are capable of giving (x,y,z) coordinates. Thus, there is no distinction between a small object that the vehicle could drive over and a large object that must be avoided. In order to construct more detailed maps for a fully autonomous system, it is necessary to expand the occupancy grid to three dimensions. Many researchers have addressed 3D mapping, but they often have problems demonstrating that their algorithm can work in real-time on a fully autonomous system. In addition, a 3D occupancy grid might require a large amount of memory that could create a problem for some autonomous systems. The second weakness of the occupancy grid is that the *ray-tracing* method can become increasingly expensive with large numbers of sensor signals and a huge environment. Implementing a 3D occupancy grid is a challenge for our fully autonomous vehicle, which requires real-time computation. This motivates us to implement Murarka and Kuipers' mapping algorithm for wheeled robots.

Murarka and Kuipers proposed a mapping algorithm for the Intelligent Wheelchair research project. They focus on using the stereo vision camera as the primary sensor rather than the laser sensors. Instead of building a complete 3D model of the environment, their goal is to construct a hybrid 3D map of safe and unsafe regions for the robot's local surrounding environment. They introduce a fast method for segmenting ground surfaces by representing the point cloud data as planar regions. Their algorithm stores data points in an x-y grid and sets the highest point's z value as the cell height. Each planar region is defined to be an *h*-meter-high set of continuous cells. Then, the algorithm segments the grid into planar regions. With the assumption that the current position of the robot is the ground surface, the computer searches for adjacent planar regions of the ground surface that the robot can traverse. The set of all traversable planar regions from the robot's current position is called a *drivability map*.

Their algorithm for segmenting and constructing a *drivability map* is very fast and has been demonstrated to work well with the wheelchair robot. However, the local surrounding area of the wheelchair is only 10 x 10 meters. On the other hand, our autonomous vehicle has to deal with a local surrounding area of 120 x 120 meters. Furthermore, the vehicle also has to interact with many more fast-moving objects on the roads than the wheelchair does. My specific research contribution is to apply, extend, and implement a revised version on Murarka and Kuipers' algorithm to define drivable road segments and obstacles using the data points from our laser sensor so that it does not require excessive computations in order to process the 100,000 data points provided every $1/10^{th}$ of a second.

This thesis describes the implementation of our revision of Murarka and Kuipers' algorithm by setting up our occupancy grid and adapting their algorithm, which was originally designed for a small system with a stereo vision camera, to our autonomous vehicle with a laser sensor. There are three main differences in our revision of their algorithm. First, we skip the *ray-tracing* method in our algorithm. The r*ay-tracing* method was designed for robots that can only send out one sensor signal at a particular rotational direction and finds the *open space* at different distances by iterating through points along that sensor signal. On the other hand, at a particular rotational direction, our laser sensor sends out 64 laser rays that are capable of finding the *open*

space at many different distances. Moreover, Murarka and Kuipers' algorithm uses a stereo vision camera that gives a lot of noise in the data whereas our laser sensor gives highly accurate data points. In their work, they still have to perform the *ray-tracing* method to eliminate the noise. With the high accuracy of our laser sensor, we can safely assume that we will not have to worry about noise and errors. Second, we make our grid as a radial grid (r-θ grid) that resembles a Polar Coordinate system for the vehicle's local surroundings rather than the conventional x-y grid. Third, we tested the algorithm mostly in outdoor environments where the local surrounding area is as far as 60 meters away from the vehicle, which is much larger than the wheelchair's local surroundings.

II. Background

1. Overview of the laser sensor – Velodyne HDL-64E

The primary laser sensor of our autonomous vehicle is the Velodyne 3D HDL-64E lidar [9], which is mounted on top of the vehicle to collect detailed information of the car's surrounding environment. This state-of-the-art sensor spins and sends out laser rays to determine the 3D coordinates of the reflected objects. The HDL-64E operates on a rather simple premise: instead of a single laser firing through a rotating mirror, 64 lasers are mounted on upper and lower blocks of 32 lasers each, and the entire unit spins. This design allows for 64 separate lasers to fire thousands of times per second, providing exponentially more data points per second and a much richer point cloud than conventional designs. The unit inherently delivers a 360-degree horizontal field of view (FOV) and a 26.8 degree vertical FOV. Additionally, the state-of-the-art signal processing and waveform analysis are employed to provide high accuracy and extended distance sensing and intensity data. The HDL-64E is rated to provide usable returns up to 120 meters.

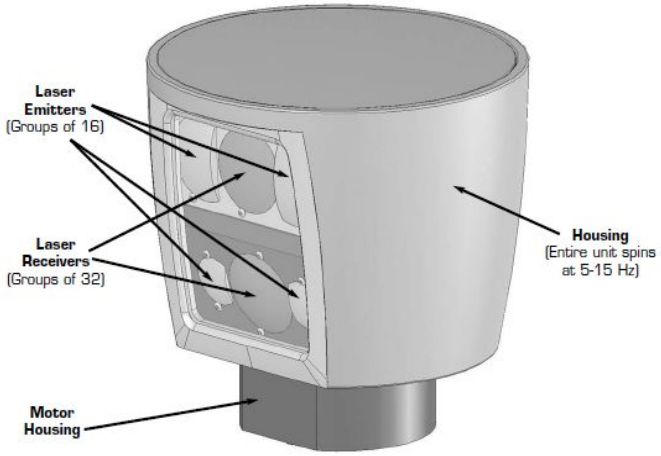

Figure 2

Based on many data sets that were collected from the Velodyne, we decided to use all data points within the radius of 60 meters from the vehicle and ignore anything outside that radius. Indeed, only a small fraction of the data points can go beyond 60 meters from the vehicle at any rotational direction. These data points are sparsely distributed and do not give us much useful information to analyze the accessible road segments. At the rate that we operate the Velodyne, it returns approximately 1 million data points in 1 second. The Velodyne finishes each revolution in $1/10^{th}$ of a second and returns approximately 100,000 data points. It provides much more data points than many sensors and gives us more detailed information about the car's local surrounding environment. However, this vast amount of data presents a computational challenge, especially since our autonomous vehicle emphasizes real-time computation.

