Traffic Sign Detection

1 Velodyne Sign Detection
Steve Cahail
Jesse Vera
Tony Daddeo

2 3D Point Cloud Projection
Nick Shelton

3 Computer Vision
Adam Menz
Jonathan Long

CS 378 Autonomous Vehicles in Traffic
Dr. Michael Quinlan
Spring 2010

Abstract

The goal of this project is to enable autonomous vehicles to detect traffic signs with no prior knowledge of their locations, and to do so using computer vision and Velodyne laser data. We have developed an algorithm that detects objects using the Velodyne HDL-64 LIDAR (herein referred to as the “Velodyne”) and returns a new point cloud containing only “regions of interest” to the visual sign detection group for determination of whether or not the objects are actually signs. The projection group then takes an incoming point cloud and maps each point to its corresponding location on a camera image. In this context, we are taking points that represent locations of possible signs and finding their location in an image produced by a camera mounted on the top of the car. The technique involves a translation and rotation matrix (camera transform) and a 3D projection of the transformed points. Finally, the part of the image the point represents is cropped and then sent for further analysis. The method for recognizing signs uses camera data and vision techniques. By using color and shape information received from the cameras, then a template matching procedure, we are able to detect and recognize signs fairly accurately. This information can then be used to alter the behavior of the vehicle, and will allow the car to drive safer in the autonomous mode.

Introduction

Marvin is an autonomous vehicle at The University of Texas capable of driving in urban environments. Marvin currently receives all of its information about traffic signs from an RNDF file, or Road Network Definition File. This file contains data about all the roads that the vehicle will be driving on, including the location and classification of traffic signs. This method works, but can be significantly improved by real time sign detection. The RNDF method puts limitations on the abilities of the car by restricting it to areas that have an accurate RNDF defined. If RNDF does not exist then the car cannot operate in this area, and this limits the usefulness of this technology. In the same way, if the car enters an area without sign data in the RNDF, the vehicle essentially becomes blind to all traffic signs. In areas with sign data in the RNDF this approach can also be problematic. If traffic signs are changed or new signs are installed, the robot will need to be updated with these changes. In the event that the car is not updated, this would result in a very unsafe situation. In order to improve the functionality of Marvin, we wanted to implement a system that is able to detect signs in real-time as the car is driving.

Our approach is different than many other sign detection procedures, mainly because we have the advantage of having laser data in addition to camera data. We are using a Velodyne and cameras to detect traffic signs. A Velodyne is a system of lasers that scans its surrounding area and gathers 3D data of this area. There is a Velodyne mounted on top of Marvin and it is currently used for much of the sensing of the car. The Velodyne group processes the data that the Velodyne is generating and identifies 3 dimensional areas that are sign candidate regions. Using this data, another group transforms the 3-dimensional area from the Velodyne into a 2-dimensional region in an image from cameras mounted on the car. This region of interest is then processed by the vision group to verify the sign detection and recognizes the sign.

1 Velodyne Sign Detection

1.1 The Velodyne HDL-64 LIDAR:

The Velodyne consists of 64 lasers and 64 laser receivers which are split up into groups: 4 groups of 16 laser emitters (two in the top half of the unit, two in the bottom half) and 2 groups of 32 laser receivers (one in each half of the unit) (see Figure 1). The entire unit spins at a rate of 5-15 Hz and has a 360-degree horizontal field of view and a 26.8 degree vertical field of view. The Velodyne collects over 1.3 million data points per second, and delivers its output via 100 Mbps Ethernet packets. Its range for pavement is 50 meters and its range for cars/foliage is 120 meters, and is accurate to within 2 centimeters[5]. The unit outputs a point cloud where each point contains distance and intensity data. The sensor returns distances and angles to every point. In our code, the Velodyne data is accessed as a "Velodyne raw scan," which is converted to x-y-z coordinates, with an "intensity" channel to make the intensity data accessible.

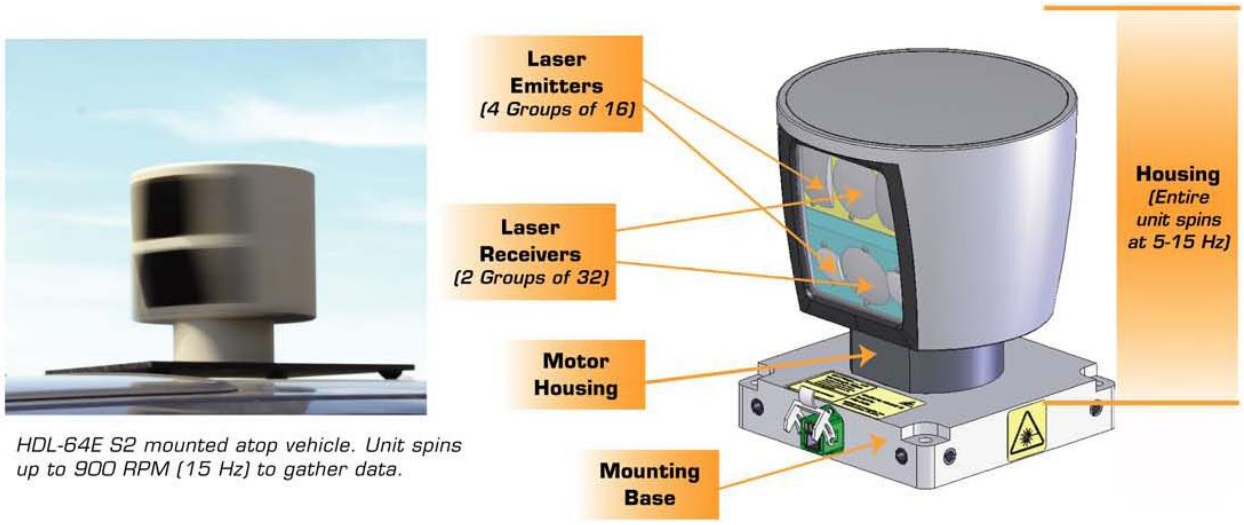


Figure 1.1 - the Velodyne unit (picture taken from Velodyne datasheet, available at www.velodyne.com/lidar/downloads/

Our algorithm groups points from this point cloud into clusters that become our "regions of interest." Before we explain this in more detail, we will mention some other work we considered before deciding to write our own algorithm.

1.2 Related Work

For this project, we used ROS (Robot Operating System), an open-source collection of software API's designed for robot research and development that is maintained by Willow Garage, Stanford University, and other universities and organizations with interest in robotics. One of the features of ROS is that there is a large database of code available for many of the problems we were likely to run into, so the first thing we did was to look around on the ROS website for clustering algorithms. Inside the ROS package point_cloud_clustering (written by Daniel Munoz and Alex Sorokin of Willow Garage), we found a class called KMeans that performs a k-means clustering on a point cloud of x, y, z coordinates.

The k-means algorithm begins by randomly selecting k "means" from a data set of points[6] Then k clusters are created by associating all of the points to the nearest mean. Then the means are assigned the values of the centroids of each of the k clusters[6]. The process is then repeated until the means are no longer assigned new values. This was unsuitable to us for the following reasons:

- It would likely require us to know how many clusters we are attempting to find before we run the algorithm. In our application, this is infeasible.
- The fact that we need to know the number of clusters would require us to set a value for k. If we set k too high, you would be breaking the desired clusters apart, and if we set k too low, you would be combining clusters together to form an undesired cluster.

We could not find a version of k-means that worked efficiently for an unknown value of k. In addition, the point_cloud_clustering package and KMeans class relies on many dependencies from the ROS library. We tried to download these and test the algorithm but could not do it, and due to time restraints and the fact that the algorithm did not seem to be useful to us, we decided to use our own algorithm, which is explained below.

1.3 Algorithm

To accomplish our task we divided up our job into 4 steps

1) Pre-Clustering Filtering
2) Clustering
3) Post-Clustering Filtering
4) Point Cloud Construction

In Pre-Clustering Filtering, we examined the points in the Velodyne raw scan and added points that were bright enough, high enough, and far enough from the car to a list of unclustered points. We only added points that were located in front of the car and on the right-hand side of the road. To do this, we determined specific thresholds for intensity, height, distance, and location and checked each point against these criteria. Finally, for each point, we calculated the distance between the point and the Velodyne sensor to eliminate points

representing the car. We stored the distance for later use during clustering. Pseudo-code for the pre-clustering step is shown below:

```
For(each point in raw scan)
        If(point satisfies the conditions )
                Add to list of unclustered points
```

To group the points into clusters that represent a particular object, we first create a cluster by taking an unclustered point as a "seed". Then, we look at the rest of the unclustered points. We add any points that are close enough to our "seed" point to our cluster. Two points are "close enough" when the distance between them is smaller than a dynamic distance threshold (the distance threshold changes based on the distance between the car and the observed points). We repeat this process every time we add a new point to the cluster. When every point has been processed, the cluster is complete. If there are still unclustered points, then there are more clusters and the clustering process needs to be repeated.

