writeup.md

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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.

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.

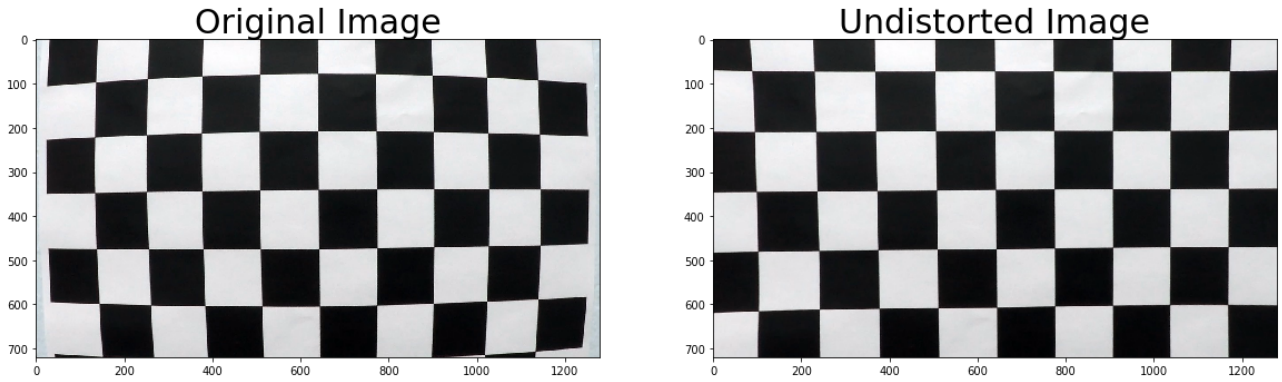
The code for this step is contained in the first code cell of the IPython notebook located in `./P4-camera-calibration.ipynb`.

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

Examples of images that visualize how the corners were detected during the camera calibration operation can be found in this folder: `./output_images/corners/`.

Moreover, here are some examples of undistorted images, after performing the camera calibration.



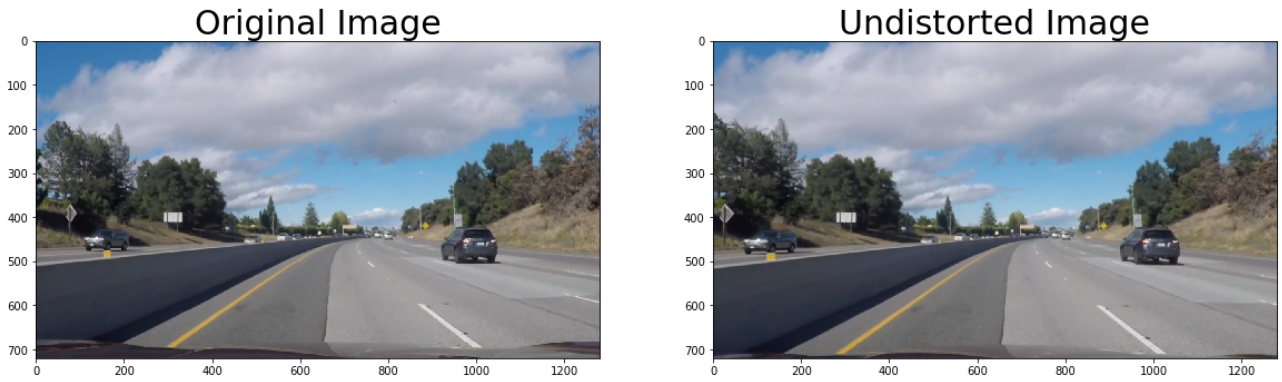
Pipeline (single images)

In order to test most of the building blocks presented here, I went on and took a couple of screenshots from all 3 provided videos in order to augment the test images that I received initially.

Most of the single images tests were performed in the `./P4-development.ipynb` notebook.

