Advanced Lane Finding

By David Rose

6/12/17

Lane Identification and tracking for autonomous vehicle video feeds

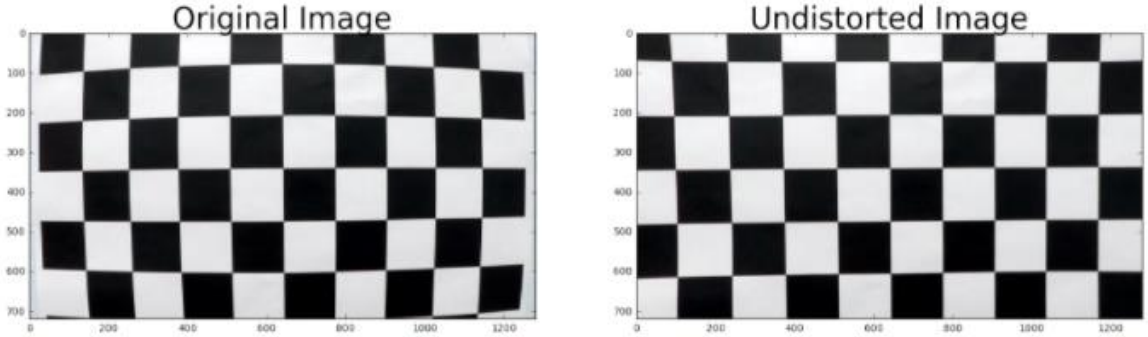
Following areas of this project include

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

STEPS

Camera Calibration

The lens of every came increases a slight portion of distortion within the images it captures, especially around the corners. Using a printed chessboard image and then tracking the corners with a function from OpenCV, the image can be transformed slightly to straighten out the lines, and then apply that transformation to subsequent images of the camera. Below is an example of image un-distortion:



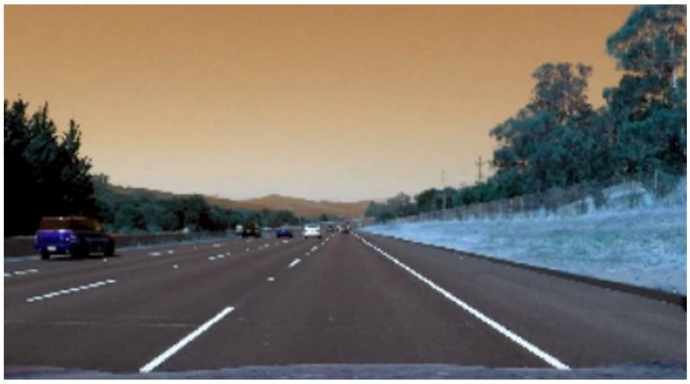
Pipeline

1. Un-Distort and Warp the Perspective

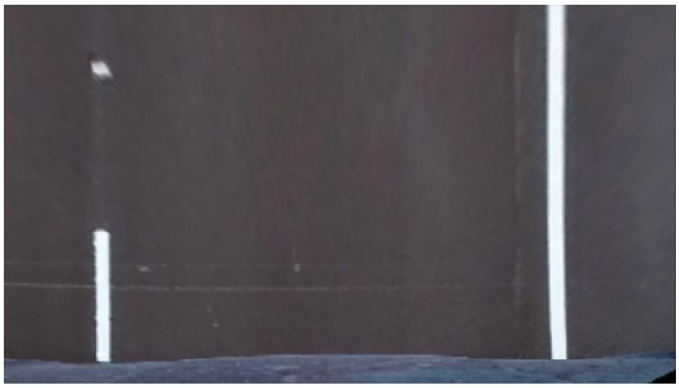
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one: (colors inverted from original)

```
def toDistort(img,mtx,dist,plot='no'):
    global undist
    undist = cv2.undistort(img, mtx, dist, None, mtx)

    if plot == 'yes':
        plt.suptitle('Before Distortion', fontsize=14, fontweight='bold')
        plt.imshow(img)
        plt.show()
        plt.suptitle('After Distortion', fontsize=14, fontweight='bold')
        plt.imshow(undist)
        plt.show()
    return undist


def toWarp(img,src,dst,undist,plot='no'):
    # Grab the image shape
    global img_size
    img_size = (toGray(img).shape[1], toGray(img).shape[0])
```