2. Approaches

As mentioned in previous sections, we take a different approach in storing the data points into the radial

grid instead of the conventional x-y grid since we have to work with a larger local surrounding area. The local surrounding area of the car contains all data points within the radius of 60 meters from its current position. If we store the data points into an x-y grid, the size of the grid has to be 120 x 120 meters. If the size of each cell is 0.25 x 0.25 meters, the grid will have 230,000 cells. The cells that are farther away from the current position of the car will have less data points and will be more likely to be empty. Storing an x-y grid of 230,000 cells wastes most of the allocated memory since the Velodyne returns approximately 100,000 data points each revolution and, at most, only 100,000 cells (less than half of the grid size) have usable information. In addition, the data points that are farther away from the car spread out more than the data points that are close to the car. Keeping the same cell size will leave a lot of empty cells that disconnect these points from the rest of the environment. Hence, we store data into a radial grid with 64 rows and 360 columns. 64 rows correspond to 64 laser rays of Velodyne and 360 columns represent the rotational angle of these 64 laser rays. The cell sizes get larger as they are farther away from the origin. This approach allows us to store a smaller size grid, 64 by 360, which has about 23,000 cells and is only 10% of the x-y grid approach. Plus, this approach minimizes the potential problem of having many empty cells and prevents data points that are farther away from being disconnected from the rest of the environment.

Since the primary goal of this research is to allow the vehicle to make optimal decisions at its current position, the computer only needs to keep its *local drivability map*. Every time the vehicle moves to a different position, it does not need to take previous *drivability maps* into consideration. Thus, at the end of each of Velodyne's revolutions, a new *local drivability map* will be created and incorporated with the path-planning algorithm to make the optimal decision. This map will not be reused for the next Velodyne revolution. Although keeping the *local drivability map* rather than the *global drivability map* might increase computational time and reduce accuracy, the algorithm is expected to run fast enough to overcome the increase in computational time and, with the vast amount of data from the Velodyne, reconstructing new *local drivability maps* each time still provides highly accurate results. This approach saves us from keeping the *global drivability map* that might accumulate a large amount of memory storage over time.

Our high-level approach is after the laser sensor finishes one revolution: the computer (1) collects detailed sensor information from the car's immediate vicinity and puts these data points into a radial grid, (2) applies this algorithm to determine accessible roads and obstacles then constructs a *drivability map*, and (3) makes an optimal choice from the current position of the vehicle to the destination. In order for this approach to be feasible, the entire process must run in real-time to guarantee that the vehicle's motion will never be interrupted by incomplete computations of the algorithm.

III. Related works

This research project is related to two research interests in Computer Science. The first one aims at using mobile robots with laser range finders to reconstruct a 3D model of an unknown environment. The second research interest is the DARPA Urban Challenge that applies a model of the environment to determine the autonomous vehicle's motion.

1. *3D model reconstruction*

There has been a huge interest in computer graphics to develop many algorithms that fully reconstruct the surfaces of 3D objects given their Point Cloud Data (PCD). Hahnel, Burgard, and Thurn [2] published their paper that applies the 3D surface reconstruction to build 3D models. In contrast with the surface reconstruction algorithm in computer graphics, the robot's sensors produce noise and errors in the data. They focus on using a mobile robot with laser range finders to collect PCD that automatically generates simplified 3D models of indoor and outdoor environments. Their method is expected to be a great utility for architecture, the video game industry, or emergency crews operating at a hazardous site. It would reduce human intervention in reconstructing 3D objects and, thus, reduce cost as well as danger. Their algorithm records data points from the robot's laser sensors. Then, they obtain a polygonal model by connecting the consecutive 3D points that are close to each other. This polygonal model is an approximation of the environment. Finally, they apply the *planar approximation* to create the smooth surfaces. This *planar approximation* starts with a randomly chosen

point in the 3D region and finds the maximum set of points in the neighborhood that can fit into a plane. The optimal plane for a set of points minimizes the sum of squared distance from points to the plane. They continuously merge the neighboring polygons in the same plane to the larger polygons until no more polygons can be merged. At the end, they obtain a simplified 3D model of the environment with smooth surfaces. The authors state that for a typical data set consisting 200,000 surfaces, a naïve implementation on a standard PC requires over 10 hours to extract all planes. We will not apply this algorithm for our vehicle that requires real-time computations. However, we will apply their *planar approximation* method to find the best fitting plane for a set of continuous data points.

In *Leaving Flatland: Toward Real-time 3D Navigation* [3], the authors present a 3D mapping algorithm that builds a polygonal model of the world using a RHex six-legged walking robot with stereo vision cameras developed by Boston Dynamics. Their algorithm decomposes 3D workspace into local 2D regions. For each PCD returned from the cameras, they construct a local octree. This octree will be divided into the cells whose sizes are no smaller than *h*. Then, in each cell of a local octree, they examine the set of points to which they will fit a plane and define a polygon. After constructing the polygons, they will find all overlapping octree cells and compare all colliding polygons. If two polygons are similar, they will compute a new polygon that is the average and best fit for them. This algorithm focuses on building a 3D map. Their system might stand for a few seconds to gather information and build the map. Building and merging polygons would provide a good approximation for the 3D model of the environment. Compared to our vehicle, which might be moving at high speeds while gathering data and computing desired output, we may not have enough time to build and merge all polygons. The authors state that their algorithm would build about ten thousand polygons before merging them, which is very difficult for our fast-moving system to process. Furthermore, their RHex six-legged robot has the capability of running at a high speed on flat ground, crawling over mountains, and climbing stairs to collect data points at any surface. On the other hand, our vehicle can only drive on flat or gradual inclined surfaces and may not gather enough data points to build a complete 3D model. Therefore, instead of building a complete 3D model of the vehicle's surrounding environment, we will concentrate on recognizing the drivable road segments of the local map.

2. *DARPA Urban Challenge*

In ten Urban Challenge papers that were published in *Journal of Field Robotics*, 2008, none of them mentioned that their algorithms would build a complete 3D model of the vehicle's surroundings. Alternatively, they provide a 2D map with additional information about traversable ground regions.

In the 2007 DARPA Urban Challenge, the Cornell team [5] and MIT team [6] took a similar approach in building the local map of the vehicle's surrounding environment. The Cornell team constructed the *vehicle-centric map* of the local surroundings of the autonomous vehicle. The MIT team built the *local frame* of the vehicle's surroundings. Both the *vehicle-centric map* and the *local frame* are in the Euclidean coordinate system, which is different from our Polar Coordinate *local drivability map*. Both teams combined data returned from many different sensors and resolved conflicts to update their local maps to find the traversable ground regions.

The Stanford's Junior team [7] relied on the Velodyne as the primary sensor for obstacle detection. They took into account the expansion and compression of concentric rings of data points. They computed the expected distance between these rings of data points on flat ground with no obstacles to measure the terrain slope. This approach is very similar to our approach of constructing the radial grid using the distance between the laser rays and the rotational angles. They still kept the environment's global map but addressed its downside for accumulating static data over time. To accommodate this problem, their algorithm performed a *local visibility calculation* to detect roads and obstacles.