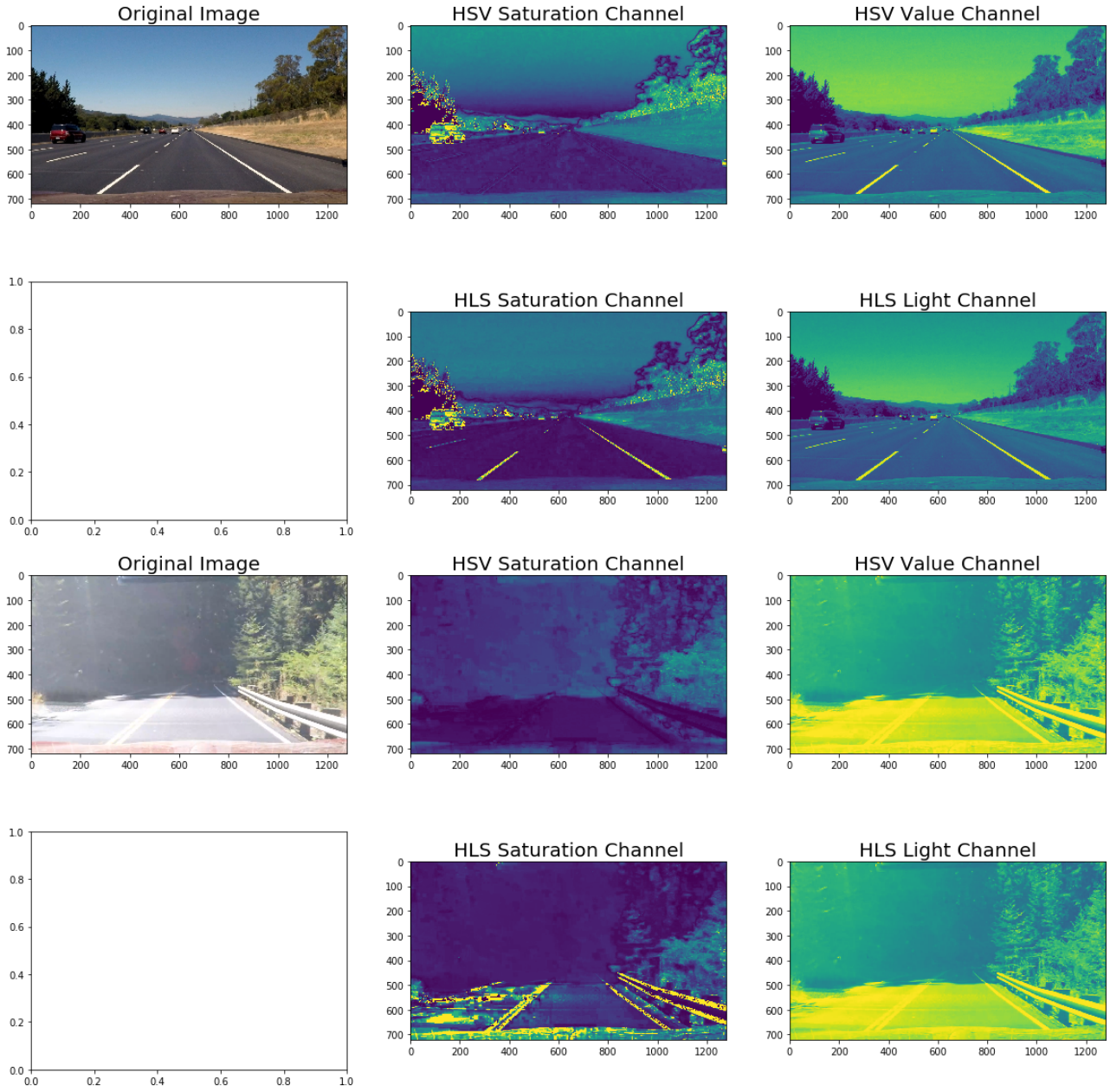
1. Provide an example of a distortion-corrected image.

Given the camera matrix calculated in the `camera-calibration` notebook, I loaded them and applied OpenCV's `undistort` method on one of my test images. Here is the result:



2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I started by replicating my approach from *Project 1*, trying to create some color masks first. I created 2 color masks, for **yellow** and **white** that I applied on images converted to the **HSV Color Space**. I also tried out the **HLS Color Space**, which I used later on with the gradient thresholds.



As it can be noted, the **Saturation** channel of the **HLS Color Space** does a pretty good job at detecting lines even in tough conditions of bright lumosity, as it is illustrated in the second image, above. **HSV*'s **Value** channel also provides good contrast, and this is the reason why I used it for applying gradient thresholds on it.

Something to note here is that I only applied color transforms and gradient thresholds on **warped** (perspective transformed) images, not on the entire input frame. I figured that in this way I will reduce computation a bit, by only focusing on the area of interest.

