Lane Detection Using Particle Filtering

Abhishek Bodas, Aditya Indoori, Gaurav Bahl, Prashant Sahu, Pushkal Reddy

CS 682: Computer Vision

Computer Science Department

The Volgenau School of Engineering

George Mason University

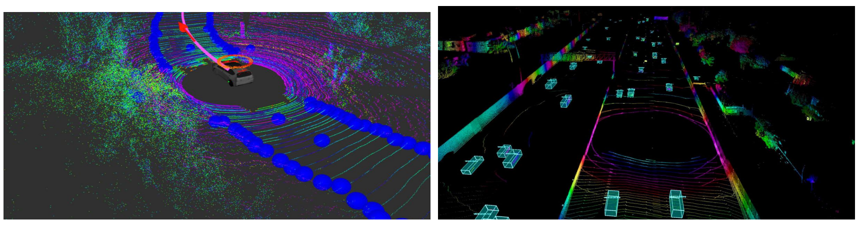
Table of Contents

Abstract

Road accidents are a leading cause of death in today's world. The ideal way to prevent them is to decrease mistakes made while driving. Enhancing a car with smart features to detect the road can assist a driver. In an intelligent vehicle system, lane detection plays a crucial role. There are several lane detection strategies, each having their own working standards, foundations, benefits, and faults. In this project, we have implemented particle filtering on KITTI datasets using a combination of various computer vision techniques and tools like OpenCV. This technique plots particles on the left and right lanes, row by row. Further by identifying a winning particle on each image for both left and right lanes leads to a collection of particles that identify the lane. In addition, these particles are combined to form a line. They are plotted on the original image to show the detected lane in different colors. This paper shows the results of the algorithm applied to one of the datasets from KITTI., The report concludes by mentioning techniques for improvement such as shadow removal, improving the image transforms, and optimizing parameters.

Introduction

Each year, around millions of people, are killed due to road and traffic-related accidents. The main reason for these accidents is human drivers in cars.
Humans easily get distracted, fatigued, and tired while driving a vehicle for a long time. This leads to a number of errors in their judgment. The most guaranteed way to save human lives is to prevent such misjudgments. A computer can never get tired or distracted even when performing an infinitely lengthy task. Using a computer to assist a human driver in a car can improve the safety of everyone on board.
In recent times, few companies have even begun rolling out cars with complete autonomous driving capabilities. Parallelly, a number of researchers are shifting their focus towards self-driving cars.
The first step for such a car is to detect the lane it is on. Without properly detecting a lane, one can almost assure that the car will cause an accident. However, if a car can properly detect a lane, it can prevent the car from sudden deviations, warn the driver when they get dangerously close to another lane, assist in changing lanes safely and lead to an overall safe driving experience in any vehicle.
Our project aims at building a lane detection and tracking system based on particle filtering. We build a geometric lane model that can be applied to not only linear roads but also to curved roads. We realize lane detection and tracking using the particle filter algorithm which is the main focus of the current project.


Insight on how a car sees its environment

Techniques and Tools

Detecting the lanes on a road translates to tracking the lane markers along a road. The field of computer vision offers a wide variety of techniques, tools, and concepts to track and recognize objects. Some popular techniques used in tracking objects are Lucas-Kanade Tracker, Kalman Filter, and Particle Filters. Some popular tools used to implement computer vision techniques are OpenCV for Python and Matlab.

In our project, we have chosen to use Particle Filters and OpenCV for Python to implement lane detection/tracking. Particle filters work great when we want to track an object that does not have a specific shape across both space and time. OpenCV for Python is a very popular Python library employed by a number of leading research institutes and computer-based companies.

Particle Filtering

The general function of a particle filter is to track a set of features based on a model that describes the motion of those particles. The particles are tracked within the same frame and through successive frames.

Particle

A particle is represented by a multi-dimensional vector, which is a combination of the location of the particle such as (x,y,z) and a descriptor such as (height, width).

In our project, we define a lane marker as a particle. Our filter processes the image row by row (along the y-axis), from top to bottom. A lane has both left and right edges. For each row, the position of the edges if defined by the column the lane is at (x-axis). Since a lane is effectively a line, we use the slope as the descriptor for our particle.

