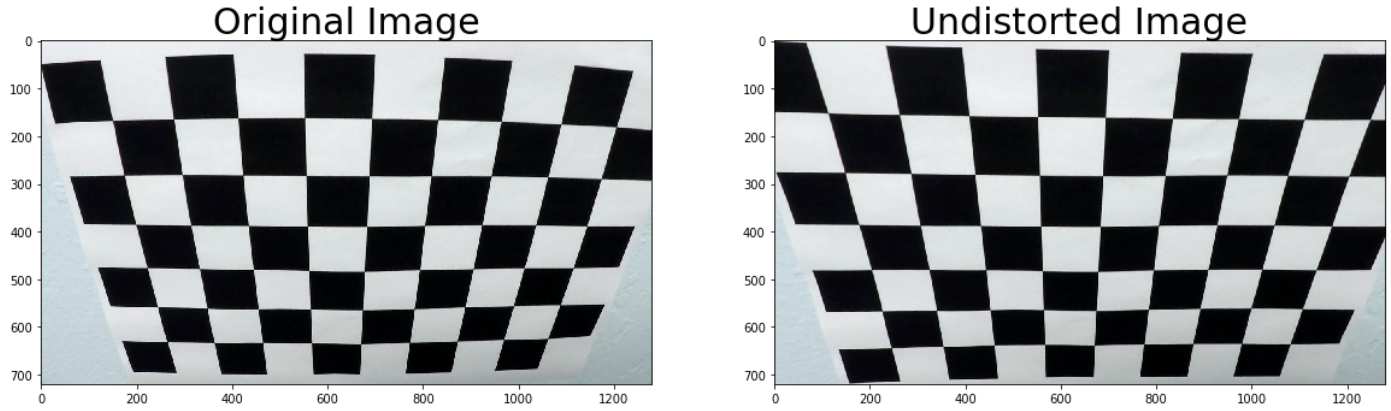
Advanced Lane Finding Project

Camera Calibration

The code for this step is contained in the function calibirateCamera located in "camera.py" in lines 13 through 36.

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:



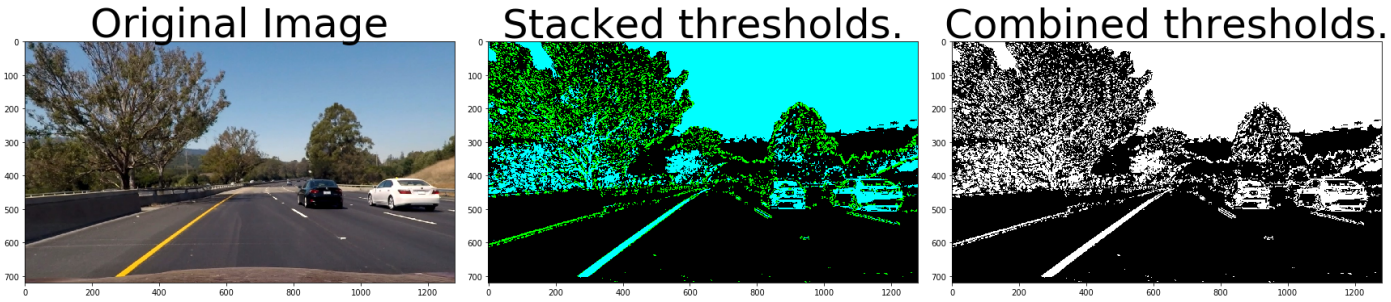
Pipeline (single images)

1. An example of a distortion-corrected image.

Following is the example of one of test images which was distortion corrected. It was implemented as per code on 38-39 in camera.py file:

2.Color transformation and gradients transformation to create a thresholded binary image.

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines 14 through 104 in `threshold.py`). Here's an example of my output for this step.



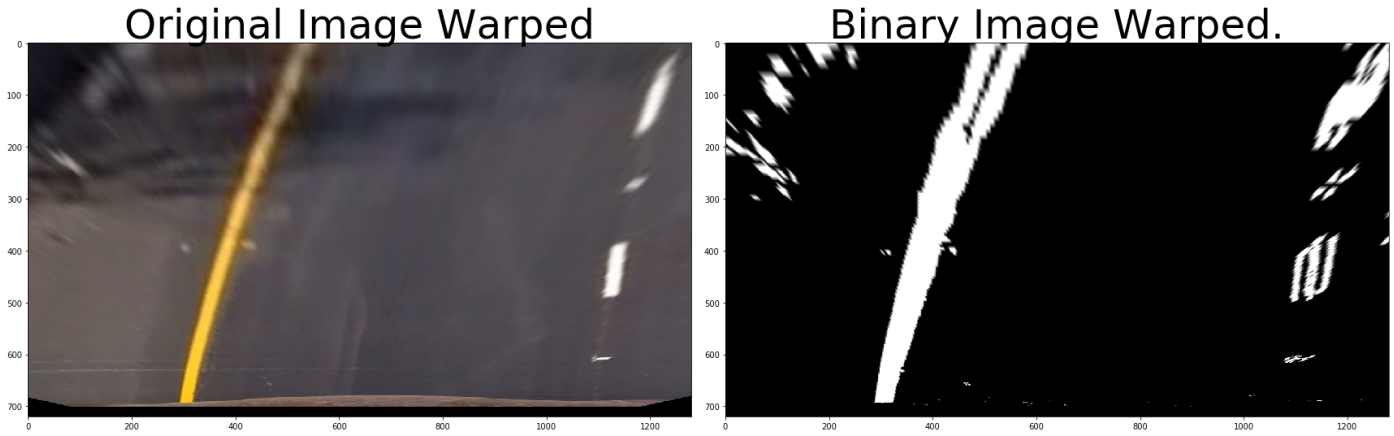
3. Perspective transformation.

