Udacity Self Driving Car Nanodegree –

Project: Vehicle Detection and Tracking

By:

Saurabh Sohoni

Udacity profile: <https://profiles.udacity.com/p/u183059>

Contents

1. Introduction 3

2. Project Goals 3

3. Implementation of Project Rubric Points 3

3.1 Write up / README 3

3.2 Camera Calibration 3

3.3 Pipeline on Test Images 4

3.4 Pipeline on Video 12

3.5 Discussion 12

4. References 13

# Introduction

This document acts in support with code and output files submitted in order to implement ‘Vehicle Detection and Tracking’ project. In this project, a pipeline is created to detect cars on a road and track their path. This is achieved with the help of Support Linear Vector (SVC) classifier, one of the widely used classifiers in machine learning for real-time applications. A camera mounted on the hood of the vehicle in center records path ahead of a car. Project pipeline created in this project is then applied to this video recording to detect other cars in ahead and beside and track their path.

# Project Goals

Following were the goals of this project:

1. Extract images containing cars and not containing cars and label the data in order to train a classifier. The cars and not cars images were provided by Udacity. These example images come from a combination of the [GTI vehicle image database](http://www.gti.ssr.upm.es/data/Vehicle_database.html) [1], the [KITTI vision benchmark suite](http://www.cvlibs.net/datasets/kitti/) [2], and few examples extracted from the video recording done by a camera mounted on the hood of the car.
2. Extract Histogram of Gradients (HOG) features from each training image and form a vector of features.
3. Normalize and randomize the feature vectors prepared in above step and feed them to a classifier.
4. Train a classifier to detect distinguish between images of cars and not cars and validate the accuracy on a test set.
5. Implement a sliding window technique to divide an image frame from the video recording into small sections. Apply the classifier created above on sections of image to search for cars and draw predictions.
6. Draw a bounding box in the corresponding section of image where cars were detected.
7. Apply car detection algorithm mentioned above to a video stream and maintain a track of cars. Reject spurious false positives by comparing their detections with track of cars.
8. Summarize the approach and results in a report.

# Implementation of Project Rubric Points

Following section lists down various rubric points for this project and also details out the implementation strategy followed:

## Write up / README

This report lists methodology used in creation of vehicle detection and tracking pipeline and supports the code written for various python methods used in this project.

## Histogram of Gradients (HOG)

#### **3.2.1 Selection of hyper parameters for HOG**

Selection of right hyper parameters is important as to ensure the algorithm results in high accuracy and the processing time is lesser. For this project, following hyper parameters were selected:

1. Color space (YCbCr) – In general, cars are more saturated in color than other objects in the background. This feature is prominent in YCbCr color space and hence, original RGB images were converted to YCbCr color space before extracting HOG features.
2. Number of HOG channels (All channels) – HOG features were calculated for each channel in YCbCr color space of images in order to ensure color, saturation and illumination features are retained.
3. Number of orientations (11) – Number of orientations corresponds to the number of directions in which gradients in an image are oriented. For this project, 11 orientations were chosen. As a result, gradient features were separated by roughly 33 degrees (360 degrees for complete circle. divided by 11 orientations).
4. Pixels per cell (16) – While calculating HOG features for an image, the image is further sub-sampled into small boxes and oriented gradients are calculated for each box. This feature of HOG provides information on the shape of objects and edges around the corner. Images in training set were of the size 64 x 64 (width x height). Pixels per cell value of 16 suited the best to achieve high accuracy and faster computation speeds.
5. Cells per block (2) - Features in neighboring cells in an image may be similar (in case of no edge) or may be very different (in case of edges). Hence the magnitudes of features may be small or large but all features are equally important. In order to reduce the effect of magnitude on features, local features in a given cell are normalized. In this project, two neighboring cells were used to create a block of 4 cells and then normalization was implemented.
6. Square root of transforms – This is another form of normalization which uses power law of ‘gamma’ normalization scheme. As it is known to reduce the effects of dark shadows and bright illuminations in the image, this technique was also applied.

