Advanced Lane Finding Project

Introduction

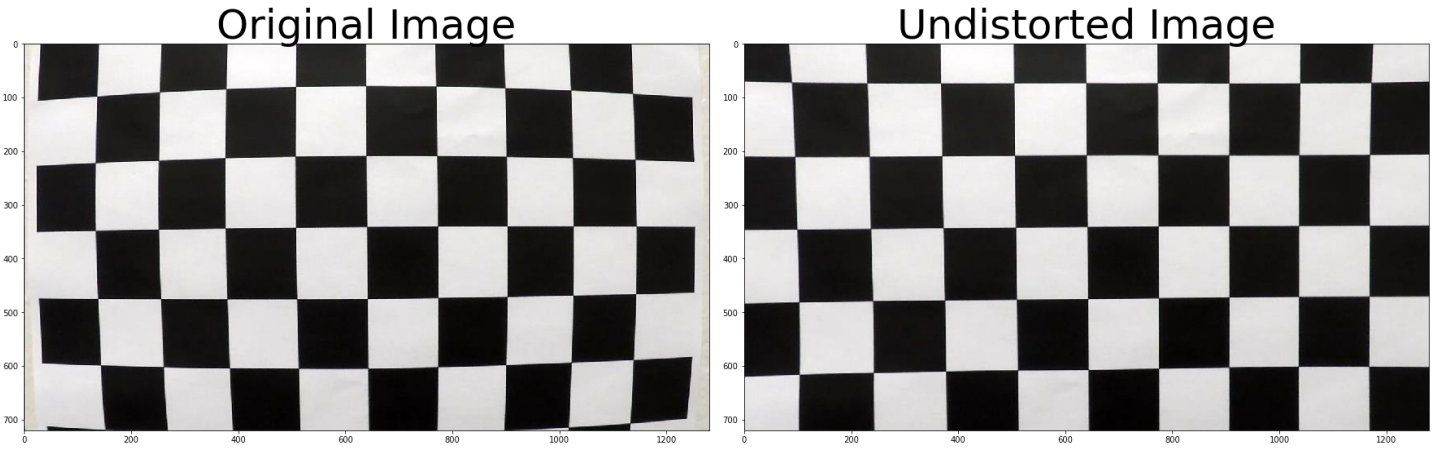
For this project, I used a Jupyter notebook to write and test the code. A significant portion of the basic image processing pipeline is borrowed from the lessons, and I have attributed accordingly in the comment in the notebook cells. I will focus on discussing my unique modifications and improvements.

Camera Calibration

I used the ***glob*** library to read in all the chess board calibration images from the camera_cal folder. I created object point data structure ***objp*** as a 6 by 9 array to reflect the number of inner corners expected in the calibration images. Using ***cv2.findChessboardCorners***, I extracted the pixel coordinates or all the inner corners from each image.

The image points (when found) and object points are loaded into arrays, which is subsequently passed to ***cv2.calibrateCamera*** to derive the distortion coefficients and camera matrix.

I tested the calibration process by undistorting calibration1.jpg, using ***cv2.undistort***. The undistorted output is saved in the output_images folder as calibration1_undistorted.jpg.