The vector representation of our particle is:
[l, r, b, g]
l: the column (x-coordinate) of the lane’s left edge
r: the column (x-coordinate) of the lane’s right edge
b: the slope(in radians) of the lane’s left edge
g: the slope(in radians) of the lane’s right edge

General Algorithm

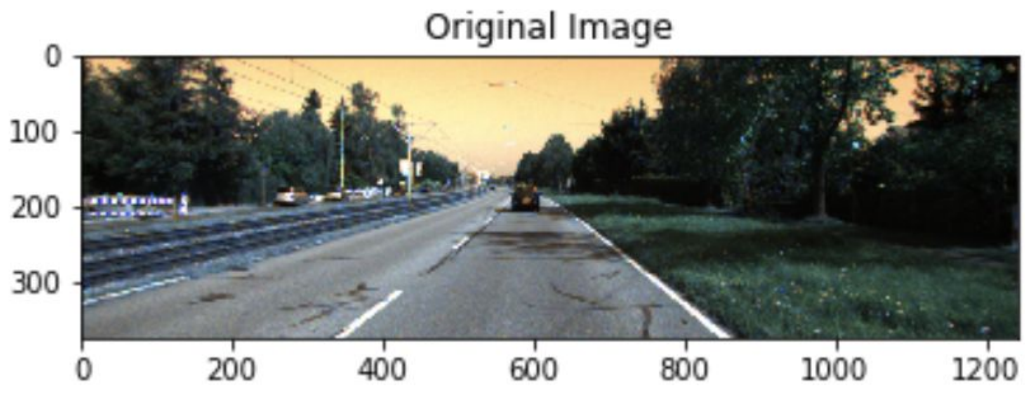
The steps involved in the particle filter, for tracking a particle across an image or multiple images are:

1) **Initialization:** We detect the (l,y) and (r,y) coordinates of the bottom-most row. We then set the initial slopes b = 0 and g = 0. Thus, we have our initial particle [l,r,b,g]
2) **Row Update & Tracking:** Beginning from the bottom row, we track our particles at every 5th row.
3) **Particle Generation:** In this step, we generate 200 random particles around the predicted particle using a uniform distribution.
4) **Weight Calculation:** Each generated particle is assigned a weight calculated based on how far it is from the predicted one.
5) **Resampling:** The resampling process decides which of the generated particles are considered for plotting the lane on the next row. We repeat the process from step 2.

Implementation

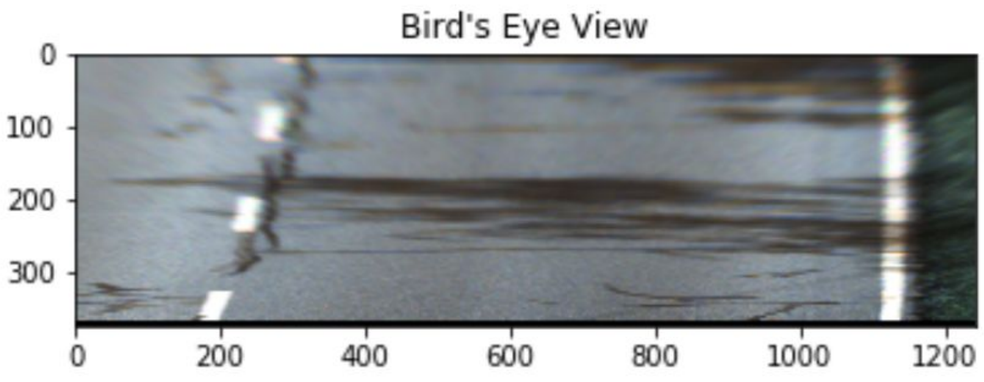
Image

In a real-life scenario, we will be getting the input from a video camera positioned on the vehicle. The video provided by the camera will be the input stream of RGB images for the Particle Filtering algorithm. In our project, we use a similar data set that was acquired from KITTI. KITTI is a project developed by Karlsruhe Institute of Technology and Toyota Technical College in Chicago. They have collected images by using a fleet of cars which they are using for autonomous driving. The data collected by them has been made available to the public for academic and research purposes. We have chosen a data set of Urban Marked Roads from their database. The data set provides RGB images with a resolution of 375 X 1242 of urban roads with marking on both sides (dotted and solid).



