Self Driving Car NanoDegree Program

Project: Advanced Lane Finding

Introduction

In this project, our goal is to write a software pipeline to identify the lane boundaries in a given video. This document details the description of the main pipeline and the project rubrics.

Steps Followed

Note: The below ten steps may be naive yet they complete the entire scope of the project.

1. Compute Calibration matrix and Distortion Coefficients.
    a. Given: Only A Set of Chessboard Images
2. Save the calibration in a file and apply the Calibration Correction to the raw images.
3. Create a thresholded binary image;
    a. Use Gradients,
    b. Color Transforms.
4. Birds-Eye-View;
    a. Apply perspective transform to rectify binary image
        i. Transform
        ii. Inverse Transform
5. Detect lane pixels.
6. Use Fit Polynomial to find lane boundaries.
7. Determine Curvature of the lane.
8. Find vehicle position with respect to center of the referenced lane.
9. Warp the detected lane boundaries back onto the original image.
10. Output;
    a. Visual display of the lane boundaries,
    b. Numerical estimation of lane curvature and
    c. Vehicle position.

How to Run the Project?

Please open a terminal in the project directory and type:

- python3 main.py
- Wait till the output video annotation is complete,
- The saved output video will be in the project directory.

Pipeline:

The my_lane_finding_advanced.ipynb notebook contains a step by step presentation of the designed pipeline and the main routines to process images and videos.

Rubric Points Explanation

1. **Camera Calibration**

Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

Solution:

I followed the steps and advice given in the Udacity Class lectures.

The function named "GetCalibrationParam(image_url)" is the main function for calibration in main.py file. It computes calibration matrix and distortion coefficients.

Chessboard Dimension: 9*6

As directed in the rubrics, OpenCV function cv2.calibrateCamera was utilized to calculate the correct camera matrix and distortion coefficients using the calibration chessboard images provided in the repository.

```
# This function computes camera Calibration Parameters
# 1. Calibration Matrix
# 2. Distortion Coefficients
def GetCalibrationParam(image_url):

    images = glob.glob(image_url)    #store images

    objp = np.zeros((6*9,3), np.float32)
    objp[:,:2] = np.mgrid[0:9, 0:6].T.reshape(-1,2)

    object_points = [] # 3d Points in real world space
    image_points = [] # 2d Points in image plane.
```

```
    corner = (9, 6) # Chessboard size to 9x6

    # Iterate over the stored images
    for image in images:
        img = mpimg.imread(image)
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        ret, corners = cv2.findChessboardCorners(gray, corner, None)

        if ret:
            object_points.append(objp)
            image_points.append(corners)

    img_size = (img.shape[1], img.shape[0])

    # Here, we will use built in cv2 function named as calibrateCamera
    # This function finds the camera intrinsic and extrinsic parameters...
    # from several views of a calibration pattern
      ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(object_points,
image_points, img_size,None,None)

    # Return Calibration matrix and Distortion Matrix
    return mtx, dist
```

As can be seen in the above code snippet, we have two “object_points” and “image_points” arrays.

1. object_points: Stores x, y, z coordinates of the chessboard corners.

Note: Here we have assumed that the chessboard is fixed on the xy plane while at z=0.

This is to make sure that the object points are the same for each calibration image.

2. image_points: This array stores (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

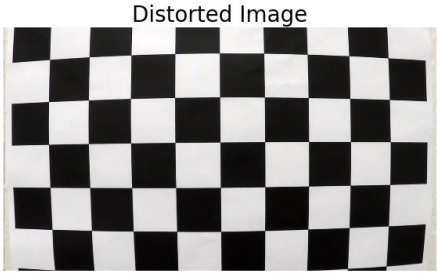
Later, these two above defined matrices are used to compute the camera calibration and distortion coefficients using the cv2 function.

Results:

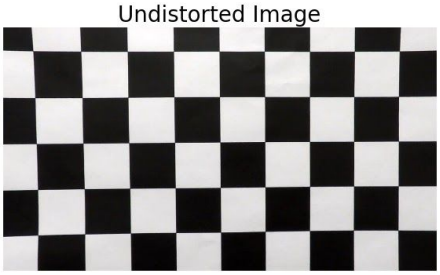
After applying the distortion correction to the test images using the cv2.undistort() from the function GetUndistortion(self, distorted_img, mtx, dist), the results are pasted below:

Note: This function is class member of "LineDetector" class defined in "line_detector.py".