Before Warp



After Warp



2. Color transformation and thresholding

I performed a transformation of color channels to HLS and then pulled out the S layer as it seemed to perform the best in making the lane line stand out. Then using gradient thresholds to generate a binary image (values are either 0 or 255). The threshold values I used were 10 and 100. Below is an example of my output for this step.

To perform the warping transform requires a bit of manual work. You have to identify the four corners of a polygon surrounding the lane and then place those closer to the edges of the frame to stretch the image out to a birds-eye view. The function used is getPerspectiveTransform from Open-CV, the particular values I ended up using (after much trial and error) are:

Source	Destination
500, 500	120, 0
800, 500	1170, 0
1280, 720	1220, 720
0, 720	0, 720

Sobel transformation code block

```
# Define a function to return the magnitude of the gradient
# for a given sobel kernel size and threshold values
def toMagSobel(img, sobel_kernel=3, mag_thresh=(0, 255), plot='no'):
    # Convert to grayscale
    if len(np.shape(img)) > 2:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    else:
        gray = img
    # Take both Sobel x and y gradients
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
    # Calculate the gradient magnitude
    gradmag = np.sqrt(sobelx**2 + sobely**2)
    # Rescale to 8 bit
    scale_factor = np.max(gradmag)/255
    gradmag = (gradmag/scale_factor).astype(np.uint8)
    # Create a binary image of ones where threshold is met, zeros otherwise
    binary_output = np.zeros_like(gradmag)
    binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1

    if plot == 'yes':
        plt.suptitle('After Binary Mag Sobel', fontsize=14, fontweight='bold')
        plt.imshow(binary_output)
        plt.show()
    # Return the binary image
    return binary_output
```

Pulling out the S layer

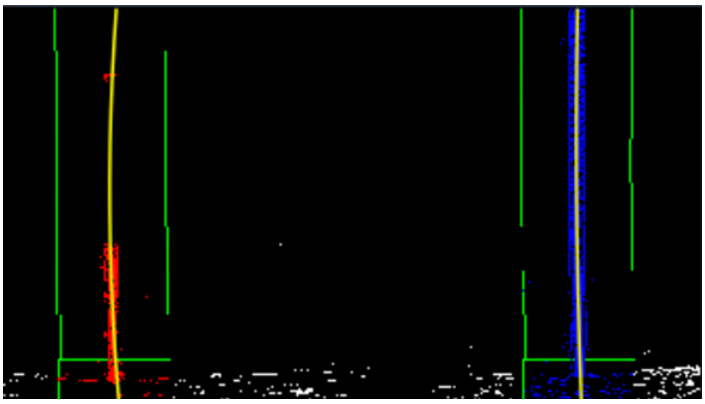
Transformed to a binary image with thresholding

4. Sliding Window Detection

Breaking up the image height-wise into a series of sliding windows I can continually find the pixel locations and begin to mark the lane locations. In the below image you can see the result of this effort. Though there were some issues drawing the window rectangles, it performed well enough in practice. It uses a total of 9 windows on each lane to then track midpoint concentration of pixels, to best fit the lane line.

The code (provided within a lesson in this project section) is a bit too long and extensive to post here in the write-up, but it can be found in the function sliding_window on lines 231 to 422.

Sliding Windows on lane lines

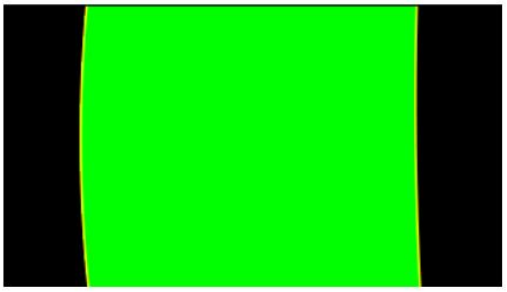


In the image above you can see the left/right borders of the sliding windows (though I am not sure why the horizontal lines keep disappearing) and the fitted yellow lines.