As an aside, we also want to create a "bounding box" for each cluster that is created. The bounding box will tell us what the maximum and minimum x-y-z values (max/min values) are in a particular cluster. This can help us determine how big an object defined by a cluster is and where its center is. To create this bounding box, we update the max/min values whenever a point is added to the cluster.

```
while( there are still unclustered points ) {
      make a new cluster with a seed point
      for( every point in the cluster "i" ) {
            for( every unclustered point "j" ) {
                  if("j" is close enough to "i" ) {
                        add "j" to the current cluster
                        update the max/min values
                  }
            }
      }
      // Post clustering filters
      add cluster to a list of clusters
}
```

As shown above, Post-Clustering Filtering is done during clustering. We chose to do these two tasks together in an effort to eliminate a particular issue we were having (points representing a sign would carry over into subsequent frames – our solution to this problem is detailed below). Post-Clustering Filtering involves examining each cluster to make sure that it meets certain criteria such as number of points and size of "bounding box". Many times there will be clusters with very few points in them. These clusters are most often noise and are disregarded. The typical size of a street sign is also known and falls within a certain range. We can check that the size of street sign falls within this range by looking at the size of the bounding box. At first, we checked these properties after all clustering was done and removed any cluster that did not satisfy these conditions from the list of clusters. This removal caused

the afore-mentioned error where points carried over into subsequent frames. To fix this, we instead do all of the checks, then add new clusters that satisfy all criteria to the list of clusters. This “check-to-add method”, while it does not fix the problem completely, seems to work much better than the “check-to-remove” method.

1.4 Testing/Results:

Testing of our algorithm was mainly done using the ROS visualization tool RViz. The RViz software could subscribe to our output and display our point cloud, along with the entire point cloud generated from the Velodyne. It could display one or the other. This tool was very useful for us as it allowed us to do most of our testing on our own machines with only a couple trips to the actual car.

Some screenshots of our algorithm running are shown below. In these images, the gray dots are the normal Velodyne point cloud. The red dots are from the point cloud returned by our algorithm. These screenshots are taken from .bag files, which are recorded files of the car driving or being driven. Using ROS, we can “play” these files and RViz can display the results of our code running on them. This simulates what happens on the actual car when a point cloud is given to RViz directly from the Velodyne sensor.

The first set of screenshots is taken from neils-thompson.bag – a recording of the car traveling down Neils Thompson Dr. at the J.J. Pickle Research Campus. The first shot includes the entire point cloud:

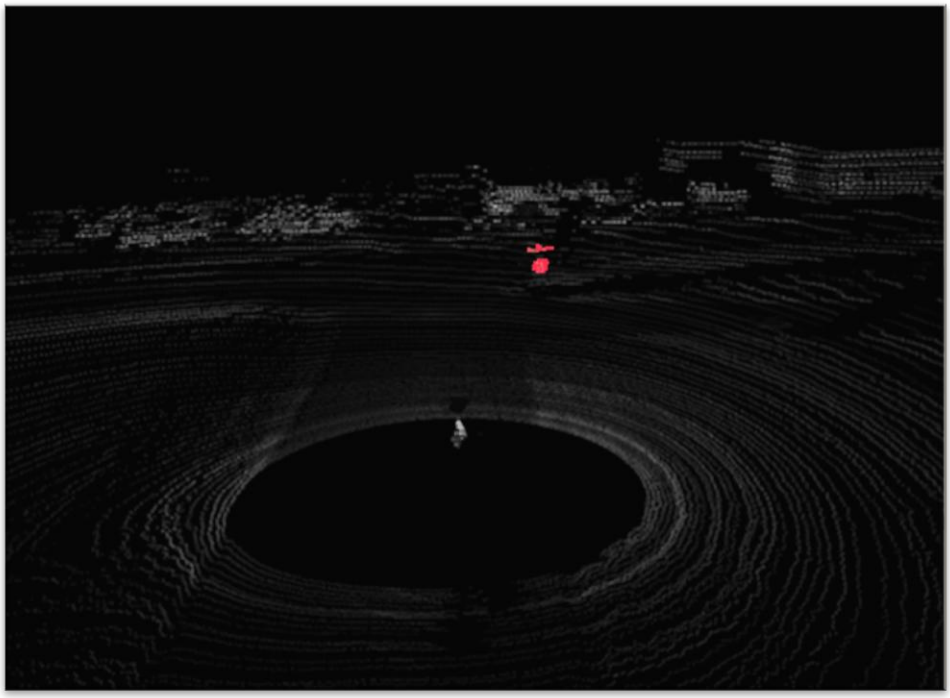


Figure 1.2 - Sign being detected on Neils Thompson Drive, with full point cloud. The red dots are the point cloud that is the output of our algorithm.

This is the same frame without the full point cloud. This is what the visual sign detection and calibration groups receive:

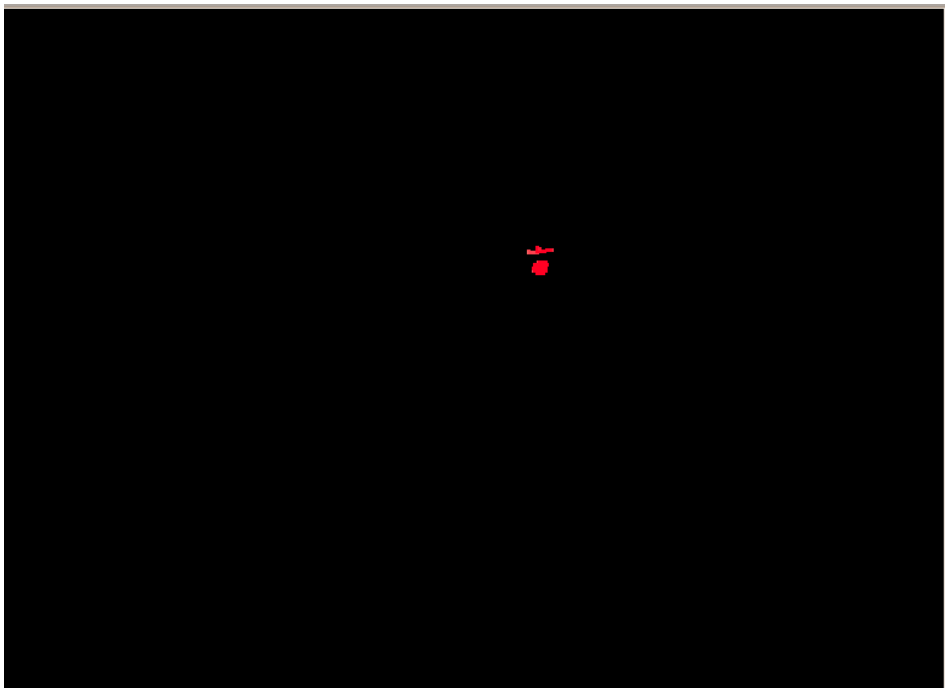


Figure 1.3 - Sign being detected on Neils Thompson Drive, with only our output shown.

Here is a pair of screenshots of the same type for a different .bag file. These are taken from road-D-east-loop.bag, another route from the J.J. Pickle Research Campus. These show that our algorithm can detect signs even while the car is turning.