Using the entire configuration parameters described above, the HOG image obtained for a car from training set is shown below:

A camera installed on the hood of the car is used to capture images of road and scene in the front. To calibrate this camera, Udacity had provided with 20 calibration images captured by the same camera with the same setup. These were the images of a 9 x 6 (9 columns and 6 rows) chessboard taken from different angels. Camera calibration was achieved by following the steps given below:

1. Object points on a chessboard in the real world were created. Object points are nothing but points in 3 dimensional space (x, y, z) on a chessboard placed in 3D world. In the scope of this project, it was assumed that chessboard, with size of 9 x 6, used for taking images was placed on a flat surface. Hence, z was 0 for all object points. The object points then looked like (0, 0, 0), (0, 1, 0) up to (5, 8, 0).
2. Each calibration image taken as input was searched to find chessboard corners. If the corners are found, these corners were used as the image points. An example is shown below:

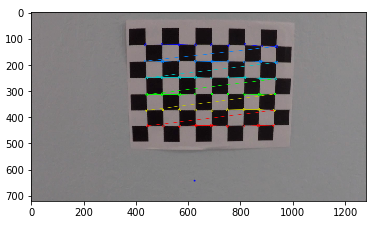
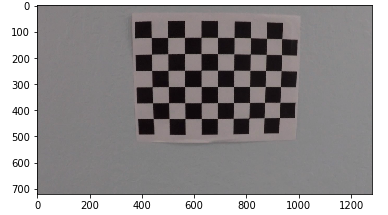


Figure 1b: Image with chessboard drawn [9b]

Figure 1a: Camera calibration image [9a]

Figure 1a is one of the calibration images. Chessboard corners found on this image are drawn with different colors.

1. Step 2 is repeated for all of the calibration images and a list of detected image points in formed. Object point are same for every image, hence, are duplicated and appended to a list object points for each image in which image points were found.
2. Lists of image points and object points are then used to generate camera matrix and distortion coefficients. These coefficients were used in next step to correct distortion in images.

Python code for camera calibration can be found in calculate\_camera\_calibration() method in project pipeline.

## Pipeline on Test Images

Lane finding pipeline comprised of several steps as described below:

#### **3.3.1 Distortion Correction (Undistort images)**

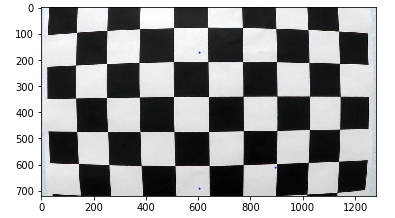
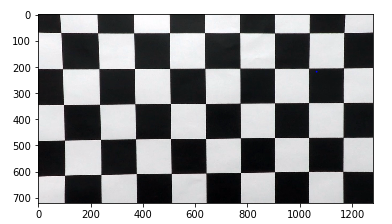
Image capture in modern cameras taken place with the help of lenses. As a result, such a camera does not follow ‘Pinhole camera model’ [1]. This may introduce tangential and radial distortion in the images captured under certain conditions. To correct this type of distortion, camera matrix and distortion coefficients calculated in camera calibration steps are used to obtain undistorted images as output. An example with a chessboard image and image taken on a road is shown below:

Figure 2b: Distortion corrected chessboard image [9d]

Figure 2a: Distorted chessboard image [9c]

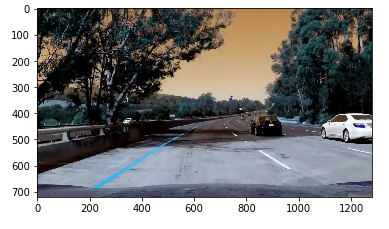
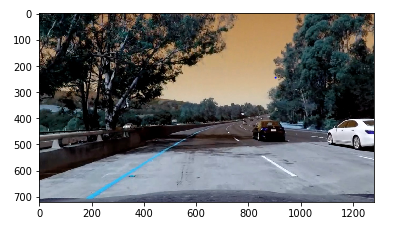


Figure 3b: Distortion corrected image of road [9f]

Figure 3a: Distorted image of road [9e]

Python code for distortion correction can be found in undistort() method in project pipeline.