Distorted Image

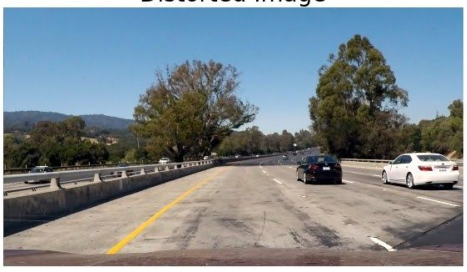


Undistorted Image



Second Example:

Distorted Image



Undistorted Image

2. Pipeline (test images)

A. Provide an example of a distortion-corrected image.

Answer

Please check the above Results (Second Example) mentioned in Camera Calibration topic.

B. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

Answer

To generate thresholded binary images, in the “line_detector.py”module:

```
    # We define a subfunction for calculating index of max sum value of
each histogram.
            def GetMaxIndexHistogram(self, histogram, left_boundary,
right_boundary, window_width=10):
            index_list = []
            side_histogram = histogram[left_boundary : right_boundary]
            for i in range(len(side_histogram) - window_width):
                    index_list.append(np.sum(side_histogram[i : i +
window_width]))
                index = np.argmax(index_list) + int(window_width / 2) +
left_boundary
            return index

        # This function calculates Histogram Thresholding for decreasing
noise from given binary images
            def HistogramThresholding(self, img, xsteps=20, ysteps=40,
window_width=10):

            xstride = img.shape[0] // xsteps
            ystride = img.shape[1] // ysteps
            for xstep in range(xsteps):
                histogram = np.sum(img[xstride*xstep : xstride*(xstep+1),
:], axis=0)
                boundary = int(img.shape[1] / 2)
                    leftindex = self.GetMaxIndexHistogram(histogram, 0,
boundary, window_width=window_width)
```

```
                        rightindex = self.GetMaxIndexHistogram(histogram,
boundary, img.shape[1], window_width=window_width)

                # mask the image
                if histogram[leftindex] >= 3:
                                img[xstride*xstep : xstride*(xstep+1), :
leftindex-ysteps] = 0
                                    img[xstride*xstep : xstride*(xstep+1),
leftindex+ysteps+1 : boundary] = 0
                else:
                     img[xstride*xstep : xstride*(xstep+1), : boundary] =
0

                if histogram[rightindex] >= 3:
                          img[xstride*xstep : xstride*(xstep+1), boundary
:rightindex-ysteps] = 0
                                    img[xstride*xstep : xstride*(xstep+1),
rightindex+ysteps+1 :] = 0
                else:
                    img[xstride*xstep : xstride*(xstep+1), boundary : ] =
0

                                 left_fit_line,  left_line_equation  =
self.CalculatePolynomial(img, 0, boundary)
                                right_fit_line,  right_line_equation  =
self.CalculatePolynomial(img, boundary, img.shape[1])

            # Return binary image after histogram thresholding
                            return img, left_fit_line, right_fit_line,
left_line_equation, right_line_equation
```