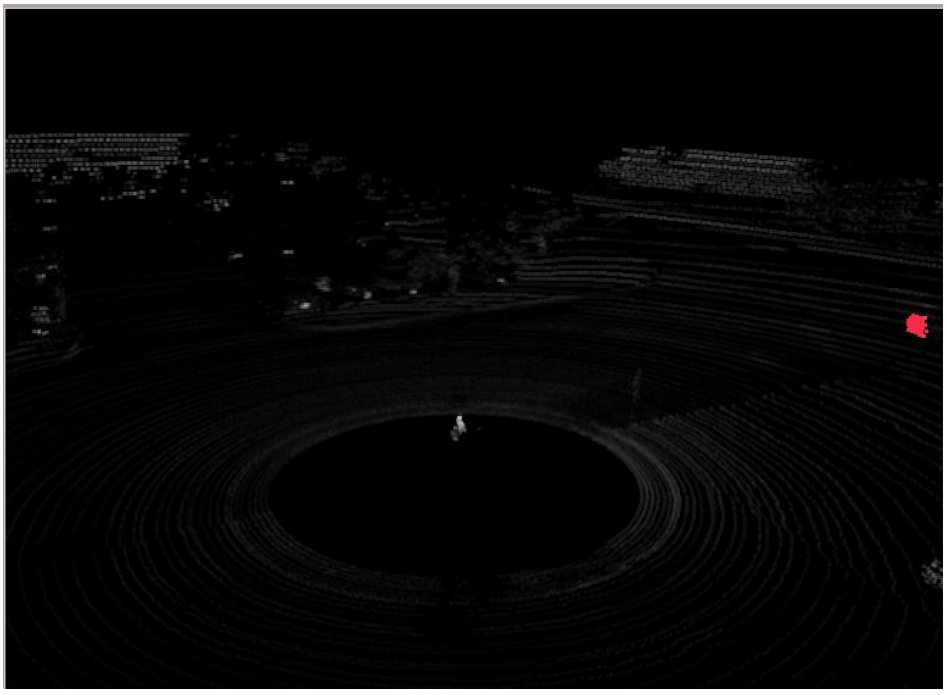


Figure 1.4 - Sign being detected while the car is making a right turn as it is approaching the sign.

Figure 1.5 - detected while the car is making a right turn as it is approaching the sign, with only the output point cloud shown

The final shot is of the car driving past some trees that are on the right side of the road. This shows that the algorithm does not detect random objects on the right side of the road as signs. This is also taken from road-D-east-loop.bag.

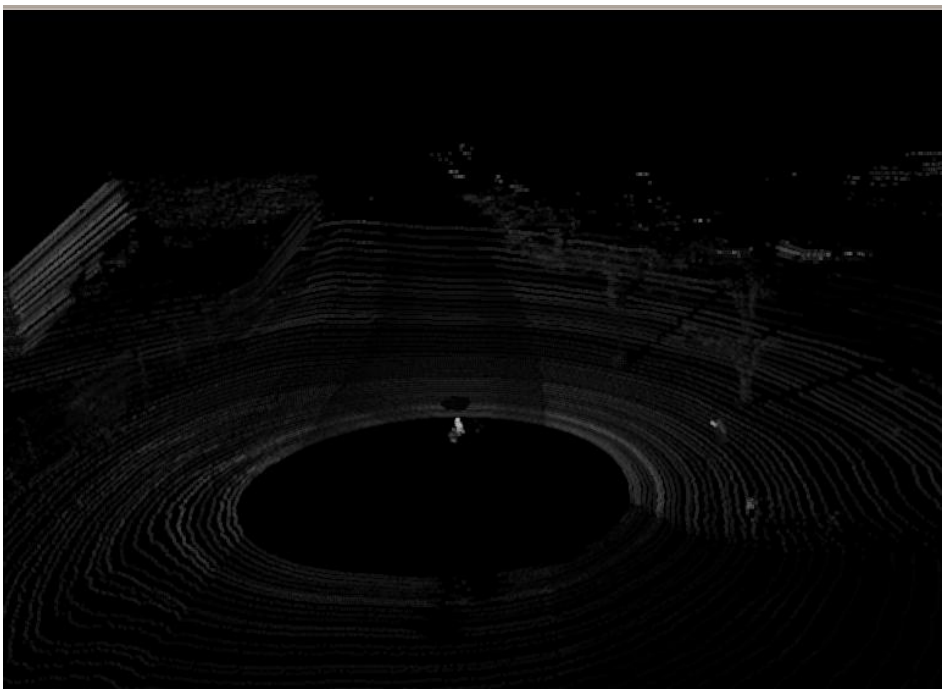


Figure 1.6 - The car driving past some trees on the right side of the road. The algorithm does not detect them as signs.

The error present in our algorithm comes in two different types, false positive and false negatives. To calculate false positives, we stepped through a bag file and counted how many clusters were published and how many of those were not signs. Cluster that do not represent a sign are quite easy to distinguish by hand so this measurement should be quite accurate. After stepping through 2 different bag files we have estimated our false positive error percentage to be roughly 11%. This error could be greatly reduced if a type of object tracking was introduced into our algorithm. Most false positives only exist in 1 or 2 frames before disappearing, while true positives remain constant in many frames. Including an object tracking ability to the algorithm and requiring an object to be present in 3 or so frames would drastically reduce the amount of false positives while barely affecting the time before a true positive is detected.

Our calculation of false negatives is not as reliable because it is difficult to define a positive reading. This definition could include when a sign is in the field of view, when a sign in your field of view is relevant, or at which distance a sign needs to be detected. For our test, we simply counted how many frames the algorithm did not detect a sign after its first sighting. This calculation yielded a false negative error of 17%. While this is a much more difficult error to correct, tweaking the thresholds would be the easiest way to reduce false negatives.

1.5 Conclusion:

We were able to meet most of our objectives for solving the problem of sign detection with the Velodyne. Our solution has an acceptable error percentage and works consistently. Not only that, but it has been relatively easy to integrate the code for this solution with the code for the other solutions mentioned in this paper to solve the overall problem of sign detection by the car. The errors that we do have are relatively small errors that do not drastically affect the quality of the overall solution (all three projects discussed in this paper combined). In the future, some ideas that we have for future Velodyne detection algorithms are object tracking, adding the ability to detect obstacles that are moving, and adding the ability to check whether obstacles are flat (this would enable us to detect certain types of objects with greater accuracy, including signs).

2 3D Point Cloud Projection

2.1 Introduction

The previous group working with the Velodyne laser scan has implemented an algorithm that takes the entire raw data scan and determines what points are likely to be signs. These points are then passed to my ROS node, in the form of a smaller point cloud, with each point having a corresponding size value. This size represents the estimated size of the sign, to determine what amount of the picture to crop.

2.2 Background

2.2.1 3D projection

3D projections are used in virtually every type of 3D graphics, from video games to drafting and engineering. Within the computer objects are calculated and manipulated in three dimensions, but must be displayed on a two-dimensional media (computer screen). Therefore, a lot of work has been done with 3D projections and they are well understood and optimized.

2.2.2 3D reconstruction

Since we as humans perceive three dimensions, 3D reconstruction is an important part of robotics research simply because a single two-dimensional camera image is not always enough data for a robot to have about its environment. Reconstruction is essentially the inverse operation of projection. On the car, we have solved the 3D reconstruction problem by adding a Velodyne laser scanner that helps us form a point cloud of objects surrounding the car, with great precision. However, now it is necessary to relate these points to visual camera data, for many possible purposes other than sign detection, even though that is what I will be focusing on.

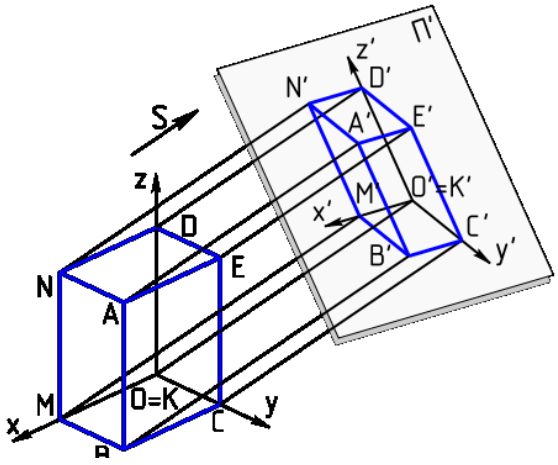


Figure 2.2.1
The general form of a 3D to 2D projection

2.2.3 ROS data

This project is implemented in ROS (Robot Operating System). A node called point_to_image has been constructed, which subscribes to two streams of data, the compressed image from the front right camera on the car and the modified point cloud from the Velodyne sign detection group. It then advertises a new image cropped from the original image based on the incoming point cloud. A problem arises with the differing frequencies of incoming data, but this discussed later. The new image advertised is then used by the computer vision group.

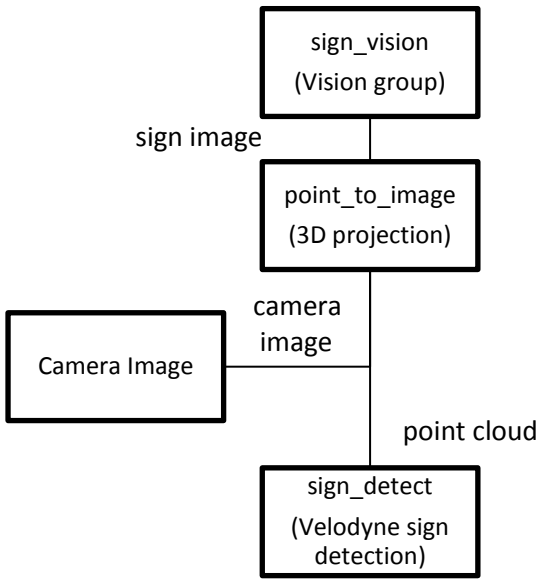


Figure 2.2.3 Algorithm Hierarchy

2.3 Related Work

As mentioned earlier, 3D projections are common in all kinds of computer graphics. A method exists within the OpenCV library in ROS called cvprojectpoints2(), which is used in this projection. It takes in arguments for the source 3D points, rotation and translation of the camera, intrinsic camera matrix, distortion coefficients and writes the projected 2D points to a cvMatrix.