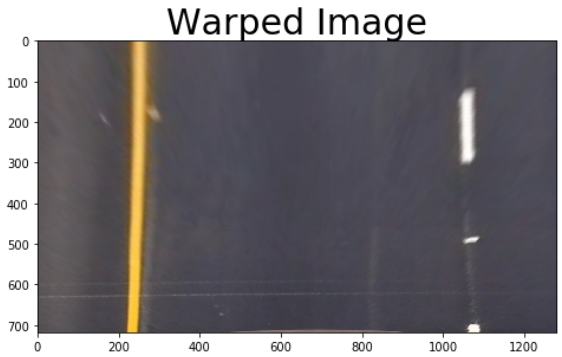
For the gradient thresholds, I used a combination of:

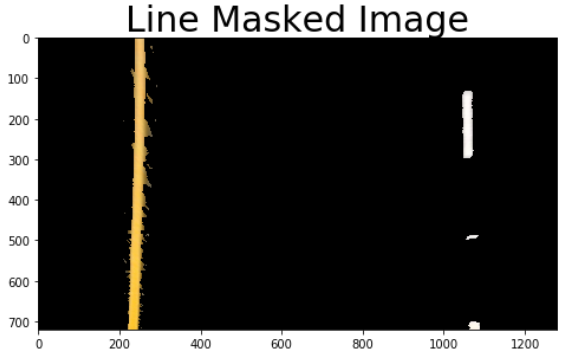
- Absolute Sobel Threshold
- Magnitude Sobel Threshold
- Direction Sobel Threshold

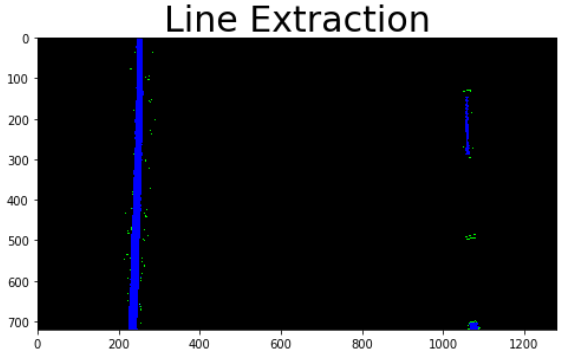
All of these were applied to the **Value** channel of the **HSV** warped image. Moreover, I played around a bit with the kernels and threshold values for some of them, specifically the *direction sobel*. Did not manage to make a big impact, but wanted to try it out.

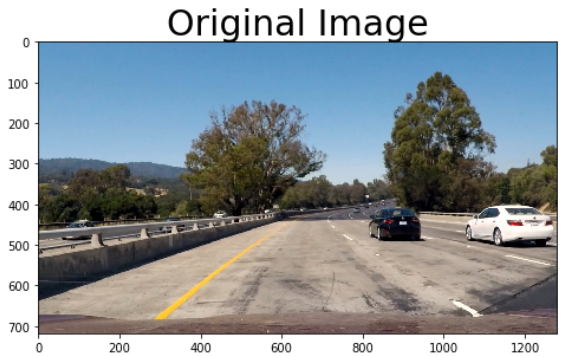
Here are some results, applied on test images.