Inverse Perspective Transform

In order to process the images and apply particle filtering, we need our lanes to be parallel. But the original images we receive have a perspective effect as the camera is mounted on the car. We can convert these images to have parallel lines by using a 4 point transform which transforms the image to a top view of the scene also known as bird's view image.



```
def inverse_perspective_transform(img):
    dst_size = (img.shape[1],img.shape[0])
    src=np.float32([(270,360),(890,360),(660,210),(520,210)])
    dst=np.float32([(10,360), (1230, 360), (1230,10), (10,10)])
    Matrix = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, Matrix, dst_size)
    # Alternate to cv2.getPerspectiveTransform() is cv2.findHomography()
#     Matrix2 = cv2.findHomography(src, dst)
#     warped2 = cv2.warpPerspective(img, Matrix2[0], dst_size)

    return warped
```

This process is effective only under the condition that the incoming images are on a flat surface. Converting the images into this format makes it easier for the algorithm to detect edges in the images being processed which directly impacts the efficacy of the process.

Sobel

The part of the image that we are interested in is the lane edges which will be running vertically (which sometimes might be curved) from the bottom of the image to the top. In order to get these edges to pop more, we apply the Sobel Edge Detection technique from the Opencv library in python. We run the edge detection technique along the y-axis of the image so that we capture the vertical edges. Fig 3.a.

Code:

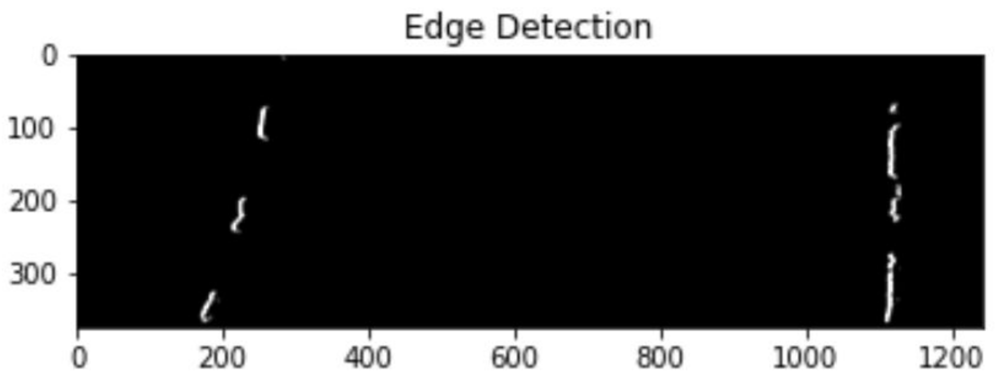
```
def edge_detect(warped):
    grayImg_warp = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)
    thres_warp = cv2.threshold(grayImg_warp, 215, 255,cv2.THRESH_BINARY)[1]
    gauss_warp = cv2.GaussianBlur(thres_warp,(11,11),0)

    sobel_warp = cv2.Sobel(gauss_warp,cv2.CV_8U,1,0,ksize=3)

    # Calculating distance transform
    dst_tfm = cv2.distanceTransform(255-sobel_warp, cv2.DIST_L2, 5)

    return dst_tfm
```

Result:



Distance Transformation on Image

A distance transformation converts a binary digital image, consisting of feature and non-feature pixels, into an image where all non-feature pixels have a value corresponding to the distance to the nearest feature pixel. We have calculated distance transformation on the edge detected image to perform particle filtering. Once the EDT is applied to the image, we find the point/particles with intensity value dropping on both sides.

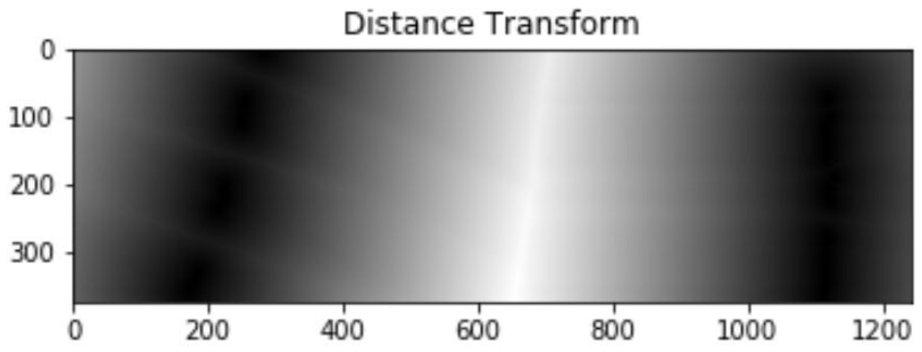
For example: 100, 50, 10, 5, 10, 40, 70

So in this case, **5** is the point where intensity on its left and right has higher values. Hence that point is considered as a valid edge point on the image.