2.4 Current Work

2.4.1 Algorithm Overview

The point_to_image node subscribes to a ROS point cloud advertised by the Velodyne group. When a point cloud is received, these points in the form (x,y,z) are copied into a cvMatrix for manipulation with the OpenCV library. cvProjectPoints2() is called on this matrix with the desired parameters. The technique used by the library method will be discussed later in this section.

Second, the image is read in and processed. The image comes in at a different rate than the point cloud,
but for now, the points are projected
when the image comes in, using the last point cloud received. This appears to work fine for our purposes but this is discussed in 6.2 of Future Work. Currently, a mutex is used to prevent concurrent modification of a point cloud while it is being projected onto the image. One thread runs every time a point cloud is read and another when the image is read.

After the points have been projected, we now have (x,y) coordinates in the video image for possible sign locations. These are located on the image (if they are within the image's pixel range), and OpenCV is used to crop the image around this Region of Interest. The cropped

image is advertised (sent to sign_vision), and the Region of Interest is reset for the original image. This process is repeated for every frame where points have been projected onto the image. If there are no points projected on the image, nothing is advertised. If there are multiple points, only the last point projected is advertised. Once again, this works fine for now as multiple sign situations are rare but in the future this will need to be corrected, as described in 6.3 of Future Work.

2.4.2 Projection

For this projection we are using the pinhole camera model. The actual projection relies on the rotation, translation and focal length of the camera.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A[R|t]\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where u, v are the pixels of the point (X, Y, Z) the image.

The camera matrix A is defined as

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

f_x and f_y are defined as the focal length of the camera in pixels vertical and horizontal directions, $(c_x\ , c_y)$ is defined as the principal point (image center). These values were determined with openCV camera calibration.

The augmented matrix $[R|t]$ (joint rotation-translation matrix) is defined as

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}\begin{bmatrix} \cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\beta \end{bmatrix}\begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Where α = camera's rotation about the x axis
β = camera's rotation about the y axis
γ = camera's rotation about the z axis

And $(\ t_x\ , t_y, t_z)$ is the offset of the camera from the frame of reference (in this case we are working with the frame of reference of the Velodyne.

The model below is equivalent while $z \neq 0$ and is used when computing. If $z = 0$ the point just lies on the principal point $(c_x\ , c_y)$.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$
$$y' = y/z$$

$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

In addition, this model takes into account the distortion of the camera lens, as determined by the OpenCV Camera Calibration. Distortion is corrected as follows:

$$x' = x/z$$
$$y' = y/z$$

$$r^2 = x'^2 + y'^2$$
$$x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1\left(r^2 + 2y'^2\right)' + 2p_2 x' y'$$

$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

Where $(k_1, k_2, p_1, p_2, k_3)$ are the distortion coefficients
The addition of the intermediate step between (x', y') and (x'', y'') accounts for tangential and radial distortion.

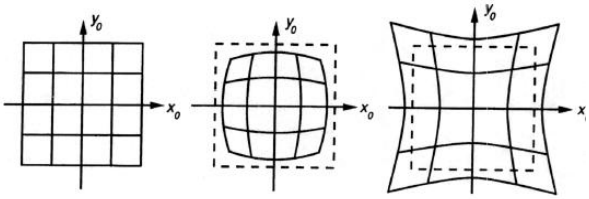


Figure 2.4.2
Radial distortion becomes worse around the edges of the image, which already has a high degree of error due to the speed of the vehicle.

2.4.3 Calibration Data and values

Through measurement, experimental methods and camera calibration, the following values have been determined for this particular camera transform (the front right camera mounted on the car). For other cameras on the car the Translation and rotation matrices will

need to be adjusted but the Distortion and Camera Matrix values will be constant because the cameras are all the same.

Distortion coefficients

$$(k_1, k_2, p_1, p_2, k_3) = (-0.291963,\quad 0.0880665,\quad 0.00284341,\quad -0.00229279,\quad 0)$$

Camera Matrix

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1072.62 & 0 & 687.063 \\ 0 & 1072.38 & 428.763 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation Matrix (with respect to Velodyne)

$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} 0.127 \\ -0.089 \\ -0.216 \end{bmatrix}$$

Rotation Matrix (with respect to Velodyne)

$\alpha = 90°$

$\beta = 270°$

$\gamma = 328°$

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 90° & -\sin 90° \\ 0 & \sin 90° & \cos 90° \end{bmatrix} \begin{bmatrix} \cos 270° & 0 & \sin 270° \\ 0 & 1 & 0 \\ -\sin 270° & 0 & \cos 270° \end{bmatrix} \begin{bmatrix} \cos 328° & -\sin 328° & 0 \\ \sin 328° & \cos 328° & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -0.52992 & -0.84805 & 0 \\ 0 & 0 & -1 \\ 0.84805 & -0.52992 & 0 \end{bmatrix}$$

2.5 Testing

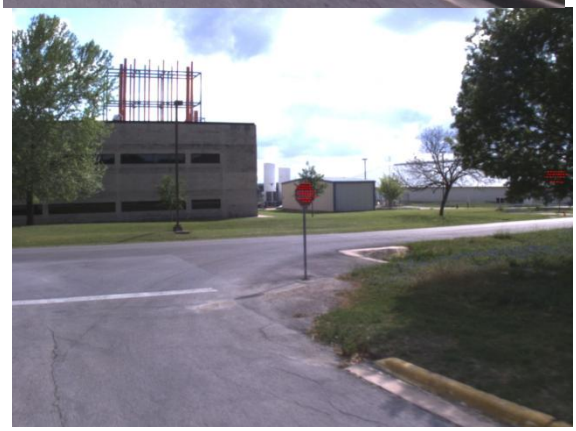
2.5.1 Running Tests

Tests were generally conducted either on the car, driving manually, or using recorded data from the car in .bag files. These files contained the point cloud data and video from the Velodyne and cameras. Therefore it was easy to run tests in a controlled environment, and to record results. However, more testing will be needed to fine tune the projection, as errors still exist.

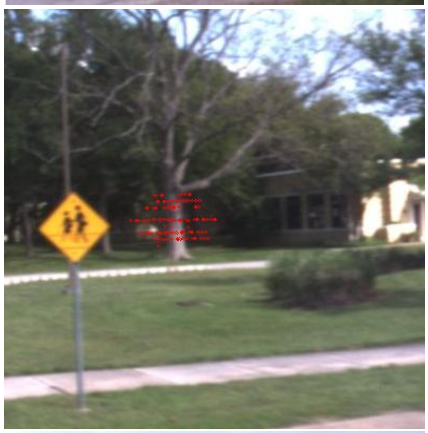
2.5.2 Cropping

These are all screenshots from running the algorithm on recorded .bag files. The red dots are points from the Velodyne algorithm that have been projected onto the image. The region of interest is computed by cropping 250 pixels around the last point considered.

