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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

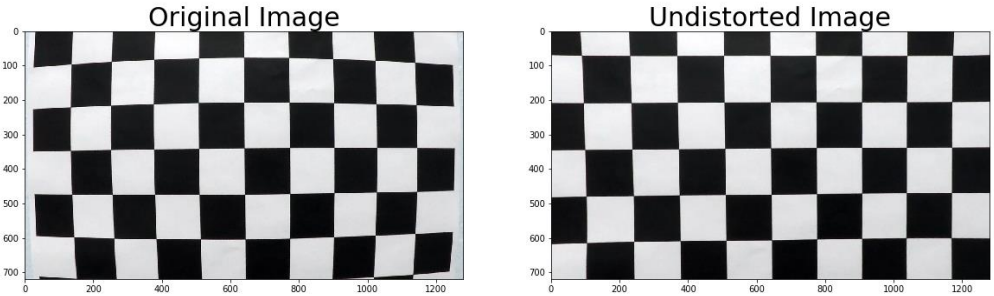
You're reading it!

Camera Calibration

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

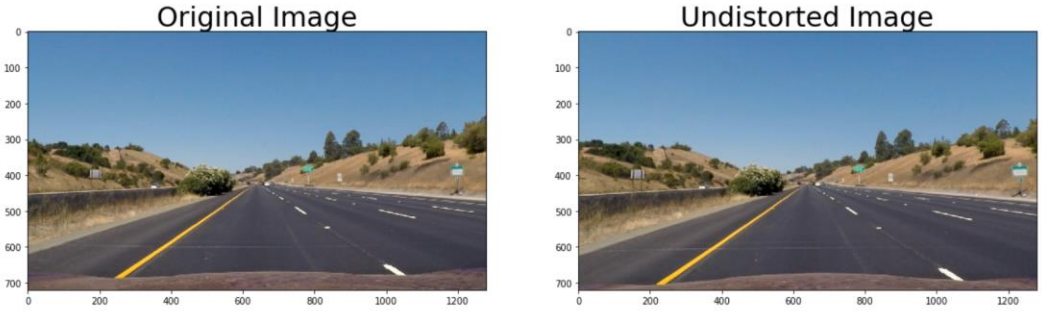
The code for this step is contained in calibrate_camera() function located in camera_calibration.py"

(I copied the description for this block from the template, because this is exactly what I have done)
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. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



1. Has the distortion correction been correctly applied to each image?

Yes. In my pipe line I start by correcting the distortion of the image as a first step



2. Has a binary image been created using color transforms, gradients or other methods?

I am using 2 thresholds for the binary image:

- b_thresh - blue/yellow threshold of a b channel from LAB color space. use it to find the yellow lines

- l_thresh - Light channel from HSL color space. I use it to find the white lines.

3. Has a perspective transform been applied to rectify the image?

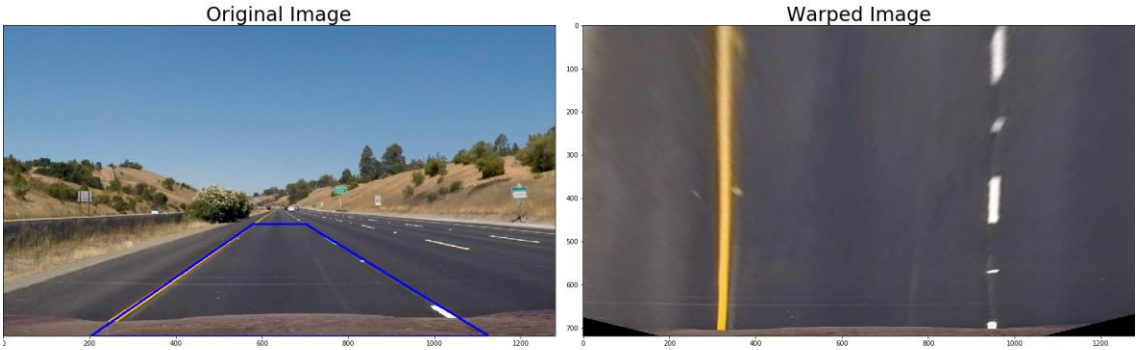
The code for my perspective transform is includes a function called set_percpective_transform() And warp_image under “lane_find_funcs.py”
(`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the image. IThese are the points that I have chosen:

```
top_left = [(img_size[0] // 2) - 60, img_size[1] // 2 + 100]
bottom_left = [(img_size[0] // 6) - 10, img_size[1]]
bottom_right = [(img_size[0] * 5 // 6) + 60, img_size[1]]
top_right = [(img_size[0] // 2 + 60), img_size[1] // 2 + 100]
```