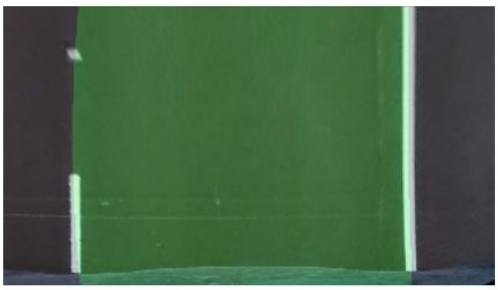
5. Create polygon to fill the lane based off sliding_window function

Once I have these lines digital lines fitted over the image of lane lines, I can use another OpenCV function to fill in a polygon and 'draw' a virtual lane into the birds-eye image. This function, fillPoly creates a basic polygon on a blank image, then I can use the addWeighted function to overlay it (with transparency) on to the original lane image, by first placing it on birds-eye then warping back to original.

Lane on blank image



Lane overlayed on birds-eye



6. Final image of lane plotted back on to the video feed, with detected lane line curves overlayed onto the image, along with the curves printed in the top-left corner.

Steps 514 to 518 show where I used overlay both the virtual lane lines, and then the text for curve values, to the image for the video pipeline.

Video Result

Here's a link to my video result

Discussion

1. Smoothing feature in the video

While implementing this in a single image is simple enough, a video can have dozens of frames a second and increases the chances for an error in the processing. One method I worked out to counter this is the idea of keeping track of the lane fit values returned in the sliding window function, and then average them.

Variables hold the positional history for both lanes and if the current detected lane diverges too much from the previous, the function will then return a recent good working lane line detection.

This was key in helping to create a smooth detected lane in video. The possible downside is turns such as 90 degree turn or hairpin turns may be too sharp for the function to handle, considering it an error. But for this purpose we are just driving on a highway which has maximum curvature requirements that will not cause any issues here.

```
global left_fitx_hist, right_fitx_hist
#MY CODE: Build left fit history in order to identify outliers
if left_fitx_hist==None:
    left_fitx_hist=[]
    left_fitx_hist.append(left_fitx)
else:
    if abs((np.mean(np.mean(np.array(left_fitx_hist), axis=0 ))) - np.mean(left_fitx))<50:
        if len(left_fitx_hist)>10:
            left_fitx_hist=left_fitx_hist[1:]
        left_fitx_hist.append(left_fitx)

#Build right fit history in order to identify outliers
if right_fitx_hist==None:
    right_fitx_hist=[]
    right_fitx_hist.append(right_fitx)
else:
    if abs((np.mean(np.mean(np.array(right_fitx_hist), axis=0 ))) - np.mean(right_fitx))<50:
        if len(right_fitx_hist)>10:
            right_fitx_hist=right_fitx_hist[1:]
        right_fitx_hist.append(right_fitx)

left_fitx_mean = np.mean(np.array(left_fitx_hist), axis=0 )[:len(ploty)]
right_fitx_mean = np.mean(np.array(right_fitx_hist), axis=0 )[:len(ploty)]
```

2. Future considerations

Drastic lane changes

As the lane detection takes into consideration only a specific portion of the frame, drastic changes in lane placement or elevation changes may hurt the correct detection. It would be more wise to take a larger swatch of the image frame as possible lane detection areas, but this would also bring with it many more false errors as it would begin to see other lanes, and maybe similar looking edges such as the medians or guardrails that will throw off the detection.

Defaulting to known lanes when losing detection

What if we go through a construction zone and lane markers are suddenly gone, or change into a different color/style than what the model is used to? It could fail to detect and jump around erratically. We could possibly train a confidence level in the lane detection, and use fallbacks when it goes below a certain threshold.