#### **3.3.2 Image Gradient and Color Thresholding**

Detection of lane lines starts with detecting edges and identifying continuous white and yellow lines. Edge detection is achieved by calculating gradients in an image and information on colored lines can be extracted using color thresholding on different color planes in different color spaces. This is described below:

1. Image gradients:

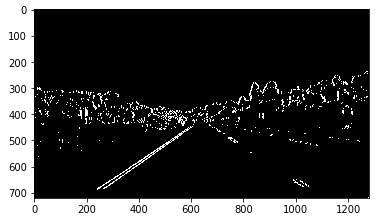
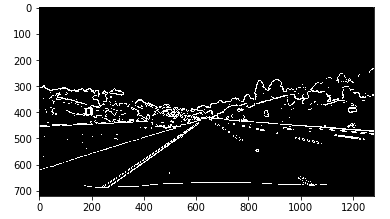
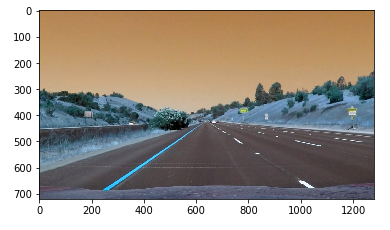
Image gradients are nothing both Sobel derivatives taken either across horizontal (X) axis or vertical (Y) axis or both in an image. These gradients are useful in detecting edges around lane lines. Since lane lines are almost vertical, this project used gradient along x direction (gradx). Also, the magnitude of combined gradient (mag) is useful to make lane lines prominent and add weight to them against other edges in the image. In this project, information about edges obtained from gradx and mag images was combined to obtain an image with lane lines. An example is shown below:

Figure 4a: Original road image [9g]

Figure 4b: Gradient in 'X' direction (gradx) [9h]

Figure 4c: Magnitude of gradients (mag) [9i]

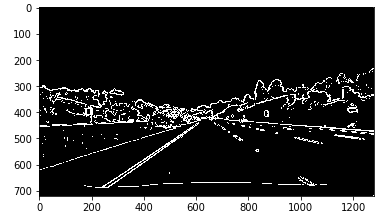


Figure 4d: Combined gradx and mag image [9j]

Python code for obtaining these gradients can be found in sobel\_threshold(),magnitude\_threshold() and apply\_grad\_threshold() methods in the pipeline. For obtaining gradx, sobel kernel of shape 11 x 11 was used with pixels in the range [40, 150] retained. For obtaining mag, sobel kernel of shape 11 x 11 was used with pixels in the range [50, 150] retained.

1. Thresholding in RGB and HLS color spaces:

Image gradients work fine in lane lines detection when the lane lines are brighter than other background pixels in the image. This is the reason gradients fail in detection of yellow lane lines as yellow is mapped to less bright pixels when converted from RGB to gray. Also, gradients fail in case a shadow is cast on the road as edges cannot be detected accurately.

Problems mentioned above can be overcome by using color thresholds in RGB and HLS images. White lane lines are detected accurately by using pixels belonging to the range [220, 255] in Red (R) channel of an RGB image. This is shown below:

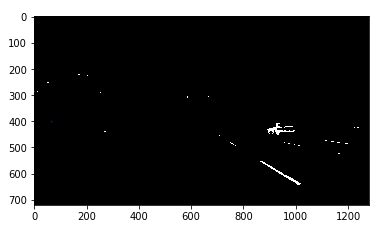


Figure 5b: R channel thresholded image [9l]

Figure 5a: Sample image of road [9k]

As evident from the result, R channel picks the white lane line effectively even in the presence of shadow. But it fails to detect yellow colored lane line.

S channel of Hue, Lightness, Saturation (HLS) color space [2] is invariant to brightness in the image. Hence, pixels belonging to [160, 200] values in S channel are used to pick yellow lane lines in the image. This is demonstrated in example below:

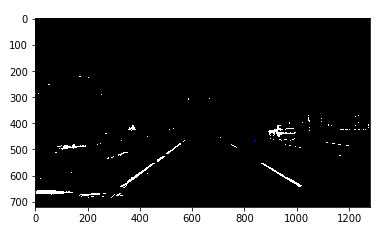
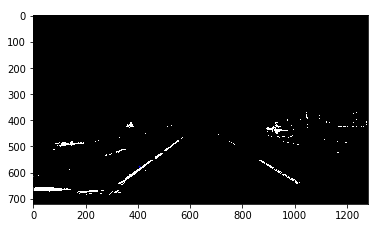


Figure 5d: R and S channel combined [9n]

Figure 5c: S channel thresholded image [9m]

In this project, information obtained from R and S channel thresholded images was combined to get an image with lane lines. An example is shown in the combined image above.

