Advanced Lane Finding Project

The goals / steps of this project are the following:

* 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.

Camera Calibration

The code for this step is contained in the first code cell of the IPython notebook located in adv_find_lane.ipynb.

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. Fig. 1 shows the 17 calibration images used to calculate camera matrix and other variables.

I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained results as shown in Fig. 2 and Fig. 3

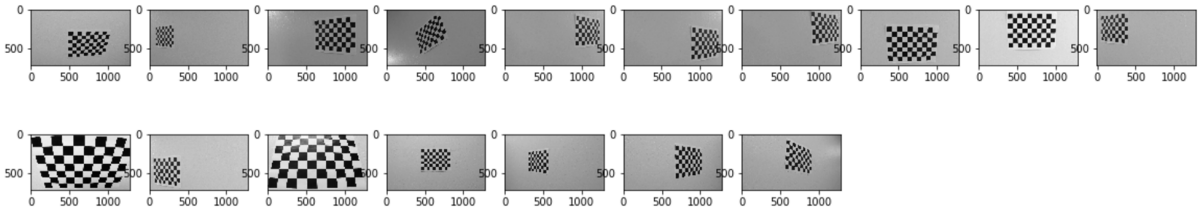

Fig. 1 Good calibration images used to calibrate camera

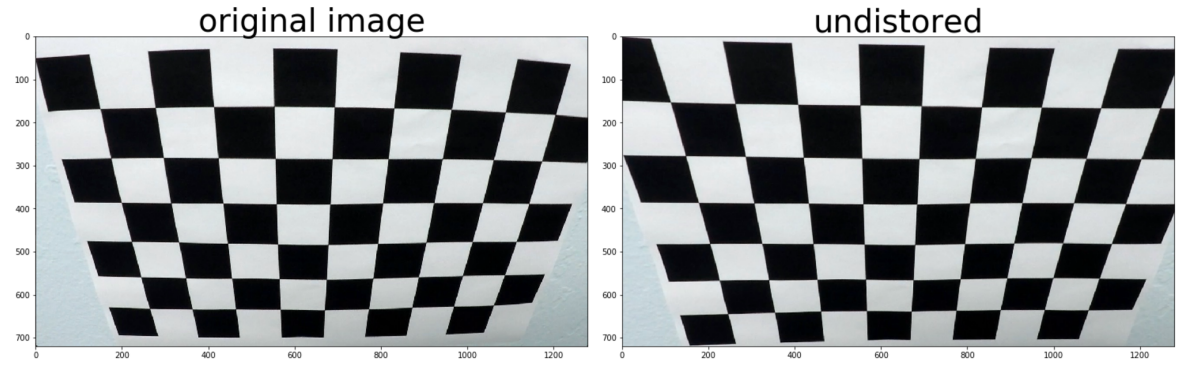

Fig. 2 Comparison between a checkerboard image and an undistorted image

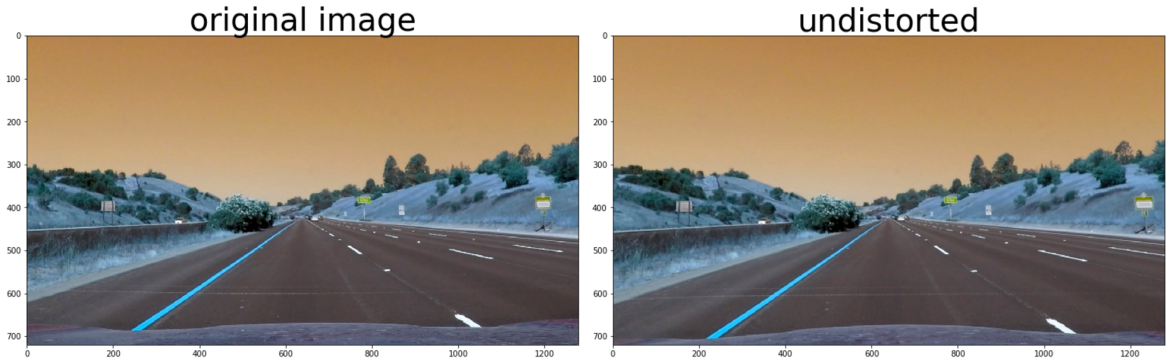

Fig. 3 Comparison between a test image and the corresponding undistorted image

Pipeline (single images)

Simply applying camera matrix and distortion coefficients to cv2.undistort() function, we can get un-distorted images.