The AnnieWay team [8] took a completely different approach. They constructed global maps instead of local maps. They used a grid that always centered at the vehicle's position and aligned it with a global coordinate system. They integrated the data from laser scanners in a 3D environment into a global 2D occupancy grid map and computed the evidence if a cell in the grid was an obstacle. Hence, the vehicle built a 2D map of the global environment while navigating.

IV. The algorithm

1. Setting up the data structure

Each data point *p* has the form of *(x, y, z)*. Each cell on the grid contains the highest point and its *segment ID*. All data points will be put into an r-θ grid in which each row represents distance from the center and each column represents the rotational angle. This is a different approach of storing data points compared to Murarka and Kuipers' algorithm, which stores data points in an x-y grid. Since there are only 64 laser rays fired at each rotational angle, our grid consists of 64 rows. Each row is expected to contain one concentric ring of data points in the absence of obstacles on flat ground. Each column represents a rotational angle of one degree and is expected to store all data points fired from 64 laser rays at that rotational angle.

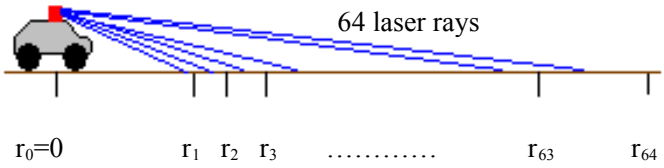

Figure 3

As the laser rays move further out, the cell size is bigger since the difference between the consecutive elements of *r_bench* will get larger, and, therefore, we will begin to lose accuracy. However, with a large number of data points from the Velodyne, any obstacle within the radius of 60 meters from the car will be detected. We will not worry about obstacles that are farther than 60 meters because the Velodyne will eventually detect those obstacles when the vehicle moves closer to them.

Each data point in one revolution of the Velodyne is converted from the Velodyne's frame of reference to the vehicle's frame of reference and processed with the following steps:

(1) Compute the distance *d* of this data point from the current position of the car using its x and y values. If *d* is greater than 60 meters, this data point will be dropped and no further processing needed. Otherwise, *d* is used to determine the row index of this data point.

(2) Compare the distance *d* with each element of *r_bench* array to find the row index of this data point. Row index of the data point is *i* if *d* is greater than *r_bench[i]* and less than *r_bench[i+1]*.

(3) Convert the data point's rotational angle from radians to degrees to find its column index.

(4) Use the row and column index to locate the cell and add this data point to this cell.

Each cell of the grid keeps the data point with highest z value and uses it to define the cell's height. When a data point is added to a particular cell, this cell processes the point by doing one of the following:

Case 1: If the cell is *empty* (has no data point), the cell is set to *not empty* and the height of the cell is the z value of the data point that is just added. This data point is stored as the highest data point of the cell.

Case 2: If the cell is *not empty*, the data point's z value is compared with the height of the cell. If this point's z value is less than or equal to the cell's height, this point is ignored. If this point's z value is greater than the cell's current height, the cell replaces its height with the new data point's z value and its highest point with this data point.

After all data points in one revolution of the Velodyne are added, the grid is a hybrid 3D model of the vehicle's *local surroundings*. Each cell's height is the highest z value of all data points added to that cell. The cell's height is the key information for segmenting the 3D model and constructing the *drivability map*.

2. Overhanging objects problem

Overhanging objects are a problem that Murarka and Kuipers do not mention in their paper. However, it is necessary to address this problem since overhanging objects might interrupt the vehicle's motion even if the vehicle can safely drive under them.

In the radial grid, each cell keeps its highest point that is used for segmenting the 3D model; thus, there is

no distinction between an overhanging object of height *h* and an obstacle of height *h*. Both of them are automatically classified as obstacles of height *h*. In some cases, an overhanging object of height *h* is high enough for the car to continue driving forward while an obstacle of height *h* blocks some parts of the road. Thus, we reconsider how high the overhanging objects can be so that the car can drive under them.

The vehicle can only drive under the overhanging objects if they are higher than the car, 2.2 meters. There are four cases shown in Figure 4: the obstacle is higher than the car (top-left), the overhanging object is higher than the car (top-right), the obstacle is lower than the car (bottom-left), and the overhanging object is lower than the car (bottom-right).

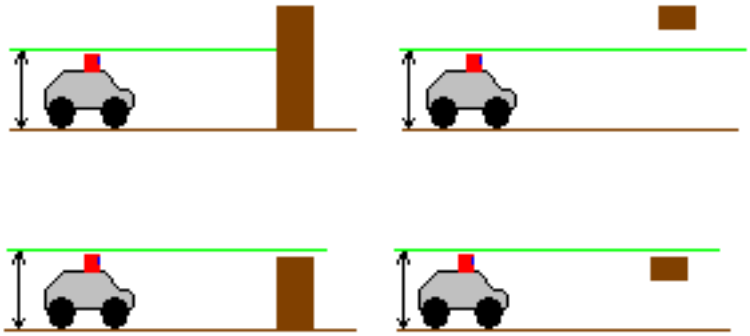

Figure 4

As demonstrated in Figure 4, the vehicle can continue driving forward when the overhanging object is higher than the vehicle (top-right). We define 2.5 meters, which is slightly higher than the car, as the cut-off height. Any data points that are higher than 2.5 meters are ignored. The computer stores the data points that are equal or below 2.5 meters. The vehicle only sees obstacles and overhanging objects below 2.5 meters and considers them as road blockages. Any overhanging objects above 2.5 meters are not considered since the car can safely drive forward without recognizing the existence of these overhanging objects. Obstacles above 2.5 meters would have the height of 2.5 meters, which makes no change in the vehicle's decision not to drive forward. This height cut-off technique saves the computer from the expensive *ray-tracing* method and quickly distinguishes obstacles and overhanging objects.