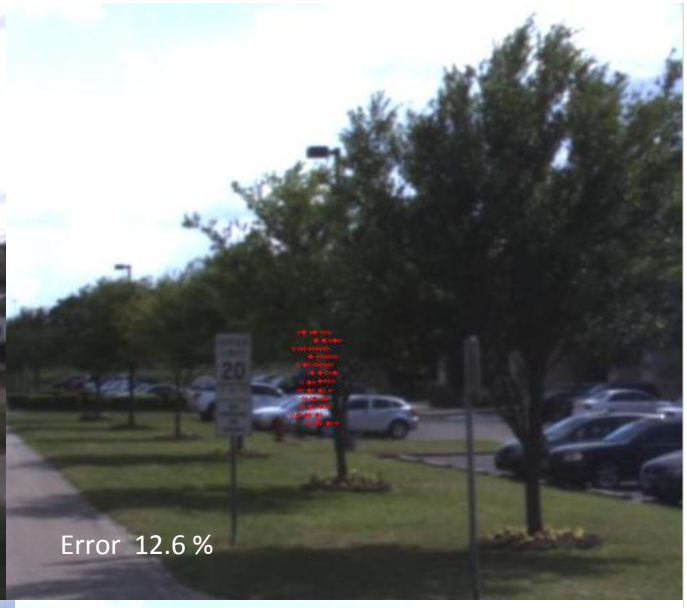
Entire image from front right camera

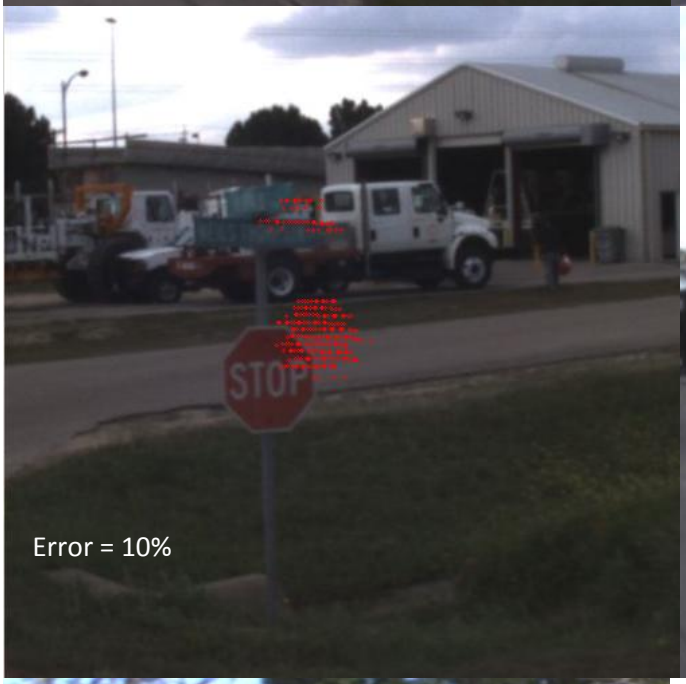


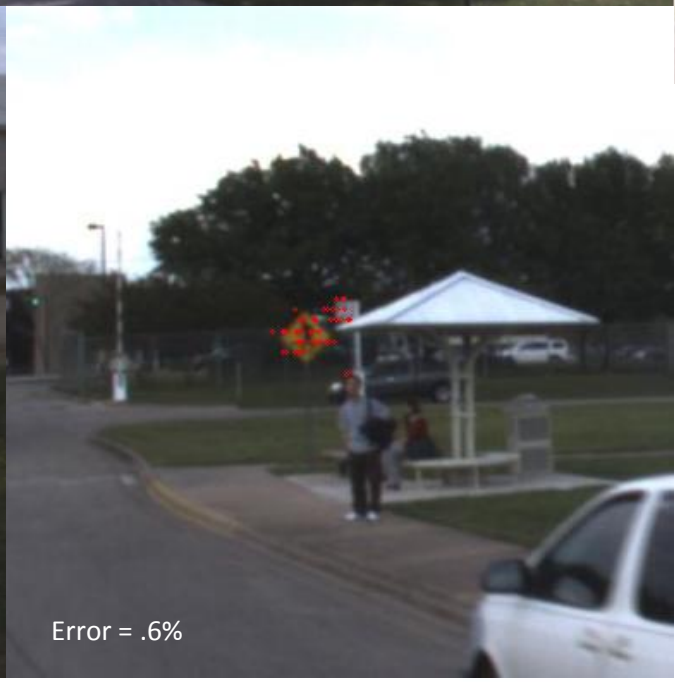
Region of interest (ROI) output



More cropped images (only ROI)







2.5.3 Error

Error is calculated by finding the pixel distance between the projected point and the actual point in the image, and rounded to the nearest pixel. This gives us a general idea of how correct the projection is.

$$Error\ \% = \sqrt{\frac{\left|x_{projection} - x_{actual}\right|}{image\ width} + \frac{\left|(y_{projection} - y_{actual})\right|}{image\ height}}$$

These errors occur when the car is turning quickly, moving at a very high velocity, or if the sign is very close to the car on either side. However, if the car is moving toward the sign and it is located near the center of the image, the projection has an error of <10%. This is important because generally all signs originate in the middle of the image (further down the road), and this will be recognized before the error becomes too great to capture the image. In addition, error can be reduced by increasing the image size output. Right now the image published is +-250 pixels of the last point projected.

Some error will always occur because of the image/Velodyne latency. These errors become amplified around the edges of the image, and at high velocities. We consider a projection error of <25% acceptable, as the sign will still appear in the output image assuming the sign takes up a reasonably small area of the non-cropped image. More on this is outlined in 6.4 Error correction.

2.6 Future Work

2.6.1 Sign Location

The obvious next step in this algorithm is passing along the original (x,y,z) coordinates of a detected stop sign to the car navigation node and have the car act accordingly. This could be implemented by simply adding a channel to the data published for sign_vision containing (x,y,z) coordinates corresponding to the point in the picture being sent. Then if vign_vision determined a sign exists in that location, it can send that on to the navigator node.

2.6.2 Data Synchronization

Messages containing point cloud data and camera frames are read in at different rates. This is because the Velodyne publishes a point cloud at a higher frequency than the camera publishes camera frames. Therefore plotting points onto an image with a very high degree of accuracy is virtually impossible, since the point clouds and camera frames are recorded at different times, and the frame of reference (Velodyne) has moved. At this stage of development, the car goes about 15 mph

Figure 2.6
Example of two point clouds being projected onto one image. Also, some delay is evident.

maximum, so this doesn't make a huge difference. However if this method was to be extended to cars at higher velocities, perhaps being driven by humans, then time synchronization would be necessary. At 60 mph (27 m/s) the latency of the Velodyne and the camera could put camera images 5 Hz behind or ahead of the point cloud, resulting in a 5 meter discrepancy between calculated sign projection and actual location. Granted, stop signs do not appear on 60 mph roads very often, but if this sign detection method were extended to speed limit signs this would be an important thing to add.

ROS offers message filtering for incoming data, and this can be used to filter messages based on timestamps. Therefore only messages within a certain threshold of time can be considered together. This offers an easy way to synchronize data inputs in the future.

2.6.3 Multiple Signs

There are situations where the Velodyne will produce multiple areas of interest that map to the image. Which areas should be cropped and advertised? Currently we advertise an image containing only the last point mapped to the image, plus or minus a threshold of pixels around that point. In the future it should be possible to publish an array of images, or multiple channels of images.

2.6.4 Error Correction

Further work should be done to use point – radius data from the Velodyne group to alter the output sign image based on image distance. This will reduce the amount of non-sign pixels being published and decrease error as signs get closer (the output image will get bigger. In addition, this will decrease error, as the image will increase in size as the sign gets closer.

3 Traffic Sign Detection and Recognition Using Computer Vision

3.1 Introduction

Our vision approach has a large advantage over other sign detection systems that have been implemented in the past. In these other systems, it is necessary for every few frames from the camera to be processed and searched for signs. By utilizing the Velodyne, it is no longer necessary to search nearly every frame. Only the frames corresponding to when the Velodyne detects a sign will have to be searched. Also, the entire frame does not have to be searched; only the region that aligns with the Velodyne detection. With only a small region of the image our algorithm will have fewer features in the image that could be a distraction, and will result in faster processing and more reliable results. This method focuses the search for signs on only places where signs are likely to be, not entire the video frame. Another benefit arises from a two-layer detection system; detection is required with the Velodyne, and also using vision. With two processes needing to be satisfied, the possibility of false positives is reduced.

Background information on vision techniques that are used is provided in Section 3.2. We describe our current procedure in Section 3.4, along with related work in Section 3.3. Our results are in Section 5, with possible improvements and future work in Section 3.7.

3.2 Background

3.2.1 HSV

HSV is a different way of representing colors than the usual RGB color space. HSV, or hue, saturation, value, represents colors in a cylindrical coordinate system, opposed to RGB's cube model. Hue is determined by rotation around the cylinder, value by the height of the cylinder, and saturation by distance from the center axis of the cylinder. Our main motivation for using this model was that it allows for simpler and more accurate color detection in a range of lighting conditions. The lighting conditions received in the camera data can vary based on the outside conditions, such as clouds, time of day, or the angle the light hits the sign. The HSV model represents color in a way that more easily allows color detection in different lighting conditions. This can be achieved by defining a specific hue range and letting the saturation and value take larger ranges. This simulates a single hue taking on different saturation and value values, as if in different lighting conditions.