Python code for obtaining these color thresholds can be found in apply\_color\_threshold() method in the pipeline.

1. Merged gradient and color thresholded image:

Finally, characteristics obtained from combined gradient and combined color thresholded images were merged to obtain a gray scale image with lane lines detected prominently. Example of such an image is shown below:

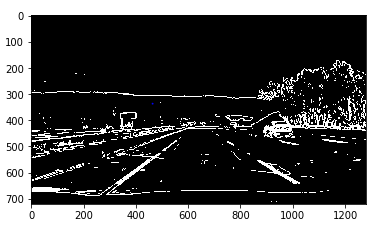


Figure 5e: Original image of road [9k]

Figure 5f: Combined gradient and color thresholded image [9o]

Python code for obtained combined gradient and color thresholded image can be found in the method detect\_lane\_lines() in the pipeline.

#### **3.3.3 Perspective transform to get a bird’s eye of lane lines**

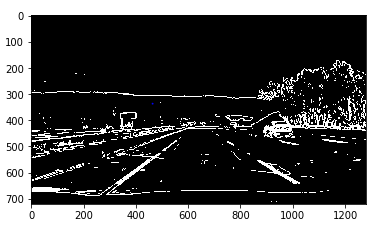
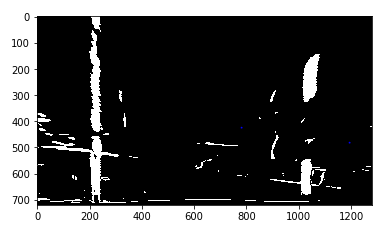
Lane lines detected by techniques described above don’t provide much information on the curvature of road. Also, parallel lane lines appear to converge in a 2D image due to phenomenon of image perspective [3]. To get some information on the lane curvature, lane detected binary images are warped to get a bird’s eye view of the lane lines somewhat similar to the one shown in example below:

Figure 6b: Image transformed to bird’s eye view [9p]

Polygon in figure 6a was transformed to rectangle bounded by (200, 720), (200, 100), (1050, 100) and (1150, 720) starting from bottom left in clockwise direction.

Figure 6a: Binary image with lane lines [9o]

Polygon bounded by (225, 700), (555, 475), (740, 475) and (1140, 700) starting from bottom left in clockwise direction is warped.

Perspective transform is obtained by choosing a polygon in lane image and mapping it to a rectangle over entire dimension of an image. Details on polygon chosen in this project is mentioned above in the image caption.

Python code for implementation of perspective transform can be found in the apply\_perspective\_transform() method in the pipeline.

#### **3.3.4 Identification of Lane lines in warped image**

As shown above, warped image is a binary and has lane line markers in bottom 70% of pixels in Y axis. Lane lines are white and can be detected by looking for non-zero pixels. To detect left and right lane lines and fit them to a second degree polynomial, sliding window technique was used. In this technique, initial guess on locations of lane lines is done by taking a histogram over bottom 70% pixels in Y axis of the warped image. Such a histogram is shown below:

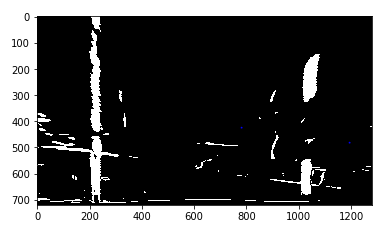
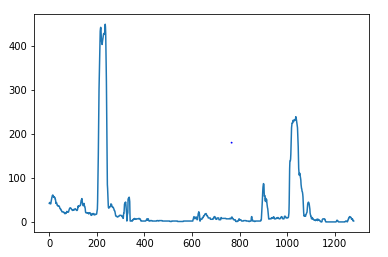


Figure 7b: Histogram taken over bottom 70% in Y axis [9q]

Figure 7a: Warped image [9p]

The two prominent peaks in left half and right half of the histogram are nothing but the locations of lane lines. The sliding window algorithm starts from bottom of warped image at locations (hist\_peak\_left, image\_y\_size) and (hist\_peak\_right, image\_y\_size) for left and right lanes respectively. Then the algorithm goes on detecting non-zero pixels in the boundary of the window and draws number of windows until it reaches the top of the image. In this process, the algorithm caches pixel coordinates and then fits them to a second degree polynomial each for the left and the right lane respectively. This is shown in the example below:

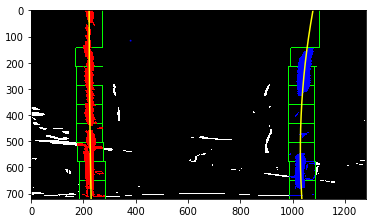


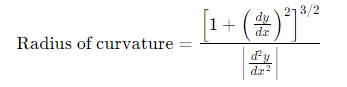
Figure 8: Sliding window search detects lane lines and fits them to polynomial [9r]

In this project, the size of window was chosen to be 100 in X axis and number of sliding windows to be 10. Also, if a cluster of 2000 non-zero pixels is detected during the search, this algorithm shifts takes the centroid of those pixels and draws a window accordingly.

Python code for implementing sliding window search can be found in sliding\_window\_search() method in the pipeline. Most of the code was reused and is attributed to code made available in class [4] by Udacity.

#### **3.3.5 Radius of Curvature and Vehicle Position**

For a curve represented by the function y = f(x), the radius of curvature can be calculated using the formula given below [5]:



Left and right lane lines detected in above section were each fitted to a second degree polynomial. A generic second degree polynomial can be represented by:

y = f(x) = Ax2 + Bx + C ; A, B, C being constants

And the first and second order derivatives are given by:

(dy/dx) = 2Ax + B

(d2y/dx2) = 2A

Hence, the radius of curvature is given by:

Rcurve = (1 + (2A + B)2 )3/2 / |2A|

For the position of the vehicle, it is assumed that the camera is mounted at the center of the car. The deviation of the midpoint of the center of the image from the center of the lane is the relative position of vehicle in the lane area.

The center or centroid of lane area is calculated by calculating moments of curve.

The radius of curvature and vehicle position thus obtained is in pixel space and needs to be converted to meters space. This was done with an approximation that lane is about 30 meters long and 3.7 meters wide adhering to US regulations [6] that require a minimum lane width of 12 feet or 3.7 meters.

Python code for calculating radius of curvature and position of vehicle can be found in extract\_radius() and advanced\_lane\_detection() methods respectively. Code for detection of radius of curvature is reused and is attributed to code made available in class [7] by Udacity.

#### **3.3.6 Result Image with Marked Lane Lines**

In this last step of the pipeline, area enclosed by left and right lane lines is filled with dark green color to form an image in bird’s eye view. This image is then warped back to original image captured by camera by taking inverse of perspective transform. Resulting images are shown below:

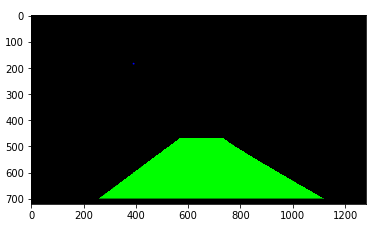
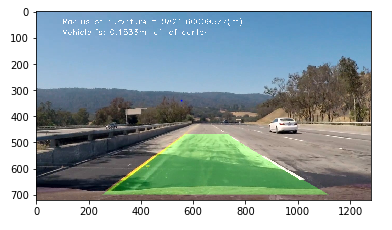


Figure 9b: Area enclosed by lane lines along with radius of curvature and vehicle position [9t]

Figure 9a: Area enclosed by lane lines in bird's eye view [9s]

Python code for warping lane lines back to original camera image can be found in warp\_lanes() method. This code is reused and is attributed to code made available in class [8] by Udacity.

## Pipeline on Video

The pipeline created for testing on images is then applied to a video (project\_video.mp4) taken around a freeway, provided by Udacity.

VideoFileClip utility from moviepy,editor Python library was used to extract image frames from the video. Pipeline applied to every frame was able to detect lane lines, area bounded by lane lines, radius of curvature of road and vehicle position effectively. The resulting video is saved in output\_video folder submitted along with this document.

Python code for implementing project’s pipeline can be found in laneDetectionInVideo() and advanced\_lane\_detection() methods.