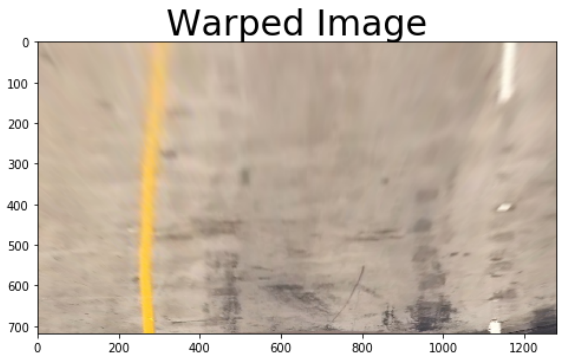

Original Image

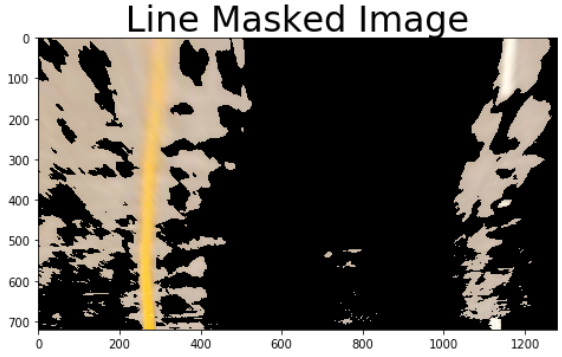

Warped Image

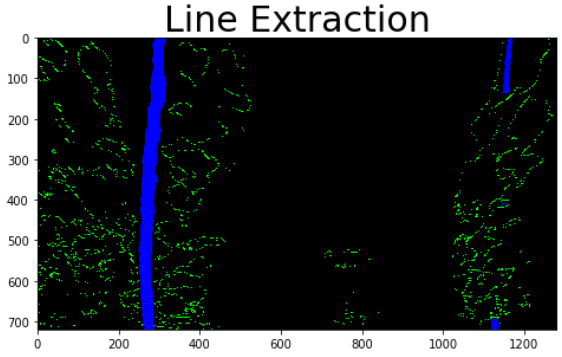

Line Masked Image

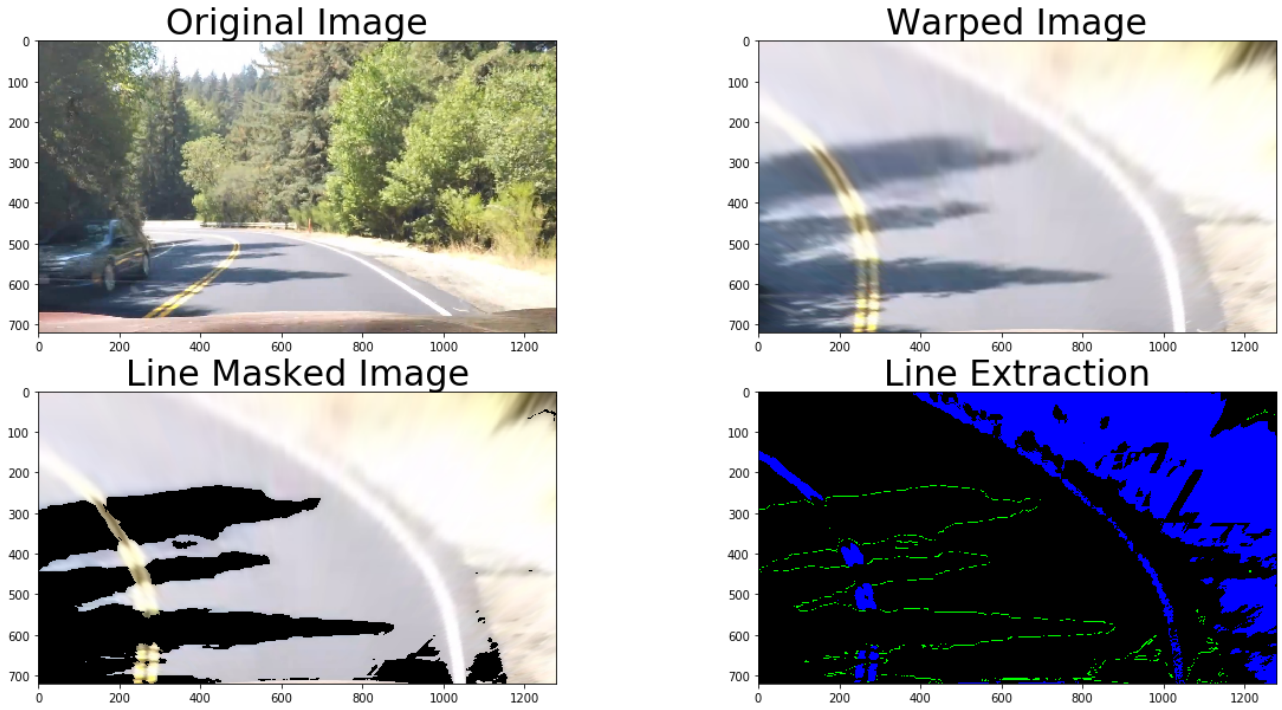

Line Extraction


Original Image


Warped Image


Line Masked Image


Line Extraction



As it can be seen, I still had quite some trobel with shadows. My initial idea was that the *direction sobel* will be able to clear these out, but I was not very successful. Given how the sliding window line detection works, the shadows (in this test case at least) did not prove to be a big issue though.

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

In order to perform perspective transform, I first selected some points of interest in the image. The reasoning for my selection was the heavily focused on trying to capture sharp turns as well. For this reason, the polygon that I decided to warm is not very "long" (i.e. it does not go to far away) and it tries to be quiet wide, in order to capture possible sharp turns, as exemplified in the `harder challenge video`.

Here are the points, visualized on test images, along with the actual transformation. As you can see, the right image is one that contains a sharp turn.

So far so good in terms of both capturing sharp turns and capturing a decent depth of straight lines.

These is the code used for capturing these points, along with the pixel coordinates.