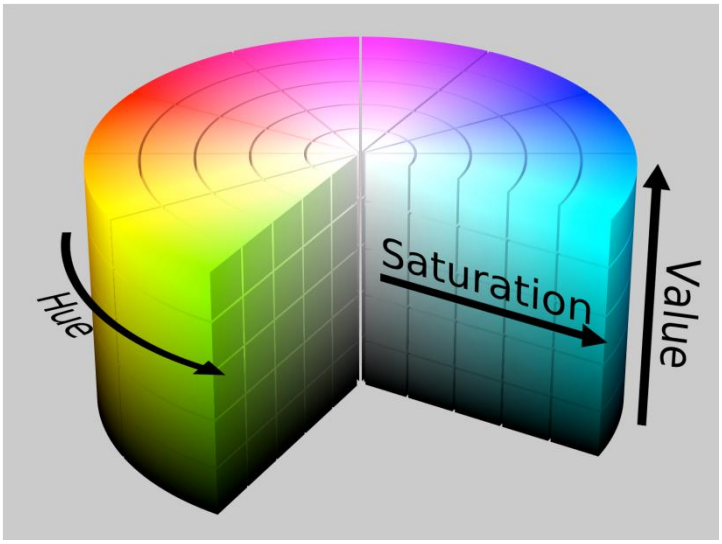


Figure 3.1. This model shows the representation of colors under the HSV color space.

3.2.2 Blobs

Our algorithm uses a blob detection process to identify signs in the images received. A blob is a region of an image that satisfies a specified property, in our case, a color range. We process an image to find regions that are dense with specified colors, and then these regions are the blobs that we use for our future processing.

Figure 3.2. The white regions are the blobs that our color detector produced.

3.2.3 Template Matching

Template matching is a technique used to find a specified template, or pattern, inside another image. The template image contains the object that is being searched for in an image. To begin the matching, the template is aligned with the top left corner of the image. A calculation is done to determine how similar the two images are, then the template is then moved over one pixel and the process is repeated. This process continues until the entire image has been evaluated. The calculated values are used to determine if the template is in the image, and, if it is in the image, where it occurs. The template matching procedure does not account for size, rotation, or any distortion of the image; therefore it is often useful to perform this matching algorithm several times with templates of different sizes, rotations, or distortions to obtain the best results.

3.2.4 Cameras

On Marvin there are two forward facing cameras, these cameras produce the images that will be processed by our algorithm. The camera data is captured in sync with the Velodyne data, so that the correlation between the two sets of data can be found. This allows the appropriate image to be produced for our algorithm to process.

3.3 Related Work

Using computer vision to detect and classify signs has been studied in the past and has been approached using many different techniques. This is an overview of some of the other work we have read about while doing our research.

The work done in [2] is very similar to what has been implemented in our system. Sign detection is achieved by scanning an image for regions that satisfy certain RGB color ratios. Properties of the blobs are calculated, such as the location of the centroid, area, and the bounding box. The blobs are then filtered using these properties and non-signs are removed. The remaining blobs are then tracked and if a blob is seen 5 frames in a row, then it is accepted as a sign. Template matching is used to classify the sign that was detected.

Other work that we have read about uses more advanced algorithms for sign detection and recognition. In [3], features of signs were generated using HOG (Histogram of Oriented Gradients) and a subset features were then chosen that were best able to detect signs, and used to generate a cascade classifier. Color was used to extract regions of the image that possibly contained signs, and then these regions were tested using the cascade classifier to determine the type of sign. The results from this work produce were very accurate, but the speed of the algorithm was very slow, averaging over nine seconds of processing for some sign types.

3.4 Current Work

3.4.1 Algorithm Overview

Our algorithm receives a region of an image that has been generated by the other subgroups. The image is first processed to detect and verify that a sign is contained in the image. If the image contains a sign, further processing is performed to recognize the type of sign in the image.

3.4.2 Sign Detection

First, a color blob filter is performed on the image. This is the process of checking each pixel and determining if the pixel's color falls within one of the two specified HSV ranges. The first color range represents the red color of "Do Not Enter", "Stop", and "Yield" signs, while the second represents the yellow color of "Warning" signs. Two binary images the same size as the original image are created, one for the first color range and one for the second color range. If a pixel in the original image satisfies a color range, then the pixel in the same location in the appropriate binary image is colored white. Otherwise, the pixel is colored black.

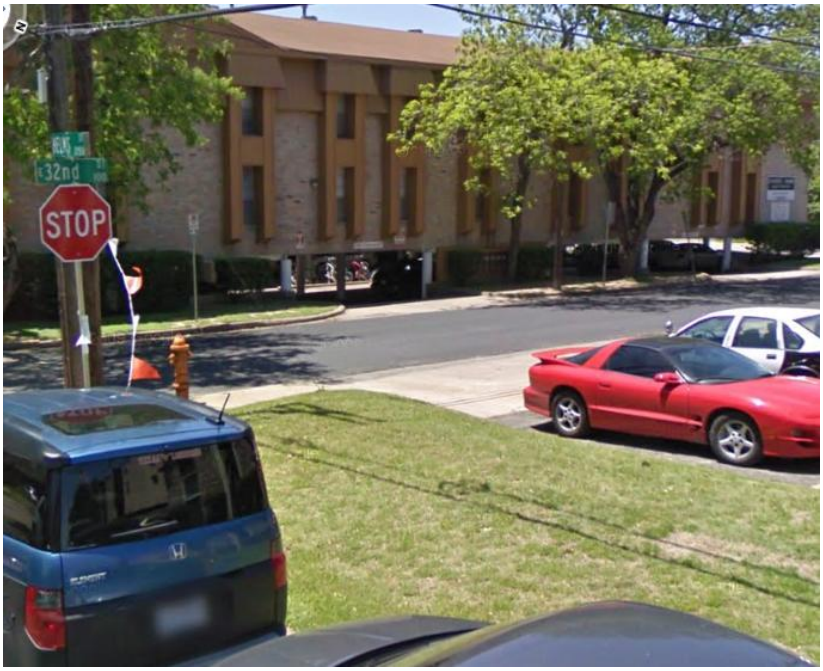


Fig 3.3 This is the original image before any processing has been applied.

Fig 3.4. This is a binary image that was generated from the color blob filter. Notice the small noisy regions of white that are scattered around the image.

These two binary images now contain regions of black and white. The white areas are the areas that have the same color as the signs that are to be detected, and everything else is black. The white regions may be very noisy and unconnected. In order to eliminate these areas, an erosion filter is run on the images. The erosion filter reduces the edges of the white areas, so small white areas are completely

removed. After that, a dilation filter is then run on the images. The dilation filter increases the edges of the white areas. This connects the many remaining white blobs into a few, larger ones.

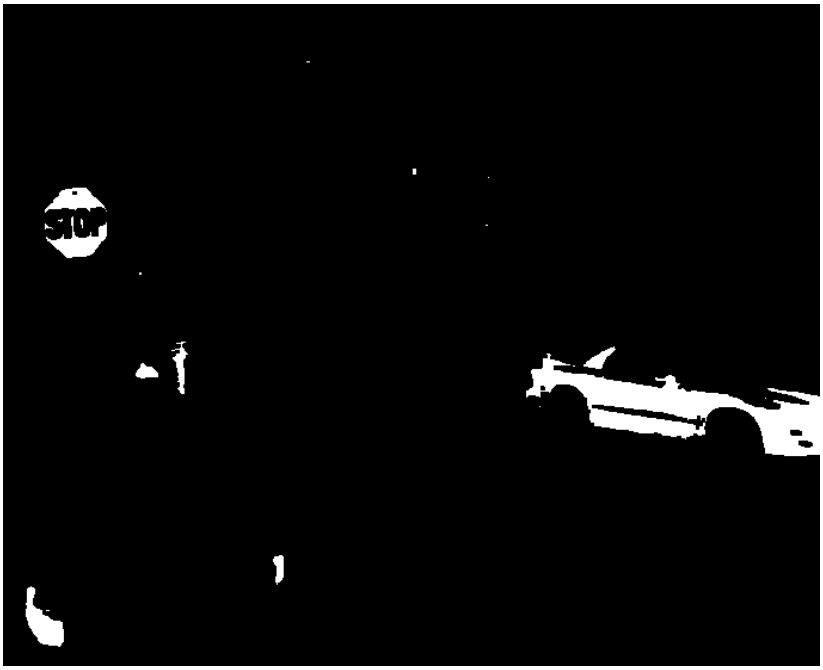


Fig 3.5. An erosion filter has been applied to the image and has removed most of the noisy regions.