## Discussion

#### **3.5.1 Challenges faced**

One big challenge faced during implementation was coming out with lower and upper thresholds for applying color thresholding to lane images. With different lighting conditions seen in test images, no unique limit value worked on all the images. This challenge was overcome with the help of information from image gradients. A combined image could detect lane lines effectively.

#### **3.5.3 Scenarios resulting in failure of pipeline**

Following are a few of scenarios where the pipeline created currently may fail:

1. Lane search technique used in this pipeline makes a guess of lane lines from the histogram of binary warped image. If this initial guess fails, the whole algorithm will fail.
2. Current algorithm is pipeline assumes that both left and right line will be detected and a second degree polynomial will be fit to each one of the lines. But in case the dashed lane lines are not seen, the whole logic of determining area bounded by lane lines will fail.
3. On country roads where lane lines are not bright enough, edge detection and color threshold will fail to detect lane lines effectively.
4. In winters, when snow is covered on some areas of road, it is very difficult to apply color thresholding techniques as snow is white in color.

#### **3.5.3 Solution to challenges and carving a robust pipeline**

To avoid failure of pipeline in scenarios mentioned above, few algorithms given below should be used along with existing pipeline:

1. An educated guess on initial position of lane lines can be made by caching last few lane line positions and comparing the current search. If a lot of deviation is found, the algorithm can fall back to lines detected in previous frame.
2. In case one of the lane lines is not found, either the algorithm should revert to lane lines found in last image frame or try to extrapolate behavior of missing lane from the one found.
3. In case of country roads, an assumption can be made that the road is narrow enough to accommodate only one vehicle. Hence the car can remain in the same lane forever and look ahead for surrounding obstacles using lasers or LIDARS.

# References

1. [1] [GTI vehicle image database](http://www.gti.ssr.upm.es/data/Vehicle_database.html)
2. [2] [KITTI Vision benchmark suite](http://www.cvlibs.net/datasets/kitti/)
3. [3] [Image Perspective](https://en.wikipedia.org/wiki/Perspective_(graphical))
4. [4] [Lane lines detection by using sliding window search in Python](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/2b62a1c3-e151-4a0e-b6b6-e424fa46ceab/lessons/40ec78ee-fb7c-4b53-94a8-028c5c60b858/concepts/c41a4b6b-9e57-44e6-9df9-7e4e74a1a49a)
5. [5] [Radius of curvature](http://www.intmath.com/applications-differentiation/8-radius-curvature.php)
6. [6] [US government specifications for highway curvature](1.%09http:/onlinemanuals.txdot.gov/txdotmanuals/rdw/horizontal_alignment.htm#BGBHGEGC)
7. [7] [Extracting radius of curvature from second degree polynomial in Python](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/2b62a1c3-e151-4a0e-b6b6-e424fa46ceab/lessons/40ec78ee-fb7c-4b53-94a8-028c5c60b858/concepts/2f928913-21f6-4611-9055-01744acc344f)
8. [8] [Visualizing area bounded by lane lines in Python](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/2b62a1c3-e151-4a0e-b6b6-e424fa46ceab/lessons/40ec78ee-fb7c-4b53-94a8-028c5c60b858/concepts/7ee45090-7366-424b-885b-e5d38210958f)
9. [9] Image references:
   1. /examples/camera-calibration.JPG
   2. /examples/chessboard-corners.JPG
   3. /examples/chessboard-distorted-image.JPG
   4. /examples/chessboard-distortion-corrected-image.JPG
   5. /examples/road-distorted-image.JPG
   6. /examples/road-distortion-corrected-image.JPG
   7. /examples/road-image-for-gradient-thresholding.JPG
   8. /examples/gradx-gradient-image.JPG
   9. /examples/mag-gradient-image.JPG
   10. /examples/merged-gradient-image.JPG
   11. /examples/road-image-for-color-thresholding.JPG
   12. /examples/r-channel-thresholded-image.JPG
   13. /examples/s-channel-thresholded-image.JPG
   14. /examples/r-and-s-channel-combined-image.JPG
   15. /examples/merged-gradient-and-color-image.JPG
   16. /examples/perspective-birds-eye-view-image.JPG
   17. /examples/histogram.JPG
   18. /examples/sliding-window-search.JPG
   19. /examples/area-enclosed-by-lanes.JPG
   20. /examples/ lane-area-and-annotated-image.JPG