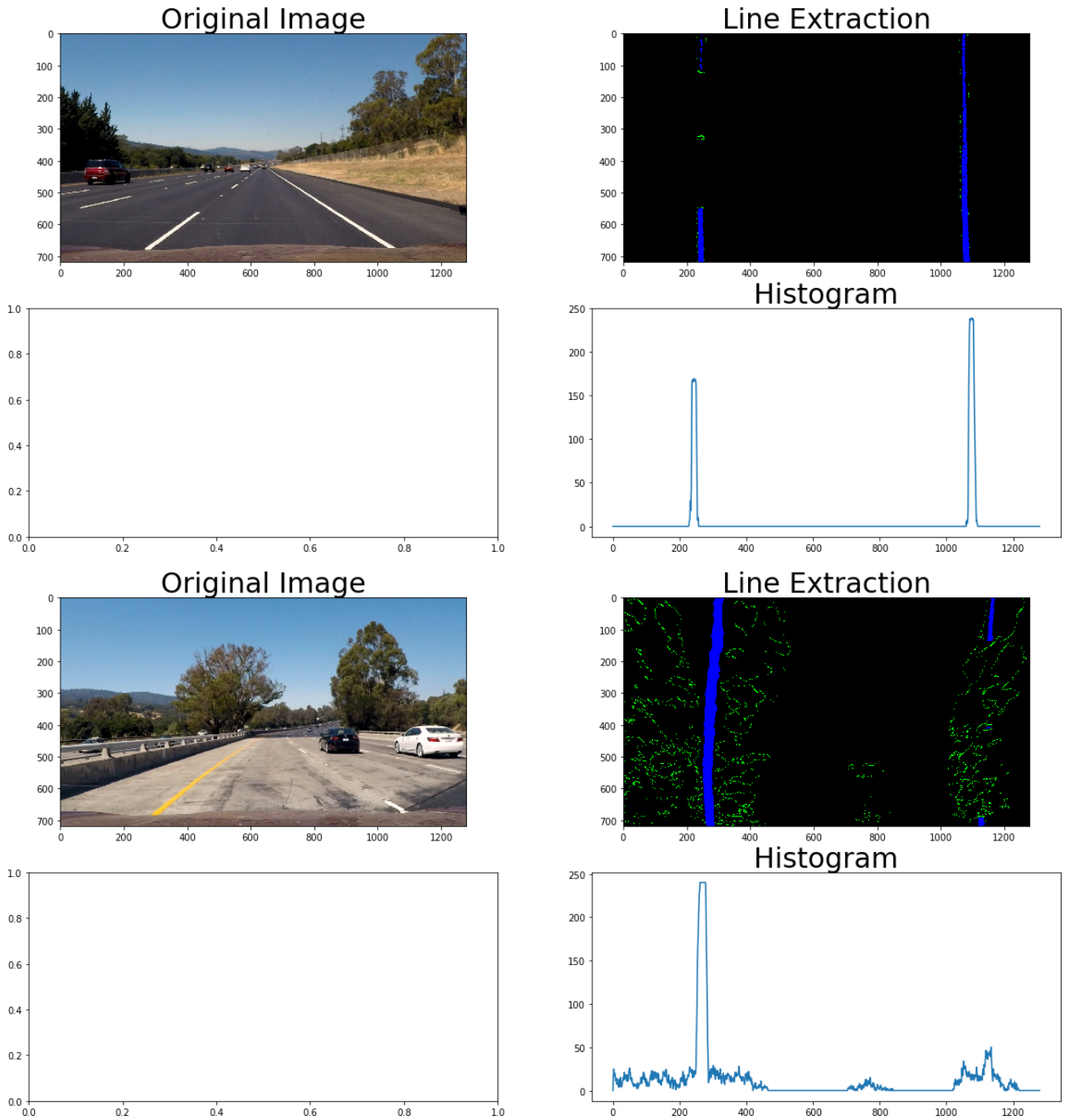
```
x1, y1 = (500, 480)
x2, y2 = (780, 480)
x3, y3 = (1200, 670)
x4, y4 = (80, 670)

src = np.float32(
    [[x1, y1],
     [x2, y2],
     [x3, y3],
     [x4, y4]])

offset = 10
dst = np.float32(
    [[offset, offset],
     [test_img_size[0] - offset, offset],
     [test_img_size[0] - offset, test_img_size[1] - offset],
     [offset, test_img_size[1] - offset]])
```

This resulted in the following source and destination points:

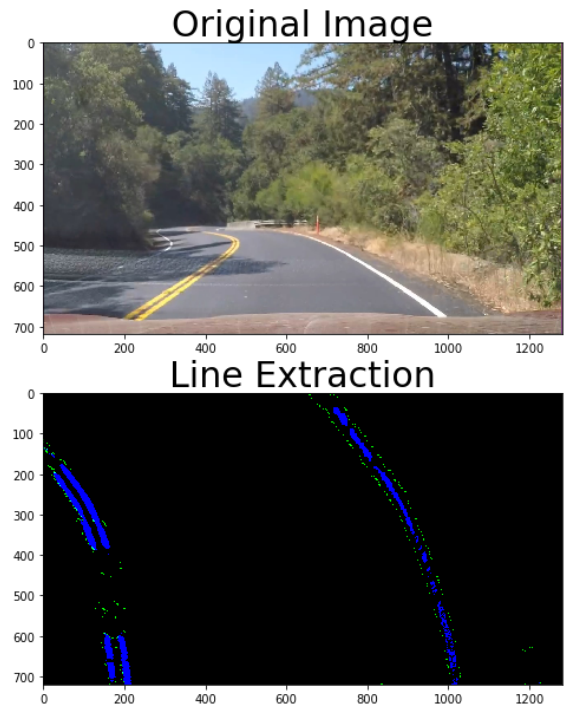
Source	Destination
500, 480	10, 10
780, 480	1271, 10
1200, 670	1271, 710
80, 670	10, 710

4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

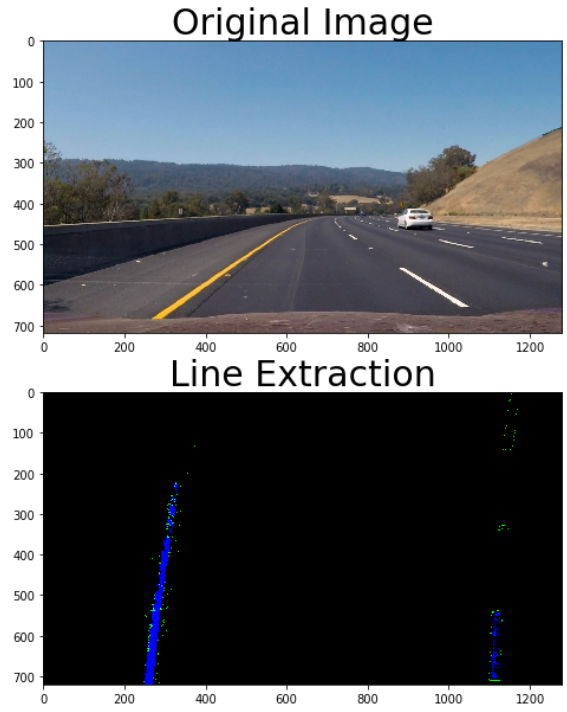
In order to identify the lines, I first calculated the histogram of each of the transformed images. This helped in detecting where to start looking for the lines in the first place.



Afterwards, in order to get the line fits, I used the sliding window approach in order to detect where are the pixels forming the lines.



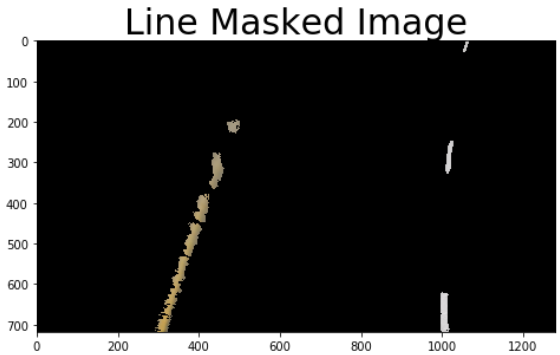
This approach starts from the base of the image, from the points of highest value on the histogram, and goes upwards trying to follow the trend of the location of pixels that form the line. If, in the following window, the location seems to shift to the left or to the right, the next window will be centered accordingly.

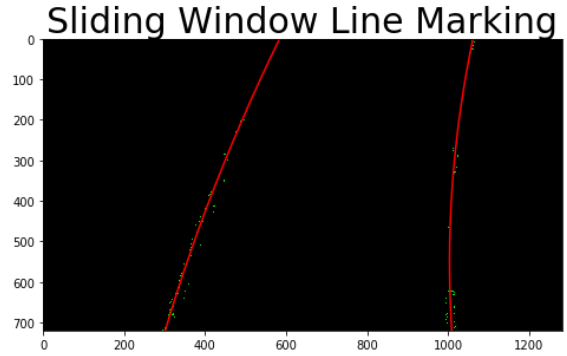


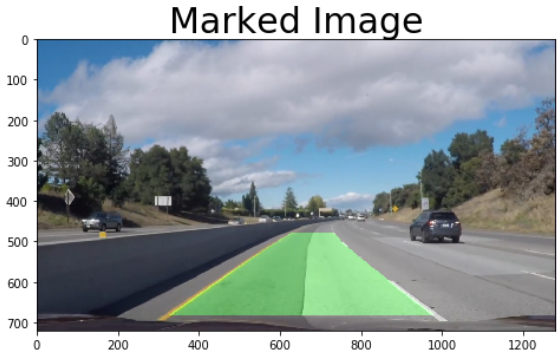
Once gathered all the pixels that formed the 2 lines, I used `numpy`'s `polyfit` to fit a 2nd degree polynomial along those points. Having the coefficients of the 2 parabolas, I then calculated the `x` and `y` points associated with the line.

