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 colour 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 centre.

* 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]: (https://review.udacity.com/#!/rubrics/571/view)

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

Camera Calibration

The code for this step is contained in the second code cell of the IPython notebook located in "ProjectCode.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.

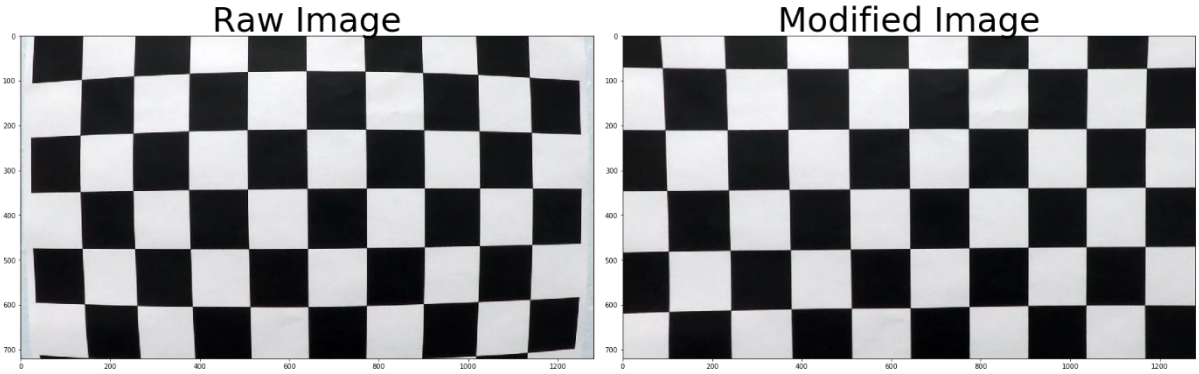
To notify me that the camera calibration was completed successfully, I added a few lines of code that would print "Camera Calibrration Complete", if data was detected in the object points.

I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

All images were produced and compared to the original to highlight affects, using the function I created, called "visual"

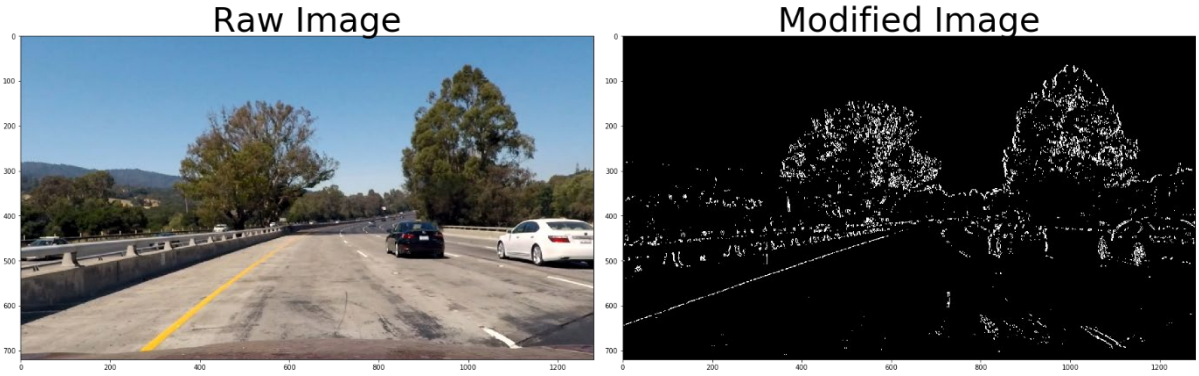
1. Provide an example of a distortion-corrected image.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

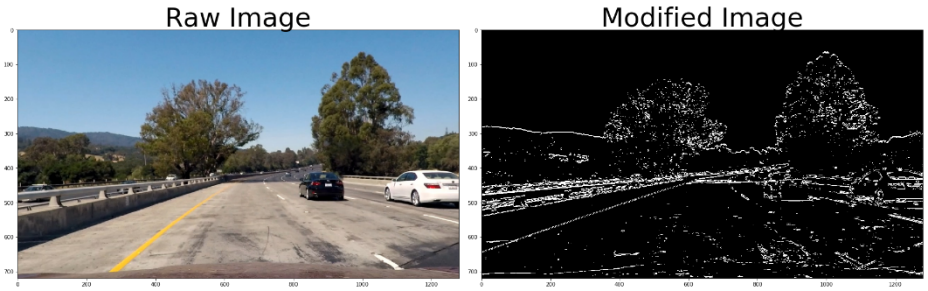


2. Describe how (and identify where in your code) you used colour transforms, gradients or other methods to create a thresholded binary image