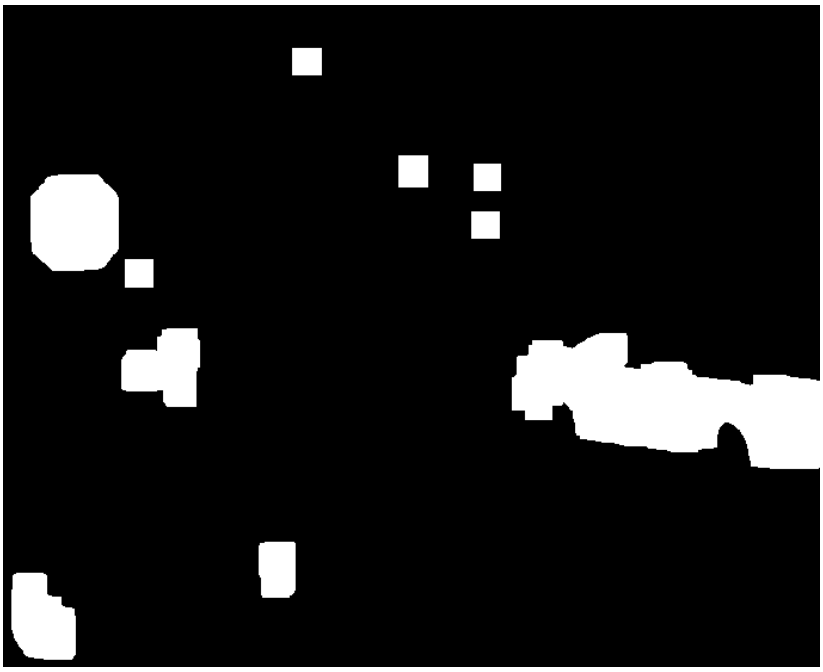


Fig 3.6. After the dilation filter is applied blobs that are close to each other become connected to create one large blob. This removes the possibility of a sign being split in half by text or images on it. Other blob regions can easily be filtered.

Now the images contain blob regions. These blobs are used to verify that the image contains a sign. A list is then complied of all the candidate blob regions. Each blob is checked for having properties that signs also have. The signs to be detected are very square in shape, so one property the blobs are checked for is having a width to height ratio that is close to a value of 1. Also, signs do not have a very irregular shape and generally fill much of the area of their bounding box, so to check this property, the exact area of the blob is calculated, the area of the bounding box is calculated, and then the areas are compared. The area of the bounding box must be less than the area of the exact area of the blob times a constant. If a blob does not satisfy one of these properties then it is removed from this list of candidates. The sign candidate list has been filtered to only include blobs that have the properties of signs. If this list is empty, then it is returned that the image does not contain any signs in it. If the list is not empty, then further processing is performed to determine the type of sign.

3.4.3 Sign Recognition

At this point, the list only contains blobs that are likely to be signs. If there are only blobs that correspond to a region that is in the "Warning" sign color range, the bounding box of the largest sign in the list and that a "Warning" sign is present is returned. Currently the procedure recognizes warning signs generally, not the specific type of warning sign.

If there are only blobs that correspond to regions of red in the original image, then sub regions are extracted by taking only the part of the image that is in the bounding box of the blob. The part of the image that does not contain a sign is then made white to improve the accuracy of the template matching. Templates for the three different types of red signs to be detected are matched against it. And for each template, several different sizes are used so signs of different dimensions can be detected.

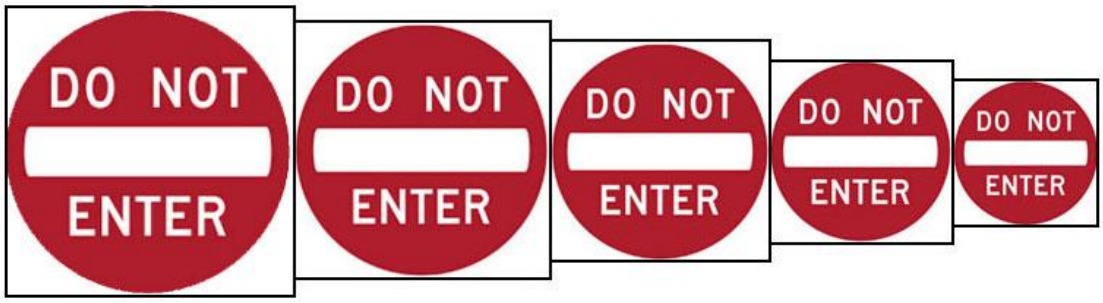


Fig 3.7 The template matching procedure is performed with several different sizes of the same template. This allows for the recognition of signs of different sizes.

As the template is slid over the image, it returns different values of the correlation between the two images. The number of values that are above a certain threshold are counted and then divided by the total number of values returned. This gives the percent of positions where the template matched well. The maximum percentage is stored, along with the template that it corresponds to. Once all the template-matching procedures are finished, the sign that had the highest average is returned.

If an image contains both yellow and red regions, then the type of sign is first set to a warning, then the template matching procedure is performed. If this generates results that are likely enough to be a different sign, then that sign will be used instead of the warning sign.

3.5 Testing and Results

To test the detection and recognition ability of the algorithm, screenshots of signs and objects likely to be detected by the Velodyne group were taken in Google Streetview. A total of 123 images were collected.

	Total Signs	Signs Detected	False Detection	Signs Recognized	False Recognition
Number	90	75	3	71	4
Percent		83.3%	2.4%	78.9%	4.44%

Many of our sources of error come from low quality images. If the image has a very low resolution or is too blurry, the algorithm is much more likely to fail the detection process.

The current results look promising, but the algorithm has not been run on images produced from the other groups. Parameters may need to be altered to have the algorithm run effectively using the cameras that are on the car.

Fig 3.8. The yellow box denotes that a warning sign was detected. Also notice that the warning sign was detected over the picture of the red octagon on the sign.

Fig 3.9. The red box denotes that a stop sign was detected. An image that is focused on a sign, as in this pictures, will improve our algorithms results.

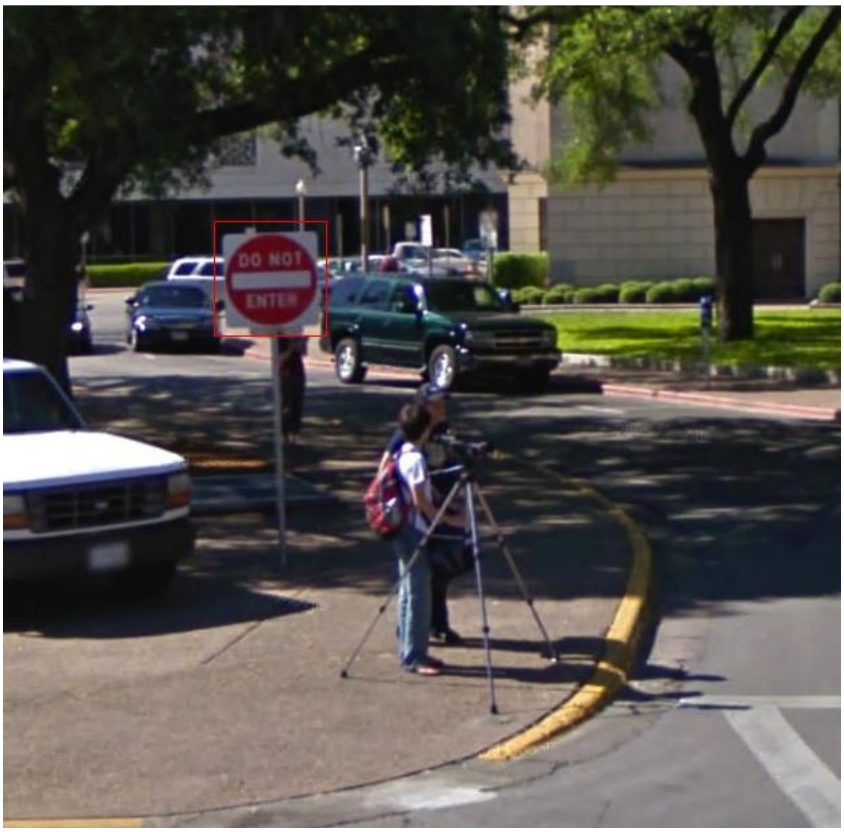


Fig 3.10. Even when a sign does not fill the whole image the algorithm is still reliable.

3.6 Conclusion

We have developed a working method for detecting and recognizing traffic signs. The procedure operates well and is capable of finding most of the signs it has been designed to detect. But it is still not enough to safely operate on the road. For the car to rely solely on this procedure it will need to have an accuracy rate that is much closer to 100% and need to detect all the types of traffic signs that could appear on the road. This work is only the first step in creating a robust traffic sign detector and recognizer for autonomous vehicles.