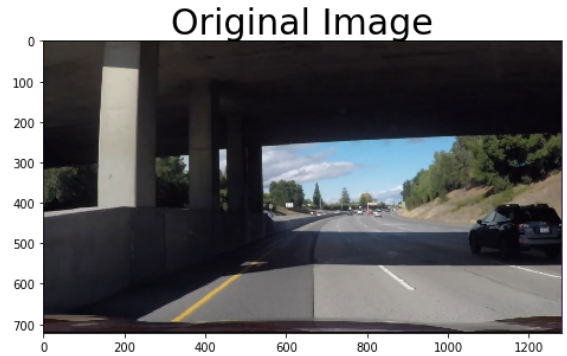
The code for this procedure can be found in the `get_line_fits` method, in the `development` notebook.

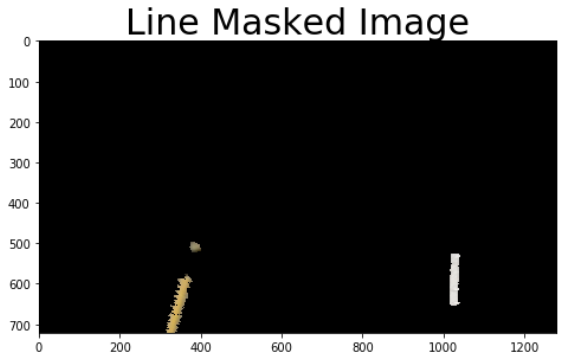
Having the `x` and `y` pixel coordinates, I was then able to draw them onto the warped image. Moreover, I drew a polygon defined by the endpoints of the 2 parabolas and unwarped the image to the original perspective. Below are examples of the result.