With an undistorted image, we can apply pipeline of color and gradient thresholds.

CV2 load image with BGR color space
Transform color space from BGR to HLS and get s channel.
s_channel = cv2.cvtColor(img, cv2.COLOR_BGR2HLS).astype(np.float)
Transform color space from BGR to LUV and get l channel.
l_channel = cv2.cvtColor(img, cv2.COLOR_BGR2LUV)[:,:,0]
Transform color space from BGR to Lab and get b channel.
b_channel = cv2.cvtColor(img, cv2.COLOR_BGR2Lab)[:,:,2]

Calculate Sobel operations
Calculate the derivative in the x direction (the 1, 0 at the end denotes x direction):
sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
Calculate the derivative in the y direction (the 0, 1 at the end denotes y direction):
sobely = cv2.Sobel(l_channel, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
Calculate the absolute value of the x derivative:

```
abs_sobelx = np.absolute(sobelx)
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
```

The magnitude, or absolute value, of the gradient is just the square root of the squares of the individual x and y gradients.
The direction of the gradient is simply the inverse tangent (arctangent) of the y gradient divided by the x gradient.

Code cell 5 implements the pipeline() by combination of all the above with default minimum and maximum thresholds in table 1. Fig. 4 shows comparison between an original image and its corresponding image processed by the pipeline.

Type	Min	Max
S Channel	180	255
L Channel	215	255
B Channel	145	200
Sobel Scale	80	255
Sobel Magnitude	50	255
Sobel Directional	0	$\pi/2$