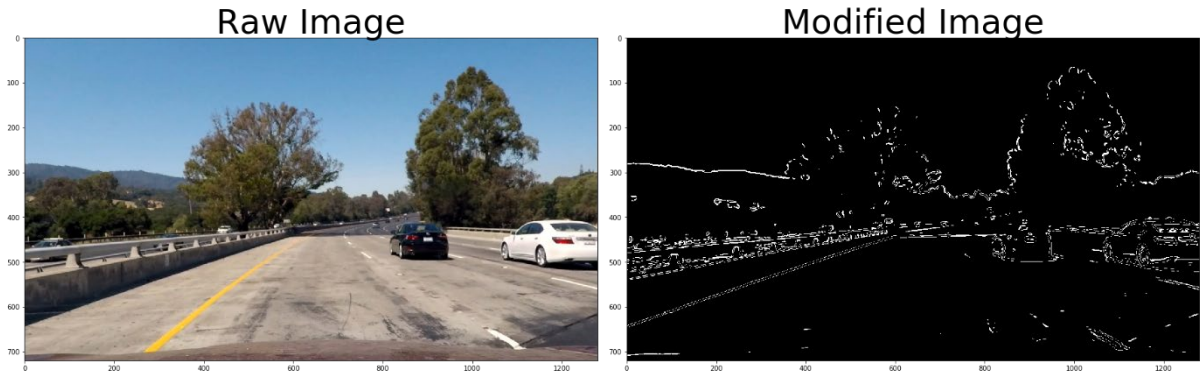
Gradient Binary[x]



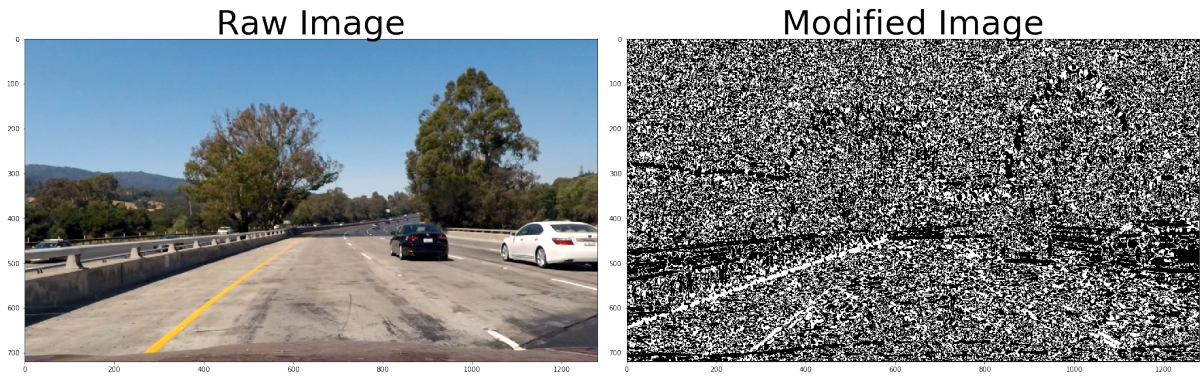
Gradient Binary[y]



Magnitude Gradient



Directioon of Gradient



HLS & Colour Thresholding

Combination of all the thresholds above

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image

The code involved in conducting a perspective transform is within the project code file, under the title 'Perspective Transform.'

I deduced source and destination points in which to conduct the transform

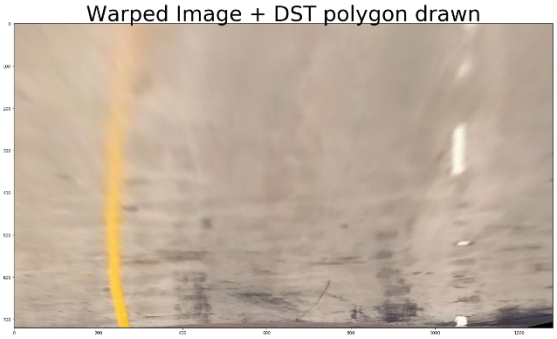
```
if src is None:
    src = np.float32(
    [[265, 700], #bottom left
    [595, 460], #bottom right
    [725, 460], #top right
    [1125, 700]]) #top left

if dst is None:
    dst = np.float32(
    [[250, 720], #bottom left
    [250, 0], #bottom right
    [1065, 0], #top right
    [1065, 720]]) #top left
```

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.