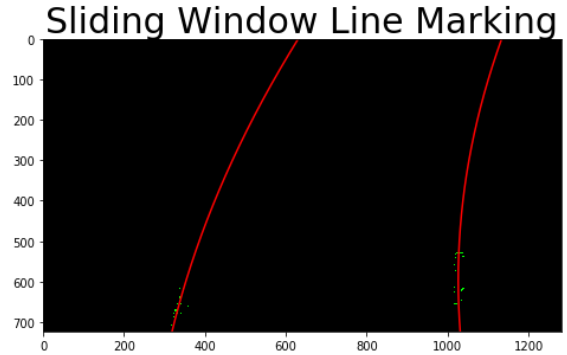


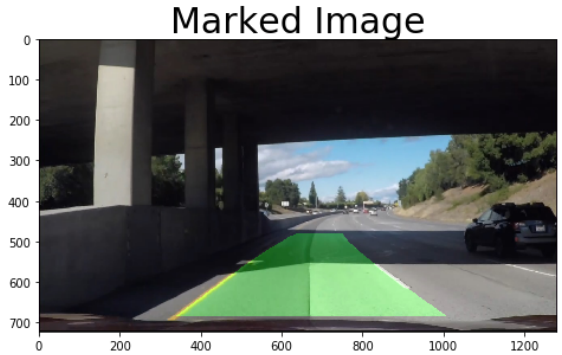




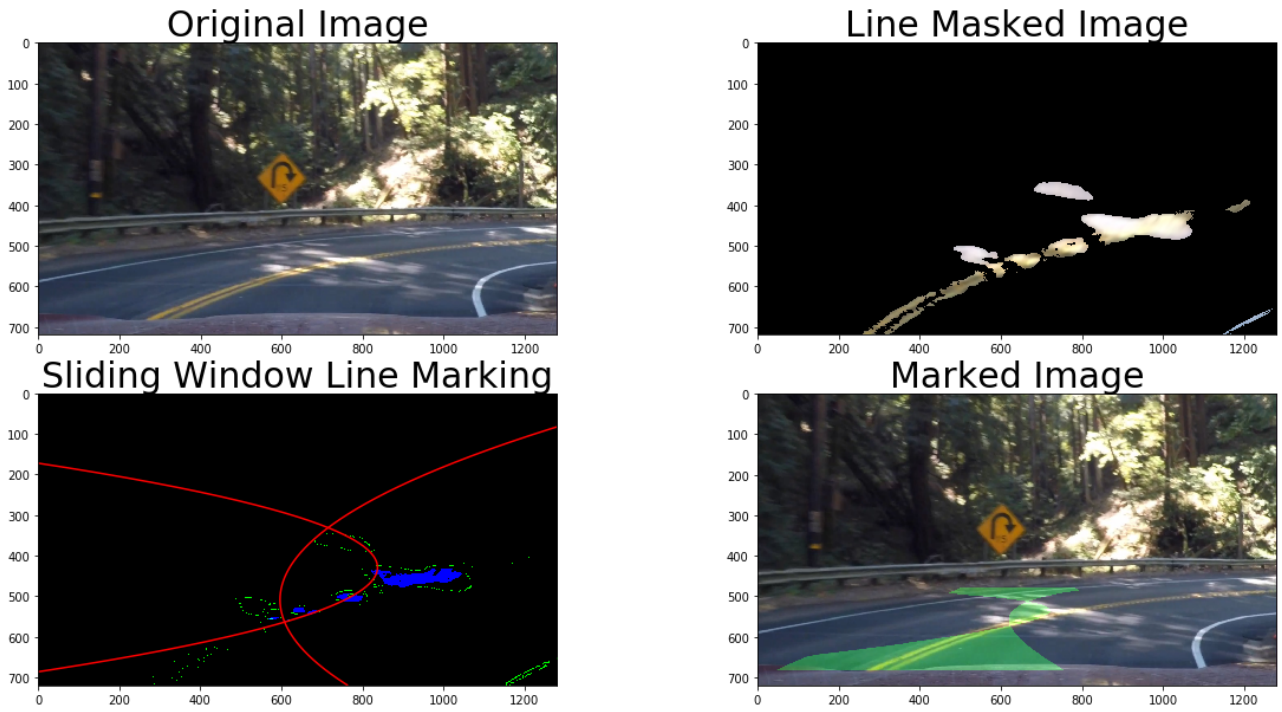








Remember my initial thought on trying to capture the sharp turns? Well, this is where everything started failing. And this is the reason why the challenge was called *harder challenge*. Basically the lumosity of the environment, along with the distortion created by the curve, made it impossible to extract the line from my color transformed and gradient thresholded warped image. Here is what happened:



5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

In order to calculate the radius of the curvature, I used the following tutorial. Moreover, I used the notes from the lecture to transform the pixel-based curvature into a meters-based curvature.

In order to calculate the center-offset of the car compared to the lane it was driving on, I calculated the distance in between the two lines (left and right), and based on that I found the center distance between them. This was the center of the lane. The position of the car was described by the center of the entire image, given that the camera was placed right in the middle of the car's width (i.e. at the center of the screen window). From this point on, I just had to calculate the difference between these two numbers, which gave me the car offset from the center of the lane.

A visual representation of these numbers is provided below, in the next sub-section.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Here are examples of the output frames from my pipeline:

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a link to my video result

Here's a link to my challenge video result

Here's a link to my harder challenge video result

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

While working on this pipeline, I noticed some issues that I will address below.

- Right now, pipeline is **very** slow, for a real-time system.
  - I think this has a mixture of causes, one being Python (and Jupyter maybe), and others being the somewhat tedious algorithms of window sliding across the image.
  - One way of improving this could be to port the algorithm to a real-time language (C++) and also to have a smarter way of using previously detected line fits when looking for the current one; this leads me to my next issue.
- Using previously found line fits.
  - For some reason, this did not work consistently well for me. It worked on the normal video alright, but not too well on the **challenge_video**. I spent some time debugging and understanding what was the issue, but I decided not to use the feature in the end, as I couldn't see a major difference in terms of processing times of the video.
- Area of interest for perspective transform.
  - I specifically picked a narrow-length and wide-width polygon in order to capture sharp turns. I think a better approach would be a dynamic one, in which you detect from previous frames if a sharp turn is about to happen (or, even better, you detect through `CNN`s that there is a *Sharp Turn* road sign ahead :)) and only then you create a area

of interest polygon wide enough to capture the edges. Otherwise you can just use a long-length and slimmer one, to capture more of the depth of the road ahead.

- Brightness is a killer.
  - There are certain portions of the *harder challenge video* where not even my eye can see where the lines are. I tried to think of what my brain looks for in those situations, and I realized that I look way farther ahead, to see the shape of the "environment" as it would go deeper in the perspective. This would be an interesting approach to try, and I assume that the industry standards look for this as well, not only the lines.
- Direction sobel threshold did not work as I expected.
  - For some reason I was not able to clean up the shadows of the trees using the direction sobel threshold. I specifically gave it to narrow it down to gradients with a slope between 45 and 135 degrees, but there were still a ton of shadow-generated gradients that were picked up. I need to do more research into how to clean them up. Maybe *Sobel* is not the best tool for this.