3.7 Future Work

One obvious enhancement would be increasing the number of signs that are detected. In order for a car to safely drive, the system must be able to detect all signs that it could encounter. With our current method it is easy to add individual signs to the process, however including every sign is impractical because that would require manually testing for the appropriate color values and features of each sign. Testing for every sign would also dramatically increase the computation time required for

detection. The system needs to run in real-time to be effective and adding a large library of signs to detect using the current method is likely to push the processing past real-time. Solutions to these problems may lie in more advanced algorithms for object detection. By using machine learning, signs could easily be added to the detection list by training the system for the new sign. This would no longer require manually testing for the color range of new signs to be detected because the system would automate this with the training process. Features of the signs to track would automatically be determined and not be prone to error from human judgment. Other work shows that detection using learning can be very accurate, but it would be necessary to ensure that real-time process can still occur.

With a working algorithm to detect and classify signs, the next step would be to use the data generated from nearby signs to modify the behavior of the car. Procedures would need to be defined for each of the different types of signs. When a sign is detected a message would be sent to the car with information about the type and location of the sign. The car would need to make the appropriate decision about how to handle the sign.

For autonomous vehicles to become the main vehicle on streets, they will need to be affordable. The current sensors on the car cost too much to be able to sell autonomous cars at a reasonable price. Computer vision may be the answer to this. Cameras are very cheap compared to other sensors on the car like the Velodyne. If a robust vision system can be implemented for the vehicle, that is advanced enough to replace the current functions of the more expensive sensors, then autonomous vehicles will be much closer to being the only vehicles on the streets. This project is a small step towards creating a completely vision-based system for autonomous navigation. Future vision projects that build upon this one will be very beneficial to the advancement of autonomous vehicles.

4 References

[1] Fisher, R. B. , K. Dawson-Howe, A. Fitzgibbon, C. Robertson, and E. Trucco. *Dictionary of Computer Vision and Image Processing*. Chichester: Wiley, 2005. Print.

[2] Michael Shneier. Road Sign Detection and Recognition. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.6874&rep=rep1&type=pdf

[3] Chia-Hsiung Chen, Marcus Chen, Tianshi Gao. Detection and Recognition of Alert Traffic Signs. [Online]. Available: http://ai.stanford.edu/~kosecka/final_report_final_traffic.pdf

[4] Wikipedia contributors, "HSL and HSV," *Wikipedia, The Free Encyclopedia*, <http://en.wikipedia.org/wiki/HSL_and_HSV>

[5] Velodyne datasheet [Online], Available: <http://www.velodyne.com/lidar/products/brochure/HDL-64E%20S2%20datasheet_2010_lowres.pdf>

[6] Wikipedia contributors, "k-means clustering," *Wikipedia, The Free Encyclopedia*, <http://en.wikipedia.org/ K-means_clustering>

All images to test Adam and Jonathan's algorithm on were taken from Google Streetview. Figures 3.1, 3.6, 3.7, and 3.8 are all screenshots taken from Streetview, and Figures 3.2, 3.3, and 3.4 are all images from street view that have been processed.

Figure 1.1 taken from the Velodyne datasheet.

5 Appendix

5.1 Remarks

Steven Cahail, Jesse Vera, and Tony Daddeo

This project was the first time any of us had worked with robotics, and we learned a lot from it, especially with regard to working on a large project, working with Linux, and working with ROS and Subversion (SVN). One of the main things we learned about robotics was exactly how broad of a field it is. There are a lot of different problems to be researched and solved in robotics.

We really enjoyed learning about the car and exactly how it works. The problem of autonomous driving is not something any of us had honestly explored before, so to see a solution to the problem in progress and understand how it worked was a very enlightening experience. To learn that the car relies heavily on the RNDF to sense important obstacles such as stop signs, especially with sensing tools as advanced as the Velodyne, was a bit disheartening. That is part of the reason why we attempted to use the Velodyne to detect signs. We decided fairly quickly that using a clustering algorithm would be a good solution to our problem, and looked at some other algorithms but concluded that the best thing to do would be to write our own. Because of this, we learned a substantial amount about how to implement a clustering algorithm.

What we learned the most about as a whole, however, is research. This project involved learning a lot of new systems and how to use them (namely ROS, Linux, and C++). It took us quite a while to get Linux and ROS working, and writing in C++ proved to be a small challenge since none of us had ever written code in C++ before. Learning to use Subversion was valuable because it is used in other research projects and in general software development. We also learned how to work in a group of more than two, with each person having a different level of programming experience. In this kind of setting, communication between everyone is extremely important, and to make sure everyone understands everything that's going on with the code is also important so one person doesn't fall behind and reach a point where they can't contribute to the project.

A summary of our code is below:

Our code is located in `svn/sandbox/velodyne_signs/src` and includes the following files:

`detect_tony.cc`: This is the file that should be used when running the project. It contains a fully working version of our algorithm and is the file that Nick's code depends on.

`detect_jesse.cc`: This file contains cleaner code and a working version of the algorithm, but was finished after final testing of the whole project, so it should not be run unless Nick's code is modified to subscribe to detect_jesse.

`detect_steve.cc`: This file should not be run with the other code. It contains an additional feature that we were working on that returns only a point and a radius for each cluster, but we decided at the very end that we did not need this feature. This code is designed to return the center point and radius of each cluster, instead of the cluster itself.

Nick Shelton

This class was different from any computer science I have ever taken or expect to take because we did something that had essentially never been done before. In other classes we have specific, boring assignments but in this class we were left to our own to devise methods to solve our problem. I feel this prepares me more for creative problem solving in the professional world. In addition, working in a group was definitely a learning experience for everyone involved. Even though my sub-project was accomplished alone, my results and input data came from other groups and it was important to understand the other groups to produce a cohesive product.

I have definitely learned more about Linux and actual project development in this class than any other experience, just being immersed in a big project with a lot of pre-existing code and a large amount of people working together. I gained experience with Linux, SVN, C++ and extensively compiling, running and testing code, working with open source libraries and ROS. All these skills definitely prepare me for future computer science research and work, and give me an advantage over freshmen who haven't had the opportunity to work on the car in this class.

My code is located in `svn/sandbox/velodyne_signs/src` and includes the following files:

`point_to_image.cc`: This file contains the code that transforms from a Velodyne point cloud that is received from the Velodyne to a correspond region in an image that is sent to the vision group, and advertises an image containing the region of interest around those points.

Adam Menz and Jonathan Long

We have learned a lot from this project, and not just about robotics, but also about working in large groups, working on a large software project, and about research in general. Our project required learning many different computer vision techniques that we would not have come across otherwise. Had we initially known the vision techniques that currently exist, we probably would have taken a different approach and used learning algorithms. In the future though, with the knowledge of these algorithms, solving vision problems will become simpler. It is also very helpful while working on a problem to look at related work and see how the problem has been approached in the past. This helps eliminate bad ideas, and will spark new ideas for a better approach. We have also discovered the frustrations that come with research, and to not get excited about something working when it first appears to be. Numerous times we thought something was working properly only to find the next day that it was a fluke that the right results were produced. Extensive testing is needed to ensure that the results you are

producing are accurate and consistent. When working with other groups on a project, it is very beneficial to have good communication between all members. Everyone should be up to date and have accurate information about the progress and results of other groups. One cannot make assumptions about the code other groups are producing.

A summary of our code is below:

Our code is located in `svn/sandbox/sign_vision/src` and includes the following files:

`sign_vision.cc`: This file contains the code that applies vision techniques to detect and recognize signs on an image that is received from Nick.

`StopTemplate.png, YieldTemplate.png, and DoNotEnterTemplate.png`: These files are all the files that are used in the template matching procedure in sign_vision.cc. The code in sign_vision.cc will need to be changed to file path that these are in on your machine to locally run the clode.

`Blop.cpp, Blob.h, BlobExtraction.cpp, BlobExtraction.h, BlobLibraryConfiguration.h, BlobResult.cpp and BlobResult.h`: These files are an external library that are used in `sign_vision.cc`

5.2 Running the code

Run roscore

```
$ roscore
```

Run the three nodes:

```
$ rosrun velodyne_signs detect_tony
```

```
$ rosrun velodyne_signs point_to_image _image_transport:=compressed
```

_image_transport := compressed allows the node to handle manipulating the compressed image published by the cameras

```
$ rosrun sign_vision sign_vision
```

Then play back a bag file (replace bagfile.bag)

```
$ rosbag play bagfile.bag
```