Setting up the radial grid
Input: S = set of data points of (x,y,z,Θ) form in one Velodyne's revolution
Output: grid = radial grid of vehicle's local surroundings in which each cell stores the highest data point

```
For each point p(x_k,y_k,z_k,Θ_k) in S
{
        d = SquareRoot(x_k*x_k + y_k*y_k);      //compute distance of point p from vehicle's position
        if (d > 60) continue;                   //ignore point more than 60 meters away
        if (z_k > 2.5) continue;                //ignore point higher than 2.5 meters
        row_index = -1;
        for j=1 to 64                           //find the row index
        {
                if (d<=r_bench[j])
                {
                        row_index = j;
                        break;
```

```
                }
        }
        new_Θ = (180/π)*Θk;              //convert rotational angle from radian to degree
        if (new_Θ>360)
          new_Θ =  new_Θ – 360;
        if ( new_Θ<0)
          new_Θ =  new_Θ + 360;
        column_index = floor(new_Θ);
        adjustHeight( grid[row_index][column_index],  p ) //add point p to the cell
}

function adjustHeight
Input: C = cell of the radial grid,
       p = data point of (x,y,z,Θ) form
Output C = cell of the radial grid with highest_point and unit_height readjusted
highest_point = data point with the highest z values added to C
unit_height = the highest point's z value in term of unit height (0.25 meters)
if (C is empty)
{
        highest_point = p;
        unit_height = floor(p.z/0.25);              //Calculate unit height of the cell
}
else
{
        if (highest_point.z < p.z)
        {
                highest_point = p;
                unit_height = floor(p.z/0.25);      //Calculate new unit height of the cell
        }
}
```

3. Segmenting

After putting all data points into the radial grid, the *segmenting* procedure is invoked. This *segmenting* procedure finds the potentially reachable cells from the vehicle's current position then separates these cells into segments. Each segment is a 0.25-meter-high set of adjacent cells. We refer 0.25 meters as 1 *unit height*. Thus, all cells in the same segment have the same *unit height* although their height difference can be up to 0.25 meters. And cells are considered as potentially reachable from each other if they are adjacent and their height difference is 1 *unit height,* from 0.25 meters to 0.5 meters.

All potentially reachable cells are kept in the *neighbor list* that is initially empty. We start exploring the local map by segmenting it from the current position of the car, namely cell *(0,0)* of the r-θ grid. Cell *(0,0)* is the first cell added to the *neighbor list* and *current segment* is initialized to 0. All cells in the grid have their *segment ID* initialized to -1.

For each cell *(r,t)* in the *neighbor list*, the *recursive segmenting* procedure is called with the value of the *current segment* to examine the adjacent cells of *(r,t)*. Each adjacent cell of *(r,t)* falls into one of the following three categories:

(1) If the adjacent cell has the same *unit height* with *(r,t)*, we set this cell's *segment ID* to the value in *current segment* and call the *recursive segmenting* procedure on this adjacent cell.

(2) If the difference in *unit height* between *(r,t)* and the adjacent cell is 1, we append this adjacent cell to the *neighbor list* to examine after we finish with the *current segment*.

(3) If the difference is more than 1 *unit height*, we ignore this cell since it is too high for the car to move

from its current position.

The r*ecursive segmenting* procedure exits when all cells that belong to the *current segment* are found. Then, we look for next cells in the *neighbor list* to find a new cell *(r,t)* that is not assigned to any segment, increment *current segment* by 1, and call the *recursive segmenting* procedure with the new cell *(r,t)* and the updated *current segment*. We only examine the next cell that is not in any segment since one cell can appear multiple times in the *neighbor list* or some of the cells in the remaining *neighbor list* are already assigned to some segments.

When all the cells in the *neighbor list* get examined, we will have the potentially drivable segments that are adjacent to each other and the height of two adjacent segments will not exceed 1 *unit height*. The remaining cells that do not belong to any segment are either empty, disconnected from the current position of the car, or their height is much higher than the potential drivable segments. Empty cells are considered as the *unknown* regions where the algorithm does not apply the next steps: plane fitting and *drivability map* construction. For any non-empty cells that are disconnected from the vehicle or have obstacles, we increment the number of segments by 1 and set the *segment ID* of these non-empty cells to the last segment, which is reserved for the unreachable cells. The plane fitting and *drivability map* construction procedures are not applied to the last segment.

4. Plane fitting

The plane fitting procedure takes the set of cells in each segment and returns the best fit 3D plane for the highest data point in each cell. We assume that the car always drives on a flat surface. Therefore, it is necessary to fit a plane into the cells that are in the same segment to determine whether the car can drive through these cells. Given the list of all cells that belong to a segment, to find the best fit plane, we extract the highest point stored in each cell, and add these points to the list of points, which is used to find the equation of the plane. Then, for each segment that has 3 or more cells, we iterate through all of these data points and apply the least square formula to get the 3D plane equation for that segment.

$$\begin{vmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & \sum 1 \end{vmatrix} * \begin{vmatrix} a \\ b \\ c \end{vmatrix} = \begin{vmatrix} \sum z_i x_i \\ \sum z_i y_i \\ \sum z_i \end{vmatrix}$$

$$A * X = B$$

$$X = A^{-1} B$$

We obtain the 3D plane equation $z = ax + by + c$ by solving the matrix equation above. We store each output a, b, and c in an array of size 3 for all segments in a list called *planes* that are indexed by the segment number. Each cell can easily access its plane equation by indexing its *segment ID* in the *planes* list. The equations of the planes determine the drivability of the segments.

```
Function Plane_fitting
Input Cell_list = list of cells belong to segment k
Output X = array of [a,b,c] form represented the plane equation z = ax + by + c
-----------------------------------------------------------------------------
A = 3 by 3 matrix, initialized to 0;
B = 3 by 1 matrix, initialized to 0;
if (size(Cell_list)<3) X = [0,0,0]
else
{
    for highest point p(xi,yi,zi,Θi) in each cell C of Cell_list
    {
        A1,1 = A1,1 + xi*xi;
```

```
        A1,2 = A1,2 + xi*yi;
        A1,3 = A1,3 + xi;
        A2,2 = A2,2 + yi*yi;
        A2,3 = A2,3 + yi;
        A3,3 = A3,3 + 1;
        B1 = B1 + zi*xi;
        B2 = B2 + zi*yi;
        B3 = B3 + zi;
    }
    A2,1 = A1,2;
    A3,1 = A1,3;
    A3,2 = A2,3;
    X = inverse(A)*B
}
```

5. Local drivability map construction

After fitting a 3D plane to each segment, a *local drivability map* is constructed. This *local drivability map* contains the set of segments that are *drivable* from segment *0*. A segment is defined to be *drivable* from another segment by three criteria:

1) The height difference of two adjacent segments' boundary cells is no more than 1 *unit height*.

2) When fitting the plane equation of segment s_1 to the boundary cells of segment s_2, the difference in z values does not exceed 1 *unit height*.

3) All continuous boundary cells of two adjacent segments have to be at least 3 meters wide so that the car can drive from one segment to the other.

If all three criteria above are met, a segment is *drivable* from another segment and we mark these segments as *drivable* from one another.

After identifying *drivable* segments using the criteria above, we start from segment *0*, which is the current position of the car, and look for the segments that are *drivable* from segment *0*.