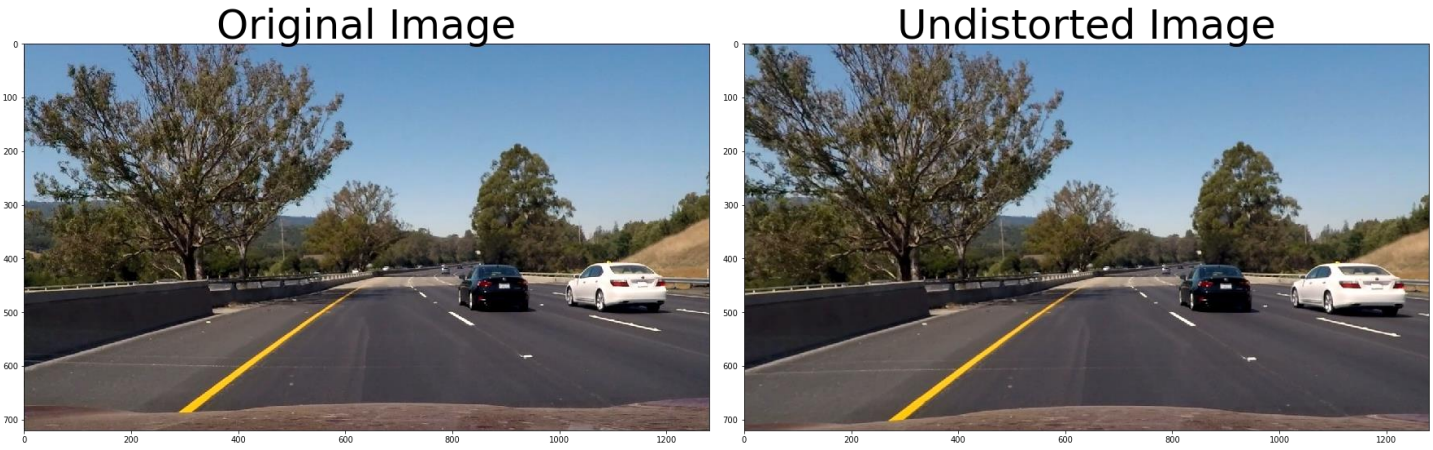


Pipeline

The following sections describe the entire transformation process from start to finish

Undistort Test Image

I tested the distortion correction algorithm on test6.jpg, the output is stored as test6_undistorted.jpg.



Perspective Transform

I created the ***bird_eye_view(img)*** function to convert the undistorted image into bird eye view.

The 4 source and 4 destination points are carefully mapped and defined in the code snippet below to achieve an accurate transform process. The transformation matrix and inverse transformation matrix is derived using the ***cv2.getPerspectiveTransform*** function. The image is then warped to bird eye view using ***cv2.warpPerspective***.

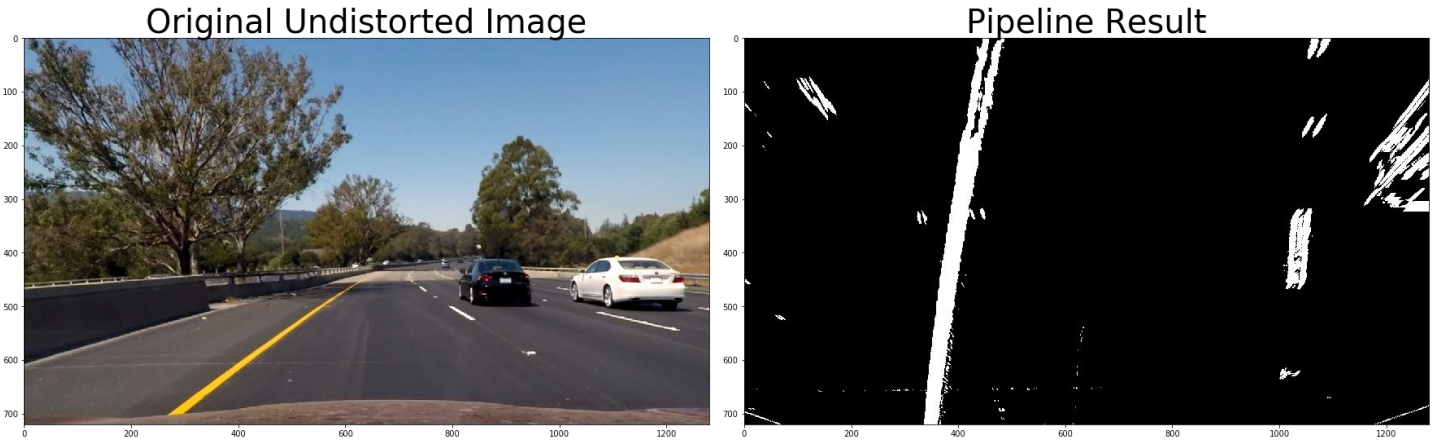
```
    # (290,670),(565,470),(720,470),(1030,670)
    # Lower corners are raised to prevent the hood from being mapped onto the transfo
rmed image
    src = np.float32([[290./1280*w,670./720*h], [565./1280*w,470./720*h], [720./1280*
w,470./720*h], [1030./1280*w,670./720*h]])
    # Map to top down view line end points for left lane and right lane
    # (320,720),(320,0)(960,0)(960,720)
    dst = np.float32([[w/4.,h], [w/4.,0], [3.*w/4.,0], [3.*w/4.,h]])

    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    bird_eye = np.array(cv2.warpPerspective(img, M, (w, h)))
```

I verified the transform algorithm on test6.jpg, the bird eye view transform is saved as test6_birdeye.jpg.