Code:

```
dst_tfm = cv2.distanceTransform(255-sobel_warp, cv2.DIST_L2, 5)
```

Result:



Applying particle filtering row by row

A particle is a point in N-dimensional space, which is technically referred to as Particle Space. Each particle has 2 characteristics: location and descriptor.

Particle Filtering uses the concept of projecting multiple particles on the image and assigning weights to a subset of the particles which were closest to the actual edge.

Algorithm and Implementation of Particle Filtering

Step 1 - Particle Generation:

We plot 500 random generated particles on the first row in the image. The descriptors (beta and gamma) of these particles are set to 0 for the initial row.

Code:

```
def generate_random_particles(row_length, selected_particles):
    predicted_particles = np.array([])
    if(len(selected_particles) == 0):
        L = uniform(loc=0, scale=row_length/2).rvs(numberOfParticles)
        R = uniform(loc=row_length/2+1,
scale=row_length/2-1).rvs(numberOfParticles)
        B = np.zeros(numberOfParticles,dtype='uint8')
        G = np.zeros(numberOfParticles,dtype='uint8')
        predicted_particles = np.stack((L,R,B,G), axis=-1)

#          print("normal: ",predicted_particles)
#          predicted_particles = np.array(np.meshgrid(L,R,B,G)).T.reshape(-1,4)
#          print("new: ",predicted_particles)

    else:
        num = int(numberOfParticles/len(selected_particles))
        for particle in selected_particles:
            L = uniform(loc=particle[0]-20, scale=40).rvs(num)
            R = uniform(loc=particle[1]-20, scale=40).rvs(num)
            B = uniform(loc=particle[2]-0.5, scale=1).rvs(num)
            G = uniform(loc=particle[3]-0.5, scale=1).rvs(num)

            if len(predicted_particles) == 0:
                predicted_particles = np.stack((L,R,B,G), axis=-1)

            else:
                predicted_particles = np.vstack((predicted_particles,
np.stack((L,R,B,G), axis=-1)))

    return predicted_particles
```

Step 2 - Get reference points from DT: An image or frame of a video contains a lot of content, but our area of interest are 2 white lanes on which the lane detection algorithm is to be applied. These reference points, denoted with ‘L’ and ‘R’ are points that denote the left and right lanes respectively. We store these values for comparison to the particles generated and to assign weights to each of them. In order to generate ‘beta’ and ‘gamma’ values for each particle, we use the previous winning particle’s ‘beta’ and ‘gamma’ values.

```
def get_actual_LR_points(arr_bt_row):
    lr = []
    left,right = 0,0
    mid_length = len(arr_bt_row)/2

    for i in range(1,len(arr_bt_row)-1):
        prev_pix_in = i-1
        next_pix_in = i+1
        next_diff = np.rint(arr_bt_row[i] - arr_bt_row[next_pix_in])
        prev_diff = np.rint(arr_bt_row[i] - arr_bt_row[prev_pix_in])
        if(arr_bt_row[i]<=10 and next_diff<=0 and prev_diff<=0):
            lr.append(i)

    central_number = 0 if len(list(filter(lambda k: k > mid_length, lr)))==0
else list(filter(lambda k: k > mid_length, lr))[0]
    central_index = lr.index(central_number) if central_number != 0 else
len(lr)-1

    if central_index>0:
        left = sum(lr[:central_index])/len(lr[:central_index])
    if central_index!=len(lr)-1:
        right = sum(lr[central_index:])/len(lr[central_index:])

    return left,right
```

Also to get beta and gamma, we are using previous particle and current actual particle and calculating the angle from the vertical straight line.

```
def get_beta(l, prev_particle):
    return math.atan((l-prev_particle[0])/delta)

def get_gamma(r, prev_particle):
    return math.atan((r-prev_particle[1])/delta)
```