Step 1: Add segment 0 to the initially empty queue.

Step 2: Extract a segment *s* from the queue, look for all segments that are *drivable* from segment *s*, and add all of them to the queue. Then, rename segment *s* as *0*. Step 2 is repeated until the queue is empty.

When the queue is empty, the *local drivability map* is constructed from all the segments that are *drivable* directly or indirectly from segment 0 and renamed as segment *0* using the procedures above. This *local drivability map* will be incorporated with the state-of-the-art path-planning algorithm to find the optimal driving direction.

V. Experiment and evaluation

Our experiment and evaluation has two parts: demonstrating the correctness of the implementation that uses the radial grid and adjusting parameters so that the algorithm can run under 0.1 seconds. We demonstrated the correctness of the implementation by taking snapshots of various locations including flat ground surface, hills with small inclined angles, and roads with overhanging tree branches. In each case, we expected this algorithm to identify the flat ground that the vehicle can drive to. Then we compared it with the existing algorithm implemented in the car, the *height difference* algorithm. The *height difference* algorithm classifies a cell in the grid as an obstacle when the height difference between the highest and lowest data points is above a certain threshold. Finally, we measured the running time of the algorithm when the car completely constructs the *drivability map* for each location and considered readjusting the size of the grid's column to achieve faster computation.

In all pictures, the blue region is the *drivable* part of the road, black regions are obstacles, and all other regions are *non-drivable*.

Figure 5 (a)

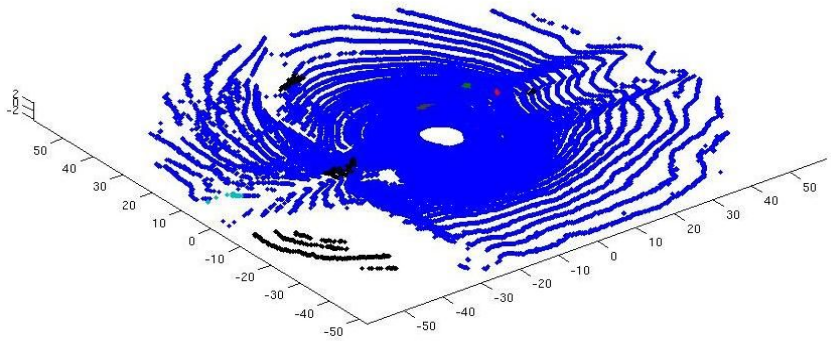

Figure 5 (b)

We began our experiment with an easy scenario in which the vehicle was on a flat ground surface and there were not many obstacles around it. In this easy case, the vehicle quickly computed the *local drivability map* that contains most of its local surroundings. Since there was not any significant difference in the height of the local surroundings, there was a small number of segments in the maps. Thus, it did not take much time to merge segments together and construct a large drivability map (Figure 5 (a) and (b)).

Next, we took the car to a road that has both upward and downward slope surfaces (Figure 6 (a)) and expected the drivability map to include all road surfaces.

Figure 6 (a)

This algorithm captures very well almost all road surfaces including downward and upward sloping surfaces of the road in Figure 6 (b).

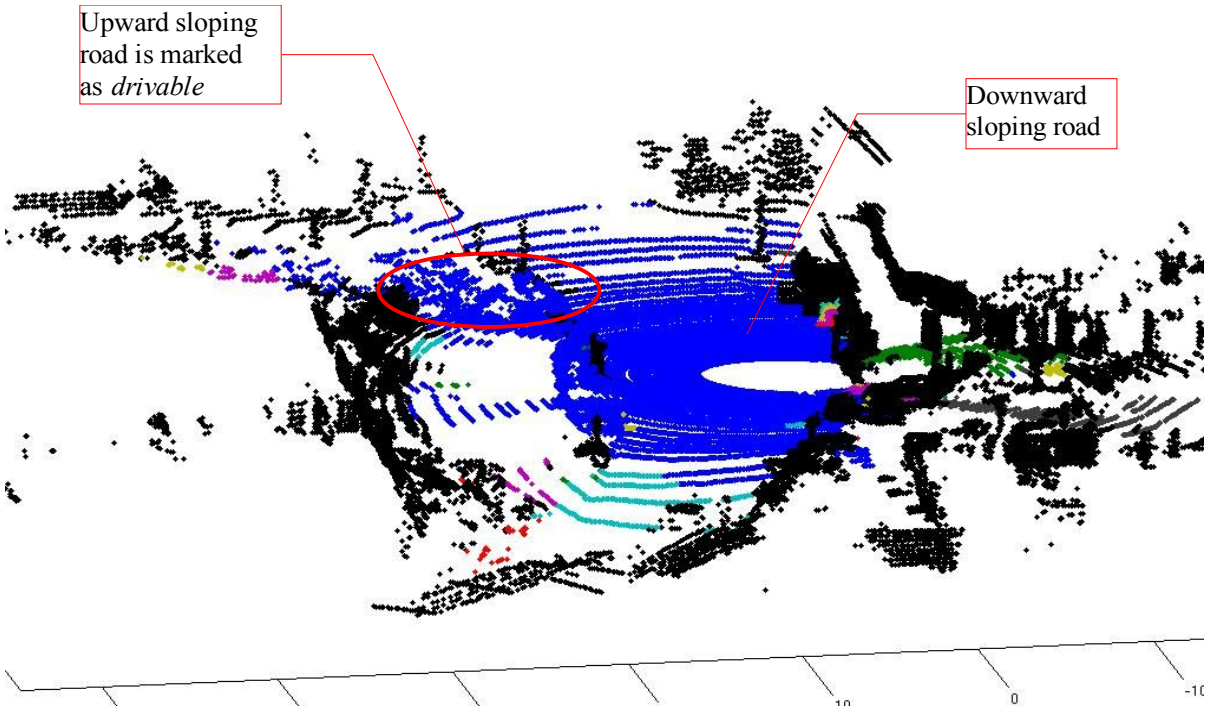

Figure 6 (b)

Compared to the *height difference* algorithm implemented in the car, the old algorithm failed to define the upward sloping road as *drivable* in Figure 6 (c) where the *drivable* region is colored in blue and the *non-drivable* region is colored in red.