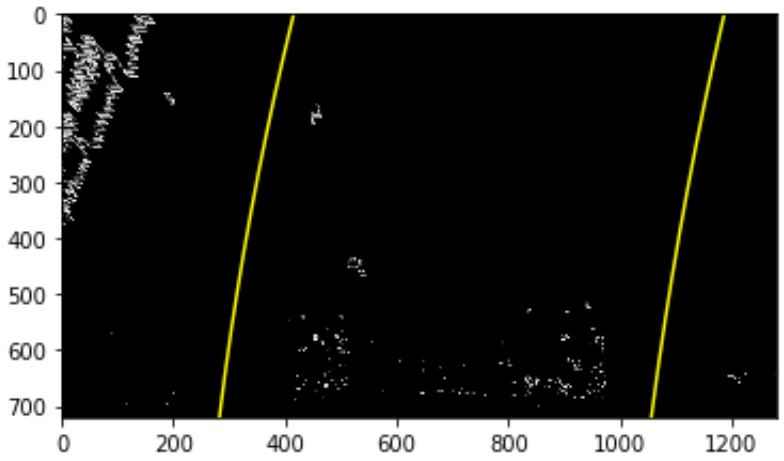
The code for my perspective transform includes a function called `warp_image()`, which appears in lines 18 through 19 in the file `perspective.py`. The `warp_image()` function takes as inputs an image (`img`), as well as mapmatrix (`M`) which is calculated either by calling perspective*transform or inverse*perspective_transform. Both take source (`src`) and destination (`dst`) points I chose to use hardcoded values for the source and destination points in the following manner:

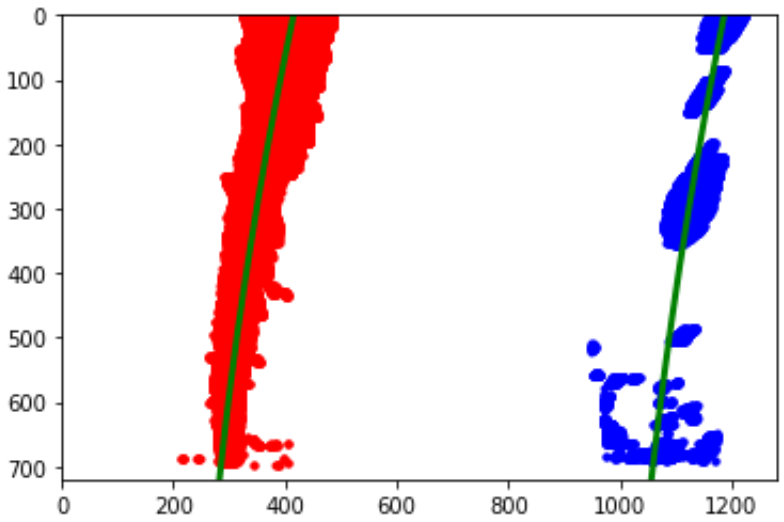
``` self.src = np.float32([[571,460],[700, 460],[1034,673],[276,673]]) self.dst = np.float32([[264,0],[1034, 0],[1034,679],[264,679]])

```

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.







4. Identification lane-line pixels and fit their positions with a polynomial.

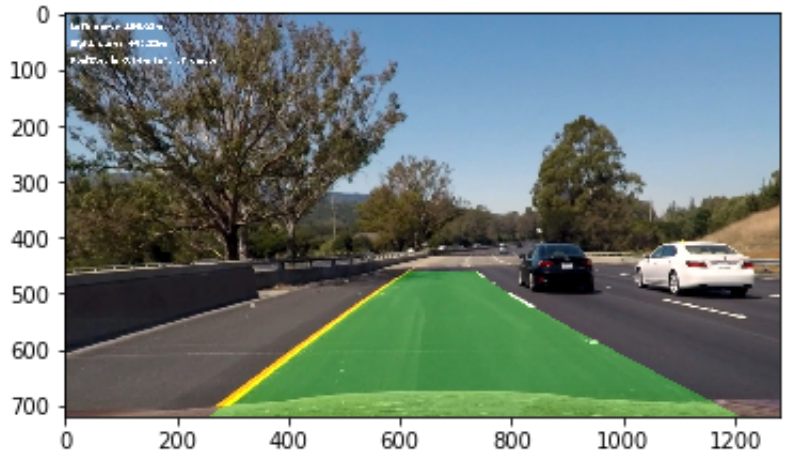
Then I implemented logic detect lane lines and fit my lane lines with a 2nd order polynomial as per code between lines 45 and 130 in file `line.py`:

5. Radius of curvature of the lane and the position of the vehicle with respect to center.

I did calculate the curvature of the lane line and relative position of the vehicle in lines 154 through 177 in my code in `line.py`

6. Example image of your result plotted back down onto the road.

I implemented this step in `CarND-Advanced-Lane-Lines-Final.html` in the function `generate_output()`. Here is an example of my result on a test image:



Pipeline (video)

Here's a link to my video result

Discussion

Some of the challenges faced while building the pipeline where around getting the correct value for different threshold so that the system can identify lane lines more robustly. Combining gradient threshold and color threshold helped me identifying the lane lines. Also I faced challenges while running the pipeline on challenge video which may require fixing the pipeline to make assumption based on previous identification and discarding or giving lower weightage to out liners. I tried implementing something on that lines but could not make it work, it will be something which I will look into separately. Also scenario where we change lines it would be interesting to see how the system behaves.