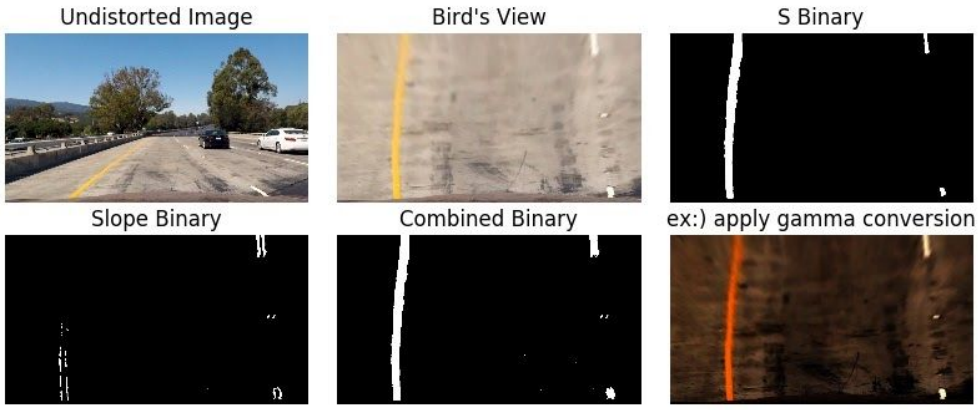
```
        # This function applies Gaussian Noise Kernal to the given binary
image
        def GaussianBlurr(self, img, kernel_size):
            # We use cv2 function GaussianBlur
```

```
            # The Gaussian filter is a low-pass filter that removes the
high-frequency components are reduced.
            return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

        # Computes S Binary Image (x, y, 1) from H and S of HLS.
        def GetSBinary(self, undist_img, thres=(110, 255)):

            # Use cv2.cvtColor and RGV2HLS Parameter
            hls = cv2.cvtColor(undist_img, cv2.COLOR_RGB2HLS)
            h = hls[:, :, 0]
            s = hls[:, :, 2]
            s_binary = np.zeros_like(s)
            s_binary[((s >= thres[0]) & (s <= thres[1])) & (h <= 30)] = 1
            s_binary = self.GaussianBlurr(s_binary, kernel_size=21)

            return s_binary

        # This function returns RGB image after applying gamma conversion
        def AdjustGamma(self, image, gamma=1.0):

            inv_gamma = 1.0 / gamma
            table = np.array([((i / 255.0) ** inv_gamma) * 255
                for i in np.arange(0, 256)]).astype("uint8")

            # We wll use a look up table LUT from cv2
            return cv2.LUT(image, table)

        # This function is basically for Gradent Thresholding
         # We First apply gamma conversion to imags for decrease noise
using 'AdjustGamma' function.
         def GetSlope(self, undist_img, orient='x', sobel_kernel=3, thres
= (0, 255)):

            undist_img = self.AdjustGamma(undist_img, 0.2)
            gray = cv2.cvtColor(undist_img, cv2.COLOR_RGB2GRAY)
            if orient == 'x':
```

```
                    slope = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0,
ksize=sobel_kernel))
            elif orient == 'y':
                    slope = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1,
ksize=sobel_kernel))
            else:
                raise KeyError("select 'x' or 'y'")

            scale_factor = np.max(slope) / 255
            scale_slope = (slope / scale_factor).astype(np.uint8)
            slope_binary = np.zeros_like(scale_slope)
               slope_binary[(scale_slope >= thres[0]) & (scale_slope <=
thres[1])] = 1

                        slope_binary = self.GaussianBlurr(slope_binary,
kernel_size=9)
            return slope_binary

# Given an undistorted image, return converted image using Color Slope
Conversion
def ColorSlopeThresConversion(self, undist_img):
       s_binary =self.GetSBinary(undist_img, thres=(150, 255))
            slope = self.GetSlope(undist_img, orient='x',sobel_kernel=7,
thres=(25, 255))

       color_binary = np.zeros_like(s_binary)
       color_binary[(s_binary == 1) | (slope == 1)] = 1

       return color_binary
```

A combination of color and gradient was implemented and to remove noise, we applied gaussian blur.

Results:



C. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

Answer

Perspective Transform

Refer to the module named "perspective_transfor.py". The class "PerspectiveTransform" takes inputs, source points and destination points, and returns a perspective transformation of the image. It uses cv2 function built in one, named: warpPerspective.

```
class PerspectiveTransform():
    def __init__(self, src, dst):
        self.src = src
        self.dst = dst
        self.M = cv2.getPerspectiveTransform(self.src, self.dst)
              self.inverse_M  =  cv2.getPerspectiveTransform(self.dst,
self.src)

    ### Here, we used the cv2 function warpPerspective
    #Applies a perspective transformation to an image.

    # This function returns a transformed image
    def Transform(self, undist):
```

```
        return cv2.warpPerspective(undist, self.M, (undist.shape[1],
undist.shape[0]))

    # This function performs inverse of 'Transform' function, it
returns original image.
   def InverseTransform(self, undist):
               return  cv2.warpPerspective(undist,  self.inverse_M,
(undist.shape[1], undist.shape[0]))
```

Results:

D. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

Answer

First, we identified peaks in the histogram of the respective image to determine location of lane lines. After that, we have to mask images by the calculated peaks. Next, we can fit a polynomial using CalculatePolynomial() function.

In the line_detector.py module,

```
# Given an image, left_boundary, right_boundary, this function calculates and fits the polynomial on it
            def CalculatePolynomial(self, img, left_boundary, right_boundary):
        side_img = img[:, left_boundary: right_boundary].copy()
        index = np.where(side_img == 1)
        yvals = index[0]
        xvals = index[1] + left_boundary
        if xvals.size != 0:
            fit_equation = np.polyfit(yvals, xvals, 2)
                        fit_line = fit_equation[0]*self.yvals**2 + fit_equation[1]*self.yvals + fit_equation[2]
            return fit_line, fit_equation
        else:
            return 0, np.array([10000., 100., 100.])
```

The trick that we can utilize here is that if we find lane lines in one frame of video, from onward, we can simply search within a window around the previous detection from the next frame of video.