Step - 3 Calculate Weights: Once the 500 random particles are generated, we assign each new particle a weight. The weight represents how likely the particle is part of the lane. We calculated the weight of each particle as the sum of mean squared differences between the particle and the previous winning particle.

```
def get_weights_MSE(particles, prev_particle):
    weights = np.array([])
    for particle in particles:
        MSE = np.mean(np.square(np.subtract(prev_particle,particle)))
        weights = np.append(weights,1/MSE)
    normalized_weights = weights/np.amax(weights)

    return normalized_weights
```

Step - 4 Resample & Plot:

Resampling (also known as condensation) is applied once the weight is assigned to each particle in the sample space and based on a certain strategy winning particle is calculated. We have calculated the winning particles by selecting the five highest weights from the 500 random particle weights which were generated. We take the mean of the two highest weights and generate a winning particle which is plotted on the image as the projected lane.

```
def get_winning_particles(weights, particles):

    weights, particles = zip(*sorted(zip(weights, np.array(particles))))

    return np.array(particles)[-5:]
```

Step - 5 Repeat for next row:

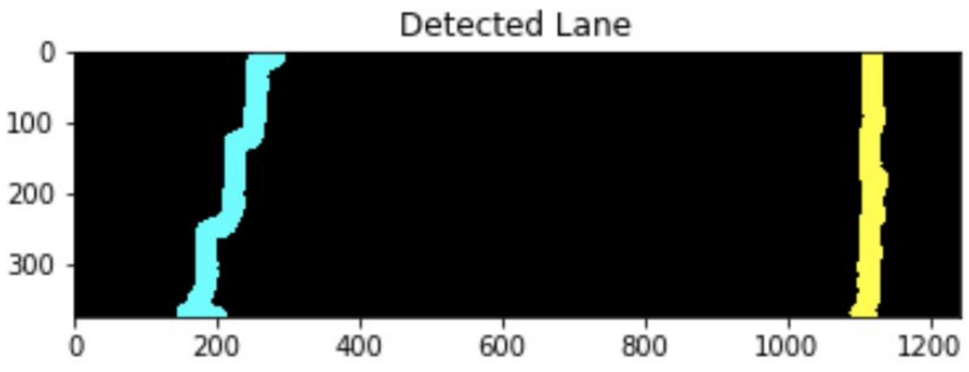
We repeat the same process for the next row. However, if we don't have L and R values present for the next row (if the lane is dotted) then we use the previous L and R values to assign weights to the new particles.

```
# Calculating the particles for each row and storing the points to plot for
left lane and right lane
    for row in range(starting_row,0,-delta):
        prev_particle, selected_particles = particle_filter(dst_tfm, row,
prev_particle, selected_particles);

        # calculating the mean of the selected (winning) particles and setting
left and right lane points
        mean_selected_particle = np.mean(selected_particles, axis=0)
        #setting left lane points
        if(int(mean_selected_particle[0])< dst_tfm.shape[1] and
int(mean_selected_particle[0]) > 0):
            test_image2[row, int(mean_selected_particle[0])] = 255
            if len(left_line_points) == 0:
                left_line_points = np.array([int(mean_selected_particle[0]),
row])
            else:
                left_line_points =
np.vstack((left_line_points,np.array([int(mean_selected_particle[0]), row])))

        #setting right lane points
        if(int(mean_selected_particle[1]) < dst_tfm.shape[1] and
int(mean_selected_particle[1]) > 0):
            test_image2[row, int(mean_selected_particle[1])] = 255
            if len(right_line_points) == 0:
                right_line_points = np.array([int(mean_selected_particle[1]),
row])
            else:
                right_line_points =
np.vstack((right_line_points,np.array([int(mean_selected_particle[1]), row])))
```

Result for one image:



Perspective Transform

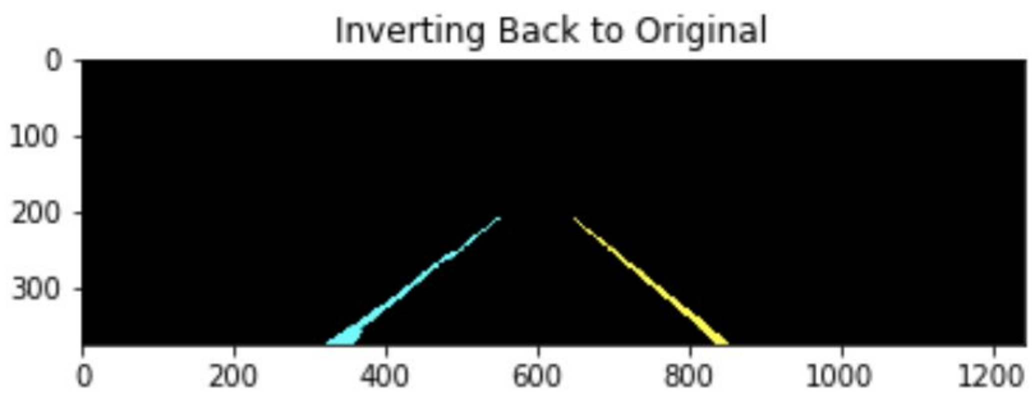
Once our particle filter detects the lane along with our entire image, we transform the detected image back to the perspective view. This is done by performing the exact opposite of the 4-point technique used in the Inverse Perspective Transform step.

```
def perspective_transform(img):
    dst_size = (img.shape[1],img.shape[0])
    dst=np.float32([(270,360),(890,360),(660,210),(520,210)])
    src=np.float32([(10,360), (1230, 360), (1230,10), (10,10)])

    Matrix = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, Matrix, dst_size)

    return warped
```

Result:



Merge with Original Image

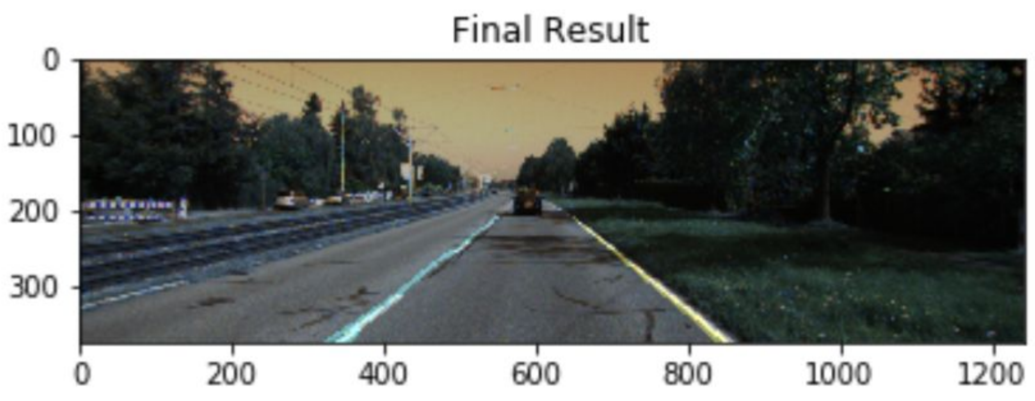
In the final step, we merge the perspective transformed image of the detected lanes with the original image. In a way, we superimpose the detected lanes on the raw data. This allows the car to detect the lanes in its actual environment.

```
def merge_images(image1, image2):
    alpha = 0.7
    beta = (1.0 - alpha)
    dst = cv2.addWeighted(image1, alpha, image2, beta, 0.0)

    plot_image("Final Result", dst)
    display_image("Result", dst)

    return dst
```

Result:



Video Demo Link

https://streamable.com/yf3nyr

Challenges Faced

Next Frame

Once our system detected the lanes in the current frame, it would start processing the next frame. The next frame was only slightly different from the previously processed frame. Yet, our system would begin from the bottom-most row and continue all the way to the top-most row. This unnecessary processing leads to a significant lag in detecting the images.

Shadows In Images

The Kitti dataset contains frames in which there are some shadows in the images due to trees and poles. This would prevent the system from properly assigning weights to the generated particles. The lanes detected in these parts were often deviating from the positions of the actual lanes. We tried some shadow removal algorithms but none worked perfectly and all increased the processing time.

Absence of lane

When lanes are not present or in case of the dotted lane we were unable to render the L and R values. So in order to handle that if the lane is not found we pick the previous L and R values and assign them to current L and R values.

Future Work

We have met most of the goals that we planned on achieving such. However, like any good project, there is always more room for improvement. With additional time and expertise, here are some things that we could have done better.