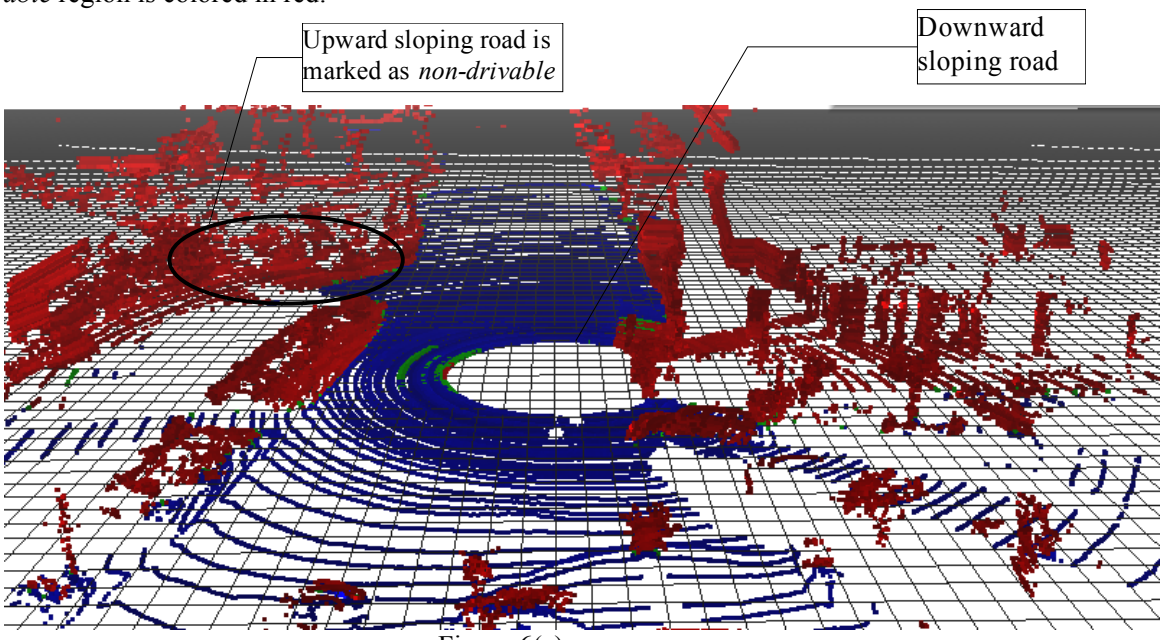

Figure 6(c)

Then we took the car to a road where there were fences, walls, or trees on either side. The vehicle was under some overhanging tree branches. We expected this algorithm to define the road and the parts under the tree branches as *drivable*.

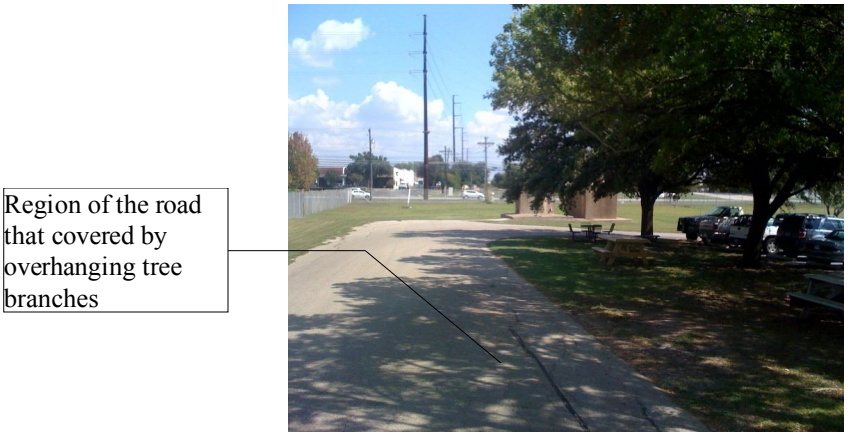

Figure 7 (a)

This algorithm effectively identified the road under the tree as *drivable* in Figure 7 (b). This algorithm defined everything behind the fence as a *non-drivable* region. This is a reasonable and expected result since the vehicle cannot reach the region behind the fence. In addition, it captured the curve of the road very well.

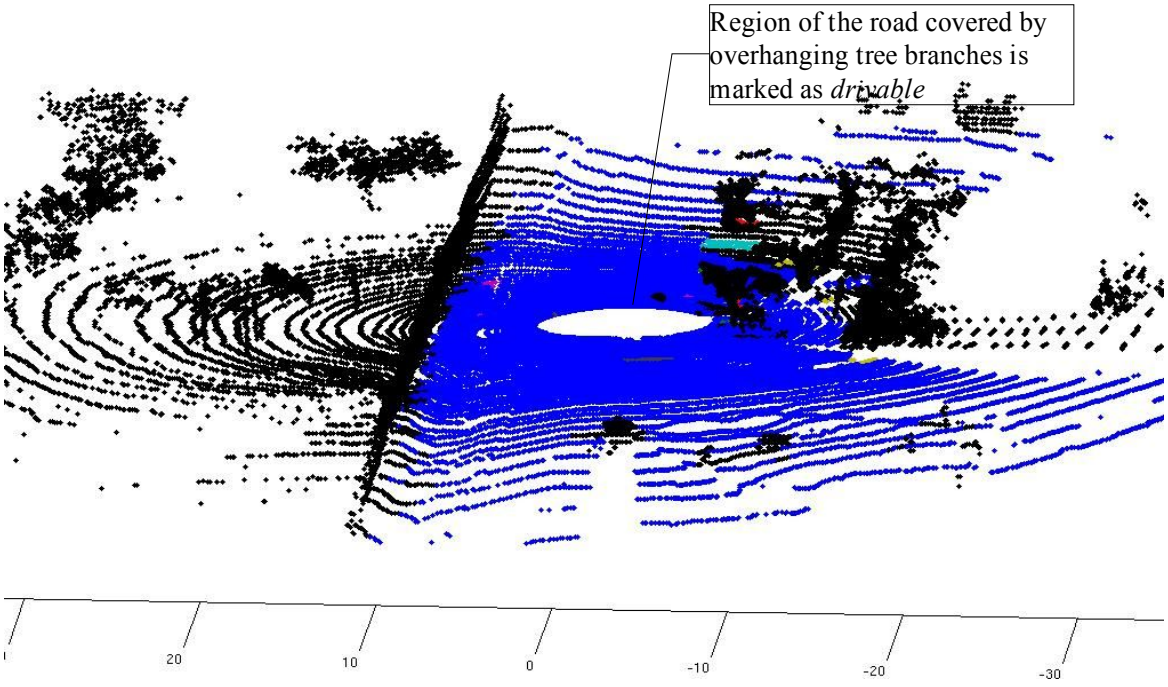

Figure 7 (b)

In Figure 7 (c), the *height difference* algorithm also failed to define the region of the road under the overhanging tree branch as *drivable*. The *height difference* algorithm classified the segment behind the fence as *drivable* but the vehicle can never reach it.