Table 1 Default Thresholds Used in the Pipeline

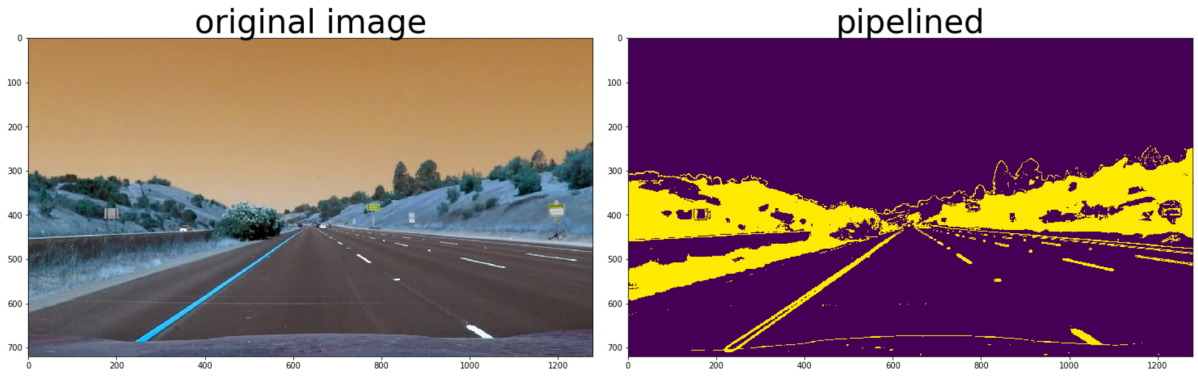

Fig. 4 Comparison between an image with corresponding pipelined

Perspective Transform

The code for my perspective transform includes a function called `apply_persepctive_transform()`, which appears in the 7th code cell of the IPython notebook. The `apply_persepctive_transform()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32([[490, 482], [810,  482], [1250, 720], [40, 720]])
dst = np.float32([[0, 0], [1280, 0], [1250, 720], [40, 720]])
```

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

The left image of Fig. 5 shows the warped image from the original image shown in Fig. 4. The right image of Fig. 5 shows the warped image after being pipeline processed.

Fig.6, Fig. 7, and Fig. 8 show compassion between original images and warped pipeline-processed images.

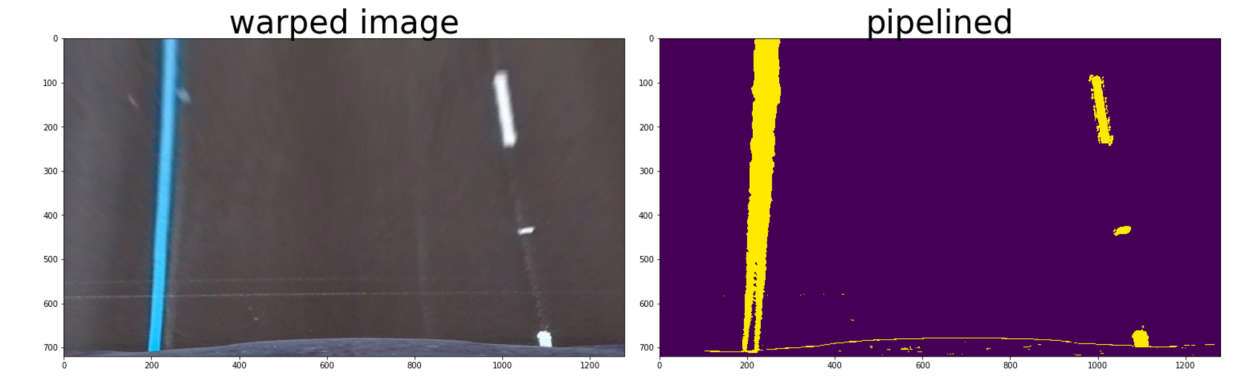

Fig. 5 Warped Image and Warped Image Processed by Pipeline

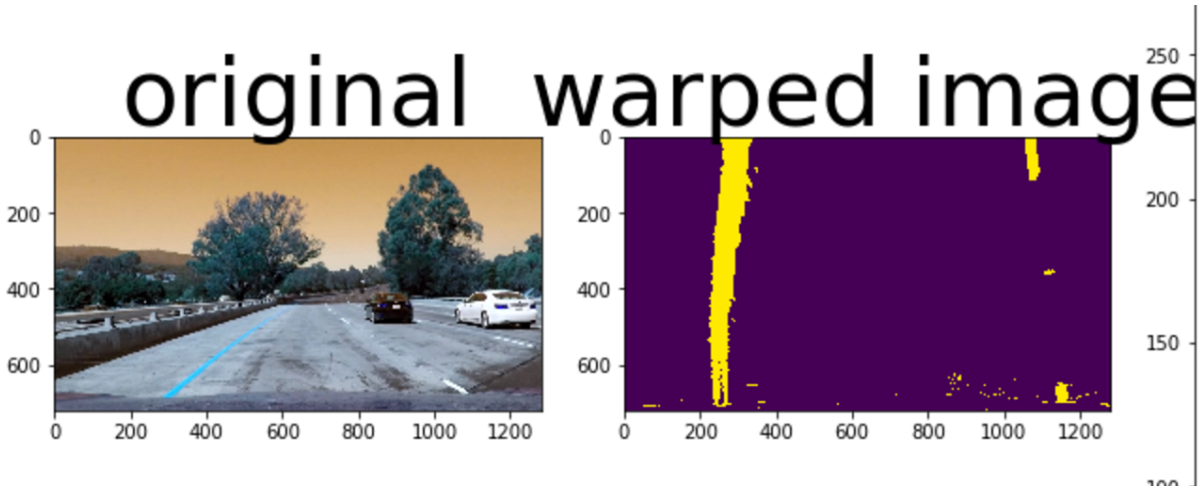