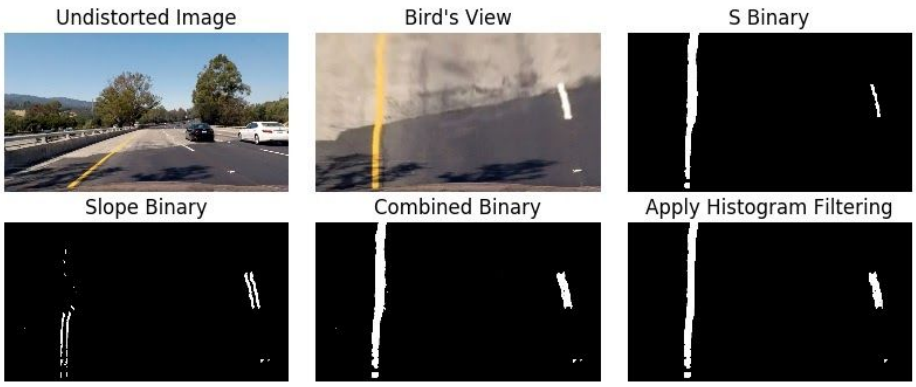
Here, we have to check for two things:

1. If the left and right lines are roughly parallel.

```
def CheckParallel(self):

      return True if (np.abs(self.left_line.current_fit[0] -
self.right_line.current_fit[0]) < 0.01) else False
```

2. The difference between previous and current line(s) curvature is negligible or little.

```
   def CheckSimilarity(self, side='left'):

       if side == 'left':

           return True if (np.abs(self.left_line.current_fit[0] -
self.left_line.best_fit[0]) < 0.0005) else False

       else:

           return True if (np.abs(self.right_line.current_fit[0] -
self.right_line.best_fit[0]) < 0.0005) else False



   def CheckLine(self):

       return self.CheckSimilarity(side='right') and
self.CheckSimilarity(side='left') and self.CheckParallel()
```



E. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Answer

The Function CalculateCurvature(self) in "line_detector.py" module computes the radius of curvature for each lane in meters.

```
# This function computes radius of curvature for each lane in meters
def CalculateCurvature(self):

       # Conversion from pixels to meters
        # By Simply multiplying the number of pixels by 3.7/700 along
x dim and 30/720 along y dim.
       ym_per_pix = 30/720
       xm_per_pix = 3.7/700

               left_fit_cr  =  np.polyfit(self.yvals  *  ym_per_pix,
self.left_line.bestx * xm_per_pix, 2)
              right_fit_cr  =  np.polyfit(self.yvals  *  ym_per_pix,
self.right_line.bestx * xm_per_pix, 2)

                    self.left_line.radius_of_curvature   =   ((1   +
(2*left_fit_cr[0]*np.max(self.yvals) + left_fit_cr[1])**2)**1.5) \
                                       /np.absolute(2*left_fit_cr[0])
                   self.right_line.radius_of_curvature   =   ((1   +
(2*right_fit_cr[0]*np.max(self.yvals) + right_fit_cr[1])**2)**1.5) \

/np.absolute(2*right_fit_cr[0])
```

Here, we utilize basic mathematics and geometrical algebra to calculate x intercepts from the respective polynomials. The conversion factor for pixels to meters was also applied.

To calculate the position of the vehicle, the function AddPlaceToImage(self, image) was used.

```
def AddPlaceToImage(self, image):
                        place    =    (self.left_line.bestx[-1]    +
self.right_line.bestx[-1]) / 2
         diff_center = np.abs((image.shape[1] / 2 - place) * 3.7 /
700)

       if place > image.shape[1] / 2:
                    cv2.putText(image, 'Vehicle is {:.2f}m left of
center'.format(diff_center), (100,80),
                  fontFace = 16, fontScale = 2, color=(255,255,255),
thickness = 2)
       else:
                    cv2.putText(image, 'Vehicle is {:.2f}m right of
center'.format(diff_center), (100,80),
                  fontFace = 16, fontScale = 2, color=(255,255,255),
thickness = 2)
       return image
```

G. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Answer

3. Pipeline (video)

Answer

The pipeline consists of, namely

a. class LineDetector(object):
b. class PerspectiveTransform():
c. class Line(object):
d. main()

Please check the video named “output_video.mp4” in the project folder.

4. Discussion

Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Answer

As I was naive at first, I had to experiment a lot with gradient and color channel thresholding. My designed pipeline failed for the hard challenge videos. Mostly because, the video had sharper turns for very short intervals.

Improvements: (to make robust)

We can improve the above pipeline by taking following measures:

1. Better perspective transform
    a. Due to sharper turns, we will have to choose a smaller section to take the transform.
    b. Next, we average over a smaller number of frames.