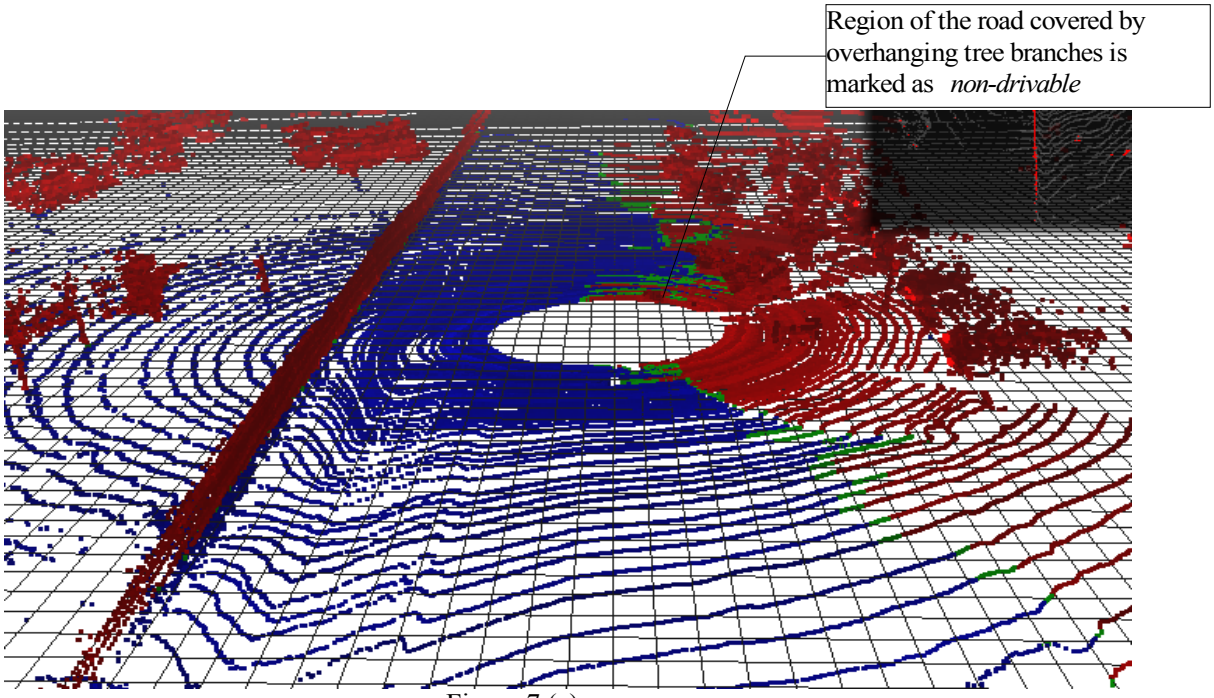

Figure 7 (c)

To further demonstrate that this algorithm can correctly identify the overhanging objects lower than 2.5 meters as obstacles and marks everything behind the obstacles as *non-drivable* (not in blue), we manually added some data points into Figure 7. It produced the desired drivability map in which the blue segment stops at the position of the overhanging object (Figure 8).

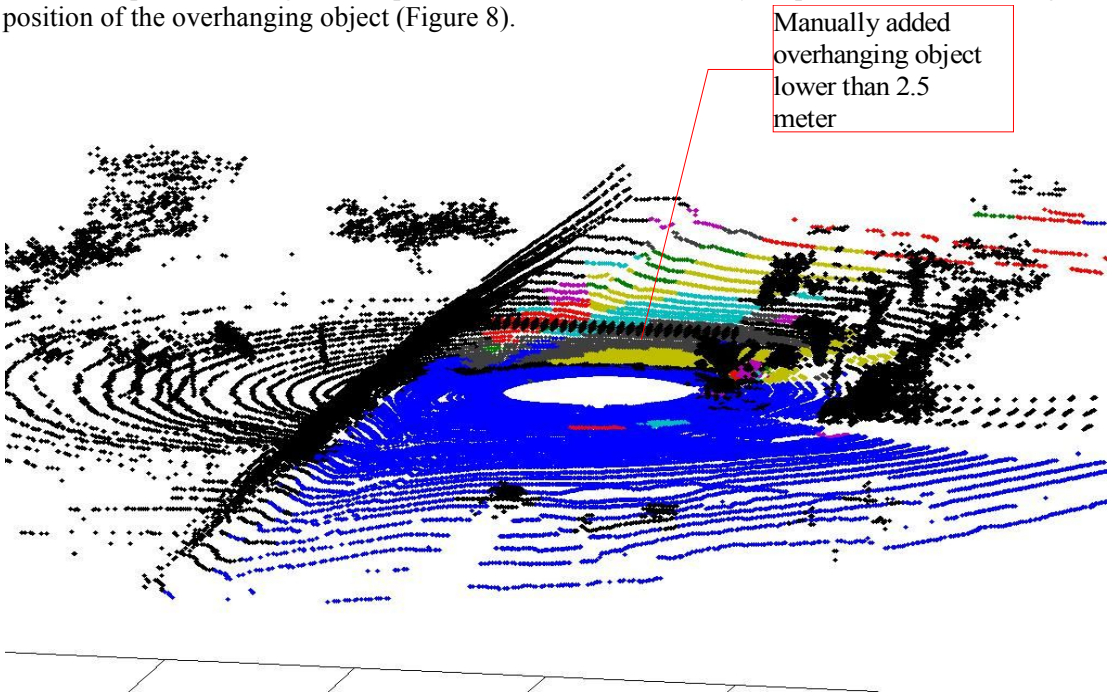

Figure 8

The percentage of the *drivability map* size in the local map varies depending on the vehicle's local surroundings. The more obstacles that existed in the local surroundings, the smaller the size of the drivability map became. Overall, the algorithm correctly defines the *drivability map* by recognizing the continuous road segments. It adapts well to the change in the road's slopes and curves.

Figure	Description	Drivability map size
Figure 5	Flat ground with no obstacles	98.90%
Figure 6	Downward slope with obstacles on both sides	63.40%
Figure 7	Flat road with obstacles on both sides	78.30%
Figure 8	Figure 8 with an overhanging object added in front of the vehicle	37.00%

The running time was measured by executing the program on the vehicle's computer. The running time of the algorithm also varies depending on the grid column size and the local map. With the grid's column size of 1 degree, it is not possible for the algorithm to run in real-time. However, when we increase the column size to 2 and 4 degree, it is more likely that the algorithm will run in real-time.