Fig. 6 Comparison Between Original Image and Warped Pipelined Image

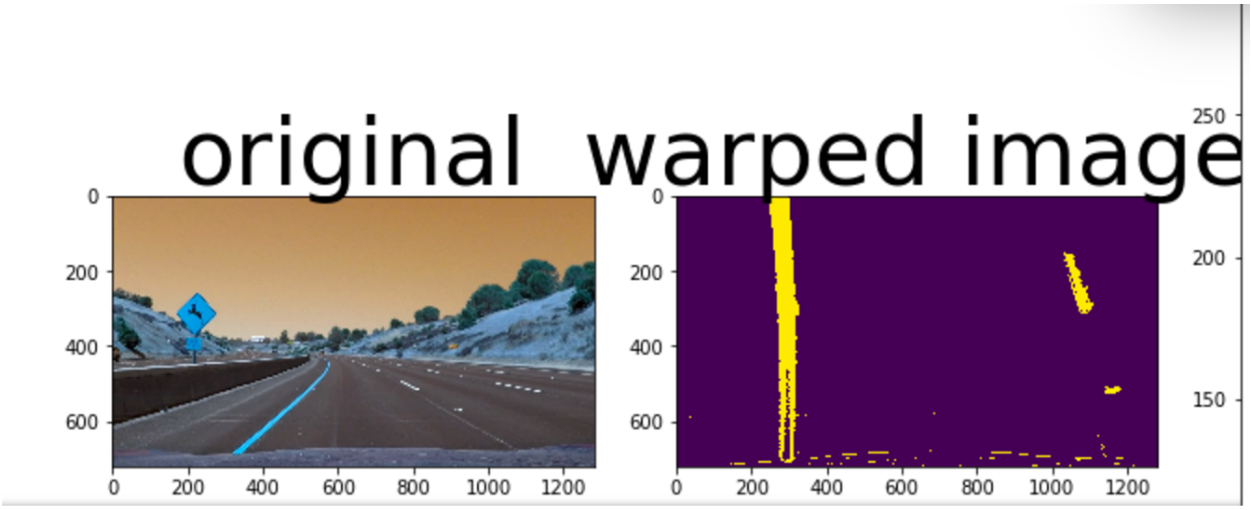

Fig. 7 Comparison Between Original Image and Warped Pipelined Image

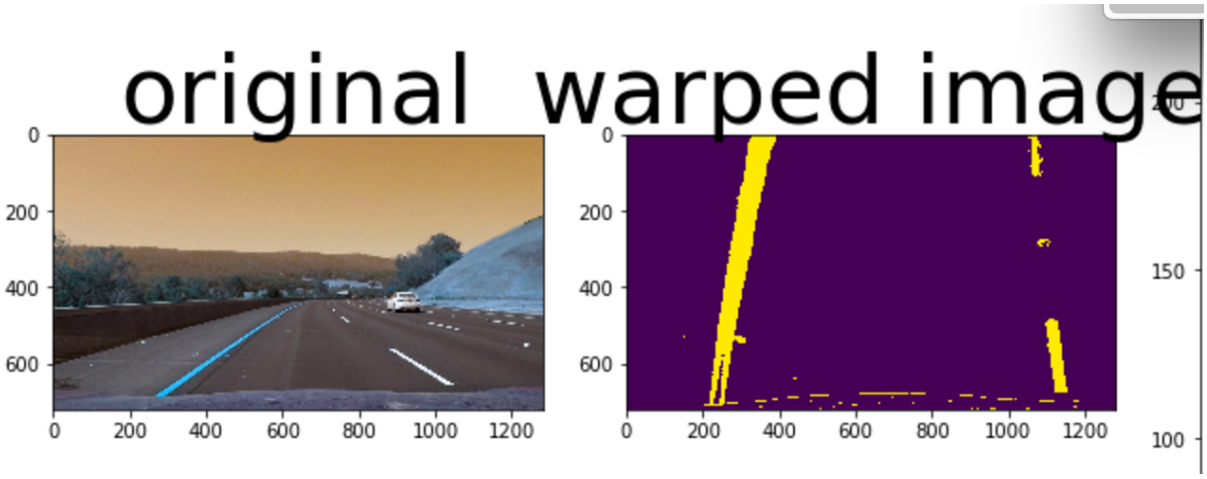

Fig. 8 Comparison Between Original Image and Warped Pipelined Image

Fill Lane

FillLane() is implemented in 11th code cell of the python notebook. Histogram is used with 2nd order polynomial to fit curves for both left lane and right lane. We identify left and right lane and highlight with green and red color respectively in the warped image. We also fill lane with green color with an overlay.

Fig. 9, Fig. 10, and Fig. 11 show results with warped images on the left and filled green lane images on the right.

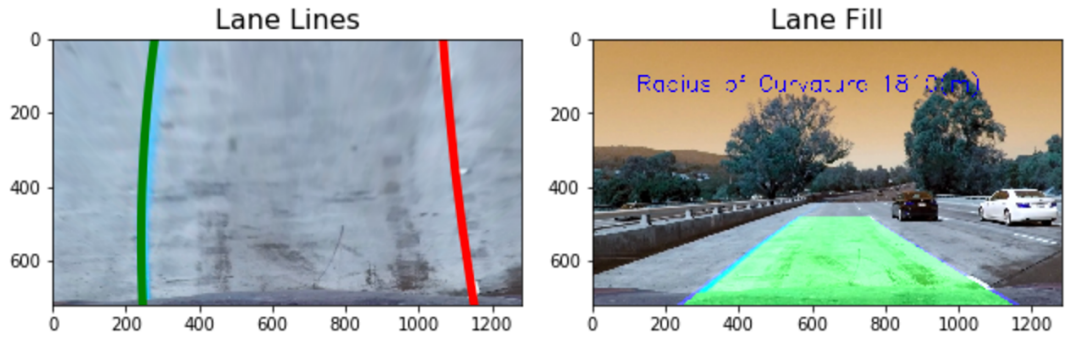

Fig. 9 Results from Lane Identification and Lane Fill

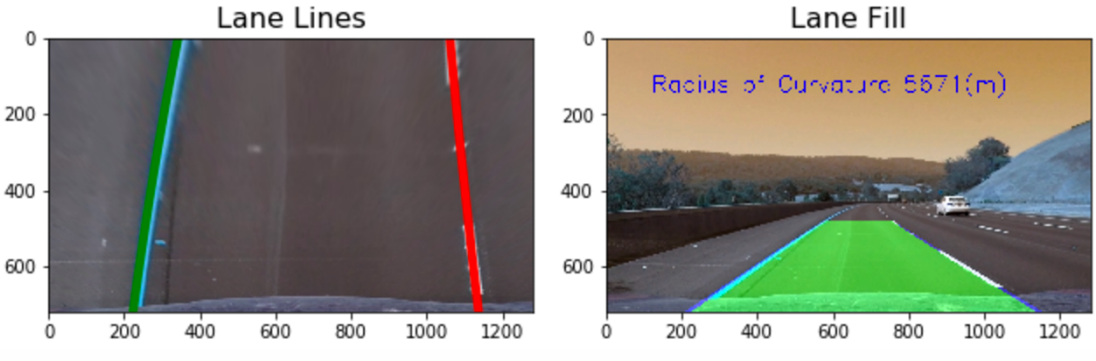

Fig. 10 Results from Lane Identification and Lane Fill

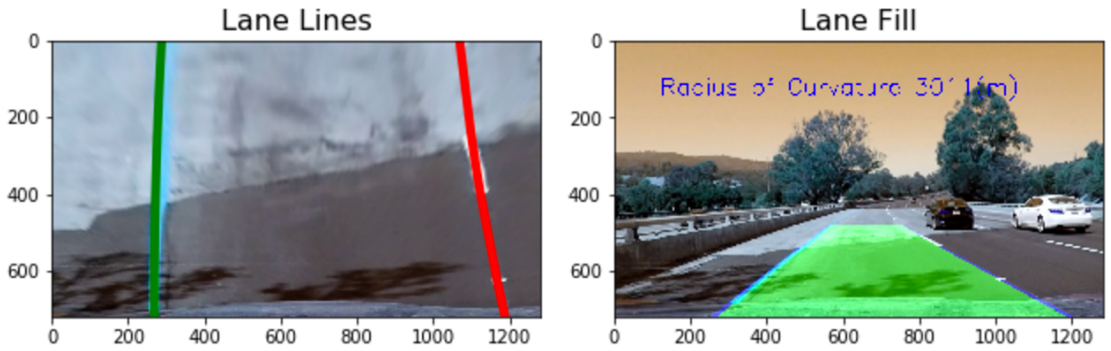

Fig. 11 Results from Lane Identification and Lane Fill

Curvature of the lane and the position of the vehicle

The formula for the radius of curvature at any point x for the curve $y = f(x)$ is given by:

$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

I did this in measure_curvature() in the 10th code cell of the notebook.
The vehicle position is first calculated by the formula, abs((640 - center-position)*3.7/700), i unit of meter. The middle of the road can be averaged by the two lanes in x-axis.

Pipeline (video)

The video results are saved under the output_images directory which uploaded to GitHub.
result_project_video.mp4 for project_video.mp4
result_harder_challenge_video.mp4 for harder_challenge_video.mp4

Discussion

There are couple of situations that are difficult to identify correctly. Bright sunshine, rain, and snow can be challenge to identify lanes properly. Motor-cycles, bicycles, animals, pedestrian, faded pavement, and new pavement can create problems for computer vision like this.

However, without question this is a good foundation for further improvement.