Increase time efficiency

On uncalibrated and non-optimized hardware, our system is able to detect lanes at a rate of 1 frame per second. We believe, by optimizing the weight function and tuning the parameters such as the number of particles, the number of rows shifted, and the shifting of rows per image, we can improve the performance on the algorithm side. Running the system using a calibrated camera and optimized CPU would further improve the rate of processing and detecting lanes.

Camera Calibration

The KITTI dataset for road-lanes provides a number of images and the calibration details of the camera used. However, for our project, we used the 4-point technique to transform the perspective of the image from front-view to top-view. We believe that by correctly using the calibration details provided by KITTI, we can improve the image obtained by the perspective transformation. An improved perspective will result in cleaner lanes, which will result in lesser noise, therefore faster processing and detection of lanes.

Contributions

Abhishek Bodas

I implemented an image transformation technique: edge-detection. Carried out image pre and post-processing steps. Helped out in documentation and creating the presentation. Played a supporting role in developing the Particle Filtering algorithm, which I consider the most complex but well-implemented part of the project. One of the best learnings from this project is playing around with the parameters to fine-tune the algorithm. With faster processing units and finer calibrations, I believe this project could be considered as an effective application of OpenCV.

Aditya Indoori

As a member of the team, I learned the importance of computer vision and gained insight into object tracking. I was able to contribute to designing the pseudo-code for particle filtering. I also helped in writing the code to read the raw data and perform inverse perspective & distance transform. I feel that we could have used additional techniques to optimize the time taken to process each image. In the end, I enjoyed and learned a lot about computer vision from this experience.

Gaurav Bahl

I learned the concept of object tracking and detection. Understood the concept of particle filtering and condensation algorithm for assigning weight and selecting the winning point out of the whole sample space of points. I helped in writing and optimizing the code using NumPy arrays. I also helped with creating reports and presentations. I feel that our results are pretty good, but further optimization can be applied to make the experience more flawless. It was a great experience to apply computer vision concepts to real-world problems of lane detection.

Prashant Sahu

During this project, I learned to apply the techniques we picked in class such as Edge Detection, Distance Transform, and Particle Filtering. It has been a rewarding experience as I got to see the impact of applying these techniques in practical applications such as lane detection. In order to make the lane detection more efficient, I worked on parameter tuning for optimization. A major issue that we ran into was that our algorithm was losing its track due to shadows and other disturbances on the road. I tried to work on shadow removal to minimize this issue but we believe that it can be further improved. Furthermore, I implemented code for 4 point Perspective Transform which converts the perspective images we received as input to the bird’s eye view.

Pushkal Reddy

Learned new techniques and concepts which are crucial for computer vision and augmented reality practices. In this project, I learned the application of particle filtering algorithms from academic papers and contributed to developing the algorithm for our project. I got to the first-hand experience in implementing concepts we learned in class such as Perspective Transform. I worked in optimizing the Perspective Transform by modifying to pick the areas of interest in the incoming frames. I implemented the code to create the Left and Right lane arrays into an image and merge that result with the original image to generate an output with detected lanes being plotted on the street. We learned that having a better camera calibration can avoid a lot of the additional processing we did to filter out noises and disturbances. This can improve our performance time and also improve the quality of detection being done by the algorithm.

Conclusion

The goal to save human lives from unnecessary road accidents can be achieved by making our vehicles smarter. Among other tasks, a smart vehicle must be able to detect the lane it is driving on. Our work is a proof that with a proper and optimal implementation, it is not so difficult to build a system just to detect lanes. However, lane detection in the real-world can get a lot more complicated due to the nuances involved in driving. The number of factors involved in the process of completely autonomous driving increases exponentially. By employing powerful tools like OpenCV and concepts like particle filtering, any vehicle can detect lanes with high accuracy.

References

[1] Sehestedt Stephan, Kodagoda Sarath, Alempijevic Alen & Dissanayake Gamini, 2007, 'Efficient Lane Detection and Tracking in Urban Environments', 3rd European Conference on Mobile Robots (ECMR 2007), pp. 78-83

[2] S. Sehestedt, S. Kodagoda, A. Alempijevic, and G. Dissanayake, "Robust lane detection in urban environments," 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA, 2007, pp. 123-128

[3] Dataset: http://www.cvlibs.net/datasets/kitti/raw_data.php?type=road