Size of radial grid's column (degree)	Running Time (second)
1	0.09 – 0.13
2	0.05 – 0.08
4	0.03 – 0.06

VI. Future Work

The remaining work of this project is to integrate the *drivability map* with the existing path-planning algorithm. With the *drivability map* being constructed, we reused the path-planning algorithm that was already implemented in the vehicle using a Voronoi-style skeleton [4]. Given any two points in a graph, the Voronoi diagram is constructed by the set of points that are equal distance to these two points. Thus, instead of finding the set of points that are equal distance to any two points, the computer finds the set of points that are equal distance to any obstacles. We call these points the *safe points*. This set of *safe points* determines the path for the vehicle to move from one point to another without hitting any obstacle. The distance from these points to the obstacles is called the *safety radius*.

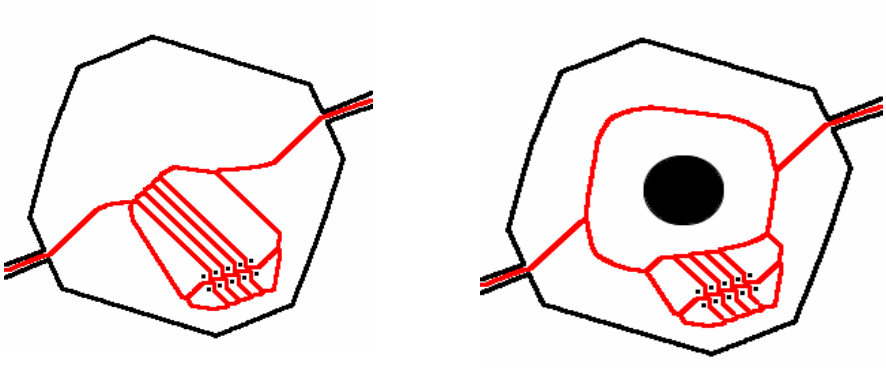

Figure 9

In the figure above, the red lines represent the set of points that are equal distance to any obstacles, and we call it the *safe path*. To move to the destination, the vehicle chooses which direction to drive based on how far

it must follow the *safe path* to reach the destination. Every time the vehicle moves, its location is updated on the *safe path*. The vehicle's location must be within the *safety radius* of its closest *safe point.* Then, the vehicles recomputes the next *safe point* on its *safe path* and drives to that point until the *safe point* is reasonably close to the destination. This state-of-the-art path-planning algorithm worked smoothly on our vehicle in 2007 DARPA Urban Challenge.

Although we tested the algorithm in various locations, there were not many interactions among the vehicle and other obstacles on the roads. We have not taken the vehicle out to roads that allow high speeds such as freeways and highways where the demand for quick responses to dynamic obstacles on the road is higher. It is also necessary to test the vehicle performance in places where there is no pre-defined road segments and the vehicles will have to truly rely on the algorithm to define the road. It might be useful to acquire more data from other laser sensors on the vehicle rather than using only data from Velodyne. More data from sensors will improve the accuracy of the *local drivability map*. However, this will again increase the amount of data, which will raise the computational challenge.

VII. Conclusion

Murarka and Kuipers' algorithm has been proven to work in real-time. However, the local surrounding area of their mobile robot is only 10 x 10 meters. This is a relatively smaller space than our autonomous vehicle's local surroundings of 120 x 120 meters. My research applies and extends their algorithm with three main differences: (1) the *ray-tracing* method is skipped due to the sufficient amount of data that is given by the Velodyne, (2) the local surrounding environment is represented by an r-θ grid rather than an x-y grid, (3) we tested the algorithm in mostly outdoor environments where there are many significant height differences among many parts of the vehicle's local surroundings. In addition, my research addresses the problem of overhanging objects and proposes a quick and simple solution that still produces the desired result. Compared to the *height difference* algorithm previously implemented in the vehicle, this algorithm is expected to improve the vehicle's performance in detecting obstacles, defining *drivable* regions, and navigating safely to destinations while interacting with other obstacles on the road in real-time. This is the important progress towards having fully autonomous vehicles in urban traffic.

VIII. Acknowledgments

Many thanks to Professor Peter Stone, Dr. Patrick Beeson, Dr. Aniket Murarka, Dr. Micheal Quinlan, and Jack O'Quin for their help and valuable instructions. My research is supported by the College of Natural Science's Summer 2009 Research Fellowship and the Computer Science Department's Undergraduate Research Opportunity Program Fall 2009.

REFERENCES

[1] Aniket Murarka, Benjamin Kuipers, “A Stereo Vision Based Mapping Algorithm for Detecting Inclines, Drop-offs, and Obstacles for Safe Local Navigation,” IROS, 2009.

[2] Dirk Hahnel, Wolfram Burgard, Sebastian Thrun, “Learning Compact 3d Models for Indoor and Outdoor Environments with a Mobile Robot,” 2003.

[3] Benoit Morisset, Radu Bogdan Rusu, Aravind Sundaresan, Kris Hauser, Motilal Agrawal, Jean-Claude Latombe, Micheal Beetz, “Leaving Flatland: Toward Real-Time 3D Navigation.”

[4] Patrick Beeson, Jack O' Quin, Bartley Gillan, Tarun Nimmagadda, Mickey Ristroph, David Li, Peter Stone, “Multiagent Interactions in Urban Driving,” Journal of Physical Agents: Multi-Robot Systems, Vol. 2, No. 1, March 2008.

[5] Isaac Miller, Mark Campbell, Dan Huttenlocher, Frank-Robert Kline, Aaron Nathan, Sergei Lupashin, Jason Catlin, Brian Schimpf, Pete Moran, Noah Zych, Ephrahim Gracia, Mike Kurdziei, Hikaru Fujishima, "Team Cornell's Skynet: Robust Perception and Planning in an Urban Environment," Journal of Field Robotics, 2008.

[6] John Leonard, Johnathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Flore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, Matthew Walter, David Barret, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, "A Perception-Driven Autonomous Urban Vehicle," Journal of Field Robotics, 2008.

[7] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Issac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, Sebastian Thurn, "Junior: The Stanford Entry in Urban Challenge," Journal of Field Robotics, 2008.

[8] Soren Kammel, Julius Ziegler, Benjamin Pitzer, Moritz Werling, Tobias Gindele, Daniel Jagzent, Joachim Schroder, Michael Thuy, Matthias Goebl, Felix von Hundelshausen, Oliver Pink, Christain Frese, Christoph Stiller, "Team AnnieWay's Autonomous System for the 2007 DARPA Urban Challenge," Journal of Field Robotics, 2008.

[9] HDL-64E Manual.