Thresholding

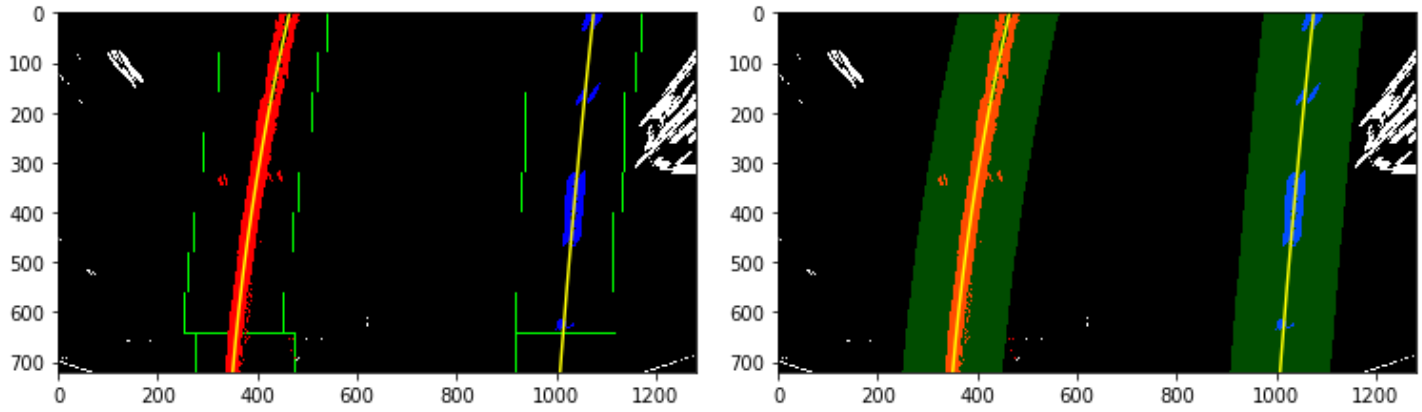
I created the thresholding function ***threshold_bin(img, s_thresh=(150, 255), sx_thresh=(20, 255))***. Code is borrowed from Udacity lesson with minor modifications to the threshold values. This algorithm utilizes the combined output of Saturation channel in HLS model and the Sobel operator in x direction to identify likely lane pixels.

I verified the transform algorithm on test6.jpg, the bird eye view transform is saved as test6_binary.jpg.



Polynomial Line Fit

The sliding window approach from Udacity lesson is used to extract the lane pixels. I used the same code with no modifications. The output is saved as test6_sliding_window_visualization.jpg and test6_polynomial_fit.jpg.



Curvature and Center Offset

Code from Udacity lesson is used with minor modifications to meter/pixel count. Note that the meter/pixel in the y axis was eventually modified to be in line with recommendations from the lesson.

```
# Define conversions in x and y from pixels space to meters
ym_per_pix = 3/150 # meters per pixel in y dimension
xm_per_pix = 3.7/670 # meters per pixel in x dimension
```

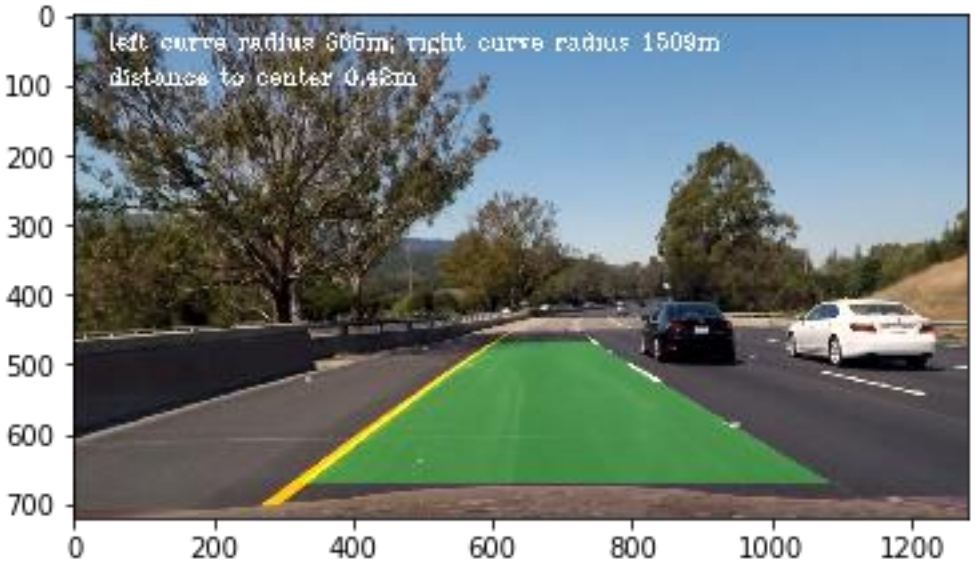
The calculated curvature for left lane, right lane and center offset are 365 m, 1589 m, and 0.42 m respectively