Which in pixel coordinates are:

Top left	(580,460)
Bottom left	(203,720)
Bottom right	(1126,720)
Top right	(700,460)

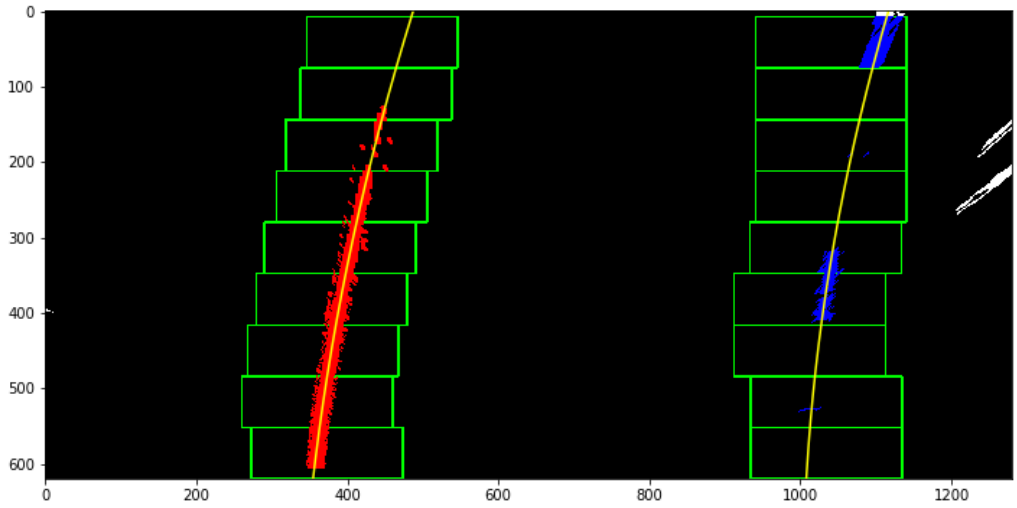
Beautiful:



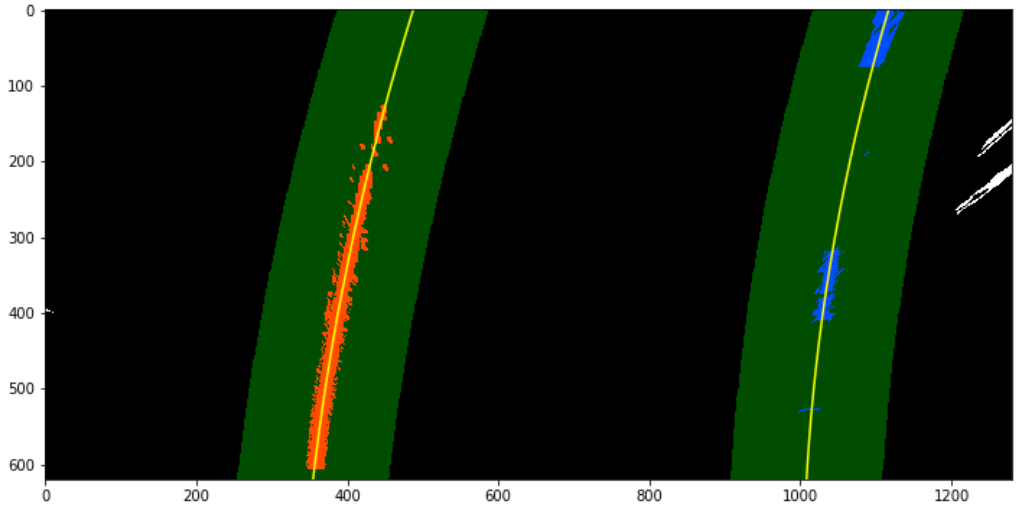
4. Have lane line pixels been identified in the rectified image and fit with a polynomial?

They sure have. They were calculated from a warped binary mask. First I needed to determine which pixels belong to the lines. I did it in tow methods:

Sliding window method: described in curv_calc_sliding_window() located at "lane_find_funcs.py". the first step is to do a histogram analysis to estimate the center of the lane, and then use a margin to filter outliers. We update the location of the box as we move along (the colored pixels will be used for calculation):



Previous polygon method: described at curv_calc_from_previous() located at "lane_find_funcs.py" described in relevant for video pipeline. We can use the lane line found on previous frame in order to determine a mask for the current frame. This method is supposed to be faster.