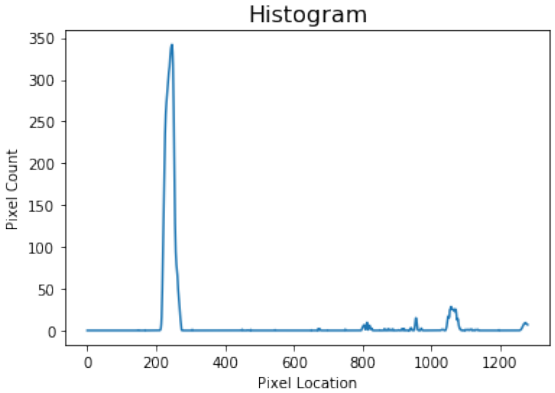
Raw Undistorted image + SRC polygon drawn

Warped Image + DST polygon drawn

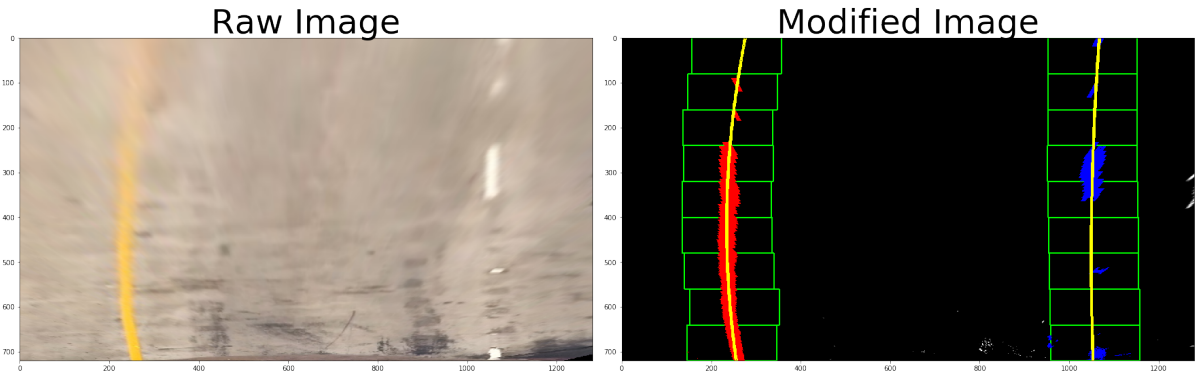


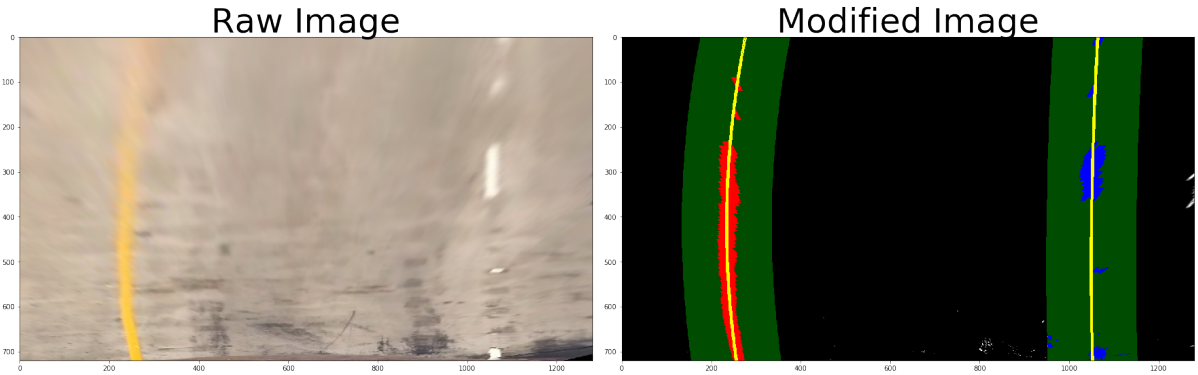
4**. Detecting Lane Lines**

Using a thresholded version of the warped image on the right, I was able to use a histogram function to count up the pixels, to detect lane lines, using the two histogram peaks.



The lane lines were then detected using the ‘prior method’ or in cases where this wasn’t feasible, the sliding windows method, in order to fit a polynomial which would represent the lane lines.





Measuring Curvature

This section is labelled clearly #6, within the notebook, and features the code used to calculate curvature, where the real lane line curvature was calculated using a conversion from pixels to metres.

Vehicle Offset

Following this, vehicle offset was determined to identify the position of the vehicle in respect to the centre of the lane.

Filling in Lane Lines

To make it clear for the vehicle, I was then able to produce code to construct a waypoint formulated from a filled polygon.

Overlaying Data

Data about the curvature and offset was then overlayed over the screen.

Pipeline

The pipeline class was constructed and then used to run on the project_video.mp4, successfully producing project_video_result.mp4.

Discussion

A lot the issues faced in this project came from the shape of the images used, because there were a lot of 3D and 1D images.

It became important to ensure I was consistent with what image format I used in each function and it meant that I had to be aware of the image sizes throughout, to prevent errors.

The code failed on the challenge video, however by potentially adjusting thresholds, it may be able to work when going underneath bridges.

It may also be ideal to create a function that decides that if the polygon fails to retain its shape, the code restarts and redetects the lane line.