Metadata Overlay

Code from Udacity lesson is used with minor modifications.

The left and right lane pixels are drawn onto the bird eye view with the function ***cv2.fillPoly***. The bird eye view is then transformed back to the original point of view of the camera using the inverse perspective transform matrix ***inv_perspective_M***. It is then overlaid on top of the original undistorted image. Finally, text overlay is added with the curvature and center offset information, using the ***cv2.putText*** function.

I ran the pipeline on test6.jpg. Output files is saved as test6_metadata_overlay.jpg.



Video

output.mp4

Discussion

The video pipeline is defined as a function called ***process_image(frame)***.

It consists mostly of code snippets from all the stages of the pipeline previously discussed. However, there are some unique features and enhancements that I have introduced to help is become more robust.

After running the standard pipeline on a few difficult sections of project_video.mp4, I realized that shadows and drastic color changes in the road are a problem. The thresholding algorithm will fail to isolate the lanes, and create “bad” frames. My algorithm essentially allows “bad” frames to be skipped and replace the polynomial line fit with data from the last “good” frame, until a maximum number allowed skipped frames has been reached.

I’m relying on two assumptions

1. Lane curvature differences between adjacent frames are minimal and visually undetectable.
2. The shadows/color transitions a brief (usually no more than 0.5s).

I will explain the high-level concept here, for details, please see jupyter notebook comments.

I first define a list of global variables to store the relevant polynomial line fit and curvature values.

```
# Define global parameters for storing line-fit polynomials, curvature and offset dat
a to replaced "bad" frames
# Only the parameter values from the last single "good" frame is stored; the paramete
rs are updated every time a "good" frame is detected
first_frame = 0
left_fit_LG = None
right_fit_LG = None
left_fitx_LG = None
right_fitx_LG = None
left_curverad_LG = None
right_curverad_LG = None
centerdist_LG = None
skip_count = 0
```

The pseudocode is as follows

*if **is_first_frame == True** or **skip_count > max_allowed**:*
    *increment **first_frame***
    *reset **skip_count** to 0*
    *run **sliding window search** from scratch on **current frame***
    *update **global variables** with **current frame data***
else:
    *run search **within a narrow band** of the **previous frame polynomial** stored in **global variables***
    *compute **deviation** between **current frame** and **previous frame***
    *if **deviation** > **threshold**:*
        *frame is **bad**, skip frame, increment **skip_count** by 1*
        *replace **current frame parameter** with **global parameter** from **previous frame***
    else:
        *frame is **good**, reset **skip_count** to 0*
        *update **global variables** with **current frame data***

The important tuning parameters are **max allowed skip_count**, **band width**, and **threshold**. After experimenting, the optimal values I came up with are **15 frames** for **max allowed skip_count**, and **20 pixels** for both **band width** and **threshold**. The algorithm seems to perform quite well on project_video.mp4 (some minor jitter remains in the shadow sections, but does not cause the overlay to drift out of the lane, and is able to recover quickly to precise tracking as soon as the lighting conditions improve).

The basic video is captured under very ideal conditions, so I expect the pipeline to fail if lighting conditions are worse due to either glare or shadow. Perspective transform may also be less accurate is the rate of change of the slope of the road is not gradual or uniform (top of the hill or bottom of the hill). It could also be susceptible to other visual noises on the road such as potholes and residual lane markings that does not overlap with newly painted markings.

The pipeline can be improved by using more sophisticated threshold detection algorithms and path planning algorithms that is able to predict lane curvatures ahead based on current and previous curvature data.