After the lanes pixels were found by any of the methods, we use a polyfit in order to find the equation that can describe this lane.

When there are not enough pixels to properly calculate the fit in one of the lanes, I take the pixels from the other one, shift them by the lane width and add them to the calculation. Doing that, I use two assumptions: the lines are parallel, and they should be separated by a constant width most of the time.

5. Having identified the lane lines, has the radius of curvature of the road been estimated?
And the position of the vehicle with respect to center in the lane?

Yes! in order to convert the results, I used the following coefficients:

```
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
```

I than converted the coordinates of the lane pixels and recalculated the fit. (Did it in the same functions that calculated the original fit described in section 4)

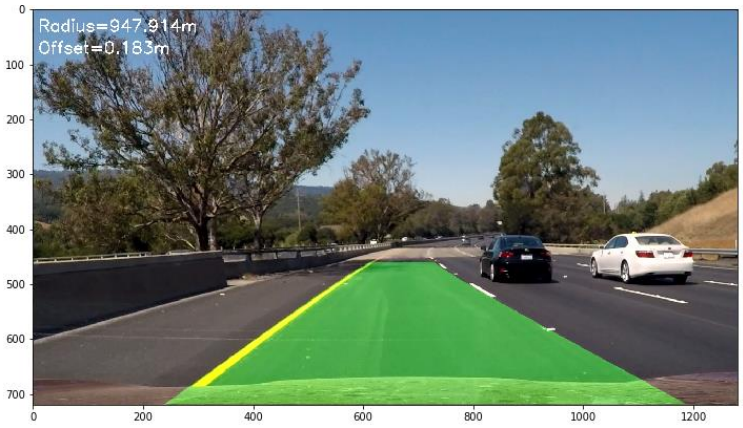
Once I had the real world dimension poly fit, I calculated the curvature at measure_curvature_and_offset() located at “lane_find_funcs.py” I used the following equation for the curvature (had to convert the y_eval pixels into meters):

```
y_eval = img.shape[0]
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
# Calculation of R_curve (radius of curvature)
left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
```

And the offset by finding the average lanes x coordinate at the bottom of the screen, and compared the distance from the middle of the frame:

```
# The offset would be the distance between the middle of the lanes at the screen bottom, and the middle of the image
lanes_center = np.mean([left_fit_cr[0]*(y_eval*ym_per_pix)**2 + left_fit_cr[1]*y_eval*ym_per_pix + left_fit_cr[2],
                right_fit_cr[0]*(y_eval*ym_per_pix)**2 + right_fit_cr[1]*y_eval*ym_per_pix + right_fit_cr[2]])
offset = lanes_center - (img.shape[1] / 2) * xm_per_pix
```

I am printing the results (including unwarped lane lines) on the output frame:



Good times!

Pipeline (video)

1. Does the pipeline established with the test images work to process the video?

Of course! I am describing the pipeline in video_frame_pipeline() located at “video.py”

The special addition I added to the video pipeline is a Line object (also described in video.py) that keeps track of the calculated lines, and applies a weighted average in order to smooth out the results or discards bad samples.

You can see the output video at “output_images/ProjectVideo.mp4”

You can also see the code for this section and the videos at the attached notebook: advanced_lane_finder.ipnb

Please tell me what you think!

Discussion

The approach I took was to use Sobel and HLS color space thresholds in order to create a binary map of the image, warp it into eagle eye view, calculate the lane lines using a poly fit and then use the fit in order to calculate the curvature radius and lane offset.

It seemed like this algorithm worked pretty well on the simple cases. In my opinion, the part that has the strongest effect is the binary mask creation. It seemed like using the HLS color space worked better than what we saw previously with RGB.

Pipeline’s shortcomings (saw them when I failed on the more challenging cases):

- when a black line was covering the lane, it became invisible to my algorithm, because I was filtering shadows with the L channel. Maybe a different approach should be tested.
- Maybe polyfit of the pixels is not the best approach. There are many outliers that are still inside the margins. Maybe RANSAC or another outlier detector can remove them.
- Should limit valid pixels for right lane to be only on the right pat of the screen, and left side only on the left, even if the calculated line twists in a crazy way.
- Should add a metric to determine if the calculated line makes sense, could be a simple range for valid poly parametes.

Anyways, had good time working on this project! Hope you like my results.